2005－CG－120（3）
2005／8／18

GPU を用いたリアルタイム高次元テクスチャマッピング

関根 真弘

（株）東芝 研究開発センター

複数のカメラ条件および光源条件によって取得したテクスチャ（高次元テクスチャ）を利用することにより，CG モデル表面の高品位な質感表現が可能となる．本稿では，グラフィックスハードウェア (GPU) を利用することによって，リアルタイムでの高次元テクスチャマッピングが実現できることを示すとともに，テクスチャデータのメモリへの配置方法によるレンダリング性能の変化を比較評価する．さらに，高次元テクスチャ圧縮に伴う画質およびレンダリング性能への影響を評価する．

Efficient High-dimensional Texture Mapping Using Graphics Hardware

Masahiro SEKINE

Corporate Research & Development Center, TOSHIBA CORPORATION

The high-dimensional texture is a set of images that is captured under various conditions. This report presents a high-dimensional texture mapping technique for realistic surface representation. The per-pixel texture mapping that depends on view- and light-direction can be realized in real-time on graphics hardware. The mapping cost can be reduced by modifying the arrangement of the high-dimensional texture data. The influence on the quality of images and the mapping cost by the high-dimensional texture compression is also evaluated.

1. はじめに

近年，実写を利用した高品位なコンピュータグラフィックス (CG) 表現の研究が広くなされており，実写の取得・分析・モデル化に対する様々な手法が提案されている．その代表的な例として，BRDF (Bidirectional Reflectance Distribution Function) や BTF (Bidirectional Texture Function) 等が挙げられる[1]. これらの手法では，実写データから取得した物体表面の光学特性を反射分布関数としてモデル化し，テクスチャマッピングの際に，反射分布関数に基づく数値計算を行なうことによって，視点や光源位置の変化に応じたリアルな表面質感を表現することができる．

しかしながらこのような関数化手法は，スペキュラ成分をもったラメ素材や自己遮蔽の多い複雑な網目素材など，視点や光源位置の変化に応じて不連続な色変化を伴う素材に対し，関数導出が困難であるという問題がある．そのため，ある素材に特化したモデル化を行なったり，視点・光源条件を限定したりする手法が提案されている．例えば，視点変化に限定した光線空間の解析手法として Light Field Rendering が知られており[2]，光源変化に限定した輝度分布の関数化手法として Polynomial Texture Mapping が知られている[3].

一方，筆者らは，複数の条件で取得したテクスチャ群（高次元テクスチャ）をそのまま保持し，CG モデルの視点／光源条件に応じてテクスチャを選択的にマッピングする手法を提案している[4,5]．この手法は，前述した関数化手法とは異なり，生のテクスチャデータを利用するため，どのような素材に対しても忠実に質感を表現できるという特徴がある．その反面，離散的なテクスチャサンプルの補間計算のためにレンダリングコストを要してしまう，データ量が膨大となってしまうといった問題があった．

本稿では，まず高次元テクスチャとそのマッピング手法を説明する．次に，レンダリングコストを抑えるために，グラフィックスハードウェア (GPU) を有効に利用し，高次元テクスチャマッピングがリアルタイムで実現できることを示す．特に，テクスチャデータのメモリへの配置方法によるレンダリング性能の変化を測定し，最適なテクスチャ配置方法を示す．また，データ量を削減するために，高次元テクスチャに対してブロックベース・テクスチャ圧縮を適用し，画質やレンダリング性能への影響を評価する．

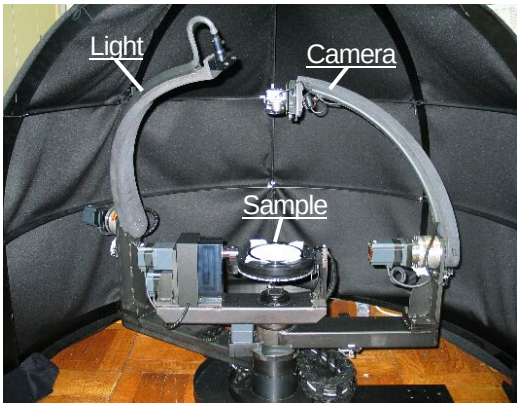

図 1　高次元テクスチャ取得装置

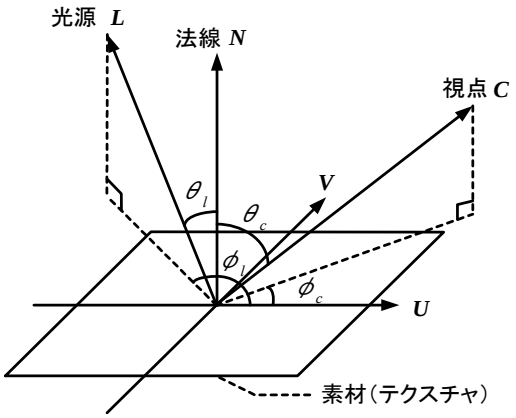

図 2　高次元テクスチャの取得条件

2. 高次元テクスチャ

本稿では，複数の異なる条件で取得したテクスチャの集まりを高次元テクスチャと呼ぶ．高次元テクスチャは，図 1 に示すような装置を用いて取得する．この装置では，図 2 に示すような極座標系でカメラ（視点）と光源をそれぞれ移動させ，複数の視点条件，複数の光源条件下で撮影したテクスチャ群を取得することができる[4,5]．一般にテクスチャとは，テクスチャ座標 (u, v) を指定することによってピクセルデータを抽出できる 2 次元データであるが，視点および光源位置に応じて変化するテクスチャ群は，6 次元データとして解釈することができる．つまり，視点の極角，方位角を θ_c, ϕ_c，光源の極角，方位角を θ_l, ϕ_l とすると，高次元テクスチャからピクセルデータを取り出すためには，6 つのパラメータ $(u, v, \theta_c, \phi_c, \theta_l, \phi_l)$ が必要となる．

高次元テクスチャの取得において問題となるのが，テクスチャのサンプリング方法である．粗い間隔でサンプリングしてしまうと，CG における質感の再現性が悪くなってしまい，逆に細かい間隔でサンプリングしてしまうと，撮影時間がかかる上データ量が膨大となってしまう．そこで，テクスチャデータのメモリ配置の容易性も考慮した上でサンプル数を 2 のべき乗とし，以下のようなサンプリング間隔でテクスチャを取得した．

- 極角　：　0～ 70° を 10° 間隔　［8 サンプル］
- 方位角：　0～360° を 24° 間隔［16 サンプル］

よって，視点／光源方向ともに 128 通りのサンプリングを行なうこととなり，合計 16,384 枚のテクスチャサンプルを取得しなければならない．テクスチャサンプルのサイズを 32×32 ピクセルとすると，データ量の合計は，64MB（4Byte/pixel として算出）となる．このデータ量は，後述する現在の GPU において，1 つのテクスチャデータとして格納できる最大のサイズである．つまり，高々32×32 ピクセルのテクスチャしか格納できないわけである．

32×32 ピクセルといったテクスチャサイズでは，CG モデルにマッピングするにはあまりにも小さい．そこで，小さなサイズの高次元テクスチャから任意サイズのシームレスな高次元テクスチャを生成する技術（高次元テクスチャ生成技術）が重要となる．この技術を用いることで，小さなサイズの高次元テクスチャデータを有効利用して，十分な品質の高次元テクスチャを生成することができる[4,5]．

高次元テクスチャ生成技術を用いるために，高次元テクスチャデータを図 3 (a), (b) に示すようなインデックスデータとコードブックデータのペアで表現する．インデックスデータとは，新規に生成された高次元テクスチャの各ピクセルがもとのテクスチャサンプルのどのピクセルデータを参照すべきかを指し示すものであり，コードブックデータとは，複数の条件で取得したテクスチャ群をまとめたものである．図 3 (a), (b) は，コーデュロイ生地を様々な視点／光源条件で撮影した例であり，32×32 ピクセルのテクスチャサンプル群からなるコードブックデータ（図 3 (b)）と，コードブックデータをもとに新規に生成した 128×128 ピクセルのインデックスデータ（図 3 (a)）である．インデックスデータは，本来，テクスチャサンプルのテクスチャ座標を示す数値データであるが，u と v の数値データをそれぞれ赤色と緑色に割り当てることによって可視化している．

以上で説明した高次元テクスチャデータを 3 次元 CG モデルにマッピングする方法を次節で論じる．

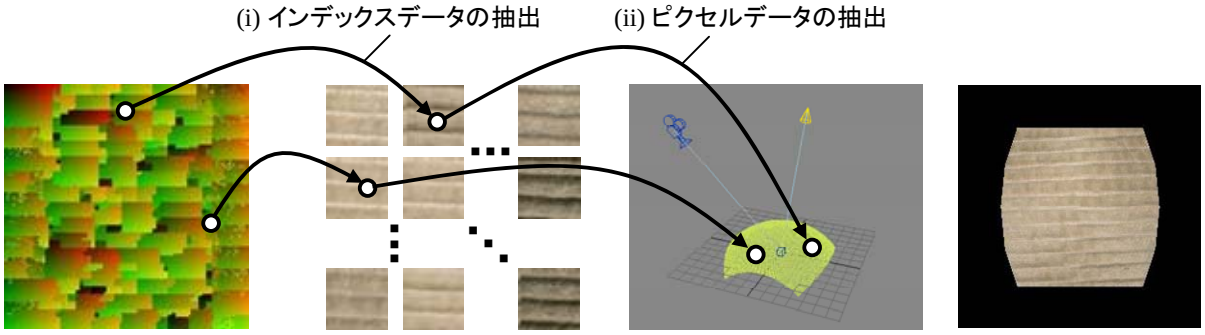

図3　高次元テクスチャデータとそのマッピング

3. 高次元テクスチャマッピング

高次元テクスチャマッピングとは，通常のテクスチャマッピングとは異なり，視点／光源条件に応じて，テクスチャサンプルを選択しながらマッピングする手法である[4,5]. 図 3 は, 高次元テクスチャマッピングの概念図である．CG モデル上のそれぞれの位置に応じて視点／光源条件が異なる（図 3(c)）ため，それぞれ異なったテクスチャが選択される．このようなマッピングによって，光が真上から照射されている部分は明るくなり，斜めから照射されている部分は暗くなるなど，視点や光源に依存して変化する素材の特性をうまく再現することができる．また前述したとおり，高次元テクスチャデータは，インデックスデータとコードブックデータで構成されているため，インデックスデータからインデックスを抽出し（図 3(i)），その後にコードブックデータから適当なピクセルデータを抽出する（図 3(ii)）といった 2 段階の処理によってデータを取得し，マッピングを行なうことになる．

それでは，マッピング処理の詳細アルゴリズムを示す．高次元テクスチャマッピングに必要となるピクセル単位のデータは以下の 5 つのデータである．

① 位置座標　$\boldsymbol{P}=(x,y,z)$
② テクスチャ座標　$\boldsymbol{T}=(u,v)$
③ 法線ベクトル　$\boldsymbol{N}=(x_n,y_n,z_n)$
④ U ベクトル　$\boldsymbol{U}=(x_u,y_u,z_u)$
⑤ V ベクトル　$\boldsymbol{V}=(x_v,y_v,z_v)$

ただし，$|\boldsymbol{N}|=1, |\boldsymbol{U}|=1, |\boldsymbol{V}|=1$ とする．通常のテクスチャマッピングでは，①と②のデータさえあれば最低限のマッピングが可能であるが，高次元テクスチャマッピングには，それ以外に，③～⑤のデータも必要となる． U ベクトル，V ベクトルとは，3 次元 CG モデル上のある頂点における接線ベクトル（binormal vector）および従法線ベクトル（tangent vector）とよく似たベクトルであるが，若干異なる．これらは，ある頂点における接平面にテクスチャがどのような方向でマッピングされるかを示すベクトルであり，テクスチャ座標 $\boldsymbol{T}$ をもとに算出される．したがってテクスチャ座標の与え方によっては，U ベクトルと V ベクトルは直交しない場合もある．また，フレーム単位のデータとして，CG のシーン内に設置されているカメラ（視点）および光源の位置座標が必要となる．

- 視点の位置座標　$\boldsymbol{P_c}=(x_c,y_c,z_c)$
- 光源の位置座標　$\boldsymbol{P_l}=(x_l,y_l,z_l)$

これらのパラメータを用いて，図 2 に示すような座標系に基づく視点／光源条件 $\theta_c, \phi_c, \theta_l, \phi_l$ の算出をピクセル単位に行なう．まず以下のように視点ベクトル $\boldsymbol{C}$ および光源ベクトル $\boldsymbol{L}$ を計算する．

$$\boldsymbol{C}=\boldsymbol{P_c}-\boldsymbol{P}, \qquad \boldsymbol{L}=\boldsymbol{P_l}-\boldsymbol{P}.$$

次に，視点ベクトル，法線ベクトル，U ベクトルを用いて， θ_c, ϕ_c を算出する．以下のように正規化した視点ベクトル $\boldsymbol{C'}$ と法線ベクトルとの内積を計算することによって， θ_c を求めることができる．

$$\boldsymbol{C'}=\boldsymbol{C}/|\boldsymbol{C}|, \qquad s=\boldsymbol{C'}\cdot\boldsymbol{N}, \qquad \theta_c=\cos^{-1}s.$$

また，以下のように法線ベクトルと視点ベクトルとの外積ベクトル $\boldsymbol{R}$ を求め，U ベクトルとベクトル $\boldsymbol{R}$ の正規化ベクトル $\boldsymbol{R'}$ との内積を計算することによって， ϕ_c を求めることができる．

$$\boldsymbol{R}=\boldsymbol{N}\times\boldsymbol{C'}, \qquad \boldsymbol{R'}=\boldsymbol{R}/|\boldsymbol{R}|,$$
$$t=\boldsymbol{U}\cdot\boldsymbol{R'}, \qquad \phi_c=\cos^{-1}t-\pi/2.$$

ここで，・は内積，×は外積を表す．厳密には，V ベ

クトルを用いた ϕ_c の補正が必要であるが，本稿では説明を割愛する．また，θ_l, ϕ_l についても同様に算出することができる．

このようにして視点／光源条件を算出した後，インデックスデータを抽出することによって，ピクセルデータを抽出するために必要な 6 つのパラメータ $(u, v, \theta_c, \phi_c, \theta_l, \phi_l)$ を揃えることができる．

最後に，この 6 つのパラメータをもとにピクセルデータの抽出を行なう．各次元のサンプルは離散的に存在しているため，正確なピクセル値を求めるには 6 次元の線形補間 (hexa-linear 補間) を行なう必要がある．各次元において，近隣の 2 つのサンプルによる補間計算を行なうため，hexa-linear 補間では，64 (=2^6) 個のサンプルを用いた計算を行なう．

以上のようにして，高次元テクスチャを CG モデルにマッピングすることができる．

4. 高次元テクスチャの配置

前述した高次元テクスチャマッピングは計算コストが高く，CPU 上での実装では，1 フレームあたりのレンダリングに数秒を要してしまう．そこで，DirectX® 9.0c および上位レベルシェーダ言語 (HLSL) による実装を行ない，グラフィックスハードウェア (GPU) 上での高次元テクスチャマッピングを実現させた．その結果，リアルタイムでのレンダリングが可能となったが，コードブックデータをどのようにメモリ上に配置するかによって，コードブックデータへのアクセス方法が変化し，レンダリング性能が大きく変化することが分かった．

本節では，コードブックデータのメモリへの配置方法に伴うレンダリング性能の変化について評価する．複数のテクスチャデータをメモリに格納するには，以下の 3 つの方法が考えられる[6].

方法 1　別々のテクスチャとして格納する．
方法 2　タイル状に並べて格納する．
方法 3　レイヤー状に重ねて格納する．

方法 1 に関しては，1 枚 1 枚のテクスチャがそれぞれ別々に格納されるため，テクスチャ座標のアドレス指定が容易である．しかし，ピクセルシェーダにおいて格納できるテクスチャ数が制限されているため，全てのテクスチャサンプルを別々のテクスチャとして格納することはできない．また，条件に応じてテクスチャ識別子によるテクスチャの指定が必要であるが，現在のピクセルシェーダの仕様では，動的分岐処理内で汎用レジスタを用いたテクスチャ命令を実行することができない．そのため，無駄な

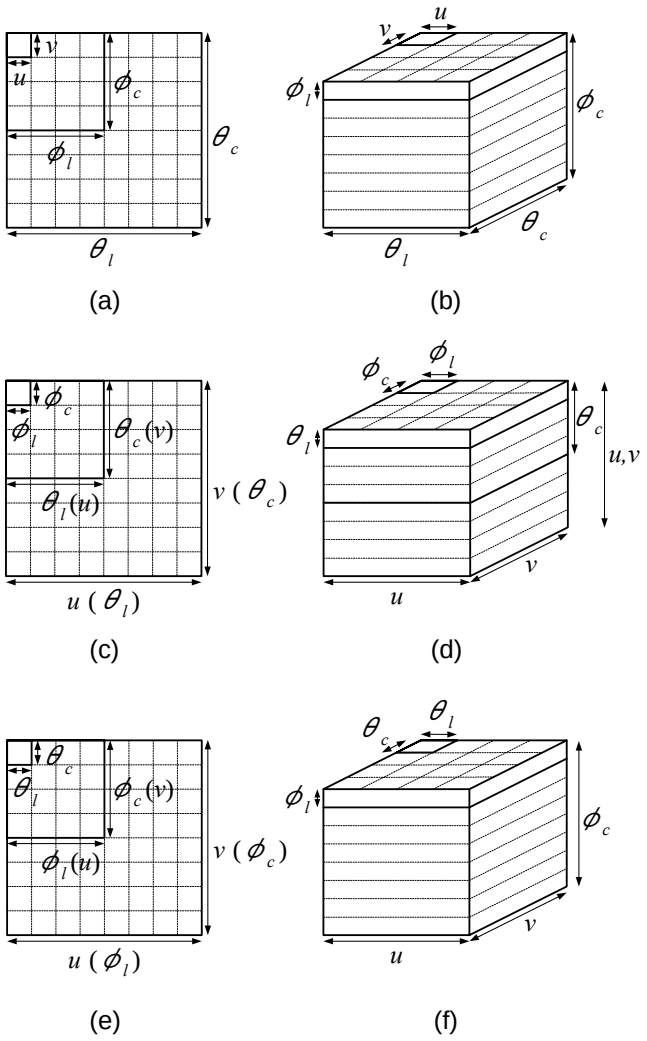

図 4　様々なテクスチャ配置方法

テクスチャロードが必要となり，レンダリング性能が著しく低下してしまうと考えられる．

方法2,3に関しては，一度に複数のテクスチャデータをまとめて格納できる．しかし，方法 2 では複数のテクスチャをタイル状に並べるために，2 次元テクスチャ座標のアドレス計算が必要であり，方法 3 では複数のテクスチャをレイヤー状に重ねるために，3 次元テクスチャ座標のアドレス計算が必要となる．

GPU ではテクスチャ命令に比べて算術命令の計算コストは小さいため，無駄なテクスチャロードを行なわない方法 2,3 を優先的に利用した方が好ましいと考えられる．2 節で説明した 64MB のコードブックデータは，4096×4096 ピクセルの 2 次元テクスチャ，もしくは，256×256×256 ピクセルの 3 次元テクスチャに格納できるサイズであるため，方法 2, 3 を併用することによって様々なパターンでタイル化・レイヤー化を行なうことができる．

そこで，様々なテクスチャ配置方法を試み，最適

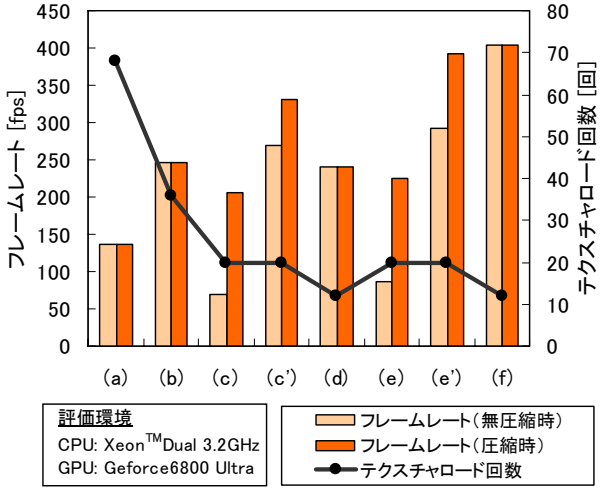

図 5　レンダリング性能

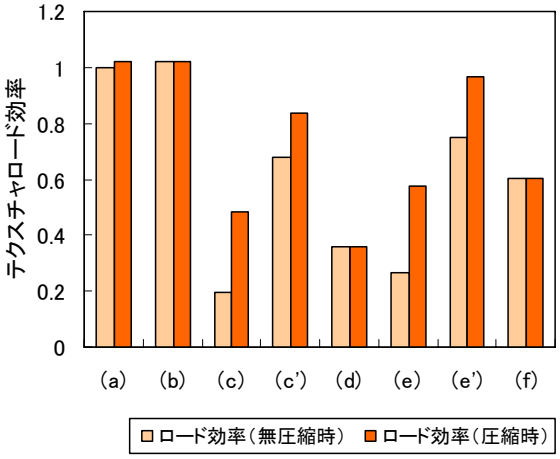

図 6　1 回あたりのテクスチャロード効率

な配置方法を検討する．図 4 に，考えられる配置方法を示す. (a), (c),(e) は 2 次元テクスチャへの配置であり，縦横方向にタイル状に並べている．(b), (d), (f) は 3 次元テクスチャへの配置であり，縦横方向にタイル状に並べ，さらに高さ方向にレイヤー状に重ねている．(a), (b) は u, v の変化を基準にしたテクスチャ（いわゆる一般のテクスチャ）をタイル化もしくはレイヤー化している．(c), (d) は ϕ_c, ϕ_l の変化を基準にしたテクスチャを, (e),(f) は θ_c, θ_l の変化を基準にしたテクスチャをタイル化もしくはレイヤー化している．また，(c), (e) において括弧内に示すような並べ方に変更したものを (c'), (e') とする．

まず，これらの配置方法で違いが分かるのは，テクスチャのロード回数である．前節で説明したとおり, 高次元テクスチャマッピングでは hexa-linear 補間を行なうために，コードブックデータを最大 64 回ロードし，シェーダ内で補間計算を行なう必要がある（この他に，インデックスデータ抽出のため 4 回のテクスチャロードが必要）．しかし, 配置方法によっては，ハードウェアの bi-linear 補間機能や tri-linear 補間機能を効果的に利用できるため，テクスチャのロード回数を減らすことができる．

それぞれの配置方法におけるテクスチャロード回数を図 5 に示す．高次元テクスチャマッピングはインデックスを参照して u, v を決定するため，u, v の変化に対してはハードウェアの補間機能を利用することができない．したがって, (a) ではコードブックデータを 64 回ロードしなければならない．(b) のようにレイヤー化することによって，高さ方向のみハードウェアの補間機能を利用できるため，コードブックデータは 32 回ロードすればよい. また (c), (e) では, ハードウェアの bi-linear 補間機能を利用できるため, コードブックを 16 回ロードすればよい. (d), (f) のようにレイヤー化することによって，ハードウェアの tri-linear 補間機能を利用できるため, コードブックデータは 8 回ロードすればよい．このようにして, テクスチャの配置方法によって, テクスチャのロード回数が大きく変化する．

それぞれの配置方法におけるレンダリング性能の評価結果を図 5 に示す．これは, Xeon™ Dual 3.2GHz (CPU), Geforce6800 Ultra (GPU) の PC にて実験し，640×480 ピクセルサイズのウィンドウに 256×256 ピクセルサイズの矩形を描画したものである．フレームレートを比較した結果，テクスチャのロード回数だけがレンダリング性能を左右するのではないことが分かった．

そこで，コードブックデータのロード回数を 8 回に統一してフレームレートを計測し, (a) の方法 (無圧縮時) を 1 としたときの値を比較した（図 6）．この値によって，各テクスチャ配置方法における 1 回 1 回のテクスチャロードの速度を比較できる．ここでは，この値をテクスチャロード効率と呼ぶ．レンダリング性能を左右するもう 1 つの原因は，このテクスチャロード効率であると考えられる．テクスチャロード効率は，テクスチャキャッシュのヒット率が影響すると考えられ，あるピクセルとその隣のピクセルを描画する際に，どれだけ近くのテクスチャ座標を参照するかがポイントとなる．図 6 での評価によって，u, v の変化に伴うピクセルデータが近くに存在するほど，ロード効率が向上することが分かった．また，θ_c, θ_l は ϕ_c, ϕ_l に比べてサンプル数が少なく変化も小さいため，θ_c, θ_l の変化でまとめた方が，ロード効率が高いことも分かった．

このようにして，1 回 1 回のテクスチャロード効

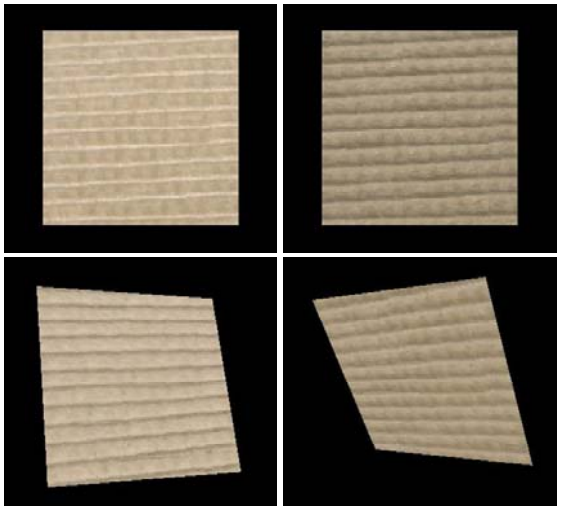

図 7　レンダリング結果

率と，テクスチャロード回数とを考慮した結果，(f) のようなテクスチャ配置を行なうことによって，レンダリング性能が格段に向上することが分かった．図 7 に，レンダリング結果の例を示す．

5. 高次元テクスチャの圧縮

本節では，高次元テクスチャのデータ量を削減するために，前節で述べた様々な実験条件に対してブロックベース・テクスチャ圧縮を適用し，画質やレンダリング効率への影響を評価する．ブロックベース・テクスチャ圧縮とは，テクスチャを一定サイズのブロックに分割し，各ブロックにおいてベクトル量子化を行なうものであり，ランダムアクセス性を失わず，テクスチャマッピングの性能を低下させない手法として知られている[7].

64MB のコードブックデータを，図 4 (a)～(f) のように配置し，DirectX® 規定のブロックベース・テクスチャ圧縮方式である DXT1 で圧縮することによって，データ量を 8MB（8 分の 1）にすることができた. 通常, テクスチャ圧縮は, (a),(b) のような u, v の変化でまとめられたいわゆる一般的なテクスチャに対して適用されるものであるが，本稿では，(c)～(f) のような視点／光源条件に依存する色変化に対してもブロックベース・テクスチャ圧縮を適用した. また DXT1 は, 4×4 ピクセルサイズのブロックごとに圧縮するものであるため，4 の倍数のテクスチャサイズでタイル化されていれば，タイル境界での圧縮の影響はない.

それぞれのテクスチャの圧縮性能（SN 比）を評価したところ，(a),(b) は 37.2dB，(c),(d) は 35.6dB，(e),(f) は 33.9dB といった結果となった. 視点／光源条件によってテクスチャ化した (c)～(f) は，若干圧縮性能が低下してしまうが，画質への影響は少ないことが分かった.

一方，図 5 において無圧縮時と圧縮時でのレンダリング性能を比較してみると，圧縮時のフレームレートは無圧縮時のフレームレートに比べて同等以上となることが分かった．図 6 において無圧縮時と圧縮時でのテクスチャロード効率を比較してみると，圧縮によるロード効率の向上がレンダリング性能の向上につながっていることが分かる．テクスチャを DXT1 によって圧縮した場合，ブロック単位でロードしてピクセルデータを抽出するため，テクスチャキャッシュへのヒット率が比較的向上するのが原因であると考えられる. 特に, (c),(e) の配置方法では，圧縮によって u, v の変化に伴うピクセルデータが接近するために, ロード効率が著しく向上している.

このように，高次元テクスチャを圧縮することによって，データ量の削減だけではなく，GPU の特性を利用したレンダリング性能の向上も図ることができた.

6. おわりに

本稿では，高品位な質感表現を行なうための高次元テクスチャマッピング手法について説明し，GPU を利用することによって高次元テクスチャマッピングがリアルタイムで実現できることを示した. また，テクスチャの配置方法を工夫することによってレンダリング性能が向上することを示した．さらに，高次元テクスチャの圧縮によって，データ量が削減でき，レンダリング性能も向上することを示した.

今後は，複雑なモデルによる性能評価を行なうとともに，視点／光源条件といった次元に限らず様々な次元への適用も進めていく.

参 考 文 献

[1] K. J. Dana, et al., “Reflectance and texture of real-world surfaces”, ACM Trans. on Graphics 18(1): pp.1-34, 1999.

[2] M. Levoy, P. Hanrahan, “Light Field Rendering”, ACM SIGGRAPH96, 1996.

[3] T. Malzbender, et al., “Polynomial texture maps”, ACM SIGGRAPH 2001, pp.519-528, 2001.

[4] Y. Yamauchi, M.Sekine, S. Yanagawa, “Bidirectional Texture Mapping for Realistic Cloth Rendering”, ACM SIGGRAPH 2003 sketch, 2003.

[5] 関根真弘, 山内康晋, “マクロ柄を考慮した双方向依存テクスチャの生成とマッピング”, 情報処理学会研究報告 2004-CG-116, pp.13-18, 2004.

[6] M. Pharr, “GPU Gems 2”, Addison Wesley, pp.521-545, 2005.

[7] A. Kugler, et al., “High Performance Texture Mapping Architectures”, Proc. of the 6th OMI Annual Conference on Embedded Microprocessor Systems, 1996.