VirtuaFighter5レンダリング技術事例

Imagire Day

最新技術に関わる技術トラック③
CEDEC2007

株式会社セガ　第二AM研究開発部　山之内 毅
株式会社CRI・ミドルウェア　研究開発部グラフィックスグループ　加束 崇

1. VirtuaFighter5の紹介

VirtuaFighter(以降VF)は、1993年に弊社(セガ)よりリリースされた格闘ゲーム。独特のメッシュキャラ表現と迫力のあるモーションで好評を博した。

翌年リリースのVF2は、テクスチャマッピングの導入と高フレームレート化(約30fps→約60fps)で、社会現象を起こすまでのヒットとなった。

以降シリーズや派生作品を重ね、VF5はシリーズ5作目として2006年に発売。2007年8月時点で、ビンゴフェスタなどを搭載したVer.Cがアーケードで好評稼働中。

VF5の開発に当たって、グラフィクスに課せられた要件は次の2点。

- **VF4と一目で差別化のできる絵**
- **60fps死守**

後者はロード中とシーケンス切替え中以外のすべてで処理落ちを許さない(デモシーンも不可)と言う、強力な制約である。

開発ターゲットのLINDBERGHのビデオカードは、GeForce6/7シリーズのハイエンドカテゴリの量産版。ビデオメモリのバス幅が256bitで帯域にいくらか余裕があるが、高解像度でFSAAかけて、凝ったマテリアル

表1. VirtuaFighterシリーズ年表

タイトル	発売年	基板	主なレンダリング技術
VF1	1993	MODEL1	フラットシェーディング, Zソート, 30fps
VF2	1994	MODEL2	4bitモノクロテクスチャ, ミップマップ, トライリニアフィルタ, 60fps
VF3	1996	MODEL3	スムーズシェーディング, Zバッファ, 16bitカラーテクスチャ, エッジアンチエイリアス
VF4	2001	NAOMI2	スペキュラ項, マルチライト, アルファブレンディング, シャドウボリューム
VF5	2006	LINDBERGH	ピクセル単位シェーディング, 法線マップ, HDRレンダリング, イメージベースドライティング, シャドウマップ

で、凝ったライティングで、何回ものマルチパスでエフェクトを…と言う欲張った実装は難しい。優先順位を付ける必要がある。

一目で判るものと言う要求に応えるため、次の手法を最優先した。

HDRレンダリング → 明るい反射とライトブルーム

画面内の高輝度部分はエミッションも少なくないが、やはり動きのあるスペキュラで魅せることが多い。法線マップとカラースペキュラマップで高輝度部分をキラキラと見せた。採用ディスプレイの高解像度化と相俟って、法線マップとカラースペキュラマップの組み合わせは有効に働いたと言って良いだろう。

図1. 製品版のスクリーンショット。ライトブルームとピクセル単位の明るい反射(スペキュラ)が、旧作VF4との大きな差別化を実現する。

図2. ライトブルームをカットした絵。

図3. スペキュラをカットした絵。

2. イメージベースドライティング

VF5のプレイフィールドは狭いリング上であり、ゲーム中にほとんど光源情報が変化しない。これを利用してイメージベースでライティングを行った。各ステージ毎にHDRIのキューブマップを用意し、用途別に事前フィルタリングを行う。

事前フィルタリングは素直に球面積分を行った。太陽のようなインパルスに近い光源があると低次の近似ではうまくいかない。

各キューブマップは整数エンコードして格納した。GeForce6/7で16ビット浮動小数点(GL_RGBA16F)のテクスチャをフィルタさせると重いため。シェーディング時にデコードして使う。

フィルタして作るキューブマップは次の6種類。

- **diffuse キーライトあり**
- **diffuse キーライトなし**
- **specular キーライトあり(高指数)**
- **specular キーライトなし(高指数)**
- **specular キーライトあり(低指数)**
- **specular キーライトなし(低指数)**

ライティングは以上のキューブマップをサンプリングして行う。一律底上げのアンビエントは存在しない。

スペキュラの指数は16と128。この間は線形補間で求める。正確な描画ではないが、法線マップやディフューズカラーマップを貼ってしまうと、アーティファクトは目立たなくなる。

セルフシャドウ用に、太陽の描かれている部分をキーライトエリアとして、その有無で2枚のマップを生成した。太陽の無いステージは、シーン内で一番強い光源をキーライトエリアとする。一般にキューブマップサンプルのライティングとセルフシャドウは相性が悪いが、この2枚を用意することで自然なシャドウを実現した。

ライトソースにステージをレンダリングしたイメージをそのまま使うと、キャラが完全に周りに馴染んでしまった。これは格闘ゲームでは視認性が悪くなり少々都合が悪い。そこで意図的にデザイナがイメージにライトを書き込んで、キャラを周囲から目立たせる工夫をした。

事前に応答を畳み込むため、マテリアル毎にBRDFを変えようとするとその数分のキューブマップが必要になる。VF5ではそこまで行わず、通常のディフューズとスペキュラのみを用意した。デザイナの用意するそれぞれのテクスチャが、プレイヤーの見る質感を決定する。

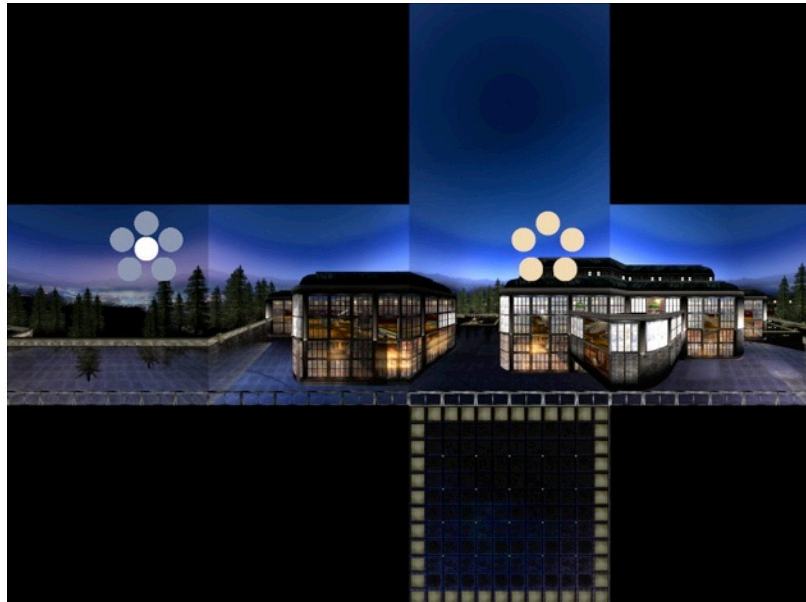


図4. イメージによるライトソース。デザイナがキーライトとカウンターライトを書き込んでいる(判りやすいようここだけトーンマップを変えてある)。左側真ん中の白丸がキーライト。

	キーライトあり	キーライトなし
diffuse		
specular (高指数)		
specular (低指数)		

図5.各フィルタ済みキューブマップ。これもキーライトの有無の差が判るよう、diffuseとspecularでトーンマップを変えてある(diffuseの方が明るめ)。

図6. 左がキーライトあり、右がキーライトなしのシェーディング。

図7. これを5章のスクリーンスペースソフトシャドウで合成する。

利点:

- 多数の光源による自然なシェーディング。
- 一律底上げのアンビエントライトの排除。
- 計算よりもテクスチャフェッチに負荷がかかる。GeForce6/7には得意分野。

欠点:

- HDRIの環境マップを用意するのが大変。XSIやPhotoshopなどのツールが対応しつつある。今後は楽になっていくだろう。
- 動的光源には対応できない。VF5のヒットエフェクトや美白ライトは別計算した。
- 光源が無限遠と仮定している。局所光源には一工夫必要。GPU Gems 1のChapter19(Kevin Bjorke) 参照。

3. HDRレンダリングとトーンマップ

3.1 HDRレンダリング

シェーディングは2章で用意したキューブマップ以降すべてHDRで行い、結果をGL_RGBA16Fのフレームバッファにストアした。浮動小数点バッファの採用で、HDRレンダリング特有の明るい反射やブルームの実装が簡単になった。アルファチャネルへの小細工や、フレームバッファの精度で悩む必要がない。レンダリングとポストプロセスの両方でコードが単純化される。

アルファブレンディングでも良い結果が得られる。低いアルファ値でも明るい反射が可能だ。

空いたアルファチャネルはオブジェクトのマスク切り出し用に使っている。直接ゲーム内で用いるわけではないが、キャラやアイテムのスクリーンショットを撮って行うデザイン作業の効率化に寄与している。

図8. HDRアルファブレンディングの例。眼鏡のレンズ部分の低いアルファ値でも、太陽の明るい反射が表現されている。

3.2 トーンマップ

開発初期はRGB毎に指数関数(1-exp(-x))を適用していた。これはコントラストが落ちる点と彩度が落ちる点が問題となった。デザイナの狙った色が出力されず、テクスチャ作成に支障が出た。

デザイナからはいっそ線形飽和でもよいのでは、と言う意見も出たが、これは高輝度部分で白飛びや色相変化が生じてしまう問題がある。そこで次の2点を改良した。

- **色空間を変換して輝度に対してトーンマップを行う。**
- **低輝度部分では線形に近い応答を返す。**

トーンマップ手順は次の通り。

1. **RGBからYCbCrへ変換**
2. **Yに指数関数とガンマを適用して0〜1の範囲へ収める**
3. **YへのスケールDE値でCbCrをスケーリング**
4. **RGBへ変換**

これにより低輝度部分はほぼ線形と同じ、高輝度部分では白飛びや色相変化を抑えたトーンマップとなった。

トーンマップは全てのレンダリングしたピクセルに対して行うので、シェーダのステップ数を極力抑えないとパフォーマンスが悪くなる。当初色空間変換を嫌ったのもパフォーマンスダウンを懸念してのことである。

RGB-YCbCr色空間変換は通常3ステップ(DP3が3命令)だが、CbCrを正規化しなくてよいなら2ステップ(DP3, SUB)で可能である。このトーンマップでもCb,Crに対してスケールをかけるだけなので正規化は不要だ。1と4 の変換を計4ステップで済ませることができた。

トーンマップカーブは 1-exp(-x) から、(1-exp(-x*2))^1.5 へ変更した。表2の通り、0.5以下の低輝度ではほぼ線形と同じ応答を返す。カーブはガンマ値などのパラメータが変わるたびにCPUで計算し、1次元テクスチャに格納している。

図9. トーンマップの比較。上からRGB毎の指数関数(彩度が落ちる)、線形飽和(高輝度部で白飛びと色相変化)、製品版のトーンマップ。

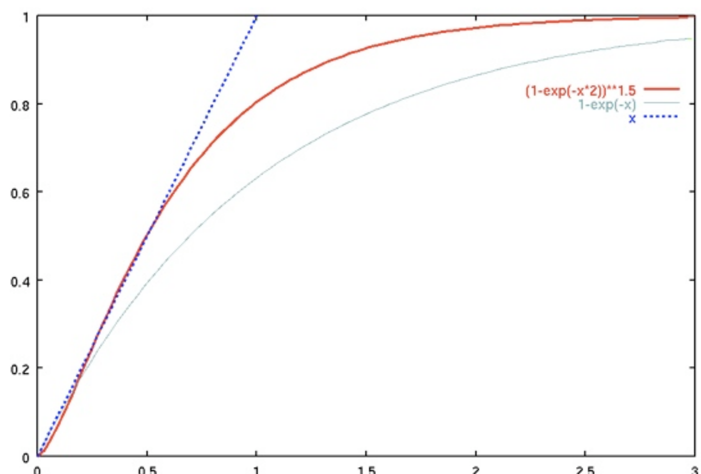


表2. 採用したトーンマップカーブ。当初の指数関数よりも線形に近い応答を返す。

実装したトーンマップには更に次の機能を備えさせた。

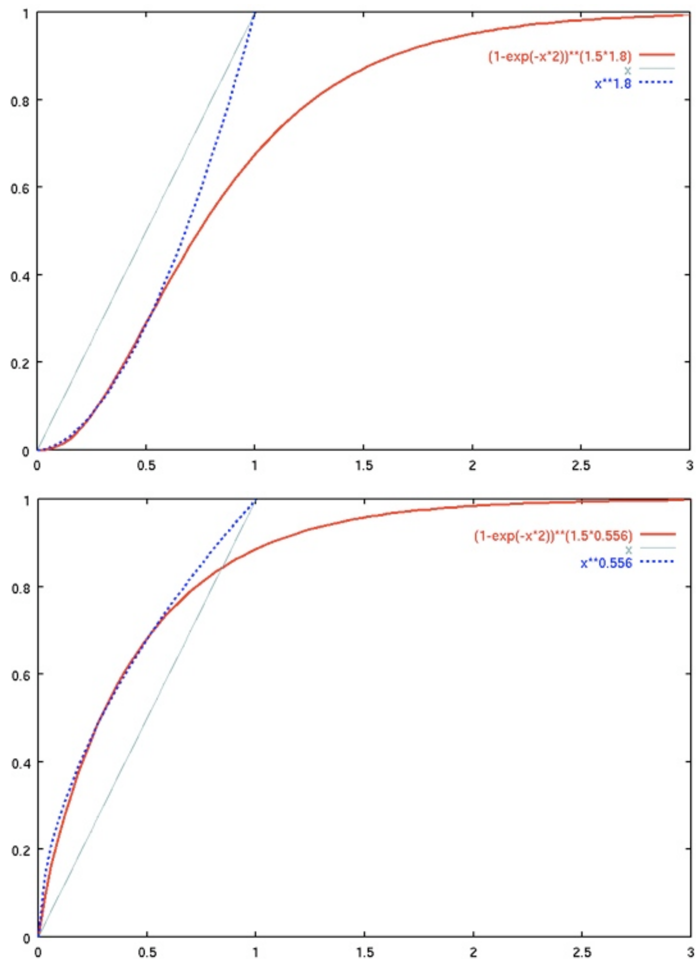
- **ガンマ調整機能**
- **彩度調整機能**
- **太陽フレア描画**

トーンマップの式を変更した時点で既にガンマを適用しているので、ガンマ調整機能をつけた。カーブはグラフの通り。低輝度部に関しては通常のガンマ処理とほぼ同じ応答を返す。

毎フレーム色空間を変更するついでに、画面全体の彩度を調整できるようにした。手順3のCbCrのスケーリングの際に調整用係数をかけている。これは廃墟ステージの落雷表現に使用した。落雷の瞬間に彩度を無くして明るさを強調している。

またトーンマップで全画面を走査するついでに、描画面積の大きい太陽のフレアテクスチャをタップしている。

表3. ガンマ適用時のトーンマップカーブ。上がガンマ1.8、下がガンマ1/1.8。



利点:

- **明るい反射。低反射率、低いアルファ値でも明るい絵が得られる。**
- **少ないパス数、簡単なブルーム実装。精度に悩まなくてよい。**
- **線形変換に近い、デザイナに違和感の少ないトーンマップ。**

欠点:

- **このトーンマップ手法に物理的根拠は無い。**
- **GeForce6/7ではポストプロセスで行うGL_RGBA16Fテクスチャのフィルタ処理が重い。**
- **GeForce6/7ではMSAAが使えない。**

  ➡後者二つはGeForce8シリーズで解消。GL_RGBA16Fがオーバースペックの場合には、32bit幅のGL_RGB9_E5_EXT、GL_R11F_G11F_B10F_EXTフォーマットが使える。

図10. 太陽フレア描画。太陽が視界に入った時はほぼ全画面にフレアのテクスチャを張るので、トーンマップ処理に組み込んでパフォーマンス改善を図った。

4. ピクセルバッファオブジェクト活用法

4.1 バッファオブジェクト解説

バッファオブジェクトとはOpenGLでグラフィックサブシステム(サーバ)側に用意されるメモリのこと。最初は頂点配列を格納するために開発された。それまでは描画の度にアプリ(クライアント)側にある頂点配列をサーバ側へ転送していた。バッファオブジェクトに書き込んで一度サーバ側に投げ込むことで、以降の転送を不要とした。頂点バッファオブジェクト(VBO)と呼ばれる。

OpenGLには昔からサーバ側へリソースを投げ込むディスプレイリスト(DL)が用意されているが、VBOももはや赤本で100ページ行かないうちに解説されるメジャーな機能となった。

DLとVBOとでパフォーマンスは一長一短。大雑把な傾向としては、描画の単位が大きい場合はDLが、小さい場合にはVBOが有利なようだ。弊社製品の例では、VF5がVBOを、AFTER BURNER ClimaxがDLを採用している。

VBOから始まったバッファオブジェクトは次第に増え、現在は表4に示す種類がある。よく話題にのぼるFBOはFramebuffer Objectであり、バッファオブジェクトの一種ではない。

バッファオブジェクトは種類によらず統一的な操作関数が用意されている。表5にその一覧を示す。ただしNV拡張のものに関しては現在のところ専用の関数を用いる。

本章ではピクセルバッファオブジェクト(PBO)のVF5での活用法を紹介する。PBOはテクスチャやフレームバッファの内容を、サーバ-クライアント間で効率的に読み書きするためのバッファである。PBOは転送方向によって次の2種類がある。

- GL_PIXEL_PACK_BUFFER
  - **サーバ→クライアントへの転送バッファ**
- GL_PIXEL_UNPACK_BUFFER
  - **クライアント→サーバへの転送バッファ**

表4. バッファオブジェクトの種類。indexはVBOに含まれる。

vertex	頂点を格納する(コア機能)
index	頂点インデックスを格納する(コア機能)
pixel	テクスチャやフレームバッファの読み書きに使う(コア機能)
texture	シェーダからテクスチャフェッチでアクセスできる(EXT拡張)
uniform	シェーダ(GLSL)からuniformでアクセスできる(EXT拡張)
parameter	シェーダ(アセンブラ)からparameterでアクセスできる(NV拡張)
transform feedback	頂点またはジオメトリシェーダで処理した結果を格納する(NV拡張)

4.2 ムービーの再生

VF5のアドバタイズはVF4までの慣例を破り、ムービー再生を採用した。リアルタイムデモは労力に見合わないこと、HD画質のムービーがリアルタイムデモに見劣りしないクォリティとなったことが、ムービー移行への大きな理由である。

アドバタイズムービーは1280x768で60fpsと言う大きな帯域と必要とする。CRI Sofdecでデコードしたイメージを、順次テクスチャメモリへストリーム転送するためにPBOを用いた。

転送方向はクライアント→サーバ、よって用いる PBO は GL_PIXEL_UNPACK_BUFFER である。PBOを二つ作り、ping/pongして同期を回避しつつストリーム転送を行った。疑似コードをリスト1に示す。

4.3 動画作成サービス

VF5ではプレイヤーのリプレイを動画で提供するサービスを行っている。

VF.NETを通じてキーリプレイを収集し、そのデータを元に動画センターでリプレイ動画を作成する。動画センターには15台のLINDBERGHを設置。それぞれがキーリプレイを元にゲームを再生、一試合分の全コマをチャプチャしてエンコーダへ投げている。

当初作成されていたキャプチャのコードの流れは次の通り。全てがシーケンシャルに実行され、10〜15fps程度のパフォーマンスしか出なかった。

- **当初の処理の流れ**
  - **ゲーム処理→フレームバッファ読み戻し→CPUでムービーサイズに縮小→ファイルに書きだし**

表5. バッファオブジェクトのOpenGL操作関数。

glGenBuffers	バッファIDの作成。
glBindBuffer	バッファのバインド。
glBufferData	バッファの初期化。
glMapBuffer	バッファのマップ。クライアント側からメモリ操作が可能になる。
glUnmapBuffer	操作済みバッファのアンマップ。
glBufferSubData	バッファの一部書き換え。glMapBufferよりも同期を軽減できる可能性がある。
glDeleteBuffers	バッファの削除。

動画センターの能力は、稼働台数と1本当たりの作成時間で決まる。実時間の4〜5倍は許される数字ではない。次の3点を改良した。

- **縮小処理をGPUへ。フィルタ付き縮小はGPUの十八番。**
- **PBOで非同期読み戻し。サーバ→クライアントなので、GL_PIXEL_PACK_BUFFER。**
- **別スレッドでファイル書きだし。I/O待ちの解消。**

改良版の処理の流れ

ゲーム処理→GPUでムービーサイズに縮小→PBOをキック→PBOで転送した前フレームのイメージを読み戻し→ファイル書き出しの別スレッドへ投げる

処理が並列化され、ほぼ60fps(ストレージで律速)で回るようになった。疑似コードをリスト2に示す。画面を等速でキャプチャできるようになり、多数のリクエストを捌くことができた。

リスト1. ムービー再生の疑似コード。PBOをバインドすることで glTexImage2D の最後の引数が、クライアント側メモリへのポインタから、サーバ側バッファオブジェクトのオフセットへと変わっている。

```
//-------- 今フレームでデコードしたイメージをPBOで転送開始 --------
// SofdecでデコードしたYCbCrのポインタとサイズを得る
MwsfdYccPlane     ycc;
int               npix_y, npix_cb, npix_cr;
mwPlyCalcYccPlane(frm.bufadr, width, height, &ycc);     // デコード情報取得
npix_y  = ycc.y_width  * height;        // Y のピクセル数
npix_cb = ycc.cb_width * height/2;      // Cbのピクセル数
npix_cr = ycc.cr_width * height/2;      // Crのピクセル数
// PBO へデコードイメージを書き込む
uint8_t     *img;
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbo[pbo_write_index]);
img = (uint8_t *)glMapBuffer(GL_PIXEL_UNPACK_BUFFER, GL_WRITE_ONLY); // 書き込み専用でマップ
if (img != NULL) {
    memcpy(img,                     ycc.y_ptr,  npix_y);      // Y のイメージを書き込む
    memcpy(img + npix_y,            ycc.cb_ptr, npix_cb);     // Cbのイメージを書き込む
    memcpy(img + npix_y + npix_cr, ycc.cr_ptr, npix_cr);     // Crのイメージを書き込む
}
glUnmapBuffer(GL_PIXEL_UNPACK_BUFFER);  // アンマップ。これでUNPACKのDMAをキック。
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);

//---- 前フレームで転送したイメージをテクスチャに定義する --------
// 転送済みPBOからテクスチャを定義
int          read_index = (pbo_write_index + 1) % num_pbo;
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbo[read_index]);
// Y テクスチャの定義
glBindTexture(GL_TEXTURE_RECTANGLE_ARB, tex_y);
glTexImage2D(GL_TEXTURE_RECTANGLE_ARB, 0, GL_LUMINANCE8, ycc.y_width, height, 0,
             GL_LUMINANCE, GL_UNSIGNED_BYTE, (char *)NULL + 0);
// Cbテクスチャの定義
glBindTexture(GL_TEXTURE_RECTANGLE_ARB, tex_cb);
glTexImage2D(GL_TEXTURE_RECTANGLE_ARB, 0, GL_LUMINANCE8, ycc.cb_width, height/2, 0,
             GL_LUMINANCE, GL_UNSIGNED_BYTE, (char *)NULL + npix_y);
// Crテクスチャの定義
glBindTexture(GL_TEXTURE_RECTANGLE_ARB, tex_cr);
glTexImage2D(GL_TEXTURE_RECTANGLE_ARB, 0, GL_LUMINANCE8, ycc.cr_width, height/2, 0,
             GL_LUMINANCE, GL_UNSIGNED_BYTE, (char *)NULL + npix_y + npix_cb);

glBindTexture(GL_TEXTURE_RECTANGLE_ARB, 0);
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);

//-------- バッファをフリップ --------
pbo_write_index = read_index;
```

リスト2. リプレイキャプチャの疑似コード。こちらもPBOのバインドで、glReadPixelsの最後の引数がPBO内オフセットへと変わっている。

```
//-------- 前フレームで転送したイメージを読み戻す --------
SS_ImageParam       *param;             // スクリーンショット構造体
param = &image_param[pbo_read_index];

// PBOから前フレームのイメージを読み戻す
glBindBuffer(GL_PIXEL_PACK_BUFFER, param->pbo);
src = (uint8_t *)glMapBuffer(GL_PIXEL_PACK_BUFFER, GL_READ_ONLY);  // 読み込み専用でマップ
if (src != NULL) {
    memcpy(param->image, src, image_size);        // ストアするイメージをPBOからコピー
}
glUnmapBuffer(GL_PIXEL_PACK_BUFFER);
glBindBuffer(GL_PIXEL_PACK_BUFFER, 0);

// ストアリクエストをワーカスレッド宛に積む
pthread_mutex_lock(&image_param_mutex);
while (image_param_queue.size() > 120) {     // 120枚・2秒分くらい溜まっている?
    pthread_mutex_unlock(&image_param_mutex);
    sleep(1);                                // ワーカの仕事が溜まり過ぎていたら待つ
    pthread_mutex_lock(&image_param_mutex);
}
image_param_queue.push_back(param);          // ストアリクエストを積む
pthread_cond_signal(&image_param_cond);      // ワーカスレッドへシグナルを投げる
pthread_mutex_unlock(&image_param_mutex);

//-------- 今フレームのレンダリングイメージをPBOで転送開始 --------
// 画像を縮小。GPUで行う。
resize();

// 読み戻しリクエスト。フレームバッファを読む。
int         write_index = (pbo_read_index + num_pbo - 1) % num_pbo;
glBindBuffer(GL_PIXEL_PACK_BUFFER, pbo[write_index]);
glReadPixels(0, 0, width, height, GL_BGRA, GL_UNSIGNED_BYTE, (char *)NULL+0); // PACKのDMA
をキック
glBindBuffer(GL_PIXEL_PACK_BUFFER, 0);

//-------- バッファをフリップ --------
pbo_read_index = (pbo_read_index + 1) % num_pbo;
```

利点:

- **CPUとGPUの非同期データ転送により、パフォーマンスアップが可能。**

欠点:

- **OpenGL実装や条件によっては同期が取られてしまうことも。ベンダの資料を良く読む必要がある。GeForce6/7であれば “Fast Texture Downloads and Readbacks using Pixel Buffer Objects in OpenGL” (http://developer.nvidia.com/object/fast_texture_transfers.html)が必読。**

5. シャドウ

シャドウ表現は、キャラクタのセルフシャドウ、キャラクタ間の相互シャドウ、建物からのシャドウ、距離によって半影度合いの変わるソフトシャドウを実現している。（図11）

図11. VF5のシャドウ表現

キャラクタのセルフシャドウは、シャドウマッピング後のイメージに対してフィルタを施し、エイリアシングの除去とソフトシャドウ化を行う手法を開発した。この手法を「Screen Space Filtering Shadow Mapping」と呼んでいる。

キャラクタがステージへ落とすシャドウは、投影マッピング時に距離に応じたミップマップレベルを選択し、シャドウの半影度合いを変化させる手法を開発した。この手法を「Blurred Mipmap Projection Shadow」と呼んでいる。

5.1 Screen Space Filtering Shadow Mapping

図12. セルフシャドウ

Screen Space Filtering Shadow Mappingは、スクリーンスペースでフィルタを適用し、シャドウマッピング特有のエイリアシングの除去とソフトシャドウ化を実現する手法である。（図12）

手順としては、通常のシャドウマッピングと同様に、ライト視点からシャドウ用オブジェクトをレンダリングし、深度マップを生成する。この深度マップを利用してシャドウマッピングのみのイメージをレンダリングし、「シャドウマスクテクスチャ」を生成する。

このシャドウマスクテクスチャに対して、エイリアシングの除去とソフトシャドウ化のフィルタを適用する。フィルタ処理は、黒部分を周囲に1pixelほど広げた後、ぼかしを加えている。

図13は、シャドウマスクテクスチャのフィルタ適用前と適用後の比較である。

図13. シャドウマスクテクスチャ。左がフィルタ適用前、右が適用後。

最後に、フィルタ適用後のシャドウマスクテクスチャを、キャラクタにスクリーンスペースで投影し、シャドウマスクテクスチャの黒部分を影、白部分を光の箇所としてライティングを行う。（図14）

VF5の場合は、シャドウマスク値によってキーライトの有無のイメージベースドライティング結果をブレンドしている。

図14. ライティングとシャドウマスクの合成

5.2 Blurred Mipmap Projection Shadow

Blurred Mipmap Projection Shadowは、投影マッピング時にミップマップを使用して距離に応じた半影度合いの変化を表現するための手法である。（図15）

ぼかし度合いの異なるシャドウのイメージを各ミップマップレベルに生成し、投影マッピング時に距離に応じてミップマップレベルを選択することで、近いほどくっ

きり、遠くほどぼやけたシャドウを表現することができる。ぼかし度合いの異なるイメージを複数段階（VF5では4段階）用意することで、ぼかしの切れ目が目立たないように滑らかにすることができる。

手順としては、まずシャドウを落とすオブジェクトをライト視点からテクスチャへレンダリングする。このテクスチャをシルエットマップと呼ぶ。このシルエットマップを元にぼかし度合いの異なる複数段階のミップマップイメージを生成していく。各ミップマップイメージは、上位レベルのミップマップイメージを元にぼかしフィルタを適用しながら下位レベルのミップマップイメージへ縮小コピーすることで生成する。最終的に最下位レベルのミップマップイメージには最もぼやけたシルエットが出来上がる。（図16）

図15. ステージへのシャドウ

図16. シルエットマップのミップマップ

次に、シャドウを受けるオブジェクト（ステージ等）に対して、シルエットマップをライト位置から投影マッピングする。投影マッピングの際、キャラクタの腰位置を基点として投影先の頂点までの距離を求め、その距離に応じてミップマップレベルを選択する。距離が遠いほどぼかしの強いミップマップレベルを選択し、距離が近いほどぼかしの弱いミップマップレベルを選択する。ミップマップレベルの選択にはシェーダのTXL命令を使用する。ミップマップレベル間の補間にはトライリニアフィルタを使用することで、ぼかしの継ぎ目を無くすことができる。

図17は、投影マッピング時に選択されるミップマップレベルを視覚化したものと、最終イメージである。

また、投影マッピング手法には、投影元から見て裏面になるポリゴンや、投影する方向とは逆側のポリゴンにも投影されてしまうという問題が存在する。これらの問題については、頂点シェーダでポリゴンの表裏や投影元との位置関係を判定することにより解決している。

図17. 距離に応じたミップマップの選択。上は選択されるミップマップを視覚化。下は最終イメージ。

6. 水面

水面では、キャラクタとのインタラクションのあるシミュレーションや水の濁り、背景の映り込みなどを実現している。（図18）

図18. 水面

波のシミュレーションはGPUを用いて高速に計算し、その結果をディスプレースメントマップとして水面のメッシュに適用している。ライティングは、ディスプレースメントマップから生成したノーマルマップを使用

し反射マッピングを行い、水の濁りはディスプレースメント度合いによって変化するようになっている。

波のシミュレーションは、256x256の浮動小数点テクスチャ(R32G32B32A32)をシミュレーション用に用意し、GPU上でセル・オートマトン方式による簡易的なシミュレーションを行っている。

キャラクタとのインタラクションについては、キャラクタの骨構造を近似した衝突検出用プリミティブを用いて、水面との衝突を検出し、モーションに沿う形で新たな波を発生させている。

生成する波の強さは、衝突箇所のモーションの速度に比例し、動きが激しいほど発生する波も大きくなり、動きが無ければほとんど波は発生しない。波の発生位置と強さをCPUで求めた後、シミュレーション用テクスチャに対して点ポリゴンをレンダリングすることで、新しい波が発生していく。

水面のレンダリング時は、まずシミュレーションの結果をディスプレースメントマップとして水面のメッシュに適用する。（図19）

水面の法線は、シミュレーションテクスチャをノーマルマップ化したものを使用する。（図20）

水面への映り込みは、あらかじめXZ平面で上下反転した簡易モデルをレンダリングして反射マップを生成し、ノーマルマップによって反射マップのUV値を摂動させ水面に投影する。

水の濁りは2枚のカラーテクスチャを使用する。（図21）水深の浅い部分と深い部分のカラーテクスチャを２枚用意し、それらを頂点のY座標によってブレンドすることで、水面が波立った際に2枚のテクスチャが混ざり合い、水が濁ったような表現となる。また、水面に落ちる静的な影は、あらかじめ浅い水深用のテクスチャに描き込んである。水面の色は主にマテリアルによって調整している。

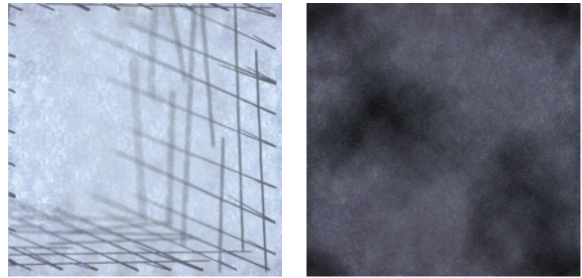


図21. 水面のカラーテクスチャ。左が浅い水深用で、右が深い水深用。

水面が半透明ポリゴンである場合、キャラクタの半透明ポリゴンとソートの相性が悪いという問題がある。そのため、水面は半透明ではなく不透明ポリゴンでレンダリングし、水面下部分を別のテクスチャとして合成することで、擬似的に透明感を出すようにしている。

水面下部分のテクスチャには、オブジェクトに対してハイトフォグを適用することで、水深の深い部分ほど濁って見えなくなり、よりリアルな濁りを表現できる。（図22）

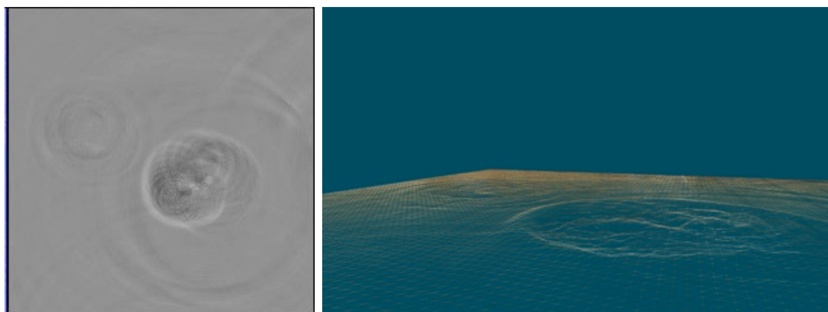


図19. ディスプレースメントマッピング。左がシミュレーションテクスチャ、右がそれによって変位されたメッシュ。

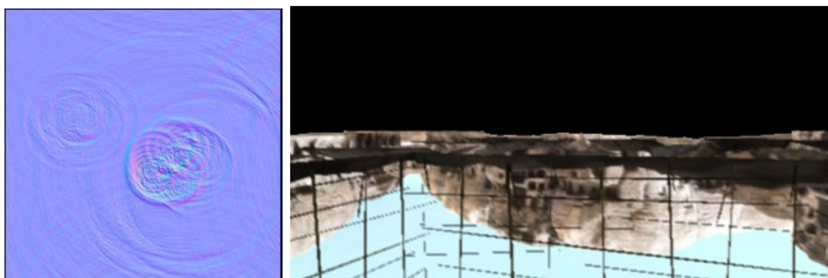


図20. 反射マッピング用のテクスチャ。左がノーマルマップ、右が背景の映り込みテクスチャ。

図22. 水面下のハイトフォグ。上が濁りを表現するハイトフォグ、下が水面との合成。

7. 雪面

雪面では、キャラクタが移動した場所の凹みをディスプレースメントマップによって表現し、質感には擬似的な表面下散乱を使用している。（図23）

図23. 雪面

雪面のディスプレースメントマップでは、凹凸の初期値と最小値をテクスチャとしてあらかじめ用意しておく。初期値テクスチャは雪面が踏み荒らされていない状態を表し、最小値テクスチャは踏み荒らされた状態を表す。（図24）

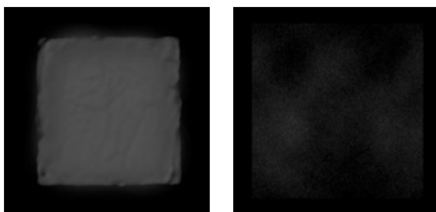


図24. あらかじめ用意するテクスチャ。左が初期値テクスチャ、右が最小値テクスチャ。

初期値テクスチャによって初期化されたディスプレースメントマップに対し、キャラクタとの接触箇所に足跡テクスチャを減算合成でレンダリングする。減算合成の際に、ディスプレースメントマップが平らにならないようにするため、最小値テクスチャと比較して最小値以下にならないようにクランプする。生成されたディスプレースメントマップを雪面のメッシュに適用する。（図25）

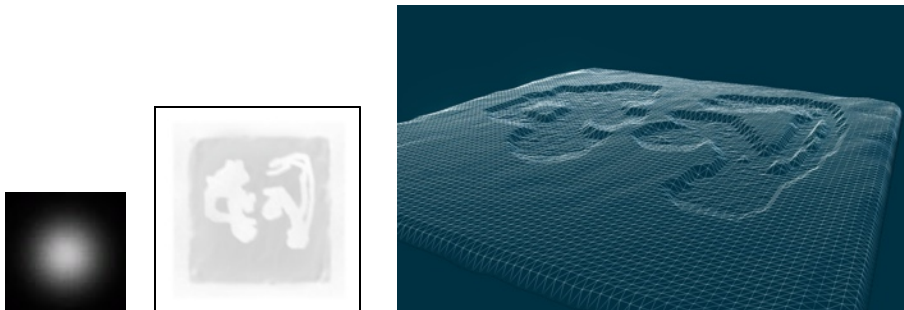


図25. 雪面のディスプレースメントマッピング。左から足跡テクスチャ、ディスプレースメントマップ、変位された雪原のメッシュ。

踏んでいない箇所と踏んだ箇所の雪面の質感の違いを表現するために２枚のテクスチャを使用する。（図26）これらのテクスチャを頂点のY座標を元にブレンドすることで、踏んでいない箇所と踏んだ箇所で質感の違いが出るようになる。また凹んだ箇所のスペキュラは弱めになるように調節している。

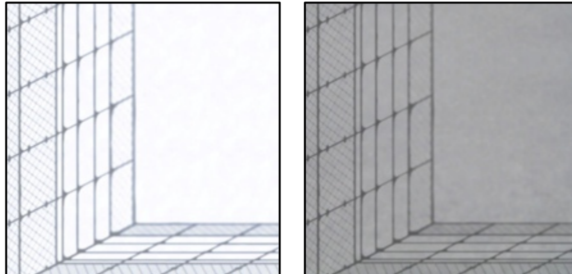


図26. 雪面のカラーテクスチャ。左が綺麗な雪面、右が踏んだ雪面。

雪面の擬似的な表面下散乱は、テクスチャスペースのぼかしによって表現している。まず、ライトベクトルや視線ベクトルをテクスチャスペースへ変換し、テクスチャスペースでdiffuseとspecularを計算する。その結果をテクスチャにレンダリングし、そこへぼかしフィルタを適用する。（図27）このテクスチャをカラーテクスチャと合成することで擬似的な表面下散乱となる。（図28）

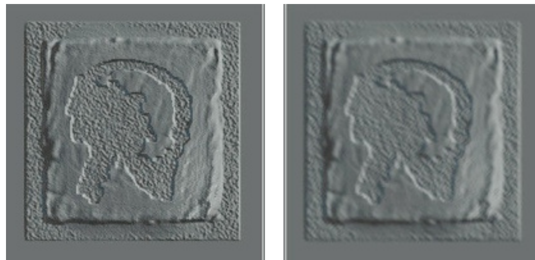


図27. 表面下散乱テクスチャ。左がフィルタ前、右がフィルタ後。

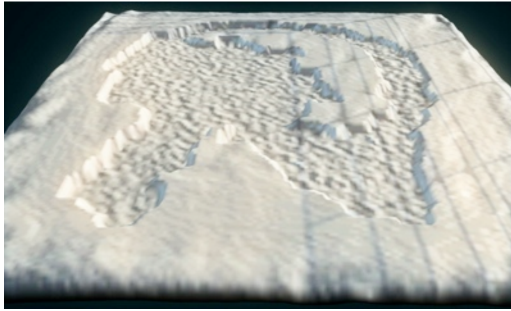
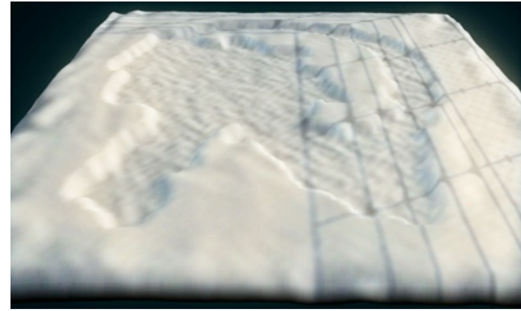



図28. 表面下散乱効果の比較。左が表面下散乱無効、右が有効。

8. フォグマップ

霧（もや）は、パーティクルベースによる簡易シミュレーションで動きを計算し、その結果をフォグマップとしてハイトフォグに使用することで表現している。（図29）

図29. フォグマップを使用したハイトフォグ。

パーティクルのシミュレーションは、ステージ上に敷き詰めたパーティクルを風によって移動させる簡易的なものである。風は、パーティクル全体を一定の方向へ流すグローバルな風と、キャラクタの動きによって発生するローカルな風を計算している。

ローカルな風は、キャラクタのモーションの速度が一定以上の場合に、モーションの軌道に沿って風のフィールドを発生させ、その周辺のパーティクルが影響を受けて移動する。

各パーティクルはポイントスプライトとしてフォグマップへレンダリングする。このフォグマップをキャラクタやステージのレンダリング時に頂点シェーダで参照し、ハイトフォグの濃度として使用する。（図30）通常のハイトフォグでは、高さが同一の頂点に対しては同じ濃度となるが、フォグマップを使用することで同一の高さの頂点でもフォグの濃度を変化させ、自然な霧を表現することができる。

また、半透明ポリゴンを何層も重ねて霧を表現する手法と比較すると、フォグマップ方式は、ポリゴンのめり込みや、別の半透明ポリゴンとのソートを考慮しなくて良いというメリットがある。

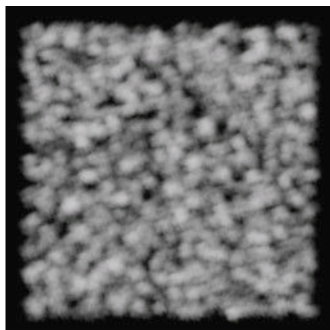


図30. 左がフォグマップ、右がそれをハイトフォグとして適用した絵。