## **Chapter 1　プログラマブルシェーダーとは**

## **1.1 固定機能**

シェーダーが生まれる前、DirectX9までは固定機能パイプラインというものが存在していました。この固定機能パイプラインはDirectX10で削除され、それ以降固定機能パイプラインは用意されなくなっています。これはOpenGL、OpenGLES、Sonyや任天堂などが提供する専用SDK(PS4、PS3、WiiUなどで使用できるDirectXのようなもの)でも同じで固定機能はグラフィックプログラミングの世界では過去のものとなっています。ではGPUで実行される処理を考えながら、固定機能とはなんなのかを考えていきましょう。

## **1.1.1グラフィックスパイプライン**

CPUからDraw命令送られてくると、下記のようにセットされた頂点バッファから頂点をフェッチ(取り出して)して、その頂点に陰影処理を行いスクリーン座標系に変換して対応するピクセルの色を決定します。

**メモリ**

**プロジェクション行列**

**ビュー行列**

**ワールド行列**

**GPU**

**頂点処理(スクリーン座標系に変換したり)**

**ピクセル処理(ライト情報などからピクセルの色を決める)**

**確定したピクセルカラーをフレームバッファに書き込む**

**頂点フェッチ(頂点取り出し)**

**頂点バッファ**

**バックバッファ(解像度が1280\*720なら、RGBA32bitの1280\*720の２次元配列)**

このように３次元データから２次元画像を作り出すまでの処理をグラフィックスパイプラインといいます。DirectX9まではポリゴンの頂点をスクリーン空間に変換する座標変換、座標変換後に、ポリゴンをスクリーン上で表示するために塗りつぶす必要のあるピクセルカラーの決定といった処理をGPUが用意してくれていました。これが固定パイプラインです。しかし、DirectX10以降では、この機能はGPU側で用意されなくなり、プログラマーがプログラミングする必要が出てきました。そして、そのプログラムがシェーダーと呼ばれるものです。

## **1.2 シェーダー**

シェーダーの導入で先ほど紹介したグラフィックパイプラインの頂点処理とピクセル処理を自分でプログラミングして、自由に頂点処理やピクセル処理を実装することができるようになりました。つまり、自分で頂点をスクリーン座標系へ変換したり、ピクセルカラーを決定するプログラムを書くことになります。つまり**DirectX10以降ではシェーダーを書かないと絵は表示できなくなりました！！！**

**メモリ**

**プロジェクション行列**

**ビュー行列**

**ワールド行列**

**GPU**

**頂点シェーダーをロードして実行する。**

**ピクセルシェーダーをロードして実行する。**

**確定したピクセルカラーをフレームバッファに書き込む**

**頂点フェッチ(頂点取り出し)**

**頂点バッファ**

**バックバッファ(解像度が1280\*720なら、RGBA32bitの1280\*720の２次元配列)**

この図のように、頂点処理とピクセル処理がシェーダーをロードして実行するという内容に変わっています。ではなぜ固定機能が削除されてシェーダーが登場したのでしょうか？せっかく用意されていたものがなくなって、同じものを作らないと絵を出せなくなったなんて面倒だと思いませんか？

## **1.3シェーダーが生まれた経緯**

固定機能しか存在していなかったDirectX7までは、マイクロソフトが用意したグラフィック表現しか行うことができませんでした。先ほどピクセルカラーを決める方法が固定されているといった話を思い出してみてください。DirectX9ではせいぜいディフューズライト、スポットライトライト、ポイントライト、アンビエントライトくらいでしょうか？ではこれらの機能を使って下記のようなアニメ調のグラフィック表現が実現できるでしょうか？



アニメ調のグラフィックを実現するためには、特殊なライティングアルゴリズムを実装する必要があります。しかし、シェーダーが生まれる前は新しいグラフィック表現を実現するためにはマイクロソフトがその処理を実装するまで待つ必要がありました。また、多数のゲーム開発者の要望に全て答えようとするとDirectXのAPIがどんどん膨らんで行くことにもなります。(ゲーム開発者というのは他とはことなるユニークな表現を行いたがるものなのです)その要望に答えるためにマイクロソフトは自分たちで処理を実装することを止めて、頂点処理、ピクセル処理を自由にプログラミングできるようにしました。これによりグラフィック表現の幅は大きく広がり、現在の高品質なフォトリアルな表現や、ナリティメットストームのようなノンフォトリアル表現まで多様な表現が実現できるようになったのです。

|  |
| --- |
| *実はこのアニメ調の表現はプログラマブルシェーダーを書かなくても実現できます。CPUで頂点をロックすれば頂点を自由に加工することができますよね？また、ピクセルカラーも単なる２次元配列に32bitのピクセルカラーを描き込んでいるだけなので、こちらもCPUでプログラミング可能です。このようにCPUでグラフィック処理を行うことをソフトウェアレンダリングといいます。ではなぜわざわざGPUでプログラミングをするのか？その答えは浮動小数点計算においてGPUはCPUに比べて圧倒的に高速に動作するからです。もし興味があれば、一度頂点のスクリーン座標変換をCPUとGPUの両方で実装してみて、速度を比較してみるといいでしょう。CPUの方は目も当てられないような動作速度になるはずです。* |

# **Chapter 2 シンプルなシェーダープログラム**

では、サンプルプログラムのChapter\_02を使用して、三角形を表示しているだけの簡単なプログラムを見ていきましょう。

## **2.1 頂点バッファを初期化する**

まずは、シェーダーを見ていく前に、三角形のポリゴンを表示するための、頂点バッファを作成しているコードを見ていきましょう。これはDirectXⅠの復習です(DirectXⅠ/教材.docのChapter3を参照)。main.cppにはTriangleDrawクラスが定義されています。TriagnelDrawクラスのコンストラクタから、InitVertexBuffer関数を呼び出して、頂点バッファを初期化しています。下記はInitVertexBuffer関数のコードです。

main.cpp(55行目)

|  |
| --- |
| /頂点バッファの初期化。  void InitVertexBuffer()  {  //頂点バッファのソースデータ。  SSimpleVertex vertices[] =  {  CVector4(0.0f, 0.5f, 0.5f, 1.0f),  CVector4(0.5f, -0.5f, 0.5f, 1.0f),  CVector4(-0.5f, -0.5f, 0.5f, 1.0f),  };  //頂点バッファの定義データを作成する。  D3D11\_BUFFER\_DESC desc = {0}; //0クリア。  desc.Usage = D3D11\_USAGE\_DEFAULT;  desc.ByteWidth = sizeof(vertices); 　//バッファの大きさ。  desc.BindFlags = D3D11\_BIND\_VERTEX\_BUFFER;//どのバッファにバインドするかを指定する。  desc.CPUAccessFlags = 0; //CPUからアクセスするか決める。0ならしない。  //頂点バッファのソースデータを指定する。  D3D11\_SUBRESOURCE\_DATA InitData = {0}; //0で初期化。  InitData.pSysMem = vertices;  //頂点バッファをVRAM上に作成する。  g\_graphicsEngine->GetD3DDevice()->CreateBuffer(&desc, &InitData, &m\_vertexBuffer);  } |

## **2.2 シェーダーをロードする**

　続いて、シェーダーをロードしているコードを見ていきましょう。TriangleDrawクラスのコンストラクタから、LoadShader関数が呼ばれています。LoadShader関数の中で頂点シェーダーとピクセルシェーダーがロードされています。

main.cpp(36行目)

|  |
| --- |
| //シェーダーをロードする。  void LoadShader()  {  //頂点シェーダーをロード。  //第一引数はロードするシェーダーソースファイルのファイルパス。  //第二引数はエントリーポイントとなる関数の名前。  //第三引数はロードしたシェーダーのタイプ。ここは頂点シェーダー。  m\_vsShader.Load(  "Assets/shader/Tutorial.fx",  "VS",  Shader::EnType::VS  );  //ピクセルシェーダーをロード。  m\_psShader.Load(  "Assets/shader/Tutorial.fx",  "PS",  Shader::EnType::PS  );  } |

## **2.3 Drawコール**

　では、続いてGPUへのDrawコールを行っている箇所を見ていきましょう。main.cppの126行目からのゲームループの中で、TriangleクラスのDraw関数が呼ばれています。

main.cpp(126行目)

|  |
| --- |
| //ゲームループ。  while (DispatchWindowMessage() == true)  {  //描画開始。  g\_graphicsEngine->BegineRender();  //三角形を描画。  triangleDraw.Draw();  //描画終了。  g\_graphicsEngine->EndRender();  } |

では、Tranagle::Draw関数を見てみましょう。

main.cpp(80行目)

|  |
| --- |
| //描画だってばよ！  void Draw()  {  //デバイスコンテキストを取得。  auto deviceContext = g\_graphicsEngine->GetD3DDeviceContext();  //デバイスコンテキストに描画コマンドを積む。  //頂点バッファを設定。  unsigned int stride = sizeof(SSimpleVertex); //頂点バッファのストライド  unsigned int offset = 0; //頂点バッファのオフセット。  deviceContext->IASetVertexBuffers(0, 1, &m\_vertexBuffer, &stride, &offset );  //プリミティブのトポロジーを設定。  deviceContext->IASetPrimitiveTopology(D3D11\_PRIMITIVE\_TOPOLOGY\_TRIANGLELIST);  //頂点シェーダーを設定。  deviceContext->VSSetShader(  (ID3D11VertexShader\*)m\_vsShader.GetBody(), //頂点シェーダー。  NULL, //NULLでいい。  0 //0でいい。  );  //ピクセルシェーダーを設定。  deviceContext->PSSetShader(  (ID3D11PixelShader\*)m\_psShader.GetBody(), //ピクセルシェーダー。  NULL, //NULLでいい。  0 //0でいい。  );  //頂点レイアウトを設定。  deviceContext->IASetInputLayout(m\_vsShader.GetInputLayout());  //ここまで設定した内容でドロー  deviceContext->Draw(  3, //頂点数。  0 //描画開始の頂点番号。大抵０。  );  } |

Triangle::Draw関数では、デバイスコンテキストに対して、頂点バッファ、プリミティブのトポロジー、頂点シェーダー、ピクセルシェーダー、頂点レイアウトを設定して、Drawコールを行っています。この中から、プリミティブのトポロジーについてだけ簡単に解説しておきます。

## **2.3.1プリミティブのトポロジー**

　設定されている頂点バッファがどのようなルールで三角形を構成しているかを指定するもの。代表的なものとして、トライアングルリストとトライアングルストリップを紹介します。

**・トライアングルリスト**

4

6

1

0

2

3

トライアングルリストは1つ1つの三角形が独立しています。

**・トライアングルストリップ**

3

1

2

0

トライアングルストリップは、三角形が独立しておらず、直前の三角形の二つの頂点を使用して、新しい三角形が作成されます。

　トライアングルリストは頂点の重複が発生して、トライアングルストリップは頂点の重複が発生しないので、頂点数を減らすことができると説明されていることが、たまにありますが、インデックスバッファを使用した場合は、トライアングルリストもトライアングルストリップでも頂点の重複が可能になるので、それは正しい説明ではありません。DirectXⅠ/教材.docの「3.3 インデックスバッファ」の項目を読んでみてください。そこで使われているサンプルでは、プリミティブのトポロジーはトライアングルリストになっていますが、頂点の重複を除去できています。

## **2.4 シェーダー**

　では、いよいよシェーダーについて見ていきましょう。Assets/shader/Tutorial.fxを開いてください。

下記のようなシンプルなコードが書かれていると思います。

|  |
| --- |
| //--------------------------------------------------------------------------------------  // 頂点シェーダーのエントリー関数。  //--------------------------------------------------------------------------------------  float4 VS( float4 Pos : POSITION ) : SV\_POSITION  {  float4 pos = Pos;  return pos;  }  //--------------------------------------------------------------------------------------  // ピクセルシェーダーのエントリ関数。  //--------------------------------------------------------------------------------------  float4 PS( float4 pos : SV\_POSITION ) : SV\_Target  {  return float4( 1.0f, 0.0f, 0.0f, 1.0f);  } |

VS関数は頂点シェーダーのエントリー関数で、描画する3Dオブジェクトの頂点の数分だけ呼ばれる関数です。引数には頂点のデータが渡されています。この関数は、座標変換後の頂点を返す必要があります。(座標変換は自由にプログラミングできるわけなので、変換しなくてもよいので、これは正確ではないが・・・)

　続いて、PS関数はピクセルシェーダーのエントリー関数で、ポリゴンに塗りつぶされるピクセルの数分だけ呼ばれる関数です。引数は頂点シェーダーから出力された頂点情報です。ピクセルシェーダーは塗りつぶすピクセルのカラーを返す必要があります。

## **2.5 実習課題**

　Chapter\_2を改造して、下記の二つの実習を行ってください。

**課題１.　頂点シェーダーを改造して、ポリゴンの大きさを倍にする。**

*ヒント　float4はhlsl言語で使用できる、4次元のベクトル型。*

**課題２. ピクセルシェーダーを改造して、ポリゴンの色を青にする。**

*ヒント　ピクセルシェーダーから返す値はrgbaの光の三原色+αを返している。*

*そして、0～255が0～1.0fにマッピングされている。*

**Chapter3**

**ShaderTutorial\_01(定数レジスタへの転送)**

Chapter2のサンプルではトランスフォーム済みの頂点(CPUでスクリーン座標系まで変換している頂点のこと)をGPUに送っているため、頂点シェーダーで頂点座標にワールド×ビュー×プロジェクション行列を乗算して、スクリーン座標系に変換するコードはありませんでした。しかし、PC、PS4、XBoxOneのような最新のゲームですと頂点数が10万を越えることはザラにあります。この頂点の座標変換をCPUで行うと、まともなパフォーマンスは出ません。そのため、ワールド、ビュー、プロジェクション行列などを転送して、GPUからアクセスできるようにする必要があります。下記のパスにデータの転送を行っているサンプルプログラムをアップしていますので、pullしてください。

<https://github.com/KawaharaKiyohara/DirectXLesson/ShaderTutorial_01>

今回のサンプルでは、頂点カラーに乗算するg\_colorをCPUから転送しています。(まだワールドビュープロジェクション行列の転送は行っていないため、トランスフォーム済みの頂点で描画を行っています。頂点シェーダーでの座標変換はChapter 4で行います)

では、まずCPUからデータを転送する方法について見ていきましょう。

main.cpp

|  |
| --- |
| VOID Render()  {  // Clear the backbuffer to a blue color  g\_pd3dDevice->Clear(0, NULL, D3DCLEAR\_TARGET, D3DCOLOR\_XRGB(0, 0, 255), 1.0f, 0);  if (SUCCEEDED(g\_pd3dDevice->BeginScene()))  {  //シェーダー適用開始。  g\_pEffect->SetTechnique("ColorPrim");  g\_pEffect->Begin(NULL, D3DXFX\_DONOTSAVESHADERSTATE);  g\_pEffect->BeginPass(0);  **//定数レジスタに設定するカラー。**  **D3DXVECTOR4 color( 1.0f, 0.0f, 0.0f, 1.0f);**  **//シェーダー側のシェーダー定数の名前で、データの転送先を指定する。**  **g\_pEffect->SetVector("g\_color", &color);**  //この関数を呼び出すことで、データの転送が確定する。描画を行う前に一回だけ呼び出す。  g\_pEffect->CommitChanges();  g\_pd3dDevice->SetStreamSource(0, g\_pVB, 0, sizeof(SVertex));  g\_pd3dDevice->SetFVF(D3DFVF\_CUSTOMVERTEX);  g\_pd3dDevice->DrawPrimitive(D3DPT\_TRIANGLELIST, 0, 1);  g\_pEffect->EndPass();  g\_pEffect->End();  // End the scene  g\_pd3dDevice->EndScene();  }  // Present the backbuffer contents to the display  g\_pd3dDevice->Present(NULL, NULL, NULL, NULL);  } |

太字で書いている箇所がGPUへのデータの転送命令を行っている箇所になります。今回はシェーダー側に記述されているシェーダー定数名を指定してデータを転送する方法を採用しています。文字列で転送先の検索が行われるため重いのですが、今回は分かりやすさを重視しています。試しにローカル変数のcolorの値を変更してみてください。ポリゴンの色が変わるはずです。では続いてシェーダー側のソースを見てみましょう。

basic.fx

|  |
| --- |
| **float4 g\_color; //カラー。これがシェーダー定数。CPUから値が転送されてくる。**  struct VS\_INPUT{  float4 pos : POSITION;  float4 color : COLOR0;  };  struct VS\_OUTPUT{  float4 pos : POSITION;  float4 color : COLOR0;  };  /\*!  \*@brief 頂点シェーダー。  \*/  VS\_OUTPUT VSMain( VS\_INPUT In )  {  VS\_OUTPUT Out;  Out.pos = In.pos;  **Out.color = In.color \* g\_color;　//頂点カラーに定数レジスタに設定されたカラーを乗算してみる。**  return Out;  } |

太字になっている箇所がCPUから送られたデータに関連する箇所になります。GPUには定数レジスタという高速にアクセスできるメモリがあり、CPUから送られたデータはこの定数レジスタにバインドされます。C++などのグローバル変数のような定義のされ方がしている変数が定数レジスタにバインドされる変数になります。定数レジスタには上限があり、ライトなども定数レジスタに送るためDirectX9世代ではライトの本数などに上限が設けられていました。DirectX11からは(10からそうだったのかも？)ストラクチャバッファなどの機能が増え、事実上この上限はなくなっています。

では、下記の二つの実習を行ってみてください。

・カラーの定数g\_addColorを追加して、頂点シェーダーで加算合成を行う。

・カラーの定数g\_mulColorを追加して、頂点シェーダーで乗算合成を行う。

**Chapter 4**

**ShaderTutorial02(ワールドビュープロジェクション行列による頂点変換)**

Chapter3でも予告していましたが、今度はワールドビュープロジェクション行列(以下WVP行列)を使用して、頂点シェーダーで頂点変換を行っていきます。サンプルプログラムを下記のパスにアップしていますので、pullを行ってください。

<https://github.com/KawaharaKiyohara/DirectXLesson/ShaderTutorial_02>

Chapter3で学んだように、WVP行列はCPUからGPUへ転送を行って、シェーダーで使用することになります。ではCPU側の転送命令を記述しているコードを見てみましょう。

main.cpp

|  |
| --- |
| VOID Render()  {  // Clear the backbuffer to a blue color  g\_pd3dDevice->Clear(0, NULL, D3DCLEAR\_TARGET, D3DCOLOR\_XRGB(0, 0, 255), 1.0f, 0);  if (SUCCEEDED(g\_pd3dDevice->BeginScene()))  {  //シェーダー適用開始。  g\_pEffect->SetTechnique("ColorPrim");  g\_pEffect->Begin(NULL, D3DXFX\_DONOTSAVESHADERSTATE);  g\_pEffect->BeginPass(0);  **//ワールド行列の転送。**  **g\_pEffect->SetMatrix("g\_worldMatrix", &g\_worldMatrix);**  **//ビュー行列の転送。**  **g\_pEffect->SetMatrix("g\_viewMatrix", &g\_viewMatrix);**  **//プロジェクション行列の転送。**  **g\_pEffect->SetMatrix("g\_projectionMatrix", &g\_projectionMatrix);**  //この関数を呼び出すことで、データの転送が確定する。描画を行う前に一回だけ呼び出す。  g\_pEffect->CommitChanges();  g\_pd3dDevice->SetStreamSource(0, g\_pVB, 0, sizeof(SVertex));  g\_pd3dDevice->SetFVF(D3DFVF\_CUSTOMVERTEX);  g\_pd3dDevice->DrawPrimitive(D3DPT\_TRIANGLELIST, 0, 1);  g\_pEffect->EndPass();  g\_pEffect->End();  // End the scene  g\_pd3dDevice->EndScene();  }  // Present the backbuffer contents to the display  g\_pd3dDevice->Present(NULL, NULL, NULL, NULL);  } |

太字になっている箇所がGPUへの転送命令を記述している箇所です。先ほどと大きな違いはないかと思います。転送するのが行列なので、関数名がSetMatrixになっている点が違うくらいでしょうか。ではシェーダー側のソースを見てみましょう。

basic.fx

|  |
| --- |
| **float4x4 g\_worldMatrix; //ワールド行列。**  **float4x4 g\_viewMatrix; //ビュー行列。**  **float4x4 g\_projectionMatrix; //プロジェクション行列。**  struct VS\_INPUT{  float4 pos : POSITION;  float4 color : COLOR0;  };  struct VS\_OUTPUT{  float4 pos : POSITION;  float4 color : COLOR0;  };  /\*!  \*@brief 頂点シェーダー。  \*/  VS\_OUTPUT VSMain( VS\_INPUT In )  {  VS\_OUTPUT Out;  float4 pos;  **pos = mul( In.pos, g\_worldMatrix ); //モデルのローカル空間からワールド空間に変換。**  **pos = mul( pos, g\_viewMatrix ); //ワールド空間からビュー空間に変換。**  **pos = mul( pos, g\_projectionMatrix ); //ビュー空間から射影空間に変換。**  Out.pos = pos;  Out.color = In.color;  return Out;  } |

黒字になっている箇所がWVP行列を使用したコードになります。Hlslではfloat4x4が行列の変数です。頂点シェーダーでmul命令を使用して、頂点の座標変換を行っています。注意点としては実はmul関数は下記のように記述することもできます。

***pos = mul( g\_worldMatrix, In.pos);***

サンプルと比較して行列とベクトルの順番が変わっているのがわかりますでしょうか。この順番を入れ替えると異なる結果になるので注意が必要です。

第一引数にベクトルがある場合は行ベクトルとして計算されます。第二引数にベクトルがある場合は列ベクトルとして計算されます。今はとりあえず、結果が変わるということだけ覚えておいてください。今後シェーダーを記述していくことがあるかと思いますが、意図しない表示なっている場合などは、乗算する順番がおかしくなっていないか確認してみてください。

では下記の実習を行ってみてください。

・WVP行列の計算をCPUで行って、GPUでの計算コストを減らしてください。

**Chapter 5**

**ShaderTutorial\_03(シェーダーを使用してXファイルを表示)**

シェーダーを使用してモデルを表示するサンプルを下記のパスにアップしていますのでpullを行ってください。

<https://github.com/KawaharaKiyohara/DirectXLesson/ShaderTutorial_03>

シェーダーを使用したモデル表示も今までの方法と大きな違いはないため、ここはサンプルの紹介にとどめておきます。

**Chapter 6**

**ShaderTutorial\_04(簡単なテクスチャ貼り付け)**

Chapter5のサンプルでは虎にテクスチャが貼られていませんでした。今回は虎にテクスチャを貼り付ける処理を説明します。今まではIDirect3DDevice9::SetTextureを実行すれば勝手にテクスチャが貼られていたと思いますが、シェーダーを使用する場合はテクスチャの貼り方までプログラミングする必要があります。ではサンプルを見ていきましょう。

まず、GPUにテクスチャのアドレスを転送しているコードです。

main.cpp

|  |
| --- |
| VOID Render()  {  // Clear the backbuffer to a blue color  g\_pd3dDevice->Clear(0, NULL, D3DCLEAR\_TARGET | D3DCLEAR\_ZBUFFER, D3DCOLOR\_XRGB(0, 0, 255), 1.0f, 0);  static int renderCount = 0;  if (SUCCEEDED(g\_pd3dDevice->BeginScene()))  {  // Turn on the zbuffer  g\_pd3dDevice->SetRenderState(D3DRS\_ZENABLE, TRUE);    renderCount++;  D3DXMATRIXA16 matWorld;  D3DXMatrixRotationY( &g\_worldMatrix, renderCount / 500.0f );  //シェーダー適用開始。  g\_pEffect->SetTechnique("SkinModel");  g\_pEffect->Begin(NULL, D3DXFX\_DONOTSAVESHADERSTATE);  g\_pEffect->BeginPass(0);  //定数レジスタに設定するカラー。  D3DXVECTOR4 color( 1.0f, 0.0f, 0.0f, 1.0f);  //ワールド行列の転送。  g\_pEffect->SetMatrix("g\_worldMatrix", &g\_worldMatrix);  //ビュー行列の転送。  g\_pEffect->SetMatrix("g\_viewMatrix", &g\_viewMatrix);  //プロジェクション行列の転送。  g\_pEffect->SetMatrix("g\_projectionMatrix", &g\_projectionMatrix);  g\_pEffect->CommitChanges();  // Meshes are divided into subsets, one for each material. Render them in  // a loop  for( DWORD i = 0; i < g\_dwNumMaterials; i++ )  {  **g\_pEffect->SetTexture("g\_diffuseTexture", g\_pMeshTextures[i]);**  // Draw the mesh subset  g\_pMesh->DrawSubset( i );  }    g\_pEffect->EndPass();  g\_pEffect->End();  // End the scene  g\_pd3dDevice->EndScene();  }  // Present the backbuffer contents to the display  g\_pd3dDevice->Present(NULL, NULL, NULL, NULL);  } |

黒字の部分がGPUへテクスチャアドレスの転送を行っているコードです。これは固定機能を使っている時と大差ないのではないでしょうか。ではシェーダーを見ていきます。

basic.fx

|  |
| --- |
| /\*!  \*@brief 簡単なテクスチャ貼り付けシェーダー。  \*/  float4x4 g\_worldMatrix; //ワールド行列。  float4x4 g\_viewMatrix; //ビュー行列。  float4x4 g\_projectionMatrix; //プロジェクション行列。  **texture g\_diffuseTexture; //ディフューズテクスチャ。 ①**  **sampler g\_diffuseTextureSampler = //テクスチャサンプラ ②**  **sampler\_state**  **{**  **Texture = <g\_diffuseTexture>;**  **MipFilter = NONE;**  **MinFilter = NONE;**  **MagFilter = NONE;**  **AddressU = Wrap;**  **AddressV = Wrap;**  **};**  struct VS\_INPUT{  float4 pos : POSITION;  float4 color : COLOR0;  float2 uv : TEXCOORD0;  };  struct VS\_OUTPUT{  float4 pos : POSITION;  float4 color : COLOR0;  float2 uv : TEXCOORD0;  };  /\*!  \*@brief 頂点シェーダー。  \*/  VS\_OUTPUT VSMain( VS\_INPUT In )  {  VS\_OUTPUT Out;  float4 pos;  pos = mul( In.pos, g\_worldMatrix ); //モデルのローカル空間からワールド空間に変換。  pos = mul( pos, g\_viewMatrix ); //ワールド空間からビュー空間に変換。  pos = mul( pos, g\_projectionMatrix ); //ビュー空間から射影空間に変換。  Out.pos = pos;  Out.color = In.color;  Out.uv = In.uv;  return Out;  }  /\*!  \*@brief 頂点シェーダー。  \*/  float4 PSMain( VS\_OUTPUT In ) : COLOR  {  **return tex2D( g\_diffuseTextureSampler, In.uv ); //③**  }  technique SkinModel  {  pass p0  {  VertexShader = compile vs\_2\_0 VSMain();  PixelShader = compile ps\_2\_0 PSMain();  }  } |

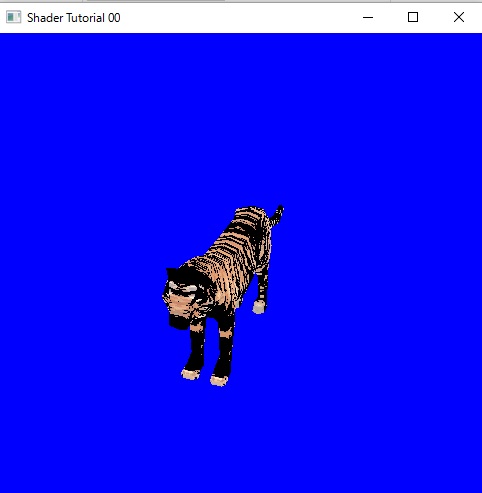
まず①の箇所がCPUから転送されたテクスチャのアドレスが格納された変数になります。そして②の箇所がテクスチャのサンプリング方法を記述したテクスチャサンプラと呼ばれるものです。テクスチャサンプラについてはここでは説明しません。今はこのように記述するものだと思っていてください。続いて③の箇所ですが、こちらがuv座標を使用してテクスチャをサンプリングしているコードになります。

実習

① サンプリングしたテクスチャカラーの明るさが1.0以上であればテクスチャカラーを出力して、1.0以下であれば黒を出力するようにピクセルシェーダーを改造してみてください。テクスチャカラーの明るさは下記のコードを使用して求めてください。

　length(float3 color)

下記のような絵になります。



② トラのテクスチャがUV座標のU方向にスクロールするプログラムを実装してください。

　実装した結果の実行ファイルを下記のパスにアップしていますので、こちらを参考に実装してみてください。

DirectXLesson\ShaderTutorial\_04\UVスクロール

③ 虎を半透明で表示してみてください。

④ 虎を点滅させてください。

**Chapter 7**

**深度テスト(Zテスト)**

このチャプターではシェーダーではなく、3Dグラフィックスのプログラミングでは欠かすことが出来ない、深度テスト(Zテスト)について解説をします。

**7.1 深度バッファ(Zバッファ)**

　まず、深度テストについて説明をする前に深度バッファについて説明をします。みなさんDirectX9で3Dモデルを表示する際に、デバイスの初期化で下記のような処理を記述していたのではないかと思います。

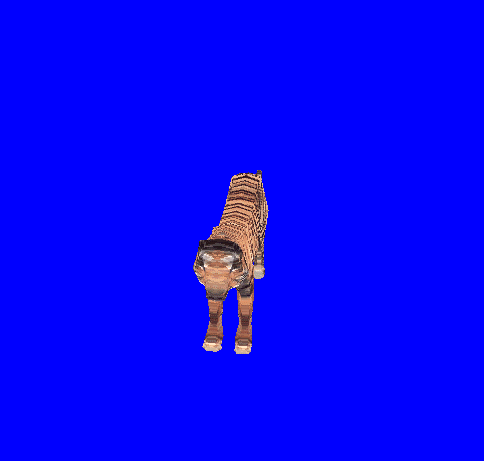
|  |
| --- |
| D3DPRESENT\_PARAMETERS d3dpp;  ZeroMemory(&d3dpp, sizeof(d3dpp));  d3dpp.Windowed = TRUE;  d3dpp.SwapEffect = D3DSWAPEFFECT\_DISCARD;  d3dpp.BackBufferFormat = D3DFMT\_UNKNOWN;  **d3dpp.EnableAutoDepthStencil = TRUE;**  **d3dpp.AutoDepthStencilFormat = D3DFMT\_D16;**  // Create the D3DDevice  if (FAILED(g\_pD3D->CreateDevice(  D3DADAPTER\_DEFAULT,  D3DDEVTYPE\_HAL,  hWnd,  D3DCREATE\_SOFTWARE\_VERTEXPROCESSING,  &d3dpp,  &g\_pd3dDevice  )))  {  return E\_FAIL;  } |

d3dppはCreateDeviceに渡す引数なのですが、この赤字になっている箇所が深度バッファに関する設定になっています。この設定を渡すと16ビットの深度バッファがフレームバッファと同じ幅と高さで作成されます。例えば1280×720のフレームバッファを作成した場合は、下記のような深度バッファが作成されます。

short depthBuffer[720][1280]; //深度バッファ

深度バッファとは、フレームバッファに絵を書き込んだ際に、その絵のZの座標を記録しておくためのバッファです。例えば図Aのような絵をフレームバッファに書き込んだ場合図Bのような深度バッファが作成されます。

図A フレームバッファ



図B 深度バッファ



**7.2 深度テスト**

深度テストとは頂点シェーダーとピクセルシェーダーの間で行われる処理で、深度バッファを参照して、新しく書き込もうとするピクセルが既に書き込まれているピクセルより手前にあるか、奥にあるかを判定するテストになります。これから書き込もうとしているピクセルが既に書き込まれているピクセルより奥にある場合、描きこむ必要がないため処理が破棄されます。この深度テストのおかげで3Dオブジェクトは正しい前後関係で描画することができるようになっています。

**7.3 深度テストを有効にする方法**

ではDirectX9で深度テストを有効にする方法を見ていきましょう。DirectX9では各種レンダリングステートの設定を行う、IDirect3DDevice9::SetRenderStateを使用することで深度テストを有効にすることができます。下記のようなコードで深度テストを有効にできます。

g\_pd3dDevice->SetRenderState(D3DRS\_ZENABLE, TRUE);

**7.4 Zファイティング**

深度バッファに保存できるZ値の精度には限界があります。そのため非常にZの値が近いポリゴンを重ねて描画すると、ポリゴンがチラ付く現象が発生します。これをZファイティングといいます。

PSPは16bitの深度バッファしか作ることができずに、深度バッファの精度が低かったため、PSPのゲームではよくこの現象が起きていました。

下記のパスにZファイティングのサンプルプログラムをアップしています。

https://github.com/KawaharaKiyohara/DirectXLesson/Zファイティングサンプル

**7.5 ZFunc**

実はZテストは必ずしも手前にあるものだけを描画するわけではありません。実はZFuncというレンダリングステートを変更すると奥にあるものを描画するということもできます。

このZFuncというのはZテストの方法を指定するレンダリングステートで、「Z値が大きいものが合格」や、「Z値が小さければ合格」というふうにZテストの方法を変更することができます。

下記にZFuncに指定できる値を列挙します。

|  |
| --- |
| D3DCMP\_NEVER  テストは常に失敗する。  D3DCMP\_LESS  新しいピクセル値が、現在のピクセル値より小さいときに応じる。  D3DCMP\_EQUAL  新しいピクセル値が、現在のピクセル値と等しいときに応じる。  D3DCMP\_LESSEQUAL  新しいピクセル値が、現在のピクセル値以下のときに応じる。  D3DCMP\_GREATER  新しいピクセル値が、現在のピクセル値より大きいときに応じる。  D3DCMP\_NOTEQUAL  新しいピクセル値が、現在のピクセル値と等しくないときに応じる。  D3DCMP\_GREATEREQUAL  新しいピクセル値が、現在のピクセル値以上のときに応じる。  D3DCMP\_ALWAYS  テストは常にパスする。 |

**Chapter 8**

**アルファブレンディング**

このチャプターでは半透明合成や、加算合成を行うために必要なアルファブレンディングについて説明します。

**8.1 アルファブレンディングとは**

　アルファブレンディングとは、ピクセルシェーダーで計算されたカラー(RGBA)をフレームバッファにどのように描き込むのかを指定するものとなります。その描き込みの際にアルファ値を使用して描き込み方を決定するため、アルファブレンディングと言われます。アルファブレンディングはZテストと同様にIDirect3DDevice9::SetRenderStateを使用して設定することができます。

アルファブレンディングを有効にするには下記のようなコードを記述します。

|  |
| --- |
| g\_pd3dDevice->SetRenderState(D3DRS\_ALPHABLENDENABLE, TRUE); |

**8.2 半透明合成**

アルファブレンディングにおいて、これから描きこもうとしているカラーをソースカラー(SRC)といいます。そして既にフレームバッファに描きこまれているカラーのことをデスティネーションカラー(DEST)といいます。半透明合成はソースアルファ(SRC\_α)を使用してソースカラーとデスティネーションカラーを混ぜ合わせることで実現されています。

半透明合成の計算式は下記のようになります。

**描き込まれるカラー = SRC×SRC\_α+DEST×(1.0f－SRC\_α)**

半透明合成を行うためのレンダリングステートの設定は下記のようなコードを記述します。

|  |
| --- |
| //ソースカラーにはソースアルファを乗算する設定。  g\_pd3dDevice->SetRenderState(D3DRS\_SRCBLEND, D3DBLEND\_SRCALPHA);  //デスティネーションカラーには1.0f-ソースアルファを乗算する設定。  g\_pd3dDevice->SetRenderState(D3DRS\_DESTBLEND, D3DBLEND\_INVSRCALPHA); |

**8.3 加算合成**

　光のエフェクト、炎のエフェクト、斬撃のエフェクトなど、光り輝くようなエフェクトは全て加算合成で実現されています。加算合成とは名前のとおり、色の加算になります。そのためポリゴンが重なれば重なるほど、白に近い色になっていきます。下記のようなエフェクトを実現するためには加算合成を行う必要があります。



加算合成は下記の計算式で実現されます。

**描き込まれるカラー = SRC×1.0f+DEST×1.0f**

加算合成を行うためのレンダリングステートの設定は下記のようになります。

|  |
| --- |
| //ソースカラーにはソースアルファを乗算する設定。  g\_pd3dDevice->SetRenderState(D3DRS\_SRCBLEND, D3DBLEND\_ONE);  //デスティネーションカラーには1.0f-ソースアルファを乗算する設定。  g\_pd3dDevice->SetRenderState(D3DRS\_DESTBLEND, D3DBLEND\_ONE); |

**8.4 実習**

下記のサンプルプログラムを使用して、下記の実習を行いなさい。

https://github.com/KawaharaKiyohara/DirectXLesson/アルファブレンディング実習

1. 重なって表示されているポリゴンを半透明合成できるようにしなさい。
2. 重なって表示されているポリゴンを加算合成できるようにしなさい。

**Chapter 9**

**ShaderTutorial\_05(ディフューズライト)**

このチャプターでは拡散反射光(ディフューズライト)を行います。ライティングの計算はシェーダープログラムの醍醐味の一つになります。ディフューズライトの計算式は下記のようになります。

**ライトの方向 = L(x,y,z)**

**ライトのカラー = C(r,g,b)**

**頂点の法線 = N(x,y,z)**

**内積を求める関数 = dot()**

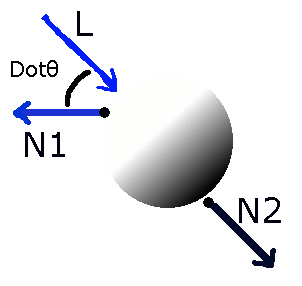
**最大値を取得する関数 = max()**

とした時に下記の計算式でライトのカラーが求まります。

**max(0,dot(-L, N)) \* C**

まず、内積の性質について説明します。内積は長さ1のベクトル(単位ベクトル)同士で計算した場合、同じ向きを向いているベクトルで計算すると1という結果を返します。また、直行しているベクトル(なす角が90度)で計算をすると0という結果を返します。最後に真逆を向いているベクトルで計算すると-1を返します。

つまり、ライトの向き\*-1と法線の向きが同じ場合(ライトを真正面から浴びている)は1を返し、ライトの向き\*-1と法線の向きが直行している場合は0を返し、ライトの向き\*-1と法線の向きが逆向きの場合は-1を返します。この結果をライトのカラーに乗算すると下記のような見え方になります。



Lがライトの方向、Nが法線です。

今回のサンプルではライトを４本使用しています。ライトを複数用意する理由は下記の絵を見るとイメージしやすいのではないでしょうか。

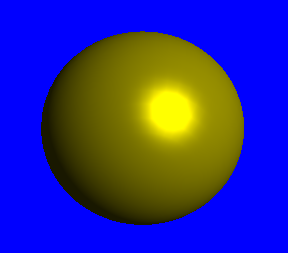


３Dモデルを綺麗に見せるためにライトの設定が重要なことが分かるかと思います。

アイドルや女優さんなどの写真集の撮影現場をイメージしてみてください。

**Chapter 10 スペキュラライト**

　前節では太陽などのような平行光源からのライティングをシミュレーションするディフューズライティングを学びました。この節では金属や人肌や髪の毛のツヤなどをミュレーションするスペキュラライトを学びましょう。



**10.1 フォン鏡面反射光**

　スペキュラライトの計算の仕方はいくつかあるのですが、ここではフォン鏡面反射と呼ばれる手法を学びましょう。

　ハイライトのような映り込みは、光源から入射した光が反射する角度から見た時が最も強く反射されます。つまり、視線が物体に当たって跳ね返った方向が、ライトの方向に近い時に明るく見えるのです。

　図１ 視線が物体に反射したベクトル

**反射した視線ベクトル**

**視線**

**法線**

**視線**

では、視線ベクトルを反射させるベクトルの計算の仕方を求めてみましょう。反射ベクトルの計算に必要な要素は下記の二つになります。

　・視線ベクトル

　・当たった面の法線

まず、視線ベクトルを求めてみましょう。視線ベクトルは視点－頂点座標で求めたベクトルを正規化することで計算することができます。この視線ベクトルをE、当たった面の法線をNとすると、反射ベクトルRは下記の計算で求まります。

**R = －E＋2×(N・Ｅ)×Ｎ**

**10.2 反射ベクトル**

　反射ベクトルの求め方は、先ほどの計算を使って求めればいいのですが、それほど難解な式ではありませんので、なぜあの式で反射ベクトルが求まるのか考えてみましょう。

**-Ｅ+２×(N・E)×N**

まず、網掛けになっている-Eがどういうベクトルなのか、下記の図を見て確認してみて下さい。

Ｎ

E

**-E**

**-Ｅ+２×(N・E)×N**

では続いて、網掛けになっている、(N・E)を見ていきましょう。N・Eで法線と視線ベクトルの内積を求めています。法線は向きだけを保持している大きさ1の正規化されたベクトルです。正規化されたベクトルとの内積を計算すると、そのベクトルに射影した時の長さが求まります。

**-E**

E

Ｎ

NとEの内積でこの長さが求まる。

**-Ｅ+２×(N・E)×N**

そして、**(N・E)×N**で、N・Ｅで求めた長さを法線Ｎに乗算していますので、下記のベクトルが求まります。

Ｎ

E

**(N・E)×Nで求まるベクトル**

**-E**

**-Ｅ+２×(N・E)×N**

そして、２×(N・E)×Nから上で求めたベクトルを２倍しているので、下記のベクトルが求まります。

**-E**

Ｎ

E

**２×(N・E)×Nで求まるベクトル**

そして、上で求まったベクトルを-Eに加算しているので、下記のベクトルが求まり、反射ベクトルが求まることになります。

E

Ｎ

Ｒ

**-E**

**10.3 スペキュラ反射の計算**

　では、本題のスペキュラ反射の計算に戻りましょう。スペキュラ反射は10.2で求めた視線の反射ベクトルがライトの方向と近いほど強く反射することになります。それをシミュレーションするために、下記の内積の性質を使用します。

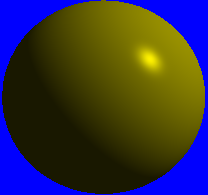
**・正規化されたベクトル同士の内積を計算すると、同じ向きのベクトルなら1を返し、直交する場合は0を返す。**

つまり、ライトの方向をL、視線の反射ベクトルをRとした場合、-L・Rは向きが同じなら1を返して、ベクトルが離れていくほど小さい値を返すようになりますので、これをスペキュラ光として利用すればうまくいきそうですね。

**10.4 スペキュラの絞り**

　では、最後にスペキュラの絞りについて見てみましょう。スペキュラの絞りとはハイライトの大きさを調整するためのものとなります。

　絞り5 絞り50

絞りはpow関数を使用することでシミュレーションできます。10.3で求めたスペキュラ光を下記のようにpow関数に渡してみましょう。

float t = max( 0.0f, dot( -L, R ) ); //10.3で求めたスペキュラ光。

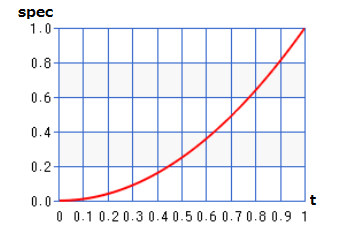
float3 spec = pow( t, 10.0f); //pow関数を使用して、最終的なスペキュラ光を求める。

pow関数は累乗を求める関数です。上記のコードですと、を求めていることになります。

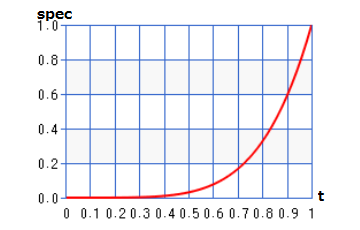
tは大きさ同士のベクトルで内積を計算しているので0～1.0を返してきます

では、tの累乗を求めるとどのような結果を返すのか下記のグラフをみて確認してみましょう。

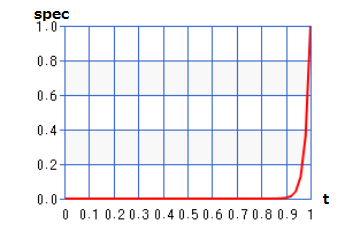
絞り2 : pow( t,2.0f )



絞り5 : pow(t, 5.0f)



絞り50 : pow( t, 50.0f )



このように、絞りが大きいほど急なカーブを描くようになるため、ハイライトが小さくなります。

**10.5 まとめ**

　スペキュラ反射を計算する手順を纏めます。

1. 視点―ワールド頂点座標で求まるベクトルを正規化して、視線ベクトルを求める。

(視点はeyePos、ワールド頂点座標はvertexPosとする)

　float3 eye = normalize(eyePos – vertexPos);

1. 視線ベクトルの反射ベクトルを求める。(ワールド頂点法線をvertexNormalとする)

　float3 R = -eye + 2.0f \* dot(vertexNormal, eye) \* vertexNormal;

1. 反射ベクトルとライトのベクトルとの内積を計算してスペキュラ光を求める。(ライトの方向をLとする)

　float3 spec = max( 0.0f, dot(R, -L ) );

1. ３で求めたスペキュラ光に絞りを適用させる。

　spec = pow(spec, 2.0f);

**10.6 実習課題**

DirextXLesson/ShaderTutorial\_06/kadai/ShaderTutorial\_06.slnを使用して、スペキュラライトを実装しなさい。

**Chapter 11 オフスクリーンレンダリング**

このチャプターではブルーム、被写界深度、SSSSS、影生成など最新の3Dグラフィックスになくてはならない表現を行うための基礎知識となるオフスクリーンレンダリングについて見ていきましょう。

**11.1 レンダリングターゲット**

レンダリングターゲットとは3Dモデルなどの描画先のことです。今まであまり意識してこなかったかもしれませんが、最終的に画面上に表示される絵はフレームバッファにレンダリングされた結果が表示されており、これもレンダリングターゲットになります。

オフスクリーンレンダリングとは、フレームバッファとは別に新しくレンダリングターゲットを作成して、そこにレンダリングを行う手法のことを言います。スクリーンに描画を行わないのでオフスクリーンレンダリングと呼ばれます。

**11.2 レンダリングターゲットの作成**

ではレンダリングターゲットの作成の仕方を見ていきましょう。まず、レンダリングターゲットとなるテクスチャを作成します。

|  |
| --- |
| //レンダリングターゲットとなるテクスチャを作成。  g\_pd3dDevice->CreateTexture(  1280, //テクスチャの幅  720, //テクスチャの高さ  1, //ミップマップの数。今は1でいい。  D3DUSAGE\_RENDERTARGET, //このテクスチャをレンダリングターゲットとして使用することを明示する。  D3DFMT\_A8R8G8B8, //テクスチャのフォーマット。ここではA8R8G8B8の32bitを指定。  D3DPOOL\_DEFAULT, //テクスチャメモリの確保の仕方。D3DPOOL\_DEFAULTでいい。  &m\_texture,　　　 //テクスチャのアドレスを格納するポインタのアドレス。  NULL //常にNULLを指定。  ); |

続いて、このテクスチャからサーフェイスを取得する。これはレンダリングターゲットを切り替えるときに必要となるもので、こういうものだと覚えておきましょう。

|  |
| --- |
| //サーフェイスの取得。  m\_texture->GetSurfaceLevel(0,&m\_surface); |

最後にデプスステンシルバッファの作成を行います。このバッファは必ずしも必要なわけではありませんが、レンダリングを行う際に3Dモデルの前後関係の判定が必要な場合は作成しましょう。

|  |
| --- |
| g\_pd3dDevice->CreateDepthStencilSurface(  1280, 　　　　　//デプスステンシルバッファの幅。必ずテクスチャと同じ解像度にする。  720,　　　　　　　　　　　　　　//デプスステンシルバッファの高さ。必ずテクスチャと同じ解像度にする。  D3DFMT\_D24X8, 　　　　　//深度バッファのフォーマット。深度バッファに24bit使用する。  D3DMULTISAMPLE\_NONE, //マルチサンプリングの設定。今回はD3DMULTISAMPLE\_NONEでいい。  　　0,　　　　　　　　　　　　 //画像の品質レベル。今回は０でいい。  TRUE,　　　　　　　　　　//TRUEでいい。  &m\_depthSurface,　　　　　//デプスステンシルサーフェイスのアドレスを格納するポインタのアドレス。  NULL //NULLでいい。  ); |

**11.3 レンダリングターゲットの切り替え**

ではレンダリングターゲットを切り替えて、オフスクリーンレンダリングを行うコードを見ていきましょう。

まず、後でもとに戻す必要があるので切り替える前に、現在のレンダリングターゲットを記録しておきましょう。

|  |
| --- |
| LPDIRECT3DSURFACE9 renderTargetBackup;  LPDIRECT3DSURFACE9 depthBufferBackup;  //元々のレンダリングターゲットを保存。後で戻す必要があるので。  g\_pd3dDevice->GetRenderTarget(0, &renderTargetBackup);  //元々のデプスステンシルバッファを保存。後で戻す必要があるので。  g\_pd3dDevice->GetDepthStencilSurface(&depthBufferBackup); |

続いてレンダリングターゲットを切り替えます。

|  |
| --- |
| g\_pd3dDevice->SetRenderTarget(0, m\_surface);  g\_pd3dDevice->SetDepthStencilSurface(m\_depthSurface);  //書き込み先を変更したのでクリア。  g\_pd3dDevice->Clear(0, NULL, D3DCLEAR\_TARGET | D3DCLEAR\_ZBUFFER, D3DCOLOR\_XRGB(255, 255, 255), 1.0f, 0); |

では、このレンダリングターゲットに対して、何か描画を行ってみましょう。描画に関しては特別なことを行う必要はありません。

|  |
| --- |
| g\_pTigerEffect->SetTechnique("SkinModel");  g\_pTigerEffect->Begin(NULL, D3DXFX\_DONOTSAVESHADERSTATE);  g\_pTigerEffect->BeginPass(0);  //定数レジスタに設定するカラー。  D3DXVECTOR4 color( 1.0f, 0.0f, 0.0f, 1.0f);  D3DXMATRIX mTigerMatrix;  D3DXMatrixRotationY( &mTigerMatrix, renderCount / 10.0f );  //ワールド行列の転送。  g\_pTigerEffect->SetMatrix("g\_worldMatrix", &mTigerMatrix);  //ビュー行列の転送。  g\_pTigerEffect->SetMatrix("g\_viewMatrix", &g\_viewMatrix);  //プロジェクション行列の転送。  g\_pTigerEffect->SetMatrix("g\_projectionMatrix", &g\_projectionMatrix);  g\_pTigerEffect->CommitChanges();  for( DWORD i = 0; i < g\_dwNumMaterials; i++ )  {  g\_pTigerEffect->SetTexture("g\_diffuseTexture", g\_pMeshTextures[i]);  // Draw the mesh subset  g\_pMesh->DrawSubset( i );  }  g\_pTigerEffect->EndPass();  g\_pTigerEffect->End(); |

最後にレンダリングターゲットを元に戻します。

|  |
| --- |
| g\_pd3dDevice->SetRenderTarget(0, renderTargetBackup); //戻す。  g\_pd3dDevice->SetDepthStencilSurface(depthBufferBackup); //戻す。 |

11.4 実習課題

ShaderTutorial\_10/kadaiを使用して、虎のオフスクリーンレンダリングを行い、レンダリングしたテクスチャを板ポリに張り付けなさい。板ポリの描画はCSpriteクラスを使いなさい。

解答例となる実行ファイルはShaderTutorial\_10/kadai/ShaderTutorial\_10/Answer.exeにあります。

**Chapter 12 投影シャドウ**

３Ｄゲームにおいて、リアルなグラフィックを実現するために影は非常に重要な要素になります。また、グラフィック面だけではなく、影は３Ｄオブジェクトの空間上の位置をユーザーに教えるための重要な要素となります。

このチャプターでは、基本となる古典的な影生成アルゴリズムの投影シャドウについて見ていきます。

**12.1 シャドウマップ**

投影シャドウはシャドウマップと呼ばれるテクスチャを使用して、影を落とすアルゴリズムの一つになります。シャドウマップというのは影が落ちる場所がテクスチャに描きこまれたものとなります。下の図を見てみて下さい。

カメラ

このように、影というのは物体によって光が遮られている箇所に発生するものだということが分かります。これは言い方を変えると、**光を放っているライトの位置から見たときに、手前に物体が存在していれば影が落ちる**と言うことができます。下記の絵を見てみて下さい。



キャラクターの影が岩に落ちているのが分かります。では、これをライトの方向から見た場合どうなるでしょうか？次の絵を見てみて下さい。

ライトの方向から見た絵



このように、ライトの位置から見てみると、影が落ちていた岩の部分は、キャラクターに遮られていることが分かります。この**ライトから見た絵**がシャドウマップです。

**12.2 ShadowMapクラス**

では、シャドウマップはどうやって作るのか？勘のいい人ならすでに気付いているかもしれませんが、これはチャプター11で勉強したオフスクリーンレンダリングを行うことで作成することが出来ます。つまり、ライトをカメラと見立てて影を生成したいオブジェクト(シャドウキャスターと呼ばれます)を、シャドウマップに対してレンダリングしてやればいいのです。ではShaderTutorial11のShadowMap.cppとShadowMap.hを開いてください。これはシャドウマップをクラス化したものになります。ShadowMapクラスには下記のメンバ変数があります。

ShadowMap.h

|  |
| --- |
| CRenderTarget renderTarget; //シャドウマップを書きこむレンダリングターゲット。  D3DXMATRIX lightViewMatrix; //ライトビューマトリクス。  D3DXMATRIX lightProjMatrix; //ライトプロジェクションマトリクス。  D3DXVECTOR3 iewPosition; //ライトビューの視点。  D3DXVECTOR3 viewTarget; //ライトビューの注視点。 |

シャドウマップを書きこむためのレンダリングターゲットや、ライトをカメラと見立てるためのパラメータがあるのが分かるかと思います。

では、ShadowMap::Update関数を見てみましょう。

|  |
| --- |
| //ライトビュープロジェクション行列を更新。  //普通のカメラと同じ。  //カメラの上方向を決める計算だけ入れておく。  D3DXVECTOR3 tmp = viewTarget - viewPosition;  D3DXVec3Normalize(&tmp, &tmp);  if (fabsf(tmp.y) > 0.9999f) {  //カメラがほぼ真上or真下を向いている。  D3DXMatrixLookAtLH(&lightViewMatrix, &viewPosition, &viewTarget, &D3DXVECTOR3(1.0f, 0.0f, 0.0f));  }  else {  D3DXMatrixLookAtLH(&lightViewMatrix, &viewPosition, &viewTarget, &D3DXVECTOR3(0.0f, 1.0f, 0.0f));  }  D3DXMATRIXA16 matProj;  D3DXMatrixPerspectiveFovLH(&lightProjMatrix, D3DXToRadian(60.0f), 1.0f, 0.1f, 100.0f); |

カメラの処理と全く同じです。ライトの位置を視点、そして注視点を使うことでライトの向きも決まります。あとは、カメラ行列とプロジェクション行列を作成するだけです。

では、最後にShadowMap::Drawを見てみましょう。

|  |
| --- |
| LPDIRECT3DSURFACE9 renderTargetBackup;  LPDIRECT3DSURFACE9 depthBufferBackup;  //元々のレンダリングターゲットを保存。後で戻す必要があるのでバックアップを行う。  g\_pd3dDevice->GetRenderTarget(0, &renderTargetBackup);  //元々のデプスステンシルバッファを保存。後で戻す必要があるので。  g\_pd3dDevice->GetDepthStencilSurface(&depthBufferBackup);  //レンダリングターゲットを変更する。  g\_pd3dDevice->SetRenderTarget(0, renderTarget.GetRenderTarget());  g\_pd3dDevice->SetDepthStencilSurface(renderTarget.GetDepthStencilBuffer());  //書き込み先を変更したのでクリア。  g\_pd3dDevice->Clear(0, NULL, D3DCLEAR\_TARGET | D3DCLEAR\_ZBUFFER, D3DCOLOR\_XRGB(255, 255, 255), 1.0f, 0);  //トラをシャドウマップにレンダリング。  g\_tiger.Draw(&lightViewMatrix, &lightProjMatrix, true, false);  g\_pd3dDevice->SetRenderTarget(0, renderTargetBackup); //戻す。  g\_pd3dDevice->SetDepthStencilSurface(depthBufferBackup); //戻す。 |

チャプター11で見た処理と大差ありません。レンダリングターゲットをシャドウマップに変更して、そこにシャドウキャスターをレンダリングしています。

**実習課題**

ここまでの内容を参考にして、ShaderTutorial11/kadaiを使用しライトから見た虎をシャドウマップにレンダリングできるようにしなさい。シャドウマップにレンダリングができているかどうかの確認が取れないと思うので、できたはずと思った時点で指導教員に確認を取るようにして下さい。

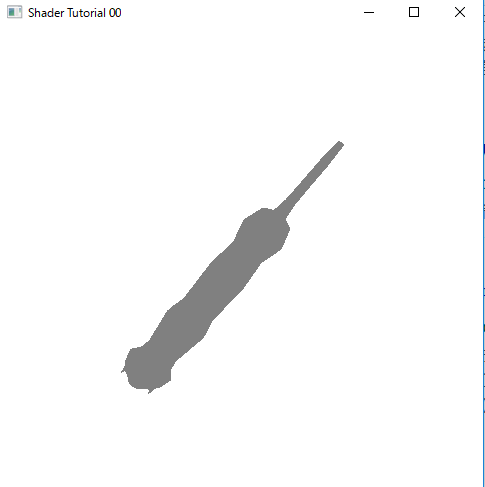
**12.3 シェーダーテクニック**

さて、前節でシャドウキャスターをライトから見た絵をシャドウマップにレンダリングすることが出来るようになりました。下記のようなシャドウマップが作成できたはずです。



ライトの方向から見た虎がレンダリングされています。しかし、投影シャドウで使用するシャドウマップは下記のようにグレースケールになっている必要があります。

(この理由は後述します。)



この節ではトラをグレースケールでレンダリングするためにシェーダーテクニックの切り替えを行っています。レンダリングターゲットに描きこまれるピクセルカラーを決めているのはピクセルシェーダーでしたね？ つまり、ピクセルシェーダーを書き換えればグレースケールでレンダリングできることになります。

今回は虎をレンダリングするためのピクセルシェーダーに通常レンダリング用と、シャドウマップへ描きこむ用の二つを用意して、それを用途に合わせて切り替えてグレースケールの描画を行っています。

では、ShaderTutorial11/basic.fxを見て下さい。96行目～117行目に下記のような記述があります。

|  |
| --- |
| /\*!  \*@brief 通常描画用のテクニック  \*/  SkinModelテクニックでは頂点シェーダーにVSMain、ピクセルシェーダーにPSMainを使用すると記述されている。  technique SkinModel  {  pass p0  {  VertexShader = compile vs\_3\_0 VSMain();  PixelShader = compile ps\_3\_0 PSMain();  }  SkinModelテクニックでは頂点シェーダーにVSMain、ピクセルシェーダーにPSRenderToShadowMapMainを使用すると記述されている。  }  /\*!  \*@brief シャドウマップ書き込み用のテクニック  \*/  technique SkinModelRenderToShadowMap  {  pass p0  {  VertexShader = compile vs\_3\_0 VSMain();  PixelShader = compile ps\_3\_0 PSRenderToShadowMapMain();  }  } |

このようにシェーダーテクニックが二つ記述されています。SkinModelが通常描画用のテクニック、SkinModelRenderToShadowMapがシャドウマップへの書き込み用のテクニックです。この二つのテクニックを見比べてみると、使用するピクセルシェーダーが異なっているのが分かります。では、PSRenderToShadowMapとPSMainを見比べてみましょう。

|  |
| --- |
| /\*!  \*@brief ピクセルシェーダー。  \*/  float4 PSMain( VS\_OUTPUT In ) : COLOR  {  //テクスチャからカラーをフェッチして、それを出力している。  float4 color = tex2D( g\_diffuseTextureSampler, In.uv );  return color;  }  /\*!  \*@brief シャドウマップ書き込み用のピクセルシェーダー。  \*/  float4 PSRenderToShadowMapMain(VS\_OUTPUT In) : COLOR  {  //灰色のカラーを出力している。  return float4(0.5f, 0.5f, 0.5f, 1.0f);  } |

通常描画で使用されているPSMainはテクスチャカラーを出力していますが、シャドウマップへの書き込みで使用されるPSRenderToShadowMapMainは灰色を出力しています。

これでシェーダー側の対応は完了です。次はCPU側でどのようにシェーダーテクニックを切り替えているのか見てみましょう。ShaderTutorial11/Model.cppのCModel::Draw関数を開いてください。

|  |
| --- |
| //シェーダー適用開始。  if (!isDrawShadowMap) {  //通常描画  g\_pModelEffect->SetTechnique("SkinModel");  }else {  //シャドウマップへの書き込みテクニック。  g\_pModelEffect->SetTechnique("SkinModelRenderToShadowMap");  } |

ID3DXEffect::SetTechniqueでテクニックを切り替えることができます。今回のサンプルでは、CModel::Draw関数の引数のisDrawShadowMapを使用して、シェーダーテクニックの切り替えを行っています。

**実習課題**

シャドウマップをグレースケールで作成できるように変更しなさい。

**12.4 シャドウマップの貼り付け**

前節でグレースケールのシャドウマップが作成できました。これで投影シャドウに使用するシャドウマップは完成したことになります。あとは、影を落とすオブジェクト(シャドウレシーバーと言われます)にシャドウマップを張り付けてやればよいことになります。では、どうやって張り付けるのか見ていきましょう。

例えば下記の絵の岩を画面に描画する際は、**岩の頂点座標×ワールド行列×カメラ行列×射影行列**という計算を行ってスクリーン座標に変換します。シャドウマップを張り付ける場合は先ほどの変換とは別に**岩の頂点座標×ワールド行列×ライトカメラ行列×ライト射影行列**という計算を行って、ライトをカメラと見立てた座標系への変換も行う必要があります。つまり岩に影を落とす場合、下記の２パターンの座標変換を行うことになります。

**カメラから見た絵　　　　　　　　　　ライトから見た絵**

では、具体的にコードを見てみましょう。ShaderTutorial\_11/basic.fxの54行目～70行目を見て下さい。

|  |
| --- |
| VS\_OUTPUT VSMain( VS\_INPUT In )  {  VS\_OUTPUT Out = (VS\_OUTPUT)0;  float4 worldPos;  worldPos = mul( In.pos, g\_worldMatrix ); //モデルのローカル空間からワールド空間に変換。  Out.pos = mul( worldPos, g\_viewMatrix ); //ワールド空間からビュー空間に変換。  Out.pos = mul( Out.pos, g\_projectionMatrix ); //ビュー空間から射影空間に変換。  Out.color = In.color;  Out.uv = In.uv;  if(g\_isShadowReciever == 1){  //シャドウレシーバー。  //ワールド座標をライトカメラから見た射影空間に変換する。  Out.lightViewPos = mul(worldPos, g\_lightViewMatrix);  Out.lightViewPos = mul(Out.lightViewPos, g\_lightProjectionMatrix);  　　}  return Out;  } |

g\_isShadowRecieverの値が1になっている場合はOut.lightViewPosにライトカメラから見た射影空間での座標が計算されています。これが頂点座標をライトから見た頂点に変換しているコードになります。

(注意 : lightViewPosのセマンティクスがTEXCOORD1になっているのは大丈夫なの？と思われるかもしれませんが。シェーダープログラマにとって、このセマンティクスは飾りです。TEXCOORDという名前ですが、UV座標以外の色々な計算結果をピクセルシェーダーに渡すときに利用されます。)

**12.4.1 正規化座標系**

では、ここで射影変換された頂点というのはどのような座標系になっているか復習してみましょう。例えば射影変換の結果、下記のような絵が描画された場合、頂点座標は下記のような縦横-1.0～1.0の座標系に変換されています。

　　　　　　　　　　(-1.0, 1.0) (1.0, 1.0)



(-1.0, -1.0) (1.0, -1.0)

このような座標系を正規化されたスクリーン座標系と呼ばれます。これがシャドウマップを張り付ける際に重要になってきますので、しっかりと抑えておいてください。

**12.4.2 UV座標に変換**

では、シャドウマップの貼り付けの話に戻りましょう。先ほど頂点シェーダーでシャドウレシーバーのオブジェクトの頂点座標にライトビュープロジェクション行列を乗算して、座標変換を行いました。そうすると岩のオブジェクトは下記のように座標変換されます。

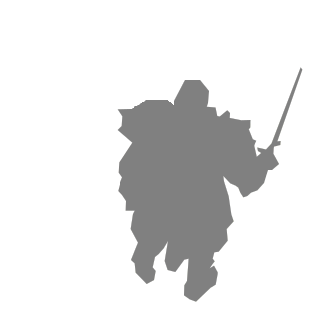
( -1.0, 1.0) (1.0, 1.0)



(-1.0, -1.0) (1.0, -1.0)

仮にこの人間のキャラクターをシャドウキャスターとした場合にどのようなシャドウマップが生成されているか見てみましょう。

(0.0, 0.0) (1.0, 0.0)



(0.0, 1.0) (1.0, 1.0)

このようなシャドウマップが生成されています。この二つの画像を見比べてみると、座標の範囲の違いはありますが、岩の影が落ちる部分はシャドウマップにマッピングするとちょうど灰色になっていることが分かります。

つまり、ライトビュープロジェクション行列で変換した座標(-1～1)をUV座標(0～1)に変換して、それを使ってシャドウマップを張り付けてやればうまくいきそうです。

座標の変換は下記のように行います。

　ライトビュープロジェクション行列で変換された座標をV、計算結果の格納先をUVとした場合に。

　UV = V × 0.5 // -1～1の範囲に0.5をかけるので、-0.5～0.5の範囲になる。

　UV = UV + 0.5 //-0.5～0.5の範囲に0.5を加算するので、0.0～1.0の範囲になる。

|  |
| --- |
| float4 PSMain( VS\_OUTPUT In ) : COLOR  {  float4 color = tex2D( g\_diffuseTextureSampler, In.uv );  if(g\_isShadowReciever == 1){  //射影空間(スクリーン座標系)に変換された座標はw成分で割ってやると(-1.0f～1.0)の  //範囲の正規化座標系になる。  //これをUV座標系(0.0～1.0)に変換して、シャドウマップをフェッチするためのUVとして活用する。  //この計算で(-1.0～1.0)の範囲になる。  float2 shadowMapUV = In.lightViewPos.xy / In.lightViewPos.w;  //この計算で(-0.5～0.5)の範囲になる。  shadowMapUV \*= float2(0.5f, -0.5f);  //そしてこれで(0.0～1.0)の範囲になってＵＶ座標系に変換できた。  shadowMapUV += float2(0.5f, 0.5f);  //シャドウマップは影が落ちているところはグレースケールになっている。  float4 shadowVal = tex2D(g\_shadowMapTextureSampler, shadowMapUV);  color \*= shadowVal;  }  return color;  } |

では実際のコードを見ていきましょう。

ShaderTutorial\_11/basic.fxの74行目～87行目までを見て下さい。

網掛けになっている箇所がシャドウマップの貼り付けを行っているコードです。シャドウマップからフェッチしてきたカラーを出力カラーに乗算しているコードを見てみて下さい。影が落ちているところはグレースケールになっているため、出力カラーが暗くなるのが分かるかと思います。これがシャドウマップをグレースケールにした理由です。

**実習課題**

シャドウマップを地面に貼り付けて影を落としなさい。

**Chapter 13 物理エンジンを活用した衝突判定**

このチャプターではBulletPhysicsという物理エンジンを活用した衝突判定について見ていきます。

**13.1 物理エンジン**

3Dゲームにおいて衝突判定は非常に難しい分野のプログラムだったのですが、物理エンジンの登場によって、その難易度は大幅に下がりました。物理エンジンで有名なものはHavokPhysics、PhysXが挙げられます。ただし、基本的な使い方はどのエンジンも大きくは変わりません。この節では、衝突判定の話に行く前に物理エンジンとはどういうものなのか簡単に説明します。

　物理エンジンにはいくつか機能的な分類があります。軟体物理、破壊物理などなど。ここでは剛体物理エンジンについて見ていきます。剛体とは形の変わらない物体のことを指します。剛体物理エンジンを使うためには物理ワールドを作成する必要があります。そして、ユーザーは剛体を生成して、そのインスタンスを物理ワールドに登録することで物理シミュレーションを行うことができます。下記は簡単な剛体シミュレーションの行い方です。

1. *物理ワールドの初期化*

|  |
| --- |
| //コリジョンコンフィギュレーションを作成。  collisionConfig = new btDefaultCollisionConfiguration();  //コリジョンディスパッチャーを作成。  collisionDispatcher = new btCollisionDispatcher(collisionConfig);  //ブロードフェーズの作成。  overlappingPairCache = new btDbvtBroadphase();  //コンストレイントソルバーの作成。  constraintSolver = new btSequentialImpulseConstraintSolver;  //物理ワールドの作成。  dynamicWorld = new btDiscreteDynamicsWorld(  collisionDispatcher,  overlappingPairCache,  constraintSolver,  collisionConfig  );  //重力を設定。  dynamicWorld->setGravity(btVector3(0, -10, 0)); |

続いて、剛体を生成して物理ワールドに登録するやり方です。

1. *剛体の生成と登録*

|  |
| --- |
| //剛体の形状を表すオブジェクトを作成する。  //このサンプルでは半径0.5の球体コリジョンを作成。  btSphereShape shape = new btSphereShape(0.5f);  //座標と回転を使って剛体の初期位置を決める。  btTransform transform;  transform.setIdentity();  transform.setOrigin(btVector3(rbInfo.pos.x, rbInfo.pos.y, rbInfo.pos.z));  transform.setRotation(btQuaternion(rbInfo.rot.x, rbInfo.rot.y, rbInfo.rot.z, rbInfo.rot.w));  //モーションステートの生成。  myMotionState = new btDefaultMotionState;  //質量、モーションステート、コリジョン形状から剛体情報を用意して、剛体を作成。  btRigidBody::btRigidBodyConstructionInfo btRbInfo(rbInfo.mass, myMotionState, shape, btVector3(0, 0, 0));  //剛体を作成。  rigidBody = new btRigidBody(btRbInfo);  //作成した剛体を物理ワールドに追加。  dynamicWorld-> addRigidBody(rigidBody); |

さて、初期化と剛体の登録までは終わりました。次は物理シミュレーションを行う方法を見ていきましょう。下記のコードを毎フレーム呼び出してください。

1. 物理シミュレーションの実行

|  |
| --- |
| //1フレームの経過時間(単位は秒)を引数に渡して、物理シミュレーションを行う。  dynamicWorld->stepSimulation(1.0f / 60.0f ); |

では最後に物理シミュレーションを行った結果をゲームに引き渡すコードを見てみましょう。

1. *ゲームオブジェクトへの引渡し*

|  |
| --- |
| btTransform trans;  myMotionState->getWorldTransform(trans);  //ゲームのオブジェクトに位置と回転を引き渡す。  position = \*(D3DXVECTOR3\*)&trans.getOrigin();  rotation = \*(D3DXQUATERNION\*)&trans. getRotation(); |

簡単な剛体物理シミュレーションの使い方を説明しましたが、このチャプターの趣旨は物理エンジンを活用した衝突判定です。なので、実はこのチャプターでは必要なのは下記の３つの処理になります。

1. *物理ワールドの初期化*
2. 剛体の生成と登録
3. 物理シミュレーションの実行

**13.2 コリジョン**

　コリジョンは衝突判定で用いられる用語で、コリジョン＝衝突という意味になります。物体同士のコリジョンを調べるためにはその物体の形状を決める必要があります。形状とは例えば下記のようなものがあります。

　・球体(SphereCollider)

　・箱(BoxCollider)

　・カプセル(CapsuleCollider)

　・3Dモデルと同じ形状をもったメッシュ(MeshCollider)

このようなコリジョンの形状のことをゲームエンジンのUnityではCollider(コライダー)と呼称しており、この教材でもその呼称を使用します。

多くのゲームではColliderに複雑な形状をもったMeshColliderを使うことはほとんどなく、SphereCollider、BoxCollider、CapsuleColliderを組み合わせて複雑な形状のColliderを作成しています。その理由はモデルから生成したMeshColliderは非常に詳細な形状を持っており、衝突判定の処理に時間がかかってしまいます。また、複雑な形状のColliderとの衝突判定は往々にして当たり抜けが発生しやすくなります。MeshColliderを使うとしても、ポリゴン数を減らした別データを用意することがほとんどです。

しかし、MeshCollider以外は実装するのが非常に簡単ですので、今回はモデルから自動的に生成することが可能なMeshColliderの作り方を見ていきます。

**13.3 MeshCollider**

DirectXLesson/ CollisionDemo/game/Physics/ MeshCollider.cppを開いてください。このソースファイルの27行目から始まっているCreateFromSkinModel関数がMeshColliderを作成しているコートです。モデルを元に作成を行っているため、第一引数にSkinModelのインスタンスのアドレスを受け取っています。第二引数のoffsetMatrixはモデルの頂点をワールド座標系に変換させるための行列です。

|  |
| --- |
| LPD3DXMESH mesh = model->GetOrgMeshFirst(); |

このコードでメッシュのアドレスを取得しています。

*頂点バッファの作成*

|  |
| --- |
| //頂点ストライドを取得。  DWORD stride = D3DXGetFVFVertexSize(mesh->GetFVF());  //頂点バッファを取得。  LPDIRECT3DVERTEXBUFFER9 vb;  mesh->GetVertexBuffer(&vb);  //頂点バッファの定義を取得する。  D3DVERTEXBUFFER\_DESC desc;  vb->GetDesc(&desc);  //頂点バッファをロックする。  D3DXVECTOR3\* pos;  vb->Lock(0, 0, (void\*\*)&pos, D3DLOCK\_READONLY);  VertexBuffer\* vertexBuffer = new VertexBuffer;  int numVertex = mesh->GetNumVertices();  //当たりデータで使用する頂点バッファを作成。  for (int v = 0; v < numVertex; v++) {  D3DXVECTOR3 posTmp = \*pos;  if (offsetMatrix) {  D3DXVec3TransformCoord(&posTmp, pos, offsetMatrix);  }  vertexBuffer->push\_back(posTmp);  char\* p = (char\*)pos;  p += stride;  pos = (D3DXVECTOR3\*)p;  }  vb->Unlock();  vb->Release();  vertexBufferArray.push\_back(vertexBuffer); |

34行目から60行目まではメッシュコライダーを作成するための頂点バッファを作成するコードです。

*インデックスバッファの作成*

|  |
| --- |
| //続いてインデックスバッファを作成。  LPDIRECT3DINDEXBUFFER9 ib;  mesh->GetIndexBuffer(&ib);  D3DINDEXBUFFER\_DESC desc;  ib->GetDesc(&desc);  int stride = 0;  if (desc.Format == D3DFMT\_INDEX16) {  //インデックスが16bit  stride = 2;  }  else if(desc.Format == D3DFMT\_INDEX32){  //インデックスが32bit  stride = 4;  }  //インデックスバッファをロック。  char\* p;  HRESULT hr = ib->Lock(0, 0, (void\*\*)&p, D3DLOCK\_READONLY);  IndexBuffer\* indexBuffer = new IndexBuffer;  for (int i = 0; i < (int)desc.Size / stride; i++) {  unsigned int index;  if (desc.Format == D3DFMT\_INDEX16) {  unsigned short\* pIndex = (unsigned short\*)p;  index = (unsigned int)\*pIndex;  }  else {  unsigned int\* pIndex = (unsigned int\*)p;  index = \*pIndex;  }    indexBuffer->push\_back(index);  p += stride;  }  ib->Unlock();  ib->Release();  indexBufferArray.push\_back(indexBuffer); |

63行目から109行目まではインデックスバッファを作成するためのコードです。

*btIndexMeshの生成*

|  |
| --- |
| //インデックスメッシュを作成。  btIndexedMesh indexedMesh;  IndexBuffer\* ib = indexBufferArray.back();  VertexBuffer\* vb = vertexBufferArray.back();  indexedMesh.m\_numTriangles = (int)ib->size() / 3;  indexedMesh.m\_triangleIndexBase = (unsigned char\*)(&ib->front());  indexedMesh.m\_triangleIndexStride = 12;  indexedMesh.m\_numVertices = (int)vb->size();  indexedMesh.m\_vertexBase = (unsigned char\*)(&vb->front());  indexedMesh.m\_vertexStride = sizeof(D3DXVECTOR3);  stridingMeshInterface->addIndexedMesh(indexedMesh);    }  meshShape = new btBvhTriangleMeshShape(stridingMeshInterface, true); |

そして最後の99行目から112行目までのコードで、作成した頂点バッファ、インデックスバッファを用いてBulletPhysicsのbtIndexMeshを生成して、btTriangleMeshShapeを生成しています。これでMeshColliderの完成です。

**13.4 RigidBodyの登録**

　MeshColliderを作成することができたので、RigidBodyを作成しましょう。ここではbulletPhysicsのbtRigidBodyを直接触らずに、ラッパークラス(\*[[1]](#footnote-1))のRigidBodyを使います。DirectXLesson/ CollisionDemo/game/MapChip.cppの50行目から58行目にかけてサンプルとなるコードがあります。

|  |
| --- |
| //スキンモデルからメッシュコライダーを作成する。  D3DXMATRIX\* rootBoneMatrix = modelData.GetRootBoneWorldMatrix();  meshCollider.CreateFromSkinModel(&model, rootBoneMatrix);  //続いて剛体を作成する。  //まずは剛体を作成するための情報を設定。  RigidBodyInfo rbInfo;  rbInfo.collider = &meshCollider; //剛体のコリジョンを設定する。  rbInfo.mass = 0.0f; //質量を0にすると動かない剛体になる。  rbInfo.pos = position;  rbInfo.rot = rotation;  //剛体を作成。  rigidBody.Create(rbInfo);  //作成した剛体を物理ワールドに追加。  game->GetPhysicsWorld()->AddRigidBody(&rigidBody); |

これで剛体を物理ワールドに登録することができました。剛体の質量を設定するときに0.0を設定していることに注意してください。質量を0.0にすることで、その剛体は静的になり、他の剛体と衝突しても動くことがなくなります。地面や、建物などのような動かない剛体は0.0を設定してください。

　これで背景などの静的オブジェクトのMeshColliderを作成することができるようになり、そのColliderを設定した剛体を物理ワールドに登録することもできるようになりました。

これで単純な剛体同士の衝突判定であれば、全て物理エンジンが計算を行ってくれるようになりました。しかし、ユーザーが操作するゲームキャラクターの挙動を全て物理シミュレーション任せにすることはできません。なぜなら、ゲームキャラクターが移動する方向と速度の決定はユーザーが決定します、このときキャラクターの挙動は物理的に正しくない挙動をすることが多々あります (物理的に正しい挙動は操作性が悪く感じることがあります) 。また、物体に衝突した時のキャラクターの挙動もゲームの状況によって異なることもあるでしょう。反発する壁とかありそうですよね。そのため、ゲームキャラクターのコリジョン処理はアプリケーション側で実装したほうが都合の良い場合が多々あります。

次の節からは、キャラクターのコリジョン処理を実装するために必要な知識のコリジョン検出とコリジョン解決を見ていきましょう。

**13.5コリジョン検出とコリジョン解決**

　キャラクターコントローラーの話をする前に、コリジョン検出とコリジョン解決の話をします。コリジョン処理には大きく分類するとコリジョン検出とコリジョン解決に分けることができます。コリジョン検出はCollider同士が衝突するかどうか、衝突する場合は衝突点を求めたり、衝突する面の法線を求めたりする処理のことをいいます。コリジョン解決は衝突する場合のめり込みの補正の処理を差します。

コリジョン検出は非常に難しいプログラムを実装することになるのですが、その難しい部分はBulletPhysicsが実装を行ってくれているので、ありがたくそれを使いましょう。下記はBulletPhysicsのレイキャストを使用した衝突判定のサンプルコードになります。

|  |
| --- |
| btTransform start;  start.setOrigin(btVctor3(10.0f, 10.0f, 10.0f)); //レイの始点を作成。  btTransform end;  end.setOrigin(btVector3(20.0f, 10.0f, 10.0f)); //レイの終点を作成。  ClosestConvexResultCallback cb //衝突したときに呼ばれるコールバックの関数オブジェクト。  dynamicWorld-> ConvexSweepTest(  shape, //レイを飛ばすオブジェクトの形状  start, 　　　　　　　　　　　　　　　　　//レイの始点  end, //レイの終点  cb 　　　　　　　　　　　　　　　　 //コールバックオブジェクト  );  if(cb.hasHit()){  //衝突した  } |

ConvexSweepTest関数にオブジェクトの形状を渡していることに注意してください。この形状はbtCapsuleShape、btBoxShape、btSphereShapeのインスタンスです。このレイキャストは下記の図のようなことをしていると考えて下さい。

レイ

始点

終点

上の図の始点と終点の間に物理ワールドに登録されている剛体がある場合は、衝突しているという結果になります。

　続いてコリジョン解決を見ていきましょう。コリジョン解決とはコリジョン検出を行った結果、衝突していることが分かった場合に、その衝突を解決するものです。例えば下記のようなケースを考えてみて下さい。

物体がこのような位置関係になる場合にコリジョン検出を行うと、衝突しているという結果が返ってきます。多くのゲームではこのようなときは下記のような位置関係に補正するプログラムが実行されていると思います。

めり込んだ分を押し戻す

青いオブジェクトが赤い矢印の方向に押し戻されています。これがコリジョン解決です。

**13.6 キャラクターコントローラー**

　最近の物理エンジンには前節で勉強した典型的なゲームキャラクターのコリジョン検出とコリジョン解決を行ってくれるキャラクターコントローラーというものがあります。しかし前述したように、キャラクターの挙動というのはゲームによって細かい調整が行われます。そのため、キャラクターコントローラーが行っている処理を理解するということは無駄ではありません。この節ではサンプルプログラムのCharacterControllerクラスを例にして、処理を説明していきます。

DirectXLesson/CollisionDemo/game/Physics/CharacterController.cppの128行目から285行目までのプログラムを見てみて下さい。この関数がコリジョン検出、コリジョン解決を行っている関数です。

*移動速度に従って、次の移動先を計算しているコード*

|  |
| --- |
| //速度に重力加速度を加える。  m\_moveSpeed.y += m\_gravity \* (1.0f / 60.0f);  //次の移動先となる座標を計算する。  D3DXVECTOR3 nextPosition = m\_position;  //速度からこのフレームでの移動量を求める。オイラー積分。  D3DXVECTOR3 addPos = m\_moveSpeed;  addPos \*= 1.0f/60.0f;  nextPosition += addPos; |

130行目～137行目までのコードは次の移動先を計算しているコードになります。

移動速度の単位はm/secになっているために、移動速度に1フレームの経過時間(1/60秒)を乗算しています。そして移動量が求まったら、それをnextPositionに加算しています。

*XZ平面でのコリジョン検出*

|  |
| --- |
| //カプセルコライダーの中心座標 + 0.2の座標をposTmpに求める。  D3DXVECTOR3 posTmp = m\_position;  posTmp.y += m\_height \* 0.5f + m\_radius + 0.2f;  //レイを作成。  btTransform start, end;  start.setIdentity();  end.setIdentity();  //始点はカプセルコライダーの中心座標 + 0.2の座標をposTmpに求める。  start.setOrigin(btVector3(posTmp.x, posTmp.y, posTmp.z));  //終点は次の移動先。XZ平面での衝突を調べるので、yはposTmp.yを設定する。  end.setOrigin(btVector3(nextPosition.x, posTmp.y, nextPosition.z));  SweepResultWall callback;  callback.me = m\_rigidBody.GetBody();  callback.startPos = posTmp;  //衝突検出。  game->GetPhysicsWorld()->ConvexSweepTest((const btConvexShape\*)m\_collider.GetBody(), start, end, callback); |

続いて、移動先が求まったのでY成分を無視したXZ平面での移動だけ考えます。移動前の座標を始点、移動後の座標を終点とするレイを作成してレイキャストを行ってコリジョン検出を行っています。

*XZ平面でのコリジョン解決*

|  |
| --- |
| //当たった。  //壁。  D3DXVECTOR3 vT0, vT1;  //XZ平面上での移動後の座標をvT0に、交点の座標をvT1に設定する。  vT0 = { nextPosition.x, 0.0f, nextPosition.z };  vT1 = {callback.hitPos.x, 0.0f, callback.hitPos.z};  //めり込みが発生している移動ベクトルを求める。  D3DXVECTOR3 vMerikomi;  vMerikomi = vT0 - vT1;  //XZ平面での衝突した壁の法線を求める。。  D3DXVECTOR3 hitNormalXZ = callback.hitNormal;  hitNormalXZ.y = 0.0f;  D3DXVec3Normalize(&hitNormalXZ, &hitNormalXZ);  //めり込みベクトルを壁の法線に射影する。  float fT0 = D3DXVec3Dot(&hitNormalXZ, &vMerikomi);  //押し戻し返すベクトルを求める。  //押し返すベクトルは壁の法線に射影されためり込みベクトル+半径。  D3DXVECTOR3 vOffset;  vOffset = hitNormalXZ;  vOffset \*= (-fT0 + m\_radius);  nextPosition += vOffset;  D3DXVECTOR3 currentDir;  currentDir = nextPosition - m\_position;  currentDir.y = 0.0f;  D3DXVec3Normalize(&currentDir, &currentDir);  if (D3DXVec3Dot( &currentDir, &originalXZDir ) < 0.0f) {  //角に入った時のキャラクタの振動を防止するために、  //移動先が逆向きになったら移動をキャンセルする。  nextPosition.x = m\_position.x;  nextPosition.z = m\_position.z;  break;  } |

この176行目～208行目までがコリジョン解決を行っているコードです。この処理は**壁ずり**と呼ばれている壁に衝突した際の典型的なコリジョン解決となります。

**13.6.1 壁ずり**

では壁ずりについて見ていきましょう。下記の図のようにキャラクターが移動した場合を考えてみて下さい。

移動した結果、この点で衝突した。

**Chapter 14 パーティクル**

**14.1 ビルボード**

ビルボードとは板ポリが常にカメラの方向を向く手法のことを言います。

常にカメラの方向を向くということは、カメラの回転行列を加えることになります。

カメラの回転行列は、カメラ行列の逆行列の平行移動成分を削除することで求めることができます。

|  |
| --- |
| D3DXMATRIX viewRotMatrix;  D3DXMatrixInverse(&viewRotMatrix, NULL, &viewMatrix); //カメラ行列の逆行列を求める。  //カメラの平行移動成分を0にする。  viewRotMatrix.m[3][0] = 0.0f;  viewRotMatrix.m[3][1] = 0.0f;  viewRotMatrix.m[3][2] = 0.0f;  viewRotMatrix.m[3][3] = 1.0fl |

上のコードでカメラの回転行列を求めることができます。

あとは、この行列を描画したいオブジェクトのワールド行列に適用することで、ビルボードが完成します。

|  |
| --- |
| D3DXMatrix m;  m = viewRotMatrix \* viewMatrix \* projMatrix; |

**14.2 パーティクルに初速度を与えて動くようにする**

UnityやUnrealEngineなどのパーティクルエンジンにはエミッターに初速度を与えることができると思います。今、みなさんに触ってもらっているパーティクルのプログラムは板ポリが一枚だけ表示されているように見えるかもしれませんが、実は何枚も同じ場所に生成されています。そこで、このパーティクルエンジンンにも初速度のパラメータを追加して、パーティクルを動かせるようにしてみましょう。

まず、パーティクルのパラメータに初速度を追加します。

ParticleEmitter.h

|  |
| --- |
| struct SParicleEmitParameter{  //初期化。  void Init()  {  memset(this, 0, sizeof(SParicleEmitParameter));  }  const char\* texturePath; //!<テクスチャのファイルパス。  float w; //!<パーティクルの幅。  float h; //!<パーティクルの高さ。  float intervalTime; //!<パーティクルの発生間隔。  **D3DXVECTOR3 initSpeed; //初速度。**  }; |

このパラメータをCParticleクラスのインスタンスに渡してやって、パーティクルを動かしてください。

恐らく、CParticleクラスに速度のメンバ変数や、座標のメンバ変数が必要になるはずです。そして。CParticle::Render関数でパーティクルのワールド行列を計算する必要が生まれます。

今回はここまでのヒントで実習にチャレンジしてみて下さい。

**14.2.1 初速度をパーティクルに引き渡す。**

パーティクルをエミットされる際に、前節で追加した構造体を使用して初速度を渡しました。各粒子の描画、ワールド行列の更新を行っているのはCParticlクラスですので、この初速度をCParticleクラスに渡してやる必要があります。恐らくCParticleクラスに速度のメンバ変数を追加することになるでしょう。また、速度を使って位置も変位させていくはずですから、位置を表す座標のメンバ変数も必要になるはずです。

|  |
| --- |
| /\*!  \* @brief パーティクル。  \*/  class CParticle{  CPrimitive 　　primitive; //!<プリミティブ。  LPDIRECT3DTEXTURE9　 texture; //!<テクスチャ。  ID3DXEffect\* 　　shaderEffect; //!<シェーダーエフェクト。  **D3DXVECTOR moveSpeed; //!<速度。**  **D3DXVECTOR3 position; //!<座標**  public:  CParticle();  ~CParticle();  void Init(const SParicleEmitParameter& param);  void Update();  void Render(const D3DXMATRIX& viewMatrix, const D3DXMATRIX& projMatrix) ;  }; |

速度を引き渡すことができたのであれば、その速度を使って座標を変位させていくだけです。今回は座標の変異はUpdate関数で行いましょう。

|  |
| --- |
| void CParticle::Update()  {  position += moveSpeed;  } |

あとはこの座標を使用して平行移動行列を作成して、ワールド行列に適用すれば完了です。

そこは自分で考えて実装してみてください。

**14.3 初速度に乱数を加えてみよう。**

今のままですと、決まった一定方向にパーティクルが飛んでいくだけになっています。もちろんそういうパーティクルもありますが、炎や煙のパーティクルはこれでは実装できません。炎や煙のような粒子のパーティクルは空気の流れなどを受けて均一の方向には流れていきません。そこでパーティクルに初速度に乱数を与えて、これを非常に簡単に近似してみましょう。

**14.3.1 乱数アルゴリズム**

　パーティクルから話がずれるのですが、パーティクルだけではなく、ゲームのクオリティを上げるための乱数アルゴリズムについて少しだけ紹介します。

　C言語の標準関数のrand関数は線形合同法と呼ばれるアルゴリズムを使用しています。このアルゴリズムはお世辞にも品質の高い乱数アルゴリズムとは言えません。そのため、ゲーム会社では自前で別の乱数アルゴリズムを使った乱数生成機を実装しています。UnityやUnrealEngineを使う場合は、エンジンが乱数生成機を実装しています。

今回、私の作成したパーティクルのデモではメルセンヌ・ツイスターと言われる乱数アルゴリズムを使った乱数生成機を実装しています。

**14.3.2 初速度に乱数を加えてみよう。**

　では本題の初速度に乱数を加える処理を考えていきましょう。初速度に乱数を加えるので、初速度を引き渡すInit関数で初速度を加工してやればいいことになります。では、私の作成したパーティクルデモのプログラムを参考までに記載します。

|  |
| --- |
| //初速度に乱数を加える。  //random.GetRandDoubleは0.0～1.0を返してくる関数。これに-0.5してから\*2.0しているので、  //-1.0～1.0の乱数を取得している事になる。この乱数に対して、パラメータで渡された諸速度の速度のランダム幅を乗算してやることで  //速度をランダムにしている。  velocity = param.initVelocity;  velocity.x += (((float)random.GetRandDouble() - 0.5f) \* 2.0f) \* param.initVelocityVelocityRandomMargin.x;  velocity.y += (((float)random.GetRandDouble() - 0.5f) \* 2.0f) \* param.initVelocityVelocityRandomMargin.y;  velocity.z += (((float)random.GetRandDouble() - 0.5f) \* 2.0f) \* param.initVelocityVelocityRandomMargin.z; |

**14.3.3 速度に重力加速度を加えてみよう。**

　加速度というのは速度を変位させていくものになります。例えば、重力加速度が9.8m/sec^2だというのは、中学生あたりで勉強したと思います。9.8m/sec^2というのは物体の落下速度が1秒ごとに9.8m加速するという意味になります。

3Dゲームでは速度は向きと大きさを表現できるベクトルで考えることがほとんどです。

例えば、マリオのようなジャンプアクションゲームで走りながらジャンプした場合、下記のような速度になります。

|  |
| --- |
| moveSpeed = D3DXVECTOR3(1.0f, 5.0f, 0.0f); // X方向に1m/frame、Y方向に5.0m/frameの速度。 |

では、この速度を使用してプレイヤーの座標を動かしてみましょう。

|  |
| --- |
| player.position += moveSpeed; //座標を変位させる。 |

このようなコードを記述することで、プレイヤーは3D空間上を動くことができます。

では、加速度について考えていきましょう。加速度というのは速度を変位させていくものになります。つまり、重力落下を実装したい場合は速度に対して加速度を適用する計算をすればいいことになります。

では先ほどのmoveSpeedに重力加速度を加えるコードを追加してみましょう。

|  |
| --- |
| D3DXVECTRO3 gravity = D3DXVECTOR3(0.0f, -0.16f, 0.0f); //Y方向に-0.16m/frame^2の重力加速度。  moveSpeed += gravity; //重力加速度をmoveSpeedに適用する。 |

では、ここまでの説明を参考にしてパーティクルに重力落下を追加してみて下さい

**14.4 加算合成**

では、パーティクルの実習の最後にパーティクルに加算合成を行える機能を追加してみましょう。

　加算合成はチャプター８で実装を行いましたので、その復習になります。

**14.4.1 レンダリングステート**

　加算合成とはアルファブレンディングの一種で、これからフレームバッファに描きもうとしているカラーを、既に書き込まれているカラーと加算して描きこむことでした。

カラーの描き込み方はIDirect3DDevice9::SetRenderState関数を使用することで設定できます。

*サンプルコード*

|  |
| --- |
| //アルファブレンディングを有効にする。  g\_pd3dDevice->SetRenderState(D3DRS\_ALPHABLENDENABLE, TRUE);  g\_pd3dDevice->SetRenderState(D3DRS\_SRCBLEND, D3DBLEND\_ONE);  g\_pd3dDevice->SetRenderState(D3DRS\_DESTBLEND, D3DBLEND\_ONE); |

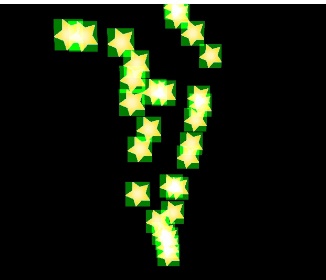
IDirect3DDevice9::DrawIndexedPrimitiveは現在のレンダリングステートの状態を使用してプリミティブを描画します。上記のレンダリングステートのコードはDrawIndexedPrimitiveの直前に記述すれば、加算合成を確認できるはずです。

**14.4.2 レンダリングステートの切り替えによるオーバーヘッド**

当然ですがレンダリングステートの切り替えというのはタダではありません。関数呼び出しのオーバーヘッドは当然かかりますし、PS4などのハードでは不要なレンダリングステートの切り替えはGPUにも悪影響を与えます。そのため、レンダリングステートの切り替えというのはラッパークラスを作成して、IDirect3DDevice9をカプセル化して制御する方がベターな場合があります。今回はわかりやすさのためにDrawIndexedPrimitiveの直前で毎回コールしていますが、まだまだ改善の必要のあるコードになっています。

**14.4.3 シェーダーの変更**

　さて、前節でレンダリングステートを加算合成を行うように変更しました。しかし、それだけでは下記のような見え方になってしまい、まだ問題があります。



このようにアルファで抜けて欲しいはずの部分が正しく抜けていません。

ではparticleDemo/ShaderTutorial\_04/ColorTexPrim.fxを開いてみてください。このシェーダーファイル

加算用のピクセルシェーダーのPSMainAdd関数の51行目を見てみましょう。

|  |
| --- |
| return float4(tex.xyz \* g\_alpha, 1.0f/g\_brightness ); |

この行がフレームバッファに描きこむソースカラーを返している処理になるのですが、実は星の画像はαで色を抜いている箇所にもカラーが埋め込まれています。そのためそのカラーの情報が加算合成されるために、上のような見え方になっていました。この行を下記のように変更してください。

|  |
| --- |
| return float4(tex.xyz\*tex.a, 1.0f/g\_brightness ); |

これできれいにαの部分が抜けるようになったはずです。

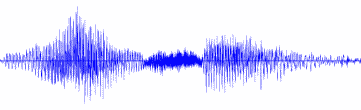
**Chapter 15 サウンド**

**15.1 XAudio2**

このチャプターではMicrosoftが提供している、WindowsおよびXbox向けの低レベルのオーディオAPIであるXAudio2を使用したサウンドプログラミングについて解説をします。

**15.2 PCMデータ(波形データ)**

　音とは空気を振動して伝わってきます。そして、その新動が鼓膜に伝わり鼓膜を揺らすことで私たちは音を認識することができます。音色はこの音の振動速さに依存します。振動は一秒間の振動の回数で測定され、単位をヘルツ(Hz)といいます。下のような図を見たことがあるのではないでしょうか。



コンピューターでは音を再生するためには、上記の音の振動のデジタル化を行う必要があり、デジタル化したデータをPCMデータ、波形データなどと呼ばれます。では簡単な波形データを見てみましょう。

|  |
| --- |
| char waveData[] = { 128, 96, 64, 32, 0, -32, -61. -89, -126, -113, -89, -48, -12, 24, 65}; |

これが波形データです。0が振動なし、そして128～-128の間で値が揺れ動いていると思います。これが振動になります。コンピューターはPCMデータを流し込まれて、そのデータ通りに振動を起こすことによって、音を再生しているのです。

**15.3 色々なサウンドファイル**

　サウンドファイルには色々なフォーマットがあるが、最も多く使われているのはwave、mp3、oggなどだと思います。このうちのwaveフォーマットはPCMデータをそのまま保存しているフォーマットになります。そのため音質の劣化がないため、品質は最も高くなります。続いてmp3やoggといったフォーマットはPCMデータを非可逆圧縮したフォーマットになります。そのため音質はwaveフォーマットに比べると若干劣化していることになります(ただし、まず気づきません)。mp3やoggはwaveと比べると1/10以下のファイルサイズになるため、外部記憶メディアの容量が足りない場合などはこれらを活用すべきでしょう。ただし、音を再生するためには圧縮されているPCMデータを復元する必要があるため、CPU不可は増大します。

**15.4 オンメモリ再生とストリーミング再生**

　音を再生するためにはPCMデータをメモリにロードして、そのデータをサウンドデバイスに流し込む必要があります。オンメモリ再生とは、再生したいサウンドファイルのデータを全てメモリ上にロードして再生する方法になります。一方ストリーミング再生とは全てのデータをメモリに乗せるのではなく、少しずつデータをロードして再生を行い、再生が完了したデータは逐次破棄していく手法となります。そのため、メモリの節約ができメインメモリの少ないハードウェアでもBGMなどのようなサイズの大きなサウンドデータを再生することが可能です。一般的にサイズの小さなSEなどのデータはオンメモリ再生、サイズの大きなBGMなどのデータはストリーミング再生を行うことが多いです。では、次の節から実際にXAudio2を活用したサウンド再生について見ていきましょう。

**15.5 XAudioの初期化**

　tkEngine/Sound/tkSoundEngine.cpp、tkEngine/Sound/SoundEngine.hでXAudioの初期化を行っています。CSound::Init関数ではマスターボリューム、リバーブエフェクト、サブミックスボイス、3Dサウンドの初期化などを行っています。

**15.6 サウンドソース**

tkEngine/Sound/tkSoundSource.cpp、tkSoundSource.hを見てみてください。このクラスはサウンドソース(音源)の処理を記述しています。音源であるため、位置を記録するCVector3型のメンバー変数も保持しています。初期化関数にはオンメモリ再生用の初期化関数のCSoundSource::Initと、ストリーミング再生用の初期化関数のCSoundSource::InitStreaming関数が用意されています。

**15.7 サウンドリスナー**

　サウンドソース(音源)があるのであれば、その音を聞くサウンドリスナーが必要です。サウンドリスナーの処理はtkEngine/Sound/tkSoundEngine.cppに記述されています。

tkSoundEngine::Update関数では登録されているサウンドソースが３Dサウンドであれば、サウンドリスナーとの位置関係によって、3Dサウンドエフェクトを計算するプログラムが記述されています。

**15.8 waveファイル**

tkEngine/Sound/ tkWaveFile.cppにはwaveファイルを読み込むためのコードが書かれています。このクラスはXAudioの機能は使っていません。WaveファイルからPCMデータを読み込むコードが記述されています。Waveファイルの読み込みは同期読み込みのCWave::Read関数と、非同期読み込みのCWave::ReadAsync関数が用意されています。CWave::ReadAsync関数は内部で読み込みスレッドを起動して、メインスレッドと並列してPCMデータの読み込みを行っています。メインスレッドをブロックしないため、ストリーミング再生の時に使用されています。

**16 法線マップ**

**16.1 概要**

このチャプターでは法線情報をテクスチャに書き込んで、低ポリゴンで細かなディテールを表現するための技術の法線マップについて見ていきます。

**16.2法線マップの登場**

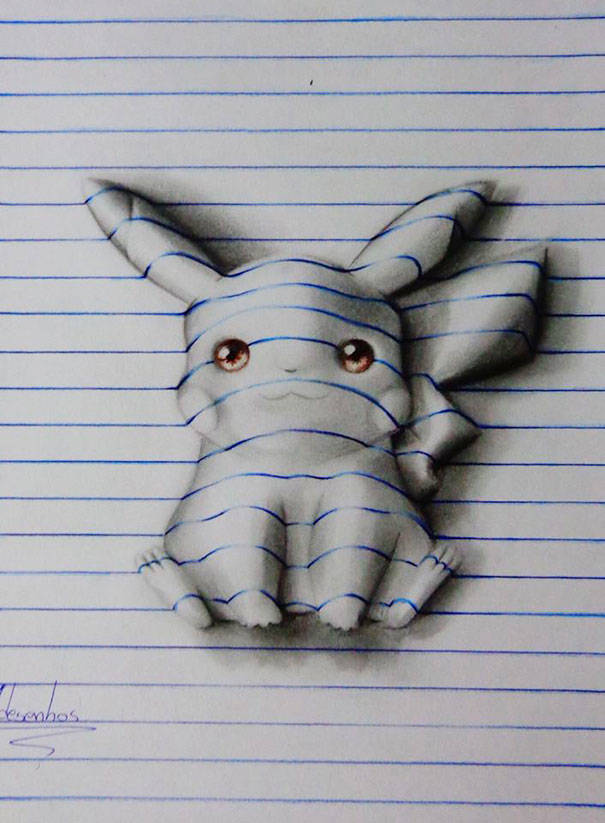
コンソールゲーム機で法線マップが本格的に使われだしたのは、PS3、Xbox360が登場してからになります。法線マップはプログラマブルシェーダーが登場してから生まれた技術のため、固定パイプラインには存在しません。そのためプログラマブルシェーダーを本格的に使用することが可能になったPS3、Xbox360の世代でコンソールゲーム機でも法線マップが活用されるようになりました。そして、現在はPS4、XboxOne、PC、スマートフォンなどでリリースされる多くの3Dゲームではあって当たり前の技術になっています。

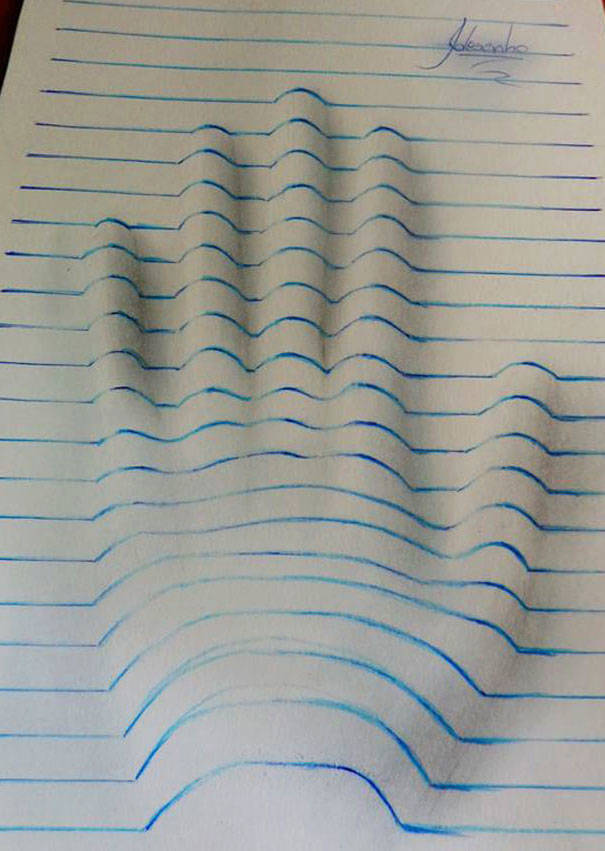
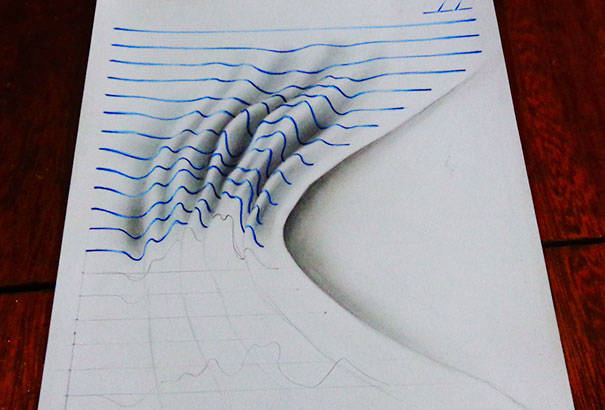
**16.3 細かいディテールの表現**

法線マップの目的は低ポリゴンで細かなディテールを表現することです。では、それを具体的に考えていきましょう。例えば道路のアスファルトを考えてみてください。アスファルトを近くでよく観察をしてみると細かな凹凸があることが分かります。もしこの凹凸をポリゴンで表現しようとすると非常に多くのポリゴンが必要になります。多すぎるポリゴンはGPUのパフォーマンスを低下させ、多くのメモリを使用してしまうことになります。このような細かいディテールの表現をポリゴンで行うことには多くの問題がありました。そこでこれらを高いパフォーマンスと少ないメモリ使用量で実現するために法線マップという技術が生まれました。

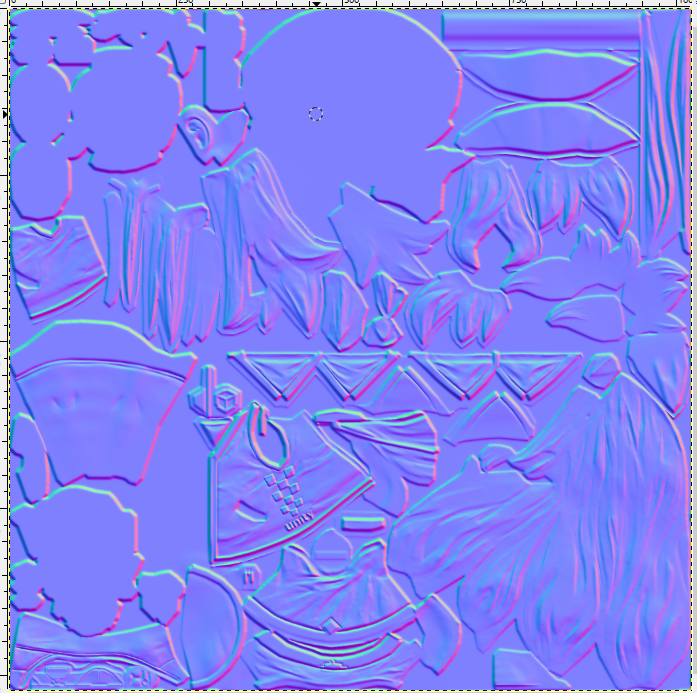
**16.4 法線をテクスチャに書き込む**

　引き続き先ほどのアスファルトの凹凸で考えていきましょう。私たちはどうやってアスファルトの凸凹を認識しているのでしょうか？ 実は凸凹を認識するために一番重要な情報は光の陰影になります。下記のだまし絵を見てみてください。

立体的に見えると思います。しかしこれらは全て平らなノートに書かれた絵であって実際には凹凸はありません。このように人間は実際に凹凸がなくても、陰影をつけることで実際に凹凸があるように錯覚します。では、ここでディフューズライトやスペキュラライトの計算式を思い出してください。ライトによって生成される陰影はライトの方向とモデルの頂点の法線によって決まっていました。つまり、ポリゴン数が少なくてもモデルの法線さえ詳細であれば、細かなディテールは表現できることになります。いわゆるだまし絵です。そこで、モデルの法線をテクスチャに書き込んでしまって、ライティングの時にはその法線を使用しようという考えが生まれます。これが法線マップです。下記はユニティちゃんの法線マップです。



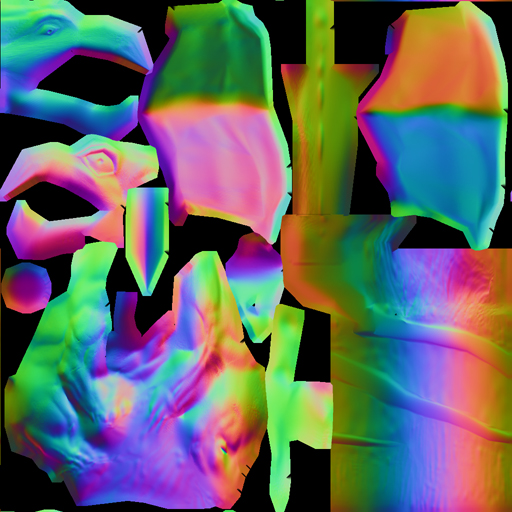
テクスチャのRGB成分に、法線のXYZ成分を書き込んでテクスチャにしたものです。

**16.5 オブジェクトスペース法線マップとタンジェントスペース法線マップ**

　法線マップには**オブジェクトスペース法線マップ**と**タンジェントスペース法線マップ**という二種類のデータ形式があります。ではこの二つについて見ていきましょう。

**16.5.1 オブジェクトスペース法線マップ**

　オブジェクトスペースはタンジェントスペースと比べるとイメージがしやすい法線マップです。この法線マップは単純にオブジェクトの法線のXYZをテクスチャのRGBに書き込んでいるだけのものです。オブジェクトスペース法線マップは下記のようなデータになります。



先ほどのユニティちゃんの法線マップとは少し違った感じのデータに見えるかと思います。実は先ほどのユニティちゃんの法線マップはタンジェントスペース法線マップです。では続いてタンジェントスペース法線マップについて見ていきましょう。

**16.5.2 タンジェントスペース法線マップ**

　タンジェントスペース法線マップは単純にモデルの法線を埋め込んだものではなく、**法線マップを貼り付けるポリゴンの法線座標系から見た法線**を書き込んでいることになります。ポリゴンの法線座標系というのはポリゴンの法線をZ軸、ポリゴンの法線と直交している接ベクトル(タンジェント)をX軸、法線と接ベクトルの外積で求めた従法線ベクトルをY軸とした空間のことです。



この画像ですと、Nが法線、Uが接ベクトル、Vが従法線ベクトルになります。この３軸を基底軸とした空間がタンジェントスペースと呼ばれるものです。

例えば、Nが(0.707, 0.0, 0.707)というベクトルだった場合、このNと同じ向きのベクトル(0.707, 0.0, 0.707)をタンジェントスペースに変換すると(0, 0, 1)になります。このように法線マップを貼り付けるポリゴンの法線と同じ向きのベクトルはタンジェントスペースに変換すると(0，0，1)に変換されます。これをテクスチャのRGBに変換して書き込むため、(0，０，255)という青いカラーが書き込まれることになります。多くのオブジェクトで法線マップに書き込む法線の方向は、貼り付けるポリゴンの法線から大きく変化することはありません。そのため、タンジェントスペース法線マップは青の成分が強く出る画像データになります。

**16.5.3 主流はタンジェントスペース法線マップ**

　現在主流となっているのはタンジェントスペース法線マップです。特に断りなく法線マップと言った場合はタンジェントスペースを指していると考えてください。

では、なぜタンジェントスペースが主流なのでしょうか。オブジェクトスペースの方が単純で分かりやすく感じたはずです。実はオブジェクトスペースには頂点の変形に対応できないという欠点があり、その欠点はタンジェントスペースであれば解決できるためです。

　今の3Dゲームではモーフィング、クロスシミュレーションなどといったオブジェクトの頂点を変形させる技術が使われています。その場合にオブジェクトスペース法線マップを使っていると下記のような問題が発生します。

オブジェクトスペース法線マップからフェッチした法線。このときは問題ないが・・・

オブジェクトが変形しているので、本来は点線の法線が正しいのだが、オブジェクトスペースの場合は実線の法線になっていまい、見た目がおかしくなる。

これがオブジェクトスペースの欠点です。一方タンジェントスペースはどうでしょうか？タンジェントスペースの法線はライティングの計算を行うときに、ワールドスペースに変換を行う必要があります。このワールドスペースへの変換を行う時に貼り付けるモデルの法線を使って変換を行うことになります。つまり、オブジェクトが変形した時にポリゴンの法線さえを正しく計算を行っておけば変形後の法線を求めることができます。

　オブジェクトスペース法線マップは非常にシンプルな考え方になるため、タンジェントスペース法線マップに比べるとGPUパフォーマンスの面では軍配があがります。しかし、頂点の変形に対応できないというデメリットを抱えることとなります。

**16.5.4 タンジェントスペースからワールドスペースへの変換**

　ライティングを行うためにはワールドスペースへ変換する必要があります。ではその変換の仕方について考えていきましょう。話を簡単にするために２次元で考えてみましょう。

例えば、基底軸ex(1,0)、ey(0,1)の座標系で考えてみましょう。

V(5,6)

V1(0,6) 

V0(5,0) 

Vは二次元上のベクトルです。このベクトルはX軸方向に5、Y軸方向に6の大きさを持っています。ベクトルVは下記のような計算が成り立ちます。

**V = V0 + V1**

また、V0とV1は下記の計算が成り立ちます。

**V0 = V.x \* ex**

**V1 = V.y \* ey**

exとeyはこの座標系の基底軸です。

この基底軸が例えばex(0.707, -0.707), ey(0.707, 0.707)の場合を考えましょう。

V(5,6)

V1 

V0

グラフは先ほどと同じですが、基底軸が違うことに注意してください。Vはワールドスペースでのベクトルではなく、基底軸ex(0.707, -0.707), ey(0.707, 0.707)の座標系でのベクトルです。これを基底軸ex(1, 0), ey(0,1)のワールド座標系に変換することを考えます。まず、VはV0とV1の合算で求まります。そこでワールド座標系でV0とV1がどのようなベクトルになるか求めてみましょう。V0とV1は下記の計算でも止まります。

**V0 = V.x \* ex**

**V1 = V.y \* ey**

つまり、V0 = 5×(0.707, -0.707)

= (3.535，－3.535)・・・・・・・・①

　　　　V1 = 6×(0.707, 0.707)

＝(3.535，3.535)　・・・・・・・・②

そしてVは下記の計算で求まります。

**V = V0 + V1**

よって、①と②から

　V = (3.535，－3.535) + (3.535，3.535)

= (7.07，0.0)

よって、(7.07，0.0)がVをワールド空間に変換したベクトルということになります。

では、タンジェントスペースの法線をワールドスペースに変換する話に戻しましょう。タンジェントスペースの基底軸はexが接ベクトル、eyが従法線、ezが法線です。各ベクト

を下記で表すこととします。

　接ベクトル

tangent

従法線

biNormal

法線

normal

法線マップからフェッチしたタンジェントスペースでの法線

　　localNormal

この時にワールドスペースでの法線worldNormalは下記の計算で求まります。

**worldNormal = localNormal.x × tangent + localNormal.y × binormal + localNormal ×normal**

**16.6 サンプルコード**

　では、この節ではサンプルコードを使って法線マップの実装の仕方を見ていきましょう。

サンプルコードはDireceXLesson/ShaderTutorial\_09です。

まず、main.cppのLoadTexture関数で法線マップのロードを行っています。

|  |
| --- |
| void LoadNormalMap()  {  if( FAILED( D3DXCreateTextureFromFileA( g\_pd3dDevice,  "normal.jpg",  &g\_pNormalMap ) ) )  　　{  std::abort();  }  if (FAILED(D3DXCreateTextureFromFileA(g\_pd3dDevice,  "diffuse.jpg",  &g\_pDiffuseMap)))  {  std::abort();  }  } |

main.cppのRender関数でロードした法線マップをGPUへ転送しています。

|  |
| --- |
| g\_pEffect->SetTexture("g\_normalMap", g\_pNormalMap); |

続いてシェーダーを見ていきましょうbasic.fxを開いてください。basic.fxに法線マップ用のテクスチャとテクスチャサンプラが追加されています。

|  |
| --- |
| texture g\_normalMap; //法線マップ  sampler g\_normalMapSampler =  sampler\_state  {  Texture = <g\_normalMap>;  MipFilter = LINEAR;  MinFilter = LINEAR;  MagFilter = LINEAR;  AddressU = Wrap;  AddressV = Wrap;  }; |

ピクセルシェーダーで法線マップからタンジェントスペース法線のフェッチとワールドスペース法線への変換を行っています。

|  |
| --- |
| //法線マップからタンジェントスペース法線をフェッチ。  float3 localNormal = tex2D( g\_normalMapSampler, In.uv );  //法線と接ベクトルの外積を求めて従法線を求める。  float3 biNormal = normalize( cross( In.tangentNormal, In.normal) );  //-1.0～1.0の範囲にマッピングする。  localNormal = (localNormal \* 2.0f)- 1.0f;  //ワールドスペースの法線を求める。  float3 worldNormal = In.tangentNormal \* localNormal.x  + biNormal \* localNormal.y  + In.normal \* localNormal.z; |

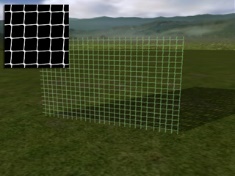
localNormalに2.0をかけてあとで-1.0を行っている箇所に注意してください。テクスチャの各要素8bitのRGBにベクトルの情報を書き込んでいるため、各成分に0-255の値しか書き込めません。そのため、負数を書き込むために0～127を負数、128～255を正数というようにマッピングしています。2.0をかけた後で1.0減算を行うことでlocalNormalの値を-1.0～1.0に変換しているのです。

**Chapter 17 いろいろなテクスチャの活用法**

　今までテクスチャといえばモデルの模様を描きこんだディフューズテクスチャのことを差していたと思います。しかし、Chapter16で学んだ法線マップはテクスチャにモデルの法線を書きこんでいます。Chapter12で学んだ投影シャドウはライトから見たスクリーンの絵をテクスチャに描きこんでいました。このようにテクスチャは単にモデルの模様だけを記録したものではなく、様々な活用のされ方があります。このチャプターではそのいくつかを簡単に紹介します。そして、最後にスペキュラマップのサンプルプログラムを解説します。

**17.1 透明度マップ**

テクスチャにモデルの透明度を書きこんだマップです。テクスチャのαを使って半透明レンダリングを行います。



**17.2 アンビエントオクルージョンマップ**

アンビエントオクルージョンとは環境光の遮蔽のことを言います。本来環境光は複雑な光の反射によって計算されます。この光の反射を計算するためにレイトレーシング、ラジオシティなど色々なアルゴリズムがありますが非常に時間のかかるアルゴリズムです。しかし、ゲームプログラムにおいて1フレームのレンダリングにかけることのできる時間は33ミリ秒～16ミリ秒です。50ミリ秒以上かかるようであれば、ユーザーにひどいストレスを与えることになるでしょう。そのためリアルタイムレンダリングでレイトレーシング、ラジオシティなどを活用することは現実的ではありません。そのため、古くから3Dゲームでは一律で光を当てるという簡略化したアンビエントライトが活用されていました。しかし一律に光をあてるアンビエントライトではリアリティにかけます。これを改善するために3dsMaxやMayaなどのツールで事前に環境光の遮蔽情報を計算してテクスチャに描きこんで、ゲームではテクスチャに描きこまれた遮蔽情報をもとにライティングを行うという手法が生まれました。この環境光の遮蔽情報が書きこまれたテクスチャをアンビエントオクルージョンマップと呼ばれます。



**17.3 スペキュラマップ**

では最後にサンプルプログラムを交えてスペキュラマップを学びましょう。Chapter10で金属のような質感を再現できるスペキュラライトを学びました。しかし今はモデル全体にスペキュラが発生しているため、例えばプレイヤーの衣服のベルトの金具の部分だけスペキュラを当てるということが困難だと思います。もしこれを実現しようとするとベルトの金具の部分だけ別モデルにするということが考えられます。では、髪の毛のツヤなどはどうでしょうか？これらもスペキュラライトを使えば実現できます。しかし単にスペキュラライトを当てただけでは髪の毛が金属のような質感になってしまいます。これらを実現するためにテクスチャにスペキュラライトの強さを書きこんでしまおうという考えが生まれます。これがスペキュラマップです。



これはskyrimというゲームの鎧のスペキュラマップです。色が白い箇所ほどスペキュラが強くなっていきます。

ではサンプルプログラムDirectXLesson/ SpecularMapを見てスペキュラマップの実装を見ていきましょう。

スペキュラマップもテクスチャなのでテクスチャをロードする必要があります。

Player.h

|  |
| --- |
| #pragma once  class Player  {  public:  Player();  ~Player();  void Init();  void Update();  void Render();  private:  SkinModel model;  SkinModelData modelData;  Animation animation;  D3DXQUATERNION rotation;  LPDIRECT3DTEXTURE9 normalMap = NULL; //法線マップ。  **LPDIRECT3DTEXTURE9 specularMap = NULL; //スペキュラマップ。**  }; |

PlayerクラスにspecularMapというメンバ変数を追加しました。

Player.cpp

|  |
| --- |
| void Player::Init()  {  　・  　　　・  　　　・  **//スペキュラマップをロード。**  **hr = D3DXCreateTextureFromFileA(g\_pd3dDevice,**  **"Assets/model/utc\_spec.tga",**  **&specularMap**  **);**  //D3DXCreateTextureFromFileAの戻り値をチェック。  if (FAILED(hr)) {  //D3DXCreateTextureFromFileAで失敗した。  MessageBox(  NULL,  "テクスチャのロードに失敗しました。指定したパスが正しいか確認をお願いします。",  "エラー",  MB\_OK  );  }  　　　　　・  　　　　　・  　　　　　・  **if (specularMap != NULL) {**  **//スペキュラマップの読み込みが成功したので、CSkinModelにスペきゅらマップを設定する。**  **model.SetSpecularMap(specularMap);**  **}**  　　　・  　　　　　・  　　　　　・  } |

Player::Init関数でスペキュラマップをロードして、CSkinModelに設定しています。

SkinModel.cppのDrawMeshContainer関数で設定されたスペキュラマップをGPUに転送しています。

|  |
| --- |
| if (specularMap != NULL) {  //スペキュラマップがあるので、シェーダーに転送する。  pEffect->SetTexture("g\_specularTexture", specularMap);  //スペキュラマップのあり、なしのフラグをtrueにする。  pEffect->SetBool("g\_isHasSpecularMap", true);  }  else {  //スペキュラマップのあり、なしのフラグをfalseにする。  pEffect->SetBool("g\_isHasSpecularMap", false);  } |

では続いてシェーダーを見ていきましょう。Assets/Shader/model.fxを開いて下さい。

|  |
| --- |
| texture g\_specularTexture;  sampler g\_specularMapSampler =  sampler\_state  {  Texture = <g\_specularTexture>;  MipFilter = LINEAR;  MinFilter = LINEAR;  MagFilter = LINEAR;  AddressU = Wrap;  AddressV = Wrap;  }; |

まず、スペキュラマップをシェーダーで使用するのでテクスチャにアクセスするためのg\_specularTextureとテクスチャのサンプリングの仕方を指定するg\_specularMapSamplerを追加しました。では、最後にスペキュラライトを計算しているピクセルシェーダーを見てみましょう。

|  |
| --- |
| float4 PSMain( VS\_OUTPUT In ) : COLOR  {  float4 color = tex2D(g\_diffuseTextureSampler, In.Tex0);  float3 normal = In.Normal;  if(g\_isHasNormalMap){  //法線マップがある。  float3 tangent = normalize(In.Tangent);  float3 binSpaceNormal = tex2D( g\_normalMapSampler, In.Tex0);  float4x4 tangentSpaceMatrix;  //法線とタンジェントから従法線を求める  float3 biNormal = normalize( cross( tangent, normal) );  //タンジェントスペースからワールドスペースに変換する行列を求める。  tangentSpaceMatrix[0] = float4( tangent, 0.0f);  tangentSpaceMatrix[1] = float4( biNormal, 0.0f);  tangentSpaceMatrix[2] = float4( normal, 0.0f);  tangentSpaceMatrix[3] = float4( 0.0f, 0.0f, 0.0f, 1.0f );  //-1.0～1.0の範囲にマッピングする。  binSpaceNormal = (binSpaceNormal \* 2.0f)- 1.0f;  //タンジェントスペースからワールドスペースの法線に変換する。  normal = tangentSpaceMatrix[0] \* binSpaceNormal.x + tangentSpaceMatrix[1] \* binSpaceNormal.y + tangentSpaceMatrix[2] \* binSpaceNormal.z;    }  float4 lig = DiffuseLight(normal);      if(g\_isHasSpecularMap){  //スペキュラマップがある  float3 spec = CalcSpecular(In.worldPos, normal);  **//スペキュラライトの強さにスペキュラマップからフェッチしてきた値を乗算する。**  **spec \*= tex2D(g\_specularMapSampler, In.Tex0).a;**  lig.xyz += spec;  }  color \*= lig;  return color;  } |

**Chapter 18 ポストエフェクトの基礎**

最近の3Dゲームではカメラのピンボケ現象をシミュレーションする被写界深度や、光が溢れ出るブルームといわれる現象を疑似的に再現しています。このチャプターではそれらの現象を表現するための手法のポストエフェクトの基礎を学んでいきましょう。

被写界深度



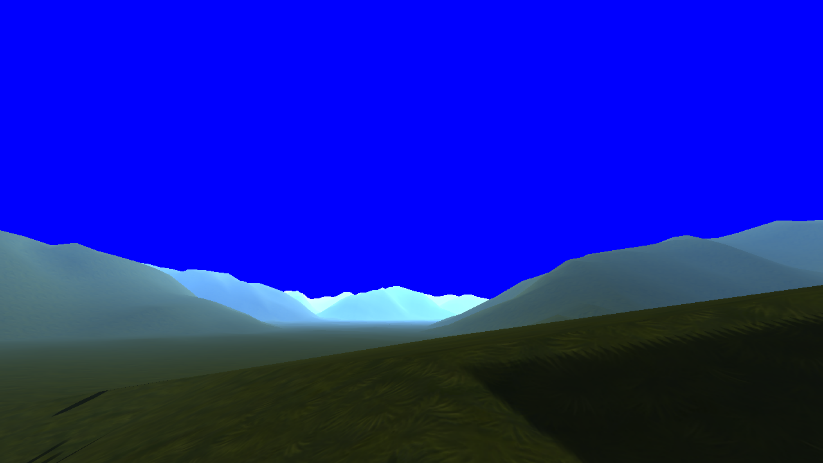
ブルーム



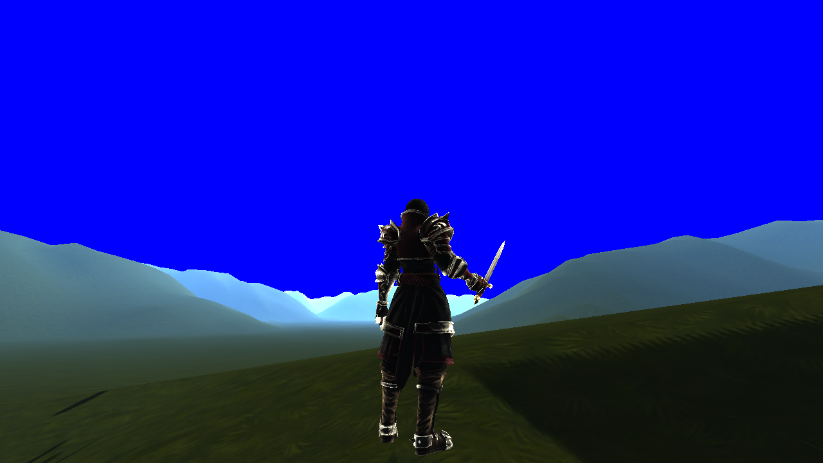
**18-1 ポストエフェクトとは**

ポストエフェクトとは一度レンダリングした画面に対して、レタッチを行ってエフェクトを追加していく処理のことを言います。画面にレンダリングを完了した後で、遅れて(ポスト)エフェクトを追加するのでポストエフェクトと呼ばれます。下記にポストエフェクトのフローの一例を記載します。

1. 地面を描画



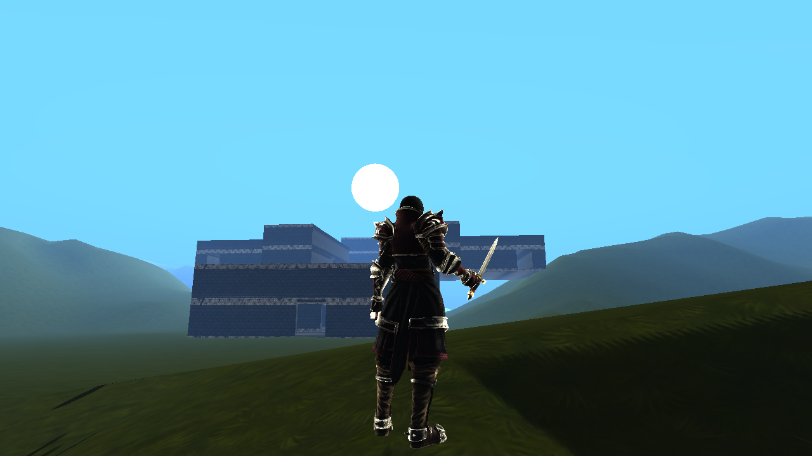
1. キャラクターを描画



1. 空を描画



1. 建物を描画

****

1. 敵を描画



1. **ポストエフェクト**

このように、3Dモデルをすべて画面にレンダリングした後でポストエフェクトを実行する流れになります。

**18-2 オフスクリーンレンダリング**

一度レンダリングした画面に対してエフェクトをかけるには、レンダリングした絵をテクスチャ化する必要があります。描画した絵をテクスチャとして使用する方法はチャプター11で学んだオフスクリーンレンダリングを使えば実現できます。つまり、今までキャラクターや背景などの3Dモデルは直接フレームバッファにレンダリングしてきましたが、直接フレームバッファにレンダリングをせずにオフスクリーンに対してレンダリングを行うのです。

　ではサンプルプログラムのDirectXLesson/PostEffect/Answerを見てみましょう。main.cppのInit関数でオフスクリーンレンダリングを行うためのレンダリングターゲットを作成しています。

**main.cpp**

|  |
| --- |
| //メインレンダリングターゲットを作成する。  mainRenderTarget = new CRenderTarget;  mainRenderTarget->Create(  FRAME\_BUFFER\_WITDH,  FRAME\_BUFFER\_HEIGHT,  1,  D3DFMT\_A8R8G8B8,  D3DFMT\_D24S8,  D3DMULTISAMPLE\_NONE,  0  ); |

そしてgame.cppのRender関数でレンダリングターゲットの切り替えを行っています。

**main.cpp**

|  |
| --- |
| //シーンの描画開始。  g\_pd3dDevice->BeginScene();  //あとで戻すためにフレームバッファのレンダリングターゲットとデプスステンシルバッファのバックアップを取る。  LPDIRECT3DSURFACE9 frameBufferRT;  LPDIRECT3DSURFACE9 frameBufferDS;  g\_pd3dDevice->GetRenderTarget(0, &frameBufferRT);  g\_pd3dDevice->GetDepthStencilSurface(&frameBufferDS);  // レンダリングターゲットをフレームバッファから変更する。  g\_pd3dDevice->SetRenderTarget(  0, //何番目のレンダリングターゲットを設定するかの引数。今回は0でいい。  mainRenderTarget->GetRenderTarget() //変更するレンダリングターゲット。  );  // デプスステンシルバッファも変更する。  g\_pd3dDevice->SetDepthStencilSurface(mainRenderTarget->GetDepthStencilBuffer());  // レンダリングターゲットをクリア。  g\_pd3dDevice->Clear(0, NULL, D3DCLEAR\_TARGET | D3DCLEAR\_ZBUFFER, D3DCOLOR\_XRGB(0, 0, 255), 1.0f, 0);  // ユニティちゃんなどをレンダリング。  game->Render();  // シーンの描画終了。  g\_pd3dDevice->EndScene(); |

**18-3 オフスクリーンにレンダリングした絵をフレームバッファに張り付ける**

前節でキャラクターや背景などの3Dモデルはオフスクリーンに対してレンダリングを行うようにしました。しかし当然ですが実際に画面に表示されるのはフレームバッファなので、このままでは画面に何も表示されません。そのため、オフスクリーンにレンダリングした絵をフレームバッファに張り付ける必要があります。フレームバッファに張り付けるには大きな四角形の板ポリにオフスクリーンレンダリングした絵をテクスチャとして張り付けてレンダリングを行えばよいのです。

**フレームバッファ**

****

では、サンプルコードを見てみましょう。main.cppのInitQuadPrimitiveで四角形の板ポリの初期化を行っています。

|  |
| --- |
| quadPrimitive = new CPrimitive;  //頂点の構造体。  struct SVertex {  float pos[4]; //頂点座標。  float uv[2]; //UV座標。  };  //頂点バッファ。  SVertex vertexBuffer[] = {  {  -1.0f, 1.0f, 0.0f, 1.0f,  0.0f, 0.0f  },  {  1.0f, 1.0f, 0.0f, 1.0f,  1.0f, 0.0f  },  {  -1.0f, -1.0f, 0.0f, 1.0f,  0.0f, 1.0f  },  {  1.0f, -1.0f, 0.0f, 1.0f,  1.0f, 1.0f  },  };  //頂点レイアウト。頂点バッファの構造がどのようになっているかをGPUに教えるためのもの。  static const D3DVERTEXELEMENT9 scShapeVertex\_PT\_Element[] = {  { 0, 0 , D3DDECLTYPE\_FLOAT4, D3DDECLMETHOD\_DEFAULT, D3DDECLUSAGE\_POSITION, 0 },  { 0, 16, D3DDECLTYPE\_FLOAT2, D3DDECLMETHOD\_DEFAULT, D3DDECLUSAGE\_TEXCOORD, 0 },  D3DDECL\_END()  };  //インデックスバッファ。  unsigned short indexBuffer[] = { 0, 1, 2, 3 };  quadPrimitive->Create(  CPrimitive::eTriangleStrip,  4,  sizeof(SVertex),  scShapeVertex\_PT\_Element,  vertexBuffer,  4,  D3DFMT\_INDEX16,  indexBuffer  ); |

main.cppのLoadShadersで板ポリを描画するときに使用するシェーダーのCopy.fxをロードしています。

**main.cpp**

|  |
| --- |
| LPD3DXBUFFER compileErrorBuffer = NULL;  HRESULT hr = D3DXCreateEffectFromFile(  g\_pd3dDevice,  "Assets/Shader/Copy.fx",  NULL,  NULL,  #ifdef \_DEBUG  D3DXSHADER\_DEBUG,  #else  D3DXSHADER\_SKIPVALIDATION,  #endif  NULL,  &copyEffect,  &compileErrorBuffer  );  if (FAILED(hr)) {  MessageBox(NULL, (char\*)(compileErrorBuffer->GetBufferPointer()), "error", MB\_OK);  std::abort();  } |

最後にmain.cppのCopyMainRTToCurrentRTで板ポリをフレームバッファに描画しています。

**main.cpp**

|  |
| --- |
| // Zテストで失敗してもらったら困るので、Zテストは無効にしておく。  g\_pd3dDevice->SetRenderState(D3DRS\_ZENABLE, FALSE);  // αブレンドもいらない。  g\_pd3dDevice->SetRenderState(D3DRS\_ALPHABLENDENABLE, FALSE);  // 頂点ストリーム0番に板ポリの頂点バッファを設定する。  g\_pd3dDevice->SetStreamSource(  0,  quadPrimitive->GetVertexBuffer()->GetBody(),  0,  quadPrimitive->GetVertexBuffer()->GetStride()  );  //18-3 コピーを行うシェーダーを使う場合はこちらを有効にする。  LPD3DXEFFECT shader = copyEffect;  //18-4 モノクロフィルターをかける場合はこちらを有効する。  //LPD3DXEFFECT shader = monochromeEffect;  //シェーダーを設定。  shader->SetTechnique("Default");  shader->Begin(NULL, D3DXFX\_DONOTSAVESHADERSTATE);  shader->BeginPass(0);  //シーンテクスチャを設定する  shader->SetTexture("g\_tex", mainRenderTarget->GetTexture());  //定数レジスタへの変更をコミットする。  shader->CommitChanges();  // インデックスバッファを設定。  g\_pd3dDevice->SetIndices(quadPrimitive->GetIndexBuffer()->GetBody());  // 頂点定義を設定する。  g\_pd3dDevice->SetVertexDeclaration(quadPrimitive->GetVertexDecl());  //　準備が整ったので描画。  g\_pd3dDevice->DrawIndexedPrimitive(  quadPrimitive->GetD3DPrimitiveType(), //プリミティブの種類を指定する。  0, //ベース頂点インデックス。0でいい。  0, //最小の頂点インデックス。0でいい。  quadPrimitive->GetNumVertex(), //頂点の数。  0, //開始インデックス。0でいい。  quadPrimitive->GetNumPolygon() //プリミティブの数。  );  shader->EndPass();  shader->End();  // 変更したレンダリングステートを元に戻す。  g\_pd3dDevice->SetRenderState(D3DRS\_ZENABLE, TRUE);  g\_pd3dDevice->SetRenderState(D3DRS\_ALPHABLENDENABLE, TRUE); |

**18-4 ポストエフェクトをかけて画面をモノクロにする。**

前節では板ポリをフレームバッファにレンダリングするときに、テクスチャカラーをそのまま出力するピクセルシェーダーを使いました。このピクセルシェーダーを改造してモノクロの色を出力するようにしましょう。そしてこれがポストエフェクトと言われるものです。ではサンプルコードを見てみましょう。main.cppのLoadShadersでMonochromeFilter.fxをロードしています。

**main.cpp**

|  |
| --- |
| //18-4 モノクロフィルターを行うシェーダーをロード。  hr = D3DXCreateEffectFromFile(  g\_pd3dDevice,  "Assets/Shader/MonochromeFilter.fx",  NULL,  NULL,  #ifdef \_DEBUG  D3DXSHADER\_DEBUG,  #else  D3DXSHADER\_SKIPVALIDATION,  #endif  NULL,  &monochromeEffect,  &compileErrorBuffer  );  if (FAILED(hr)) {  MessageBox(NULL, (char\*)(compileErrorBuffer->GetBufferPointer()), "error", MB\_OK);  std::abort();  } |

続いて、main.cppのCopymainToCurrentRTで使用されているシェーダーをモノクロフィルター用のシェーダーに変更します。

main.cpp

|  |
| --- |
| //LPD3DXEFFECT shader = copyEffect;　　　//18-3 コピーを行うシェーダーを使う場合はこちら。  LPD3DXEFFECT shader = monochromeEffect;　//18-4 モノクロフィルターをかける場合はこちらを使用する。 |

最後にモノクロフィルターをかけるシェーダーを見てみましょう。Assets/Shader/ MonochromeFilter.fxを開いてください。ほとんどのコードがCopy.fxと同じですが、ピクセルシェーダーだけ異なっています。

**MonochromeFilter.fx**

|  |
| --- |
| float4 PSMain( VS\_OUTPUT In ) : COLOR0  {  float4 color = tex2D( TextureSampler, In.uv );  float Y = 0.29900 \* color.r + 0.58700 \* color.b + 0.11400 \* color.b;  return Y;  } |

このモノクロにするための計算はwebで公開されているものをコピーしてきたもので、重要な意味はありません。重要なのはこの処理で一度レンダリングしたシーンテクスチャに対してレタッチを行えたことです。そしてこれがポストエフェクトと呼ばれるものです。

18-5 実習課題

　サンプルプログラムを改造してシーンにセピア調のフィルターをかけられるようにしなさい。セピア調への変換は下記のコードで行える。なおcolorにはシーンテクスチャーのカラーが設定されているものとする。

float Cb = -0.2f;

float Cr = 0.1f;

　float Y = 0.299f \* color.r + 0.587f \* color.g + 0.114f \* color.b;

　color.r = Y + 1.402f \* Cr;

color.g = Y – 0.34414f \* Cb – 0.71414f \* Cr;

color.b = Y + 1.772f \* Cb;

**Chapter 19 ブルーム**

このチャプターでは光が溢れる現象のブルームについて見ていきましょう。

**19.1 HDR**

オブジェクトに対して強い光が当たる、もしくはオブジェクト自身が強い光を放っている場合、その光は下の図のように溢れ出して周囲に影響を与えます。



このような現象がブルームと呼ばれます。では強い光とは一体どの程度のもののことをいうのでしょうか？私たちはここまでいくつかのライティングの計算を学んできました。ディフューズライト、スペキュラライト、アンビエントライトなど。これらのライトを強く当てることで光は強くあたります。明るさはNitsという単位で表すことができます。例えば室内の床の明るさは0.08Nits、廊下の壁の明るさは77Nits、蛍光灯の光は6,000Nits、コンクリートの照り返しは10,000Nits、直射日光を反射した車のボディの輝きは300,000Nitsです。このように現実世界に存在する光の強さのレンジは非常に大きいです。しかし、これまではどんなに強い光を当てても、フレームバッファに書き込むことができる光の範囲は0～255までです。これでは表現できる光の強さに限界があります。仮に200以上を強い光として判断する場合、世界の多くのオブジェクトが光り輝くことになってしまいます。そのためブルームを表現するためには、もっと表現できる光の強さの範囲を広くする必要があります。これがHDR(high dynamic range)と呼ばれるものです。R8G8B8A8の32bitで表現できる範囲はLDR(low dynamic range)と呼ばれます。

19.2 浮動小数点バッファ

　ブルームはHDRを活用したポストエフェクトになります。そのため、フレームバッファに対してR8G8B8A8の32bitよりも広いレンジの光を書き込めるようにする必要があります。DirectXはR16G16B16A16やR32G32B32A32の浮動小数点バッファを使うことができます。今回は16bitの浮動小数点バッファを使用しましょう。まず、前節で作成したメインレンダリングターゲットのフォーマットを下記のように浮動小数点バッファに変更します。PostEffectTutorial\_01/main.cppを見てください。

main.cpp

|  |
| --- |
| void InitMainRenderTarget()  {  mainRenderTarget = new CRenderTarget;  mainRenderTarget->Create(  FRAME\_BUFFER\_WITDH,  FRAME\_BUFFER\_HEIGHT,  1,  **//レンダリングターゲットのフォーマット。16bitの浮動小数点バッファを指定する。**  **D3DFMT\_A16B16G16R16F,**  D3DFMT\_D24S8,  D3DMULTISAMPLE\_NONE,  0  );  } |

これでメインレンダリングターゲットのテクスチャは16bitの浮動小数点バッファになります。これで広いレンジの光を書き込めるようになりました。

**19.2 アルゴリズムの流れ**

　ブルームを行うためのアルゴリズムは下記のような流れになります。

1. シーンをレンダリング



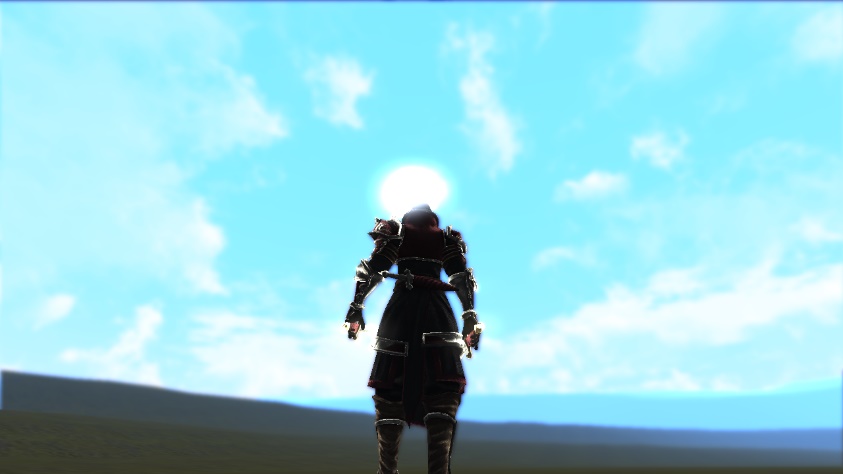
1. 輝度をテクスチャとして抽出



1. 抽出したテクスチャにブラーをかけてぼかす



1. ブラーをかけたテクスチャをシーンに加算合成



**19.3 輝度の抽出**

　まず最初のステップとして輝度を抽出する必要があります。これはレンダリングしたシーンをテクスチャとして、輝度だけ抽出してテクスチャを生成することで抽出することができます。ではPostEffectTutorial\_01/bloom.cppを見てみましょう。

　まず、輝度を抽出するためのレンダリングターゲットを作成します。

bloom.cpp

|  |
| --- |
| //輝度抽出用のレンダリングターゲットを作成する。  luminanceRenderTarget.Create(  FRAME\_BUFFER\_WITDH,  FRAME\_BUFFER\_HEIGHT,  1,  D3DFMT\_A16B16G16R16F,  D3DFMT\_D16,  D3DMULTISAMPLE\_NONE,  0  ); |

そして輝度を抽出するために、輝度抽出のためのシェーダーをセットして板ポリにシーンのテクスチャを貼り付けて描画を行います。

bloom.cpp

|  |
| --- |
| // αブレンドもいらない。  g\_pd3dDevice->SetRenderState(D3DRS\_ALPHABLENDENABLE, FALSE);  //輝度抽出用のレンダリングターゲットに変更する。  g\_pd3dDevice->SetRenderTarget(0, luminanceRenderTarget.GetRenderTarget());  g\_pd3dDevice->SetDepthStencilSurface(luminanceRenderTarget.GetDepthStencilBuffer());  //黒でクリア。  g\_pd3dDevice->Clear(0, NULL, D3DCLEAR\_TARGET | D3DCLEAR\_ZBUFFER, D3DCOLOR\_XRGB(0, 0, 0), 1.0f, 0);  // 輝度抽出テクニックをセット。  effect->SetTechnique("SamplingLuminance");  effect->Begin(NULL, D3DXFX\_DONOTSAVESHADERSTATE);  effect->BeginPass(0);  //シーンテクスチャを設定する。  effect->SetTexture("g\_scene", mainRenderTarget->GetTexture());  //定数レジスタへの変更をコミット。  effect->CommitChanges();  DrawQuadPrimitive();  effect->EndPass();  effect->End();  // 変更したレンダリングステートを元に戻す。  g\_pd3dDevice->SetRenderState(D3DRS\_ALPHABLENDENABLE, TRUE); |

では、輝度抽出のシェーダーを見てみましょう。PostEffectTutorial\_01/Assets/Shader/ bloom.fxを見てください。

bloom.fx

|  |
| --- |
| VS\_OUTPUT VSSamplingLuminance( VS\_INPUT In )  {  VS\_OUTPUT Out;  Out.pos = In.pos; //トランスフォーム済み頂点なのでそのまま  Out.tex = (In.pos \* 0.5f) + 0.5f;  Out.tex.y = 1.0f - Out.tex.y;  return Out;  } |

頂点シェーダーは特別何もしていません。頂点データにUV座標がないので、頂点座標からUVを計算しているくらいでしょうか。

|  |
| --- |
| float4 PSSamplingLuminance( VS\_OUTPUT In ) : COLOR  {  float4 color = tex2D(g\_SceneSampler, In.tex );  //輝度を計算する。tに輝度が入っている。  float t = dot( color.xyz, float3(0.2125f, 0.7154f, 0.0721f) );  　//輝度が1.0以下ならピクセルキル  clip(t - 1.001f);  color.xyz \*= (t - 1.0f);  return color;  } |

続いてピクセルシェーダーです。シーンテクスチャからフェッチしたカラーとfloat3(0.2125f, 0.7154f, 0.0721f)との内積を計算して輝度を求めています。シーンテクスチャは16bitの浮動小数点バッファになっているため、1.0より大きな明るさを格納しています。そのため、光が強いほど輝度の値は大きくなります。

clip関数は引数の値が負数になった場合、ピクセルを破棄します。つまり輝度が1.0以下の場合は負数になるため、ピクセルキルが行われ以降の処理が破棄されます。輝度が1.0以上の場合はシーンのカラーを書き込んでいます。

**19.4 ブラー**

輝度をテクスチャに抽出ができたら、そのテクスチャに対してブラーをかけます。ブラーは、ぼかしたいテクスチャを板ポリに貼り付けて別のレンダリングターゲットに描画するときのピクセルシェーダーで行います(モノクロフィルターとかと同じです)。

　最もシンプルなブラーは複数のテクセルをサンプリングしてその平均を計算することです。次のコードは近傍4テクセルをサンプリングして、その平均を出力するピクセルシェーダーです。

|  |
| --- |
| float4 PSMain(VS\_OUTPUT In) : COLOR0  {  //近傍4テクセルをサンプリングする。  　float2 baseUV = tex2D(luminanceSampler, In.uv);  tex += tex2D(luminanceSampler, baseUV + float2(1.0f/g\_texSize.x, 0.0f ) ); //右のテクセルをサンプリング。  tex += tex2D(luminanceSampler, baseUV + float2(-1.0f/g\_texSize.x, 0.0f ) ); //左のテクセルをサンプリング。  tex += tex2D(luminanceSampler, baseUV + float2( 0.0f, 1.0f/g\_texSize.y) ); //下のテクセルをサンプリング。  tex += tex2D(luminanceSampler, baseUV + float2( 0.0f, -1.0f/g\_texSize.y)); //上のテクセルをサンプリング。  //平均を取る  tex /= 4.0f;  return tex;  } |

g\_texSizeはテクスチャのサイズを格納した、float2型の定数レジスタです。1.0をテクスチャサイズで割ってやることで、1テクセル分UVがずれることになります。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 例えば解像度が4×4のテクスチャの場合、１テクセル移動させるUV座標は下記の図が示すように、1.0を4等分しているので1.0÷4=0.25となる。つまり解像度が512×512であれば１テクセル移動させるUV座標は1.0÷512となる。  0.0 0.25 　0.5 0.75 1.0   |  |  |  |  | | --- | --- | --- | --- | |  |  |  |  | |  |  |  |  | |  |  |  |  | |  |  |  |  | |

**19.4.1 ガウシアンフィルター**

　前節のブラーでは広い範囲の綺麗なブラーを出すのが困難です。そこでブルームエフェクトのブラー処理では、比較的高速で綺麗なブラーをかけることができるガウシアンフィルターが使われることが多いです。ガウシアンフィルターは重み付きでテクセルをブレンディングしていく手法です。例えばテクセル0は0.7の重み、テクセル1は0.2の重み、テクセル2は0.1の重みの場合、下記の計算で最終カラーが計算されます。

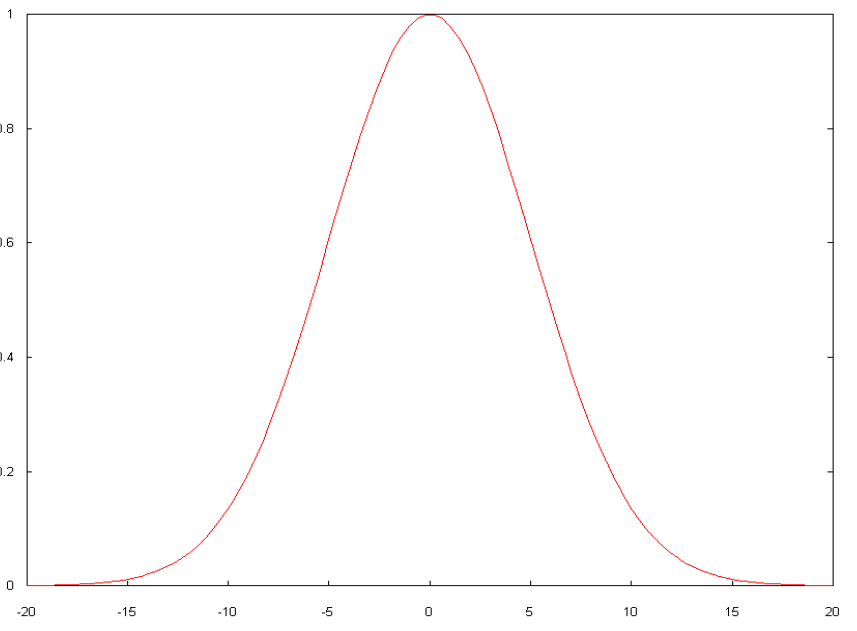
***最終カラー = テクセル0×0.7 + テクセル1×0.2 + テクセル2×0.1***

このテクセルの重みの計算にガウス関数を使用するためガウシアンフィルターと呼ばれます。

　ガウス関数は下記の形をしています。

xは基準テクセルからの距離、Nは規格化定数(あとで説明)、σは分散具合を決めるため

の変数です。この関数を使用すると下記のようなグラフができます。



横軸がxで、縦軸がガウス関数の結果です。今回はガウスフィルターではガウス関数の結果

を重みとして使用します。xは基準テクセルからの距離を表しています。Xの値が大きくな

るということは、基準テクセルから離れていっているということになるので、重みが弱くな

っていっています。σの値を変えると分散具合が変わります。σの値が大きいほどグラフは

緩やかになり、小さくなるほど鋭利になっていきます。

　では、実際にガウスフィルターのコードを見ていきましょう。今回は基準テクセルから16

テクセル離れたサンプリングしてブラーをかけています。

まずBloomクラスに重みを格納するためのメンバ変数を追加します。

bloom.h

|  |
| --- |
| static const int NUM\_WEIGHTS = 8; //!<ガウシアンフィルターの重み。  float weights[NUM\_WEIGHTS]; //!<ガウシアンフィルターで使う重みテーブル。 |

16テクセルサンプリングするのに、重みのテーブルのサイズが半分の8なことに注意して

ください。ガウス関数のグラフを見ればわかりますが、距離が正または負であってもガウス

関数の結果は変わりません。そのため、16テクセルの半分の8個の重みだけ計算すればい

いのです。

　重みの計算はBloom::UpdateWeight関数です。

bloom.cpp

|  |
| --- |
| void Bloom::UpdateWeight(float dispersion)  {  float total = 0;  　 //exp(-x^2/2σ^2)を計算している箇所。  //引数のdispersionが分散。この値を大きくするとボケが強くなる。  for (int i = 0; i<NUM\_WEIGHTS; i++) {  weights[i] = expf(-0.5f\*(float)(i\*i) / dispersion);  total += 2.0f\*weights[i];  }  // 規格化  　// ここで使用している規格化定数のtotalは重みの合計。  // 重みを重みの合計で除算しているので、除算後の重みの合計は1.0になる。  for (int i = 0; i < NUM\_WEIGHTS; i++) {  weights[i] /= total;  }  } |

　重みの計算が完了したら、次は描画になります。ガウシアンフィルターは横ブラー、縦ブラーという流れで行います。Bloom::Render関数を見ていきましょう。

|  |
| --- |
| //輝度を抽出したテクスチャをXブラー  {  　//ダウンサンプリングを行う(後述)。  g\_pd3dDevice->SetRenderTarget(0, downSamplingRenderTarget[0].GetRenderTarget());  effect->SetTechnique("XBlur");  effect->Begin(NULL, D3DXFX\_DONOTSAVESHADERSTATE);  effect->BeginPass(0);  float size[2] = {  (float)(luminanceRenderTarget.GetWidth()),  (float)(luminanceRenderTarget.GetHeight())  };  float offset[] = {  16.0f / (float)(luminanceRenderTarget.GetWidth()),  0.0f  };  effect->SetValue( "g\_luminanceTexSize", size, sizeof(size));  effect->SetValue( "g\_offset", offset, sizeof(size));  effect->SetValue( "g\_weight", weights, sizeof(weights));  effect->SetTexture( "g\_blur", luminanceRenderTarget.GetTexture());  effect->CommitChanges();  DrawQuadPrimitive();  effect->EndPass();  effect->End();  }  //輝度を抽出したテクスチャをYブラー  {  //ダウンサンプリングを行う(後述)。  g\_pd3dDevice->SetRenderTarget(0, downSamplingRenderTarget[1].GetRenderTarget());  effect->SetTechnique("YBlur");  effect->Begin(NULL, D3DXFX\_DONOTSAVESHADERSTATE);  effect->BeginPass(0);  float size[2] = {  (float)(downSamplingRenderTarget[0].GetWidth()),  (float)(downSamplingRenderTarget[0].GetHeight())  };  float offset[] = {  0.0f ,  16.0f / (float)(downSamplingRenderTarget[0].GetHeight())  };  effect->SetValue("g\_luminanceTexSize", size, sizeof(size));  effect->SetValue("g\_offset", offset, sizeof(size));  effect->SetValue("g\_weight", weights, sizeof(weights));  effect->SetTexture("g\_blur", downSamplingRenderTarget[0].GetTexture());  effect->CommitChanges();  DrawQuadPrimitive();  effect->EndPass();  effect->End();  } |

ガウシアンフィルターは一気に縦横ブラーをかけるのではなく、縦ブラーをかけたあとで、横ブラー(逆も可)をかけることで綺麗にブラーがかかります。

では、頂点シェーダーを見ていきましょう。

bloom.fx

|  |
| --- |
| /\*!  \* @brief Xブラー頂点シェーダー。  \*/  VS\_BlurOutput VSXBlur(VS\_INPUT In)  {  VS\_BlurOutput Out;  Out.pos = In.pos;  float2 tex = (In.pos \* 0.5f) + 0.5f;  tex.y = 1.0f - tex.y;  tex += float2( 0.5/g\_luminanceTexSize.x, 0.5/g\_luminanceTexSize.y);  Out.tex0 = tex + float2( - 1.0f/g\_luminanceTexSize.x, 0.0f );  Out.tex1 = tex + float2( - 3.0f/g\_luminanceTexSize.x, 0.0f );  Out.tex2 = tex + float2( - 5.0f/g\_luminanceTexSize.x, 0.0f );  Out.tex3 = tex + float2( - 7.0f/g\_luminanceTexSize.x, 0.0f );  Out.tex4 = tex + float2( - 9.0f/g\_luminanceTexSize.x, 0.0f );  Out.tex5 = tex + float2( -11.0f/g\_luminanceTexSize.x, 0.0f );  Out.tex6 = tex + float2( -13.0f/g\_luminanceTexSize.x, 0.0f );  Out.tex7 = tex + float2( -15.0f/g\_luminanceTexSize.x, 0.0f );  return Out;  } |

　16テクセルサンプリングを行うので、頂点シェーダーでは基準テクセルから8テクセル分のUV座標を計算しています。残りの8テクセルは基準テクセルを挟んで正反対の位置にあるため、計算は行っていません。

　では、続いてピクセルシェーダーです。

bom.fx

|  |
| --- |
| /\*!  \* @brief Xブラーピクセルシェーダー。  \*/  float4 PSXBlur( VS\_BlurOutput In ) : COLOR  {  float4 Color;  Color = g\_weight[0] \* (tex2D( g\_blurSampler, In.tex0 )  + tex2D( g\_blurSampler, In.tex7 + g\_offset ));  Color += g\_weight[1] \* (tex2D( g\_blurSampler, In.tex1 )  + tex2D( g\_blurSampler, In.tex6 + g\_offset ));  Color += g\_weight[2] \* (tex2D( g\_blurSampler, In.tex2 )  + tex2D( g\_blurSampler, In.tex5 + g\_offset ));  Color += g\_weight[3] \* (tex2D( g\_blurSampler, In.tex3 )  + tex2D( g\_blurSampler, In.tex4 + g\_offset ));  Color += g\_weight[4] \* (tex2D( g\_blurSampler, In.tex4 )  + tex2D( g\_blurSampler, In.tex3 + g\_offset ));  Color += g\_weight[5] \* (tex2D( g\_blurSampler, In.tex5 )  + tex2D( g\_blurSampler, In.tex2 + g\_offset ));  Color += g\_weight[6] \* (tex2D( g\_blurSampler, In.tex6 )  + tex2D( g\_blurSampler, In.tex1 + g\_offset ));  Color += g\_weight[7] \* (tex2D( g\_blurSampler, In.tex7 )  + tex2D( g\_blurSampler, In.tex0 + g\_offset ));  return Color;  } |

16テクセルサンプリングを行い、シェーダー定数のg\_weightを使用してブレンディングを

行っています。このg\_weightはbloom.cppでガウス関数を使用して求めた重みです。

今回はXブラーのみを記載しましたが、Yブラーはサンプリングするテクセルが上下の方

向になるだけのため、Xブラーを改造すれば簡単に記述できますので、各自で確認してみて

ください。

**19.5 ダウンサンプリング**

ガウスフィルターのサンプルコードにコメントでダウンサンプリングというキーワード

が出てきました。ダウンサンプリングとは解像度の高いテクスチャを解像度の低いレンダ

リングターゲットに対して縮小してレンダリングすることを言います。例えば1024×1024

の解像度のテクスチャを256×256のレンダリングターゲットに対してレンダリングする

場合などです。

1280×720のテクスチャ



**板ポリに貼り付けて描画！**

640×360のレンダリングターゲット

　この描画を行うときにバイリニアフィルタといわれるテクスチャフィルターを設定すると、テクスチャがボケて貼り付けられることになります。

上の画像の例のように縦横1/2のレンダリングターゲットに描画すると、画素の総数がちょうど1/4になります(640×360 = 230400、1280×720=921600)。つまり640×360のレンダリングターゲットの1ピクセルは4テクセルに相当することになります。しかし当然1ピクセルに表現できる画素は1画素です。そのため何もしなければ、3テクセル分の画素は失われることになってしまいます。しかし、この時にバイリニアフィルタを使用すると1ピクセルの画素は、４テクセルの平均のカラーにすることができます。19.4で近傍4テクセルのブラーが行われるとイメージしてください。バイリニアフィルタはHLSLのテクスチャサンプラのMinFiterにLinearを設定するだけで行うことができます。

|  |
| --- |
| texture g\_scene; //シーンテクスチャ。  sampler g\_SceneSampler =  sampler\_state  {  Texture = <g\_scene>;  MipFilter = LINEAR; //ミップマップテクスチャのためのフィルター。  **MinFilter = LINEAR; //テクスチャを縮小する時のフィルター。**  MagFilter = LINEAR; //テクスチャを拡大する時のフィルター。  }; |

このサンプルコードの黒字になっている箇所がテクスチャを縮小する時のテクスチャフィルターの設定です。ではガウスフィルターを行ったときのレンダリングターゲットを見てみましょう。まずはコンストラクタで行われているレンダリングターゲットの作成です。

bloom.cpp

|  |
| --- |
| //ブラーをかけるためのダウンサンプリング用のレンダリングターゲットを作成。  //横ブラー用。  downSamplingRenderTarget[0].Create(  **FRAME\_BUFFER\_WITDH / 2, //横の解像度をフレームバッファの半分にする。**  FRAME\_BUFFER\_HEIGHT,  1,  D3DFMT\_A16B16G16R16F,  D3DFMT\_D16,  D3DMULTISAMPLE\_NONE,  0  );  //縦ブラー用。  downSamplingRenderTarget[1].Create(  **FRAME\_BUFFER\_WITDH / 2, //縦と横の解像度をフレームバッファの半分にする。**  **FRAME\_BUFFER\_HEIGHT / 2,**  1,  D3DFMT\_A16B16G16R16F,  D3DFMT\_D16,  D3DMULTISAMPLE\_NONE,  0  ); |

　横ブラーのレンダリングターゲットは横の解像度がフレームバッファの幅の1/2になっています。縦ブラーのレンダリングターゲットは縦と横の解像度がフレームバッファの半分になっています。縦ブラーは横ブラーをかけたテクスチャに対して実行されるので横も1/2になっていることに注意してください。では最後に縦横ブラーをかけていく様子を画像で確認してみましょう。

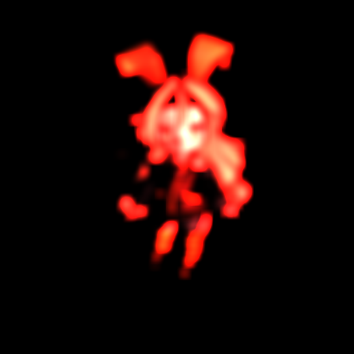
元画像



横ブラー

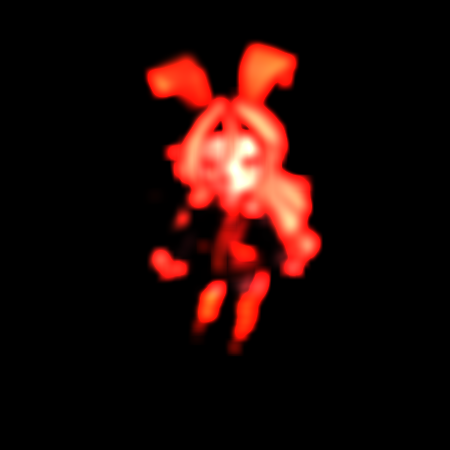


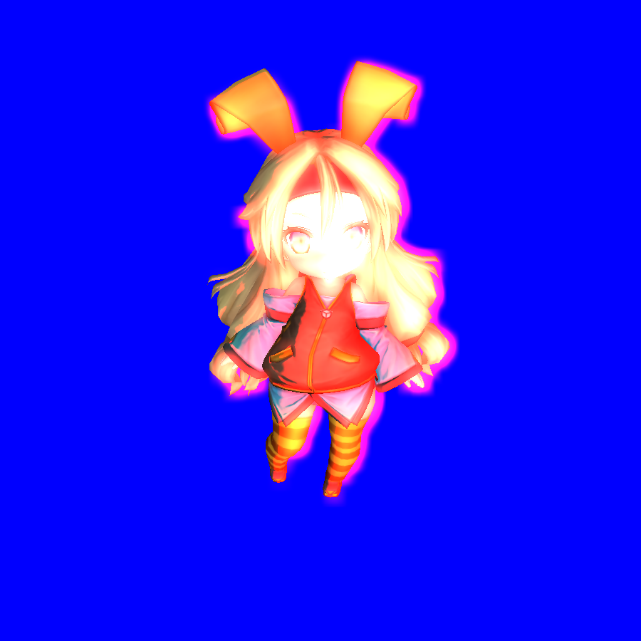
縦ブラー



**19.6 シーンテクスチャとぼかした画像を加算合成**

いよいよ、最終ステップです。シーンテクスチャと19.5で作成した画像を加算合成することでブルームエフェクトが完成します。



では加算合成を行うコードを見ていきましょう。bloom.cppのRender関数を見てください。

**bloom.cpp**

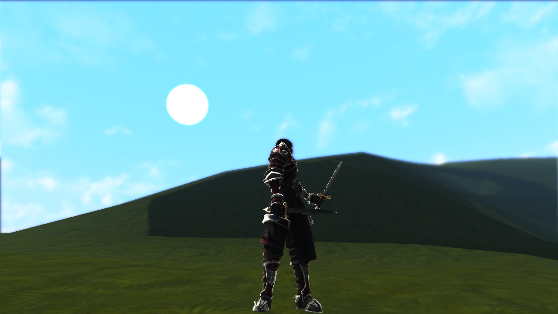
|  |
| --- |
| //最終合成。  float offset[] = {  0.5f / downSamplingRenderTarget[1].GetWidth() ,  0.5f / downSamplingRenderTarget[1].GetHeight()  };  **//レンダリングターゲットをメインレンダリングターゲットに戻す。**  **g\_pd3dDevice->SetRenderTarget(0, mainRenderTarget->GetRenderTarget());**  **g\_pd3dDevice->SetDepthStencilSurface(mainRenderTarget->GetDepthStencilBuffer());**  **//加算合成のレンダリングステートを設定する**  **g\_pd3dDevice->SetRenderState(D3DRS\_ALPHABLENDENABLE, TRUE);**  **g\_pd3dDevice->SetRenderState(D3DRS\_SRCBLEND, D3DBLEND\_ONE);**  **g\_pd3dDevice->SetRenderState(D3DRS\_DESTBLEND, D3DBLEND\_ONE);**  effect->SetTechnique( "Final");  effect->Begin(NULL, D3DXFX\_DONOTSAVESHADERSTATE);  effect->BeginPass(0);  effect->SetTexture("g\_blur", downSamplingRenderTarget[1].GetTexture());  effect->SetValue( "g\_offset", offset, sizeof(offset));  effect->CommitChanges();  DrawQuadPrimitive();  effect->EndPass();  effect->End();  g\_pd3dDevice->SetRenderState(D3DRS\_ALPHABLENDENABLE, FALSE);  g\_pd3dDevice->SetRenderState(D3DRS\_SRCBLEND, D3DBLEND\_SRCALPHA);  g\_pd3dDevice->SetRenderState(D3DRS\_DESTBLEND, D3DBLEND\_INVSRCALPHA); |

**19.7 川瀬式ブルームフィルタ**

さらにブルームを綺麗に見せるために現シリコンスタジオ所属の川瀬正樹氏が2002年にGDC(Game Developers Conference)で発表した川瀬式ブルームフィルタを紹介します。このアルゴリズムは高速で品質の高いブルームを表現できるため多くのゲームで採用されたものとなります。

川瀬式のブルームフィルタは512×512、256×256、128×128、64×64、32×32というように複数のダウンサンプリング用のレンダリングターゲットを用意して、ガウスフィルターをかけて縮小していきます。そして、それらを同解像度に拡大合成して、あふれテクスチャを生成する手法です。次のページには処理の流れを記載します。

①シーンをレンダリング



②輝度抽出(1280×720)



③ダウンサンプリング(640×360)



④ダウンサンプリング(320×180)



⑤ダウンサンプリング(160×90)



⑥ダウンサンプリング(80×45)



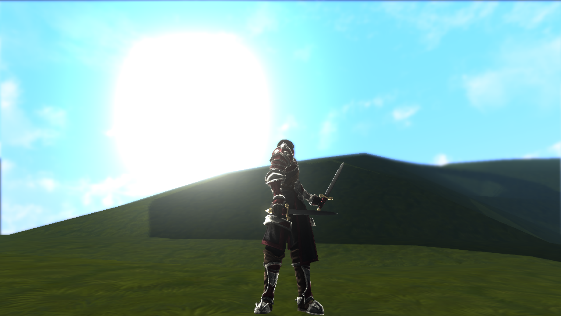
⑦ダウンサンプリング(40×22)



⑧5枚のテクスチャの平均を取って合成



⑨シーンに加算合成



1. (\*) ラッパークラスとは、既存のクラスのインスタンスを保持して、そのクラスを使いやすくなるようにインターフェースを定義したり、使用できる機能を制限したりすることが目的のクラスのことです。 [↑](#footnote-ref-1)