**Chapter 1 DirectX11**

**1.1 シェーダーモデル5**

DirectX9のシェーダーステージは頂点シェーダーとピクセルシェーダーのみでした。しかし最新のシェーダーモデル5ではジオメトリシェーダー、ハルシェーダー、ドメインシェーダー、コンピュートシェーダーの４つのシェーダーステージが追加されています。

**1.2 Windows8SDKへの統合とD3DXの廃止**

DirectXはWindows8からWindows8SDKへ統合され、DirectXSDKは廃止されました。また、この統合の際にD3DXは廃止され、D3DXから始まるAPIはすべてなくなっています。(つまりD3DXVECTOR3、D3DXMATRIX、D3DXVec3Lengthなどの関数はなくなっています。)

**Chapter1 簡単なプリミティブ描画**

ではTutorial01を使って、簡単なプリミティブ描画の仕方を勉強していきましょう。

Tutorial01/Tutorial.cpp(22行目～40行目)

|  |
| --- |
| bool Start() override  {  //頂点シェーダーをロード。  m\_vsShader.Load("Assets/shader/Tutorial.fx", "VS", CShader::EnType::VS);  //ピクセルシェーダーをロード。  m\_psShader.Load("Assets/shader/Tutorial.fx", "PS", CShader::EnType::PS);  //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),  };  //VRAM上に頂点バッファを作成。  m\_vertexBuffer.Create(3, sizeof(SSimpleVertex), vertices);  //VRAM上に定数バッファを作る。  m\_constantBuffer.Create(NULL, sizeof(SConstantBuffer));  return true;  } |

CTriangleDraw::Start関数でプリミティブを描画するための初期化処理を記述しています。まず、頂点シェーダーとピクセルシェーダーをロードしています。続いて、プリミティブをレンダリングするための頂点バッファの作成。そして最後に定数バッファを作成しています。定数バッファはDirectX9の時のシェーダー定数とほとんど同じものです。

　では、続いて実際に描画を行っている箇所を見てみましょう。

Tutorial01/Tutorial.cpp(45行目～69行目)

|  |
| --- |
| void Render(CRenderContext& renderContext) override  {  //頂点バッファを設定。  renderContext.IASetVertexBuffer(m\_vertexBuffer);  //プリミティブのトポロジーを設定。  renderContext.IASetPrimitiveTopology(D3D11\_PRIMITIVE\_TOPOLOGY\_TRIANGLELIST);  //頂点シェーダーを設定。  renderContext.VSSetShader(m\_vsShader);  //ピクセルシェーダーを設定。  renderContext.PSSetShader(m\_psShader);  //入力レイアウトを設定。  renderContext.IASetInputLayout(m\_vsShader.GetInputLayout());  SConstantBuffer cb;  cb.color.x = 1.0f;  cb.color.y = 0.0f;  cb.color.z = 0.0f;  cb.color.w = 1.0f;  //定数バッファの内容を更新。  renderContext.UpdateSubresource(m\_constantBuffer, color);  //定数バッファをPSステージに設定。  renderContext.PSSetConstantBuffer(0, m\_constantBuffer);  //描画。  renderContext.Draw(3, 0);  } |

頂点バッファを設定したり、シェーダーを設定したりと、行っていることはDirectX9のころと大きな違いはありません。プリミティブのトポロジーといのは、これからレンダリングを行うプリミティブがトライアングルリストなのか？トライアングルストリップなのか？ということを設定しています。これもDirectX9のプリミティブタイプと同じものです。UpdateSubresourceを使用して定数バッファの内容を更新しています。更新した定数バッファはPSSetConstantBufferを用いてPSステージに設定しています。DirectX9の時はどのステージに設定するかなどは気にしなかったと思いますが、実はあれはD3DXのヘルパー関数が行ってくれていただけです。DirectX11からはD3DXは廃止されているので、自分で適切なステージに設定を行う必要があります。

続いてシェーダーを見てみましょう。

Tutorial01/Assets/shader/ Tutorial.fx(5行目～25行目)

|  |
| --- |
| //定数バッファ。  cbuffer c0 : register(b0){  float4 color; //カラー。  } ;  //--------------------------------------------------------------------------------------  // 頂点シェーダーのエントリー関数。  //--------------------------------------------------------------------------------------  float4 VS( float4 Pos : POSITION ) : SV\_POSITION  {  return Pos;  }  //--------------------------------------------------------------------------------------  // ピクセルシェーダーのエントリ関数。  //--------------------------------------------------------------------------------------  float4 PS( SOutputVS In ) : SV\_Target  {  return color;  } |

頂点シェーダーは入力された頂点を出力しているだけです。ピクセルシェーダーは定数バッファに設定されているcolorの値を返しています。このcolorが先ほどCPUで設定していた定数バッファの値です。DirectX9の定数レジスタと大きな違いはありません。

**1.1 定数バッファ**

　定数バッファの用途はDirectX9の定数レジスタと同じです。C言語の構造体とよく似ていて、下記のように記述します。

***cbuffer* [*バッファの名前(C言語の構造体名のようなもの)*] : register(b#)**

**{**

**[変数] ;**

**...**

**};**

ここで気になるのはregsiterというキーワードだと思います。これは定数バッファをどのレジスタに割り当てるのか？というものを指定するものです。例えば下記のような定数バッファであれば定数バッファはbというレジスタの0番目に割り当てられます。

|  |
| --- |
| cbuffer Camera : register(b0)  {  float4x4 mView;  float4x4 mProj;  }; |

PSSetConstantBufferの1番目の引数はこのレジスタの番号を表しています。

**実習(２０分)**

Tutorial01を改造して、ポリゴンを任意の拡大率で拡大縮小できるようにしなさい。ポリゴンの拡大率は定数バッファを使用してGPUに転送するようにしなさい。

**Chapter 2 GPGPU**

**2.1 GPGPUの始まり**

GPGPUとはGeneral-purpose computing on graphics processing unitsの略でGPUに汎用計算を行わせることを指しています。汎用計算とは画像処理以外の処理のことを指します。例えばコリジョン処理など。

本来GPUは画像処理を行うために進化してきましたが、プログラマブルシェーダーの登場により、GPUでそれなりに複雑なロジックを実行することができるようになりました。また、CPUはGPUとは異なる進化を遂げてきたため、単純な計算であればCPUとは比べ物にならない速度で処理することができます。そのため、GPUに画像処理だけさせるのは勿体ない、もっと色々な処理をGPUに実行させよう！という考えが生まれます。これがGPGPUの始まりです。

**2.2 コンピュートシェーダー**

DirectX9の時代はGPUで行った計算の結果はテクスチャに対してしか出力することができません。また、両隣の計算結果を参照して処理を行うということも１パスでは行うことができません。このようにDirectX9でGPGPUを行う場合は多くの制限を受けていました。そこでDirectX10ではより高度なGPGPUを行うことができるように、コンピュートシェーダーという新しいシェーダーステージが追加されました。

コンピュートシェーダーはhlslで記述することができ、より柔軟に計算結果を出力することができるようになっています。

**2.3 データの入力と出力**

　多くの処理ではいろいろな計算を行うためにはデータを入力する必要があります。そしてその入力に対する計算結果を出力する必要もあるでしょう。クラスの平均点を求める関数であれば、下記のような実装になると思います。

**int** CalcAvarage( **int numStudent, int\* scoreArray** )

{

出力

入力

int totalScore = 0;

for( int I = 0; I < numStudent; i++ ){

totalScore += scoreArray[i];

}

return totalScore / numStudent;

}

このクラスの平均点を求める計算をGPUに行わせる場合も同様にGPUに対して生徒数と点数の配列を送ってやる必要があります。そして、GPUで行った計算結果を出力する必要があります。

**2.4 メモリ**

前節で示したように、GPGPUを行うためにはGPUへのデータを入力と、GPUからのデータの出力を行う必要があります。ではどのようにしてデータの入力、出力を行うのでしょうか？これを理解するためにはGPUとCPUとメモリについて学んでいく必要があります。

**2.4.1 VRAM**

少しPCに詳しい人であればVRAMという言葉を聞いたことがあると思います。VRAMとはGPUがアクセスできるメモリです。今まで君たちがC++で扱ってきたメモリはCPUがアクセスできるメモリ(メインメモリ)です。GPUがアクセスできるメモリはVRAMだけなので、テクスチャ、頂点バッファ、インデックスバッファ、そして定数レジスタなどのデータはすべてVRAMに載っている必要があります。今までDirextX9のAPIを使用してテクスチャや頂点バッファなどの作成を行っていたと思いますが、実はこれらのメモリはVRAM上で確保されていました。なので、GPGPUを行うための入力データはメインメモリからVRAMに転送する必要があります。DirectX9時代の定数レジスタをイメージするとわかりやすいでしょう。

　では出力データはどうなるのでしょうか？GPUから書き込みができるメモリはVRAMだけです。GPUで行った計算結果をCPUで使用するためには、VRAMからメインメモリに転送する必要があります。

**2.4.2 ユニファイドメモリ**

　さて、前節ではGPUがアクセスできるメモリはVRAMでCPUがアクセスできるメモリはメインメモリだと言いました。しかし、ユニファイドメモリを採用しているアーキテクチャではGPUとCPUが同じメモリ空間にアクセスすることができます。PS4はユニファイドメモリです。ユニファイドメモリのアーキテクチャではVRAMにデータを転送する必要はなく、直接メモリアドレスを指定するだけでよくなるため、実装がシンプルになります。

**2.4.3 DirectX11でのメモリの扱い**

　DirectX11でユニファイドメモリの場合とそうでない場合の両方のコードを記述することができます。しかし、ユニファイドメモリ前提のコードを記述した場合、そうではない場合に意図しない挙動をすることになるので、PCのようなユーザーによって多様なアーキテクチャが存在する環境で開発を行うのであれば、ユニファイドメモリではない場合のコードを記述するのが無難です。ですので、これ以降はメモリ空間が別になっているものとして話を進めていきます。

**2.5 GPUで平均点を計算するサンプルプログラム。**

ではGPUで平均点を計算するサンプルプログラムを見ていきましょう。

Tutorial01.cppのCComputeTest::Start関数で初期化を行っています。まず、コンピュートシェーダーをロードします。

|  |
| --- |
| //コンピュートシェーダーをロード。  m\_csShader.Load("Assets/shader/BasicCompute11.fx", "CSMain", CShader::EnType::CS); |

VRAM上に生徒の平均点を転送するためにStructuredBuffer(後述)を作成します。

|  |
| --- |
| D3D11\_BUFFER\_DESC desc;  ZeroMemory(&desc, sizeof(desc));  //SRVとしてバインド可能。  desc.BindFlags = D3D11\_BIND\_SHADER\_RESOURCE; //SRVとしてバインド可能。  desc.ByteWidth = NUM\_STUDENT \* sizeof(SInputParam);　 //バッファのサイズ。  desc.MiscFlags = D3D11\_RESOURCE\_MISC\_BUFFER\_STRUCTURED;  desc.StructureByteStride = sizeof(SInputParam);　　　//一要素のサイズ。  //StructuredBufferを作成。VRAM上にメモリを確保して入力データを転送。  m\_inputBuffer.Create(score, desc); |

VRAM上に出力用のStructureBufferを作成します。

|  |
| --- |
| D3D11\_BUFFER\_DESC desc;  ZeroMemory(&desc, sizeof(desc));  desc.BindFlags = D3D11\_BIND\_UNORDERED\_ACCESS; //UAVとしてバインド可能。  desc.ByteWidth = sizeof(SOutputParam); //バッファのサイズ。  desc.MiscFlags = D3D11\_RESOURCE\_MISC\_BUFFER\_STRUCTURED;  desc.StructureByteStride = sizeof(SOutputParam); //一要素のサイズ。  //StructuredBufferを作成。VRAM上に出力用のメモリを確保する。  m\_outputBuffer.Create(NULL, desc); |

出力結果をCPUで見るためにメインメモリ上にStructuredBufferを作成します。

|  |
| --- |
| D3D11\_BUFFER\_DESC desc;  ZeroMemory(&desc, sizeof(desc));  desc.CPUAccessFlags = D3D11\_CPU\_ACCESS\_READ; //CPUから読み込み可能に設定する。  desc.Usage = D3D11\_USAGE\_STAGING;//GPUからCPUへのデータコピーをサポートする。  desc.BindFlags = 0; //どこにもバインドしない。  desc.MiscFlags = 0;  desc.ByteWidth = sizeof(SOutputParam); //バッファのサイズ。  desc.StructureByteStride = sizeof(SOutputParam); //一要素のサイズ。  desc.MiscFlags = D3D11\_RESOURCE\_MISC\_BUFFER\_STRUCTURED;  //StructuredBufferを作成。メインメモリ上に作成する。  m\_outputBufferCPU.Create(NULL, desc); |

最後に各StructuredBufferがGPUからどのように見えるかを指定するビューを作成します。

|  |
| --- |
| //SRVを作成。  m\_inputSRV\_0.Create(m\_inputBuffer);  //UAVを作成。  m\_outputUAV.Create(m\_outputBuffer); |

続いて、コンピュートシェーダーの実行を見ていきましょう。CComputeTest::Render関数で毎フレームコンピュートシェーダーを実行しています。

|  |
| --- |
| //コンピュートシェーダーを設定。  renderContext.CSSetShader(m\_csShader);  //SRVを設定。  renderContext.CSSetShaderResource(0, m\_inputSRV\_0);  //UAVを設定。  renderContext.CSSetUnorderedAccessView(0, m\_outputUAV);  //コンピュートシェーダーを実行。  renderContext.Dispatch(1, 1, 1);  //CPUからアクセスできるバッファにコピー。  renderContext.CopyResource(m\_outputBufferCPU, m\_outputBuffer);  //コンピュートシェーダーの結果を取得。  CMapper<CStructuredBuffer> mapper(renderContext, m\_outputBufferCPU);  SOutputParam\* p = (SOutputParam\*)mapper.GetData();  if (p) {  char text[256];  sprintf\_s(text, "平均点 %d\n", p->avarage);  MessageBox(NULL, text, "結果", MB\_OK);  DeleteGO(this);  } |

ではコンピュートシェーダーを見てみましょう。Assets/shader/BasicCompute11.fxを開いてください。

|  |
| --- |
| #define NUM\_STUDENT 30  /\*!  \* @brief 入力構造体。  \*/  struct SInputParam  {  int score;  };  /\*!  \* @brief 出力構造体。  \*/  struct SOutputParam  {  int avarage;  };  //入力バッファ。  StructuredBuffer<SInputParam> InputBuffer : register(t0);  //出力バッファ。  RWStructuredBuffer<SOutputParam> OutBuffer : register(u0);  [numthreads(1, 1, 1)]  void CSMain( uint3 DTid : SV\_DispatchThreadID )  {  //平均点を計算する。  int totalScore = 0;  for(int i = 0; i < NUM\_STUDENT; i++ ){  totalScore += InputBuffer[i].score;  }  OutBuffer[0].avarage = totalScore / NUM\_STUDENT;  } |

InputBufferが入力データ、OutBufferが出力データです。

**2.6 StructuredBuffer**

　StructuredBufferとは構造化バッファと呼ばれるもので、構造体の配列のように扱うことができます。シェーダーへの入力、出力のバッファとして使用できます。

**2.7 ビュー**

　ビューとはStructuredBufferやテクスチャをGPUでどのようなリソースとして扱えば良いのかを指定するためのものです。今回のサンプルでは入力用のStructuredBuffer はShaderResourceView(SRV)、出力用のStructuredBufferはUnorderedAccessView(UAV)に設定しています。基本的にSRVは読み込み専用、UAVは読み書き用となります。そのため、コンピュートシェーダーの出力するバッファはUAVに設定されることとなります。

**実習(20分)**

Tutorial02/Questionを改造して、最高得点、最低得点をGPUで調査するようにしなさい。そして、その結果を平均点と同じようにメッセージボックスで表示しなさい。

**Chapter 3 コンピュートシェーダーの並列処理**

このチャプターではGPGPUによる画像のモノクロ化のサンプルプログラムを通して、コンピュートシェーダーの並列処理について学びます。

**3.1 スレッド**

すべてのプログラムはスレッドによって実行されています。驚くかもしれませんが、実はこれまで、あなたが記述したmain関数はメインスレッドから呼ばれていたのです。スレッドは複数作成することができて、作成されたスレッドは並列に動作します。例えば、あるゲームの１フレームの処理が下記のようになっていると考えてみましょう。

|  |  |
| --- | --- |
| ０ミリ秒 | メインスレッド |
|  | **プレイヤーのUpdate** |
|  | **エネミーのUpdate** |
|  | **ギミックのUpdate** |
|  | **物理シミュレーション** |
|  | **描画** |
| 24ミリ秒 |  |

このゲームではメインスレッドに記述されているゲームループから順次処理を呼び出していて、1フレームの処理が完了するまでに24ミリ秒かかっています。残念ながらこのゲームは60fpsを達成できていないため、ゲームプレイの滑らかさは多少損なわれています。では、スレッドを二つ立てて、物理シミュレーションと描画処理を並列に処理してみましょう。

|  |  |  |  |
| --- | --- | --- | --- |
| ０ミリ秒 | メインスレッド | 物理スレッド | 描画スレッド |
|  | **プレイヤーのUpdate** | **物理シミュレーション** | **描画** |
|  | **エネミーのUpdate** |  |  |
|  | **ギミックのUpdate** |  |  |
| 14ミリ秒 |  |  |  |

物理シミュレーションと描画処理を並列に実行するようにしたことで、１フレームの処理時間が14ミリ秒まで短縮できました！これで60fpsが達成できました！

(現実はデータの排他など非常に難解なことを考える必要が出てくるため、マルチスレッドにするということはここまで簡単な話ではないので注意してください。)

コンピュートシェーダーもスレッドから実行されており、ディスパッチされると、スレッドが生成されます。生成されるスレッドの数はコンピュートシェーダーのコードで指定することができます。

|  |
| --- |
| /\*!  \* @brief メイン関数。  \*/  **[numthreads(2, 2, 1)] //これがスレッドの数！！！**  void CSMain( uint3 DTid : SV\_DispatchThreadID)  {  　　　・  　　　・  　　　・  } |

numthreads(x,y,z)という文が一つのスレッドグループで生成されるスレッド数を指定しています(スレッドグループについては後述します)。この場合生成されるスレッド数は2×2１の4スレッド生成されます。numthreads(4, 1, 1)と同じ意味になるのですが、例えば入力データを二次元配列のように扱いたい場合、このように指定したほうが扱いやすくなります。

**3.2 スレッドグループ**

　スレッドグループとはスレッドをひとまとめにしたものです。スレッドグループの数はコンピュートシェーダーをディスパッチするときに指定することができます。

|  |
| --- |
| //3×3のスレッドグループが作られる！  renderContext.Dispatch(3,3, 1); |

renderContext.Dispatchの引数がスレッドグループの数です。この場合スレッドグループの数は3×3の9グループとなります。では3.2の4スレッド生成するコンピュートシェーダーを3×3の9グループでディスパッチするとどのようになるか見てみいきましょうまずディスパッチされると(０,0)のグループの4スレッドが起動してCSMainが実行されます。

**並列に動作する**

|  |  |  |
| --- | --- | --- |
| グループ |  |  |
| **0,0** | **thread(0,0)** | **CSMain** |
| **thread(0,1)** | **CSMain** |
| **thread(1,0)** | **CSMain** |
| **thread(1,1)** | **CSMain** |

(０,0)のスレッドがすべて終了すると(0,1)のグループが実行されます。

|  |  |  |
| --- | --- | --- |
| グループ |  |  |
| **1,1** | **thread(0,0)** | **CSMain** |
| **thread(0,1)** | **CSMain** |
| **thread(1,0)** | **CSMain** |
| **thread(1,1)** | **CSMain** |

続いて(0,2)、(1,0)と実行されて最後に(2,2)を実行するとコンピュートシェーダーは終了します。

では、なぜスレッドの数とスレッドグループをxとyに分けて指定していたのか見ていきましょう。コンピュートシェーダーのメイン関数の引数にSV\_DispatchThreadIDセマンティクスが指定されているものがあります。このセマンティクスが指定された引数にはスレッドを識別するためのユニークなIDが設定されます。次の図を見てみてください。

|  |  |  |  |
| --- | --- | --- | --- |
| グループ番号 |  |  | SV\_DispatchThreadID |
| 0,0 | thread(0,0) | CSMain | **x=0、y=0** |
| thread(0,1) | CSMain | **x=0、y=1** |
| thread(1,0) | CSMain | **x=1、y=0** |
| thread(1,1) | CSMain | **x=1、y=1** |
| 0,1 | thread(0,0) | CSMain | **x=0、y=2** |
| thread(0,1) | CSMain | **x=0、y=3** |
| thread(1,0) | CSMain | **x=1、y=2** |
| thread(1,1) | CSMain | **x=1、y=3** |
| 0,2 | thread(0,0) | CSMain | **x=0、y=4** |
| thread(0,1) | CSMain | **x=0、y=5** |
| thread(1,0) | CSMain | **x=1、y=4** |
| thread(1,1) | CSMain | **x=1、y=5** |
| 1,0 | thread(0,0) | CSMain | **x=2、y=0** |
| thread(0,1) | CSMain | **x=2、y=1** |
| thread(1,0) | CSMain | **x=3、y=0** |
| thread(1,1) | CSMain | **x=3、y=1** |
| 1,1 | thread(0,0) | CSMain | **x=2、y=2** |
| thread(0,1) | CSMain | **x=2、y=3** |
| thread(1,0) | CSMain | **x=3、y=2** |
| thread(1,1) | CSMain | **x=3、y=3** |
| 1,2 | thread(0,0) | CSMain | **x=2、y=4** |
| thread(0,1) | CSMain | **x=2、y=5** |
| thread(1,0) | CSMain | **x=3、y=4** |
| thread(1,1) | CSMain | **x=3、y=5** |
| 2,0 | thread(0,0) | CSMain | **x=4、y=0** |
| thread(0,1) | CSMain | **x=4、y=1** |
| thread(1,0) | CSMain | **x=5、y=0** |
| thread(1,1) | CSMain | **x=5、y=1** |
| 2,1 | thread(0,0) | CSMain | **x=4、y=2** |
| thread(0,1) | CSMain | **x=4、y=3** |
| thread(1,0) | CSMain | **x=5、y=2** |
| thread(1,1) | CSMain | **x=5、y=3** |
| 2,2 | thread(0,0) | CSMain | **x=4、y=4** |
| thread(0,1) | CSMain | **x=4、y=5** |
| thread(1,0) | CSMain | **x=5、y=4** |
| thread(1,1) | CSMain | **x=5、y=5** |

SV\_DispatchThreadIDは下記の計算式で求まります・

SV\_DispatchThreadID.x = グループID.x×スレッド数.x+スレッドID.x

SV\_DispatchThreadID.y = グループID.y×スレッド数.y+スレッドID.y

SV\_DispatchThreadID.z = グループID.z×スレッド数.z+スレッドID.z

SV\_DispatchThreadIDを２次元配列の添え字として使用することで２Ｄテクスチャや二次元配列のデータへのアクセスが便利になることがあります。では画像ファイルのモノクロ化のサンプルプログラムを通して具体例を見ていきましょう。

**3.3 画像ファイルのモノクロ化のサンプルプログラム**

このサンプルは１テクセルずつモノクロ化をコンピュートシェーダーで行っていくサンプルです。テクセル間でデータのやり取りなどが不要のため、並列化に向いている処理だといえます。一度に生成できるスレッドの数に上限がない場合、例えば400×400の解像度の画像であれば、160000個のスレッドを立てて並列に実行できます。

　では、Tutorial03/Assets/shader/monochromeCS.fxを開いて1スレッドグループにいくつのスレッドが生成されるのか見てみましょう。

Tutorial03/Assets/shader/monochromeCS.fx(25行目)

|  |
| --- |
| [numthreads(2, 2, 1)] |

1スレッドグループあたり2×2=4個のスレッドが生成されます。ではスレッドグループの数を見ていきましょう

Tutorial03/Tutorial.cpp(72行目)

|  |
| --- |
| //コンピュートシェーダーを実行。  const int NUM\_THREAD = 2;  const DirectX::TexMetadata& texMetadata = m\_textureData.GetMetadata();  renderContext.Dispatch(texMetadata.width/NUM\_THREAD,texMetadata.height/NUM\_THREAD, 1); |

グループ数は(テクスチャの幅÷NUM\_THREAD)×(テクスチャの高さ÷NUM\_THREAD)です。NUM\_THREADはコンピュートシェーダーで指定したスレッド数と同じ値になっていることに注意してください。グループの数をNUM\_THREADで除算することで、生成されるスレッドの総数がテクスチャの画素数と同じになります。

では、再度Tutorial03/Assets/shader/monochromeCS.fxを開いてください。

Tutorial03/Assets/shader/monochromeCS.fx(25行目～39行目)

|  |
| --- |
| [numthreads(16, 16, 1)]  void CSMain( uint3 DTid : SV\_DispatchThreadID)  {  **float4 color = inTexture.Load(uint3( DTid.x, DTid.y, 0));**  float Y = 0.29900 \* color.r + 0.58700 \* color.g + 0.11400 \* color.b;  color.r = Y;  color.g = Y;  color.b = Y;  color.a = 1.0f;  //float4をRGBA32フォーマットに変換。  //画像の縦幅は512で固定。  outputBuffer[DTid.x + DTid.y \* 512] = PackedFloat4ToRGBA32(color);  } |

**float4 color = inTexture.Load(uint3( DTid.x, DTid.y, 0));**に注目してください。テクスチャからカラーを取得している処理ですが、DTid.xとDTid.yを利用して取得する取得するテクセルの座標を指定しています。これはスレッドＩＤを利用する典型的な処理です。

**実習課題**

1. Tutorial03/Question/Tutorial.cppの18行目の#define USE\_GPUをコメントアウトして、モノクロ化をCPUとGPUで行った場合の処理時間の差を確認しなさい。
2. 下記のコメントアウトを削除して、モノクロ化をCPUとGPUで行った場合の処理時間の差を確認しなさい。

Tutorial03/Question/Tutorial.cpp 119行目と125行目

Tutorial03/Question/Assets/Shader/ monocromeCS.fx 29行目と35行目

1. １グループのスレッド数を16に変更して、モノクロ化をCPUとGPUで行った場合の処理時間の差を確認しなさい。
2. モノクロ化を行うコンピュートシェーダーを改造して、画像をセピア調にコンバートできるようにしなさい。
3. モノクロ化を行うCPU側のコードを改造して、CPUでも画像をセピア調にコンバートできるようにしなさい。(ヒント：color.rgbaは0.0～1.0の浮動小数型である必要がある。)

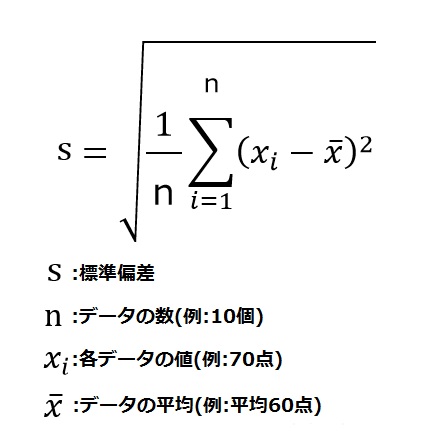
セピア調への変換プログラム例

|  |
| --- |
| 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; |

**小テスト 01(制限時間 2時間半)**

1. MiniTest\_01/mondai\_01はあるクラスの生徒の点数をランダムに生成するプログラムです。このランダムに生成された生徒の点数を用いてクラスの標準偏差をコンピュートシェーダーを使用して求めなさい。

Tutorial.cppのMondai\_01::Start関数に必要な初期化処理を、Mondai\_01::Render関数にコンピュートシェーダーのディスパッチとコンピュートシェーダーで行った実行結果をメッセージボックスで表示するプログラムを記述しなさい。標準偏差は下記の公式で求まります。(30点)



1. MiniTest\_01/mondai\_02を使用して、MiniTest\_01/mondai\_02/original.jpgを白黒化した画像を生成して、MiniTest\_01/mondai\_02/kekka.jpgとして保存するプログラムを記述しなさい。モノクロ化はコンピュートシェーダーを用いて実装しなさい。(ヒント：Tutorial03のプログラムを丸々コピーではうまくいきません。ただし、少し修正するだけで白黒化することができるようになります)(30点)
2. MiniTest\_01/mondai\_03を使用して、MIniTest\_01/mondai\_03/original.jpgにガウスブラーをかけた画像を生成して、MiniTest\_01/mondai\_02/kekka.jpgとして保存するプログラムを記述しなさい。ガウスブラーはコンピュートシェーダーを用いて実装しなさい。(40点)

**Chapter 4 Forward Rendring vs Diferred Rendering**

最近のレンダリング手法は大きくFoward系とDiferred系の二つに分けることができます。このチャプターではForwardRenderingとDiferredRenderingについてみていきます。

**4.1 ForwardRendering**

ForwardRenderingは3Dゲーム黎明期から進化してきたレンダリング手法で、今まで皆さんが実装してきたものがそれにあたります。FowardRenderingを端的に説明すると、**「ポリゴンをレンダリングする時にライティングの計算を行う。」**というものです。モデルのDrawを行うと頂点シェーダー、ピクセルシェーダーが実行されてピクセルカラーが決まります。このタイミングでライティングを行うのがFowardRenderingです。

**4.2 Deferred Rendering**

DeferredRenderingはxbox360、PlayStation3のころに生まれたレンダリング手法で、比較的新しい手法となっています。特にPlayStation3はこの手法が向いているアーキテクチャだったため、DeferredRenderingが採用されているゲームがいくつかありました。

DeferredRenderingを端的に説明すると**「ポリゴンをレンダリングする時にはライティングの計算は行わずに後で行う。」**というものです。Deferredは遅延という意味なので、遅延レンダリングとも呼ばれます。Forward系に慣れ親しんでいるとピンと来ないかもしれません。ライティングの計算をポストエフェクト的に行うと言った方が理解しやすいかもしれませんね。

**4.2.1 G-Buffer**

　ではDeferredRenderingで使用されるG-Bufferについてみていきましょう。DeferredRenderingではポリゴンをレンダリングする時にはライティングの計算は行わずにMRT(MultiRenderingTarget)を活用して、複数枚のテクスチャ(G-Bufferと呼ばれる)にテクスチャカラー、法線情報、スペキュラ強度、深度値などを書き込みます。下記はPlayStation3のKillzone2のG-Bufferの内容です。



FowardRenderingでは「モデルを描画する=ライティングを行ってフレームバッファに書き

込む」だったのですが、DeferredRenderingでは「モデルを描画する=ライティングに必要

な情報をG-Bufferに書き込む。」というものになります。

そして、G-Bufferを使用してポストエフェクト的にライティングの計算を行って、最終画

象をフレームバッファに書き込みます。

**4.2.2 Deferred Renderingのメリット**

　なぜ、PlayStation3やXbox360のころからDeferredRenderingを採用しているゲームが増えてきたのでしょうか？当然増えてきたのには何か理由があります。この節ではその理由についてみていきましょう。

　PlayStation3、Xbox360が登場したことによって、家庭用ゲーム機の世界にもプログラマブルシェーダーの波が押し寄せました。映像を少しでもいいものに、ほかのゲームと違うグラフィック表現などなど、いろいろな工夫が凝らされるようになった結果、1ピクセル当たりの計算量がどんどん増えていきました。例えばPlayStation3が発売されたばかりのころの１ピクセルのプログラムは下記のようなものだったと思ってください。

|  |
| --- |
| /\*!  \*@brief ピクセルシェーダー。  \*/  float4 PSMain( VS\_OUTPUT In ) : COLOR  {  //ライトを計算。  float4 lig = 0.0f;  float3 normal = normalize(In.normal);  lig.xyz = CalcDiffuse( In.normal );  lig.xyz += CalcSpecular( In.worldPos, normal );  lig += g\_ambientLight;  float4 color = tex2D( g\_diffuseTextureSampler, In.uv );  color.xyz \*= lig;  return color;  } |

これが時とともにもっと良いものを求めていった結果複雑化していき、下記のようなコードになっていきました。

|  |
| --- |
| /\*!  \* @brief ピクセルシェーダー。  \*/  PSOutput PSMain( VS\_OUTPUT In )  {  float4 color = 0.0f;  float4 diffuseColor = tex2D(g\_diffuseTextureSampler, In.Tex0);  color = diffuseColor;  float3 normal = normalize(In.Normal);  if(g\_flags.x){  //法線マップあり。  float3 tangent = normalize(In.Tangent);  float3 binSpaceNormal = tex2D( g\_normalMapSampler, In.Tex0);  float4x4 tangentSpaceMatrix;  float3 biNormal = normalize( cross( tangent, normal) );  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;  normal = tangentSpaceMatrix[0] \* binSpaceNormal.x + tangentSpaceMatrix[1] \* binSpaceNormal.y + tangentSpaceMatrix[2] \* binSpaceNormal.z;    }  float4 lig = DiffuseLight(normal);  if(g\_flags.z){  //リムライト。  lig.xyz += CalcLimLight(normal, g\_light.limLightDir, g\_light.limLightColor.xyz);  }  if(g\_flags.w){  //スペキュラライト。  lig.xyz += SpecLight(normal, In.worldPos\_depth.xyz, In.Tex0);  }  if(g\_flags.y){  //影  lig \*= CalcShadow(In.worldPos\_depth.xyz);  }  //自己発光色  lig.xyz += g\_light.emission;  color \*= lig;  //大気錯乱  if(g\_flags2.y == AtomosphereFuncObjectFromAtomosphere)  {  color = In.rayColor + color \* In.mieColor;  }  //ポイントライト。  color.xyz += diffuseColor.xyz \* PointLight(normal, In.worldPos\_depth.xyz, g\_flags.z);  //アンビエントライトを加算。  color.xyz += diffuseColor.xyz \* g\_light.ambient.xyz;  if(g\_fogParam.z > 1.9f){  //高さフォグ  float h = max(In.worldPos\_depth.y - g\_fogParam.y, 0.0f);  float t = min(h / g\_fogParam.x, 1.0f);  color.xyz = lerp(float3(0.75f, 0.75f, 0.95f), color.xyz, t);  }else if(g\_fogParam.z > 0.0f){  //距離フォグ  float z = length(In.worldPos\_depth.xyz - g\_cameraPos);  z = max(z - g\_fogParam.x, 0.0f);  float t = min( z / g\_fogParam.y, 1.0f);  color.xyz = lerp(color.xyz, float3(0.75f, 0.75f, 0.95f), t);  }  PSOutput psOut = (PSOutput)0;  psOut.color = color;  psOut.depth = In.worldPos\_depth.w;  if(g\_flags2.x){  psOut.velocity.xy = In.velocity.xy / In.velocity.w-In.screenPos.xy / In.screenPos.w;  psOut.velocity.xy \*= 0.5f;  psOut.velocity.xy += 0.5f;  psOut.velocity.zw = 0.0f;  }else{  //速度なし。  psOut.velocity = 0.5f;  }  return psOut;  } |

これが１ピクセルに実行されるプログラムです。これによりピクセル単位のプログラムの処理時間が増大していきました。そこで、無駄なピクセルのプログラムは実行しないようにできないか？という考えから生まれてきたのがDeferredRenderingです。

ForwardRenderingは先にライティングを行うため、不要なピクセルのライティングが計算されてしまいます。例えば3Dモデルを奥から手前にレンダリングした場合のことを考えてみてください。

この図は顔のモデルを描画した後で、その手前に三角形の板ポリを描画している図です。これがForwardRenderingで行われた場合、三角形の板ポリに遮蔽された部分のライティングの計算が無駄になってしまいます。DeferredRenderingでは、モデル描画のパスではライティング計算は行わずに、G-Bufferの内容を使って、ポストエフェクト的にライティングの計算を行うため、無駄な計算は発生しません。例えば1280×720の解像度であれば、どんな順番でモデルを描画したとしても1280×720回のライティングの計算で完了することになります。

**4.2.2 Deferred Renderingのデメリット**

先ほどはDeferredRenderingのメリットを見ていきましたが、すべてのケースでForwardRenderingより優れているわけではありません。ではDeferredRenderingのデメリットについてみていきましょう。

1. メモリの速度と容量の問題

Deferred系ではMRT(multiRenderingTarget)を活用するため、Forward系に比べるとメモリ使用量が増大します。また、モデルの描画パスで複数のG-Bufferに対して書き込みを行うため、メモリの書き込み速度も問題になってきます。CPUやGPUの演算速度とメモリの読み書きの速度の差はどんどん大きくなってきています。そのため、アーキテクチャによってはDeferred系の方が遅くなるというのは十分考えられます。実はDeferred系がPlayStation3では増えていたが、Xbox360では増えなかったのはこれが理由です。メモリの速度とVRAMの容量がxbox360はPlayStation3より劣っていたため、Deferredに不向きでした。

1. 半透明描画の問題

　Deferred系は半透明オブジェクトの描画に弱いため、半透明のオブジェクトを描画する場合、不透明オブジェクトをDeferredで描画した後でForwardで半透明オブジェクトを描画するなどといった工夫が必要になります。そのため、半透明オブジェクトを多用する日本のゲームには向いていないといわれています。

　このようにいいことばかりではなく、Deferredの方がForwardより遅くなるケースは実はたくさんあります。最近ですとVRゲームは4K解像度の絵を右目用と左目用の２枚レンダリングを行う必要があり、Deferredでレンダリングを行うと4K解像度の複数枚のG-Bufferへの書き込みが発生してメモリ帯域を圧迫します。そのため、VRゲームではDeferredよりもForwardの方が速くなる傾向があります。(UE4はVRのためにForwardのレンダリングエンジンを用意しましたね。)

**実習課題**

1. Tutorial04/Questionのサンプルプログラムを改造して、法線マップを出力するためのG-Bufferを追加して、モデルの描画パスでG-Bufferに対して法線を出力しなさい。
2. Tutorial04/Questionのサンプルプログラムを改造して、Deffered Lightinのパスでディフューズライティングの計算を行いなさい。ディフューズライティングの計算はDirectX\_3/ShaderTutorial\_06を参考にしなさい。
3. Tutorial04/Questionのサンプルプログラムを改造して、Deffered Lightinのパスでスペキュラライティングの計算を行いなさい。スペキュラライティングの計算はDirectX\_3/ShaderTutorial\_06を参考にしなさい。

**Chapter 5 TBR(Tile based rendering)**

　このチャプターでは大量のポイントライトを高速に扱うための手法のTBR(Tile based rendering)についてみていきましょう。

**5.1 ポイントライト**

　TBRの説明に入る前にポイントライトについて見ていきましょう。ポイントライトは位置、カラー、減衰率を持っているライトです。豆電球を想像すれば分かりやすいかと思います。

　ディフューズライトの強さはライトの方向とライトが当たる面の法線との内積を取ることで計算が出来ました。

|  |
| --- |
| ライトの方向をL  ライトが当たる面の法線をN  ライトの強さ = max(0.0,(-L・N)) |

ポイントライトの強さの計算もディフューズライトと似ており、ライトの方向とライトの当たる面の法線を使用します。

**5.2.1 ポイントライトの方向**

　ポイントライトは全方位に光を放出しているので、ライトの方向を求める場合は、ライトの位置とポイントライトの位置を使って下記のように計算します。

|  |
| --- |
| ライトの位置をPL  ライトが当たる面の位置をＰ  Normalizeはベクトルを正規化する関数とすると、  L = normalize(P – PL ) |

**5.2.2 ポイントライトの強さ**

　前節でポイントライトの方向が分かりました。ポイントライトの強さはディフューズライトと同じような計算を行います。つまり、ライトの向きとライトが当たる面の法線の内積を計算するのです。

|  |
| --- |
| ライトの方向をL  ライトが当たる面の法線をN  ライトの強さ = max(0.0,(-L・N)) |

しかし、このままではポイントライトからどれだけ離れていてもライトが当たってしまいます。では最後に光の減衰を見ていきましょう。

**5.2.3 ポイントライトの減衰**

光は光源から離れていくと減衰していきます。つまり光の強さは光源との距離と反比例していくこととなります。もっとも簡単な光の減衰の計算は下記のようなものです。

|  |
| --- |
| ライトの方向をL  ライトが当たる面の法線をN  ライトとライトが当たる面の距離をDとすると  ライトの強さ = max(0.0,(-L・N)) **/ D** |

ただし、この計算のままだと距離が0の時に0除算が発生してしまいます。そこで下記のような計算にしてみましょう。

|  |
| --- |
| ライトの方向をL  ライトが当たる面の法線をN  ライトとライトが当たる面の距離をDとすると  ライトの強さ = max(0.0,(-L・N)) **/ (1.0 + D )** |

**5.2.3 ポイントライト まとめ**

　では最後にポイントライトの疑似コードを示します。

|  |
| --- |
| //ライトの位置をlightPos  //ライトのカラーをlightColor  //ライトが当たる面の位置をpos  //ライトが当たる面の法線をnormalとすると  //ライトまでの距離を求める。  float3 lightDir = pos – lightPos;  float len = length(lightDir);  //lightDirを正規化してライトの方向を求める。  lightDir = normalize(lightDir);  //ライトの強さを計算する。  float lightPower = max(0.0f, -dot(lightDir, normal));  lightPower /= (1.0f + len); //ライトまでの距離で減衰させる。  //最後にライトの色を決定する。  float3 lig = lightPower \* lightColor; |

**実習課題**

1. サンプルプログラムのTutorial05/Questionを改造してポイントライトを実装しなさい。ライティングの計算はディファードライティングのパスで行うように。

ヒント①

ポイントライトを計算するためには、ライトが当たる面のワールド座標が必要に

なる。そのため、モデル描画パスでG-Bufferにワールド座標を出力する必要があ

る。

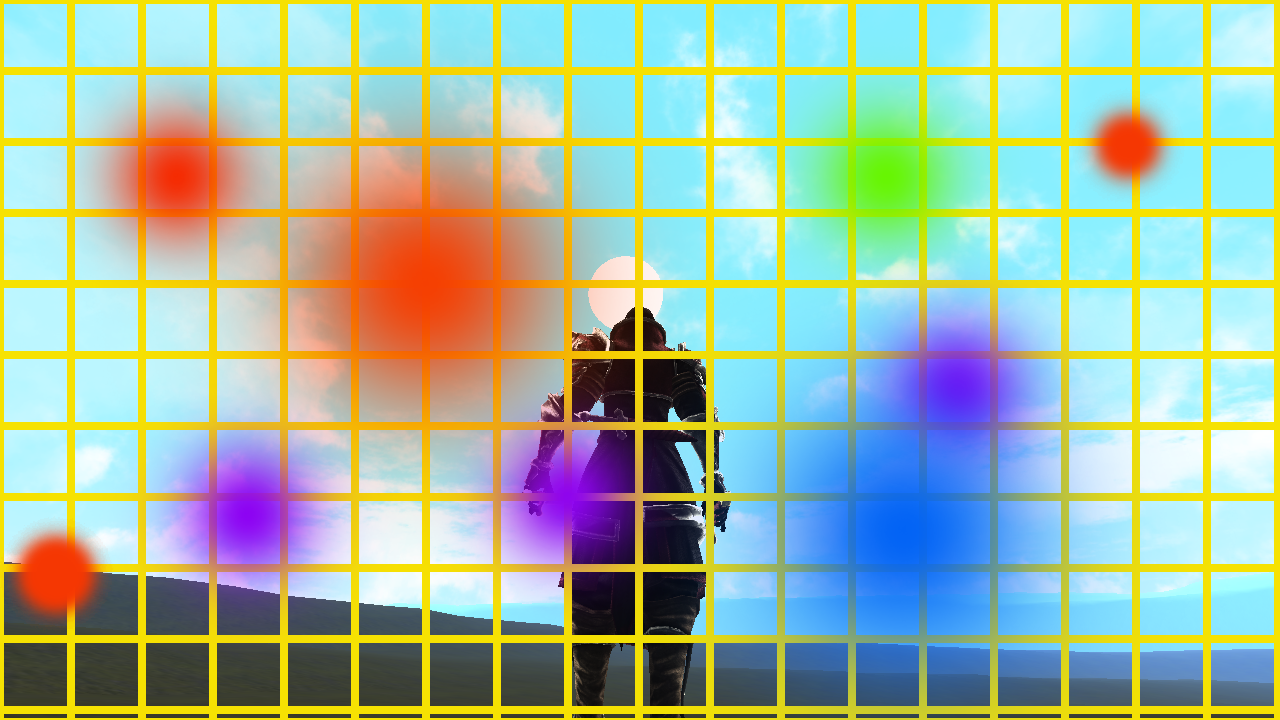
**5.3 TBDR(Tile based deffred rendering)**

　ではいよいよTBR技術の一つTBDRについて見ていきましょう。TBDRはここまで勉強してきたディファードレンダリングの進化版で、DirectX11世代のGPUが乗っているプラットフォームで使われている技術です。

　DirectX９世代のGPUでは動的光源が増えると処理がどんどん重くなっていくという問題がありました。Forward系に比べるとDeffered系は多少マシですが、それでも100も200も光源を増やすとまともなパフォーマンスは出ません。しかし、現実世界には多数の動的光源があります。これを解決するために考えられたのがDeffredRenderingの進化版となるTBDR(Tile based deffered rendering)です。

**5.3.1 アルゴリズム概要**

　TBDRのアルゴリズムを簡単に説明すると、「スクリーンをタイル状に分割して、そのタイルに影響を与える光源のリストを作成する。そのあとで、そのタイル含まれるピクセルに影響を与える光源だけでライティングを計算する。」というものです。



**5.3.2 アルゴリズム詳細**

　ではアルゴリズムの詳細を見ていきましょう。TBDRのアルゴリズムは下記のような流れになります。

1. モデル描画パスでＧ-Bufferを作成する。
2. スクリーンをタイル状に分割して、各タイルごとに影響を与える可能性のある光源のリストを作成する。
3. ②で調べた光源のリストとG-Bufferを使用してポストエフェクト的にライティングを行っていく。

では、各ステップを詳細に見ていきましょう。

1. **モデル描画パスでＧ-Bufferを作成する。**

これは今までのディファードレンダリングと変わりありません。

1. **スクリーンをタイル状に分割して、各タイルごとに影響を与える可能性のある光源のリストを作成する。**

これはコンピュートシェーダーで行います。各タイルの視推台を求めて、その視推台とポイントライトが接触しているかを調べます。

1. **②で調べた光源のリストとG-Bufferを使用してポストエフェクト的にライティングを行っていく。**

やっていることは今までのディファードレンダリングと同じですが、TBDRではライティングの計算もコンピュートシェーダーで行います。②と③を一つのコンピュートシェーダーで行うことによって、プラットフォームによってはVS->PSという二つの処理をまたぐより効率的に処理を行えることがあります。

**実習**

1. Tutorial06/DeferredRendering/Tutorial.cppの12行目のNUM\_POINT\_LIGHTの数を変更してFPSの変化を確認しなさい。(最大512)
2. Tutorial06/TBDR/Tutorial.cppの12行目のNUM\_POINT\_LIGHTの数を変更してFPSの変化を確認しなさい。(最大512)
3. VisualStudio2015の機能のグラフィック診断を使って、TBDRの処理を追いかけなさい。

**5.4 TBFR(Tile based forward rendering)**

ではTBR系の勉強の最後に、TBFR(Tile based forward rendering)について見ていきましょう。TBDRのようなDeferred系のレンダリング手法は下記のようなデメリットがあります。

　・半透明描画に弱い。

・多彩なマテリアルの表現に弱い。

　　　->多彩なマテリアルに対応するためには、マテリアルの情報を書き込むための

G-Bufferが必要になったりする。

　・G-Bufferの肥大化によるメモリ使用量の増加。

　・etc

このように、Deferred系は柔軟性が低く、特にノンフォトリアルな絵を作りたい場合は不向きな面があります。そこでforward系を改良してはどうか？という考えが生まれます。そこから生まれてきたのがTBFR(別名forward+)といわれるレンダリング手法です。

**5.4.1 アルゴリズム概要**

　TBFRもアルゴリズムの概要はTBDRと同様です。スクリーンをタイル状に分割して、GPGPUでタイルに衝突しているライトのリストを作成します。そしてそのリストを使用して、ピクセルに当たる可能性のあるライトに対してだけライティングを計算するというものです。

**5.4.2 アルゴリズム詳細**

　ではアルゴリズムの詳細を見ていきましょう。TBFRはZPrepassという描画パスが必要になります。これはDeferred系でG-Bufferと同じようなものですが、深度値のみ出力する場合にZPrepassという言葉が使われます。TBFRでもGPGPUでライトのリストを作成するときに、各タイルの視推台を作成するときに深度値が必要になります。そのため、ライトカリングを行う前に、ZPrepassというパスで深度テクスチャを作成します。TBFRの処理の流れは下記になります。

1. ZPrepassで深度テクスチャを作成する。
2. GPGPUでタイルごとのライトカリング。
3. ②で作成されたライトのリストを使用してモデルを描画。(モデル描画パス)

では各ステップを詳細に見ていきましょう。

1. **ZPrepassで深度テクスチャを作成する。**

　Deferred系のG-Bufferの作成をイメージしてください。それの深度値を作成するだけのものがZPrepassです。

1. **GPGPUでタイルごとのライトカリング。**

スクリーンをタイル状に分割して、タイルと接触しているライトのリストを作成します。こちらもTBDRと同じです。ただし、TBDRではライトのリストを作成した後で、そのままコンピュートシェーダーでディファードライティングを行っていましたが、TBFRではここではライティング計算は行いません。ライトのリストを作成するだけです。

1. **②で作成されたライトのリストを使用してモデルを描画。(モデル描画パス)**

　モデルの描画パスです。TBFRはforward系のレンダリングですので、モデル描画パスでライティングの計算を行います。このライティングの計算の時に②で作成したライトのリストを使用します。

**実習**

1. Tutorial07/ForwardRendering/Tutorial.cppの12行目のNUM\_POINT\_LIGHTの数を変更してFPSの変化を確認しなさい。(最大512)
2. Tutorial06/TBFR/Tutorial.cppの12行目のNUM\_POINT\_LIGHTの数を変更してFPSの変化を確認しなさい。(最大512)
3. VisualStudio2015の機能のグラフィック診断を使って、TBFRの処理を追いかけなさい。

**Chapter 6 深度テクスチャの利用**

Chapter 5でZPrepassという描画パスで深度テクスチャを作成して、それを利用して大量点光源を実装しました。このように色々な中間データを用いて様々な画像処理を行うことができます。このチャプターでは深度テクスチャを利用したいくつかの画像処理について見ていきましょう。

**6.1 トゥーンシェーダーのエッジ描画**

アニメ調の表現をする際に輪郭線の描画を行うことがあります。輪郭線の抽出アルゴリズムはいくつか存在します。

　・法線の利用

　・ＩＤマップの利用

　・モデルの反転描画

　・**深度値の利用**

今回は深度値を利用したエッジ抽出を見ていきましょう。

**6.1.1深度値によるエッジ抽出**

　深度値によるエッジ抽出を行う場合、深度テクスチャを作成するパスが必要になります。例えば、ポストエフェクトエフェクト的なエッジ抽出の場合は、モデル描画パスでMRTを利用して、深度テクスチャに深度値を書き込めばいいでしょう。

**6.1.2 ポストエフェクトで深度値によるエッジ抽出を行うアルゴリズム**

**・アルゴリズム手順**

1. **モデル描画パスでMRTを活用して、深度テクスチャに深度値を書き込む。**
2. **ポストエフェクトで深度テクスチャを参照してエッジを描いていく。**

ポストエフェクトを使用したエッジ抽出が最も一般的な手法化と思います。しかし、ポストエフェクトを使用した場合、シンプルに実装すると、すべてのモデルにエッジが出てしまう問題があります。ゲームによっては背景にはエッジを描きたくないなどといったこともあるでしょう。そのような場合、一工夫必要になります。

**6.1.3 モデル描画パスで深度値を参照してエッジ抽出を行うアルゴリズム**

　そこで、今回はZPrepassを利用して、モデル描画パスでエッジを描くアルゴリズムを紹介します。モデル描画パスでエッジ抽出を行うため、エッジを描くモデル、描かないモデルの切り替えが容易になります。

**・アルゴリズム手順**

1. **ZPrepassで深度テクスチャを作成。**
2. **モデル描画パスで深度テクスチャを参照して、エッジを描画。**

**実習**

　Tutorial08/Questionを使用してトゥーンシェーダーを実装しなさい。

**6.2 被写界深度**

　深度テクスチャを使用した代表的なポストエフェクトに被写界深度というものがあります。ゲーム業界で被写界深度というキーワードが出てきた場合はほとんどの場合、カメラのピンボケ現象のことをさしていると考えて問題ありません。カメラに詳しい人が聞いたら怒るかもしれませんが。下記のような現象です。



カメラにはピントが合う範囲というものがあって、その範囲外の被写体は上の画像のようにボケて移ります。

今回実装する被写界深度はカメラ座標系でのZ値を使って、つまり深度テクスチャです、ボケ画像を生成して、元々のシーンに合成する手法です。アルゴリズムの詳細は下記です。

1. ZPrepassまたはモデル描画パスでMulti Rendering Targetなどを利用して深度テクスチャを作成する。
2. 被写界深度のポストエフェクトを実行する。ポストエフェクトの手順は下記のようになります。

・深度テクスチャとシーンテクスチャを参照して、ボケが発生するピクセルを抽出し

たテクスチャを作成する。

・作成された画像にブラーをかけてボケ画像を作成する。

・作成したボケ画像とシーン画像を合成して完成。

**実習**

Tutorial09/Questionを使用して、被写界深度を実装しなさい。