**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; |