

シンプルなライティングサンプル（DX12）

*このサンプルは、Windows 10 Anniversary Update SDK（14393）と互換性があります。*

# 概要

このサンプルでは、静的および動的ランバートライティングを使用して照らされたインデックス付きジオメトリを描画するために、静的Direct3D 12頂点、インデックス、および定数バッファを作成する方法を示します。

このサンプルは、立方体としても表される、2つのライト（1つは白で1つは赤）で照らされている大きな立方体をレンダリングします。白色光は静止していて、赤色光は中央の立方体の周りを周回しています。中央の立方体も回転します。この動きによって、色のついた光の効果をさまざまな角度から観察することができます。



# サンプルの使用

|  |  |  |
| --- | --- | --- |
| 作用 | ゲームパッド | キーボード |
| 終了 | 表示ボタン | ESC |

# 実装上の注意

## シェーダ

## このサンプルでは、シーンをレンダリングするために3つのシェーダ頂点シェーダ（ "TriangleVS"）と2つのピクセルシェーダ（ "LambertPS"、 "SolidColorPS"）を使用しています。コンパイルされたシェーダブロブはCreateDeviceDependentResourcesにロードされ、各シェーダの組み合わせに対してPipeline State Objectsを作成するときに参照されます。 すべてのシェーダは同じHLSLインクルードファイル "SimpleLighting.hlsli"で定義されていて、3つのスタブシェーダがこのファイルをインクルードします。各スタブシェーダは、3つのシェーダブロブを作成するために、異なるエントリポイントに対してコンパイルされます。

## パイプライン状態オブジェクト（PSO）

シンプルなライティングサンプルには2つのユニークなシェーダの組み合わせがあります。1つはLambertPSと組み合わせたTriangleVSで、もう1つはSolidColorPSと組み合わせたTriangleVSです。DirectX 12では、一意のシェーダの組み合わせごとにPipeline State Object（PSO）を作成する必要があります。名前が示すように、PSOは、特定のシェーダセットを使用して、場合によっては複数の描画呼び出しに必要なすべてのパイプライン状態をカプセル化します。PSOは、ルートシグネチャ、パイプラインのさまざまなステージのシェーダ、ラスタライザ状態、深度ステンシル状態、ブレンド状態などの状態設定を組み合わせたものです（詳細については、MSDNのドキュメントを参照してください）。

## ルートシグネチャ

ルートシグネチャは、グラフィックスパイプラインにバインドされているリソースの種類とリソースのレイアウト方法を定義します。ルートシグネチャは、API関数シグネチャに似ています。パラメータのタイプ、パラメータの順序、およびレイアウトについては述べていますが、実際のパラメータインスタンスは定義していません。ルートパラメータは、ルートシグネチャの要素に対応する実際のデータインスタンスです。サンプルの頂点シェーダは、シェーダ定数に対して単一の構造体しか必要としないため、ルートシグネチャは非常に単純です。ルートシグネチャには、ConstantBufferView型の単一のルートパラメータが含まれています。

## ジオメトリ

## シーンのジオメトリは、立方体の6つの四角形を表現している24の頂点のデータを含む静的頂点とインデックス配列で構成されています。2つの配列はSample::CreateDeviceDependentResourcesの内部で宣言されており、ID3D12Device::CreateCommittedResourceによって直ちに使用されてバッファのID3D12Resourcesが作成されます。簡潔にするために、サンプルではD3D12\_HEAP\_TYPE\_UPLOADを使用しています。これにより、単一ステップのデータで初期化しながら各リソースを作成できるようになります。ただし、\_UPLOADヒープはジオメトリデータにとって最適とは言えない場所です。より効率的な実装では、ジオメトリデータにD3D12\_HEAP\_TYPE\_DEFAULTを使用します。\_DEFAULTヒープを初期化するには、\_UPLOADヒープを使用する必要があり、2つのヒープを使用することになり、実装が複雑になります。

ジオメトリ用のバッファが作成されると、サンプルは頂点用にD3D12\_VERTEX\_BUFFER\_VIEW、インデックス用にD3D12\_INDEX\_BUFFER\_VIEWを作成できます。ID3D12GraphicsCommandList::IASetVertextBufferおよびID3D12GraphicsCommandList::IASetVertextBufferへの呼び出しを介して入力アセンブラを設定するときに、Sample::Renderでビューが使用されます。

## シェーダ定数の管理

この非常に単純なシーンでは、すべてのシェーダ定数は、次のものを含む単一の定数バッファにまとめられます。

* ワールド、ビュー、射影マトリックス
* 光の方向と色
* 単色

より複雑なシーンでは、定数が更新される頻度に応じて、通常、定数を複数のバッファに分割します。

大きな立方体と赤いライトはアニメーション化されているので、シェーダー定数は、描画呼び出しの間にフレームごとに複数回更新する必要があります。各描画呼び出しで使用するためにシェーダ定数のコピーを参照する必要があるタイプConstantBufferViewの単一のルートパラメータが含まれているルートシグネチャのリコールです。CPUとGPUは並行して動作するので、GPUがそれを使い終えるまでCPUは定数バッファを更新を試みてはいけません。定数バッファが1つしかない場合、GPUが描画を終了するまでCPUはブロックする必要があります。これは、定数を複数の描画呼び出しで更新する必要があるため、現実的ではありません。したがって、サンプルではいくつかの定数バッファを使用しているため、GPUの描画中に、CPUがGPUに定数を送信し続けることができます。

サンプルは単純であり、コンパイル時に知られているフレームあたり描画呼び出しを決められた数だけ有しています。このサンプルは、スワップチェーン内のバックバッファーの数を掛けたぶん、描画呼び出しごとに1つのバッファーを作成しています。この数は、CPUが書き込むための空きバッファが常にあることを保証します。すべての定数バッファは、Sample::CreateDeviceDependentResourcesで作成された単一の連続アップロードバッファに格納されています。アップロードバッファはCPUアドレス空間とGPUアドレス空間の両方のベースメモリアドレスを取得するために直ちにマッピングされます。

Sample::Renderメソッドでは、定数は、定数アップロードヒープのCPUベースアドレスからインデックスされた場所に書き込まれます。ID3D12GraphicsCommandList::SetGraphicsRootConstantBufferViewへの呼び出しを介して、バッファーをパイプラインにバインドするために同じインデックスがGPUアドレスと組み合わされます。インデックスは、描画数だけでなくバックバッファの頻度も考慮する必要があります。詳細はSample::Renderの実装をご覧ください。

## CPU/GPU の同期

CPU が GPU がコマンドリストを処理することができるよりも早く GPU にコマンドリストを発行することができる場合、結局 CPU は GPU が追いつくのを待つ必要があります。このサンプルはラウンドロビン方式でコンスタントバッファメモリを使用します。つまり、バッファスロットは一定期間後に再利用されます。一般に、共有リソースを再利用する前に、リソースが使用中でないことを確認するためにある種の同期戦略を使用することが重要です。 サンプルは CPU と GPU を同期させるためにID3D12Fenceを使用します。CPU は、フレームインデックス値を用いてフェンスオブジェクトを「知らせる」ために、コマンドをコマンドキューに挿入します。提供されたフレームインデックス値は、signalコマンドがGPUで処理されるとすぐにCPUから見えるようになります。これにより、ＣＰＵは、現在のフレームインデックスと、ＧＰＵによってシグナリングされた最後のフレームインデックスとを比較して、GPUがCPUと比較してどれだけ遅れているかを判断することができます。GPUフレーム数とCPUフレーム数の差がバックバッファの数を超えると、CPUは待機しなければならなくなります。

# 更新履歴

最初のリリース、2018 年 4 月