# もくじ


まえがき 006

サンプルについて 008

Chapter 1 プログラミングを始める前に 022

1.1 コンピュータグラフィックスとテクノロジーの発展 022
1.2 Direct3D概論 023
1.3 画面にポリゴンが表示されるまでの工程 025
1.3.1 大まかな流れ 025
1.3.2 描画コマンドの作成 026
1.3.3 レンダリングパイプライン 027
1.3.4 コンピュートパイプライン 030
1.4 Direct3D 12はどこが変わった？ 031
1.4.1 コマンドリストとコマンドキュー 031
1.4.2 パイプラインステート 031
1.4.3 ディスクリプタ 032
1.4.4 ルートシグニチャ 032
1.4.5 バンドル 033
1.4.6 インダイレクトコマンド 033
1.4.7 シェーダモデル 6.0 034
1.4.8 レイトレーシングパイプライン 035
1.4.9 可変レートシェーディング 037
1.4.10 メッシュシェーダパイプライン 038
1.4.11 サンプラーフィードバック 038
1.5 開発環境の構築 039
1.5.1 開発環境について 039
1.5.2 Visual Studioのインストール 039
1.5.3 グラフィックスツールの設定 041

Chapter 2 ウィンドウの表示 042

2.1 プロジェクトの作成 042
2.2 アプリケーションクラスの作成 045
2.2.1 Appクラスの実装 046
2.2.2 Appクラスの呼び出し 057
2.3 Direct3D 12の初期化処理 058
2.3.1 デバイスの生成 063
2.3.2 コマンドキューの生成 065
2.3.3 スワップチェインの生成 066
2.3.4 コマンドリストの生成 071
2.3.5 レンダーターゲットビューの生成 072
2.3.6 フェンスの生成 077
2.4 Direct3D 12の描画処理 079
2.4.1 コマンドの記録開始 080
2.4.2 書き込みのためのリソースバリア 080
2.4.3 レンダーターゲットビューの設定 083
2.4.4 表示のためのリソースバリア 085
2.4.5 コマンドの記録終了とコマンドリストの実行 085
2.4.6 表示処理 086
2.5 Direct3D 12の終了処理 088
2.6 リファクタリング 090
2.7 スマートポインタの使用 092
2.8 デバッグフラグの追加 094
2.9 メモリリークチェック 094

Chapter 3 ポリゴンの表示 096

3.1 シェーダの準備 097
3.1.1 シェーダファイルの作成 098
3.1.2 頂点シェーダの実装 099
3.1.3 ピクセルシェーダの実装 103
3.2 描画に必要なインタフェースの生成 104
3.3 頂点バッファの生成 109
3.3.1 頂点データの準備 109
3.3.2 リソースの生成 110
3.3.3 頂点データの書き込み 114
3.3.4 頂点バッファビューの設定 115
3.4 定数バッファの生成 115
3.4.1 定数バッファデータの定義 116
3.4.2 ディスクリプタヒープの生成 116
3.4.3 リソースの生成 116
3.4.4 定数バッファビューの生成 117
3.5 ルートシグニチャの生成 119
3.5.1 ルートパラメータの種類 120
3.5.2 ルートシグニチャの生成 122
3.6 グラフィックスパイプラインステートの生成 124
3.6.1 入力レイアウトの設定 125
3.6.2 ラスタライザーステートの設定 127
3.6.3 ブレンドステートの設定 129
3.6.4 シェーダの設定 132
3.6.5 パイプラインステートの生成 134
3.7 描画コマンドの生成 135
3.8 インデックスバッファ 138
3.8.1 インデックスバッファの生成 139
3.8.2 描画処理の変更 143
3.9 複数図形の描画 144
3.9.1 Zバッファ法 146
3.9.2 深度ステンシルビューの生成 147
3.9.3 深度ステンシルステートの設定 149
3.9.4 描画処理の変更 152

Chapter 4 テクスチャの表示 156

4.1 テクスチャとは？ 156
4.1.1 画像形式 156
4.1.2 テクスチャ座標 157
4.1.3 テクスチャアドレッシングモード 157
4.1.4 ミップマップ 159
4.1.5 テクスチャフィルタリング 160
4.2 DirectXTKライブラリの利用 162
4.2.1 DirectXTK12のインストール 162
4.2.2 テクスチャデータの準備 163
4.3 テクスチャの描画 166
4.3.1 シェーダの準備 167
4.3.2 頂点バッファの変更 171
4.3.3 ルートシグニチャの変更 172
4.3.4 テクスチャの生成 176
4.3.5 描画関数の変更 182

Chapter 5 メッシュの表示 184

5.1 assimpライブラリを利用したモデルの描画 184
5.1.1 assimpライブラリのインストール 184
5.1.2 メッシュファイルの準備 186
5.1.3 ファイルのロード処理 186
5.1.4 メッシュデータの変換処理 192
5.1.5 メッシュの描画 194
5.2 フレームワークの整備 196
5.2.1 ディスクリプタプールクラス 197
5.2.2 テクスチャクラス 207
5.2.3 定数バッファクラス 208
5.2.4 メッシュクラス 210
5.2.5 マテリアルクラス 210

Chapter 6 基本的なライティング 218

6.1 光と材質 218
6.2 Lambertモデル 219
6.3 Phongモデル 227
6.4 バンプマッピング 231
6.4.1 法線マップとは？ 232
6.4.2 法線マップの計算方法 232
6.4.3 接線空間への基底変換 234
6.4.4 バンプマッピングの実装 236
6.4.5 ワールド空間への基底変換 241

Chapter 7 光の数値化 244

7.1 立体角 244
7.2 光 246
7.3 放射分析学 246
7.3.1 放射エネルギー(radiant energy) 246
7.3.2 放射束(radiant flux) 247
7.3.3 放射照度(irradiance) 247
7.3.4 放射発散度(radiant exitance) 248
7.3.5 放射強度(radiant intensity) 248
7.3.6 放射輝度(radiance) 249
7.4 測光学 249
7.4.1 錐状体 250
7.4.2 分光視感度(spectral luminous efficacy) 250
7.4.3 光束(luminous flux) 251
7.4.4 光度(luminous intensity) 251
7.4.5 照度(illuminance) 252
7.4.6 光束発散度(luminous exitance) 252
7.4.7 輝度(luminance) 252
7.5 放射量と測光量の単位変換 252

Chapter 8 光伝達のモデル化 254

8.1 光の伝達を表現するには？ 254
8.2 どのぐらい光が出射される？ 255
8.2.1 双方向散乱面反射率分布関数 256

8.2.2 双方向反射率分布関数 256
8.3 BRDFモデル 257
8.3.1 Lambertモデル 257
8.3.2 Phongモデル 262
8.4 オフスペキュラー 265
8.4.1 フレネル反射 265
8.4.2 シャドウイングとマスキング 268
8.5 マイクロファセットに基づくBRDF 269
8.5.1 マイクロサーフェイス 270
8.5.2 マイクロサーフェイスの射影 271
8.5.3 マイクロファセットに基づくBRDFの導出 273
8.5.4 シャドウイング・マスキング関数 275
8.5.5 Cook-Torranceモデル 276
8.5.6 GGXモデル 279


Chapter 9

## 色を適切に扱う 282


9.1 色を表現する 282
9.1.1 混色 282
9.1.2 等色実験 283
9.1.3 CIE 1931 RGB表色系 284
9.1.4 CIE 1931 XYZ表色系 285
9.1.5 xy色度図 286
9.2 ディスプレイへの出力 287
9.2.1 ガンマ補正 287
9.2.2 伝達関数 289
9.2.3 色空間の変換 290
9.3 色の規格 292
9.3.1 IEC 61966-2-1規格(sRGB規格) 292





9.3.2 ITU-R BT.709勧告 293
9.3.3 ITU-R BT.2100勧告 293
9.4 トーンマッピング 296
9.4.1 Reinhardトーンマッピング 298
9.4.2 HDRディスプレイとトーンマッピング 299
9.4.3 Reinhardトーンマッピングの拡張 300
9.4.4 GTトーンマップ 301
9.5 プログラムの実装 303
9.5.1 HDRサポートのチェック 303
9.5.2 HDR画像の準備 307
9.5.3 レンダーターゲットのフォーマット変更 308
9.5.4 色域変換 308
9.5.5 トーンマッピングの適用 314
9.5.6 OETFの適用 316


Chapter 10

## ライトの種類と表現 318


10.1 光束と放射束の単位変換 319
10.2 パンクチュアルライト 320
10.2.1 ポイントライト 320
10.2.2 スポットライト 324
10.2.3 光の減衰 328
10.2.4 ディレクショナルライト 331
10.3 フォトメトリックライト 333
10.3.1 配光測定のための座標系 334
10.3.2 IESプロファイルの仕様 334
10.3.3 IESプロファイルの変換 338
10.3.4 IESプロファイルの適用 345
10.4 イメージベースドライト 347

10.4.1 逆変換法 348
10.4.2 モンテカルロ積分法 354
10.4.3 ライトプローブのフィルタリング 357
10.4.4 ライトプローブの評価 376

Chapter 11 メッシュシェーダパイプライン 384

11.1 メッシュシェーダとは？ 384
11.2 ポリゴンの描画 385
11.2.1 メッシュシェーダの実装 385
11.2.2 パイプラインステートの生成 396
11.2.3 描画コマンドの発行 403
11.3 メッシュレットを用いた描画 405
11.3.1 メッシュレットの生成 405
11.3.2 シェーダの実装 420
11.3.3 描画メソッドの修正 423

Appendix A デバッグの仕方 426

A1 基本的なデバッグ操作 426
A1.1 ブレークポイント 428
A1.2 データウォッチ 429
A1.3 ステップ実行 431
A1.4 特定の場所に移動する 432
A1.5 コールスタック 432
A1.6 メモリの表示 433
A1.7 データブレーク 433
A1.8 CRTデバッグライブラリ 434
A1.9 並列スタック／並列ウォッチ 435
A2 Direct3D 12のデバッグ 435
A2.1 デバッグレイヤーの有効化 436
A2.2 GPUクラッシュの解析 437
A3 PIX for Windowsを用いたグラフィックスデバッグ 439
A3.1 インストールとプログラムの起動方法 439
A3.2 GPUキャプチャー方法 440
A3.3 Pipelineビュー 442
A3.4 頂点シェーダのデバッグ 444
A3.5 ピクセルシェーダのデバッグ 447

Appendix B DirectX Agility SDKの使用 449

B1 Agility SDKの使用方法 449
B2 Agility SDKの動作について 450

あとがき 452
参考文献 456
索引 460

Chapter 1

# プログラミングを始める前に

この本を手に取られた読者は，グラフィックスプログラミングに興味がある方でしょう。もしかしたらゼロからグラフィックスプログラミングについて学びたいという方かもしれませんし，ある程度グラフィックスプログラミングは知っているけど，Direct3D 12は使ったことがないから学んでみたいという方かもしれません。

プログラマーは，プログラミングを行うにあたって，最初にプログラミング言語の文法やライブラリの使い方を学びます。グラフィックスプログラマーであれば，これらに加えて様々な表現を実現するためのアルゴリズムやテクニック，そしてDirect3Dなどのグラフィックスライブラリについての知識を身に着けておく必要があります。特に，ゲーム開発者を目指す学生や開発現場に配属されたばかりで経験が浅い社会人ならば，こうした基礎知識は「ゲームを作る」という実戦を経験する前に体得しておいた方が開発の際に色々と役立つ場面があるかもしれません。本書は，そのような方々のために基本事項を確認しながらDirect3D 12を利用してグラフィックスプログラミングについて学習するための本です。

## 1.1 コンピュータグラフィックスとテクノロジーの発展

Direct3Dプログラミングを始める前に，まずはコンピュータグラフィックスの歴史の移り変わりを抑えながら「Direct3D 12とはどのようなものか？」についての簡単な説明をしておきます。

コンピュータグラフィックス（CG）とは，コンピュータおよびその周辺機器を用いて作成された画像または映像を指します。今日，我々の生活ではパソコンやスマートフォンなど様々なコンピュータに触れる機会が多くなっています。読者の皆様の中にもテレビや映画を鑑賞する人や動画やゲームなどのコンテンツを楽しむ人がいるかと思いますが，それらも「コンピュータを通じて可視化を行う」コンピュータグラフィックスです。今やCGは非常に身近なものになってきたと言えるでしょう。

一口にコンピュータグラフィックスといっても色々な切り口があり，以下のようにいくつかの分類で分けることができます。

- 2次元か3次元か？
  2次元：平面画像に対するカラーフィルタなど
  3次元：3次元モデルに対するスケルタルアニメーションなど
- 対話性が求められるかどうか？
  対話性が求められる（リアルタイム）：ゲームなど
  対話性が求められない（オフライン）：映画やアニメなどの映像作品
- 写実的か非写実的か？
  写実的（フォトリアリスティック）：写真のようなリアルさを感じられるもの
  非写実的（ノンフォトリアリスティック）：絵画やアニメ調のようなもの

CGの採用事例が比較的多く見受けられるのは，コンピュータゲームと映画，アニメの3つです。ただし，アニメはCGを使わずにすべて手書きで作成しているものもあり，CGが必須の分野とは言えません。映画でも，最初から最後まで全てCGで作成されているというものは比較的少なく感じます。それに対して，現在のコンピュータゲーム開発とCGは非常に密接に関係しており，ゲーム開発をする上でCG技術の利用は避けて通れなくなっています。

さて，今やコンピュータグラフィックス技術が密接に関係するコンピュータゲームですが，筆者の目線では3DCG技術が盛んに使われるようになったのは，1990年代後半のPlayStation（PS1）の登場以降です。PS1登場前もポリゴンを用いて描画するゲームはゲームセンターのアーケードゲームをはじめいくつか存在したと思いますが，やはりPS1の登場によって一気にお茶の間に普及した印象を受けます。昔のコンピュータゲームと今のコンピュータゲームの画像を比べると，コンピュータグラフィックス技術の発展や進化が見て取れるでしょう。

テレビもまた進化を遂げました。昔は白黒のブラウン管でしたが，時代が経つにつれてカラーテレビになり，その後プラズマテレビや液晶テレビが登場します。また，解像度も昔と比べると大きく上がりました。ファミコン時代は240p（256×240ピクセル）で，今は1080p（1920×1080ピクセル）のフルHDが主流の時代です。最近の4Kテレビや8Kテレビの登場もあり，今後も解像度はより高くなると見込まれます。また，解像度が高くなる一方で色の表現力を高めるHDRテレビの登場により解像度だけではない新しい進化軸が表れ始めています。こうした表示機器の性能向上により，コンピュータゲームや映像作品の表現も変化してきています。より高い品質が求められる時代になり，今やごまかしが効きづらい時代へと突入しました。

ところで，コンピュータグラフィックスで高品質な画像や映像を生み出すためにはシミュレーションなどの計算に割り当てられる時間（処理時間）を増やす必要があります。勿論，割り当てられる処理時間は有限ですので，ある程度のごまかしは必要になります（ごまかし自体は悪いことではありません）。GPU（Graphics Processing Unit）の登場により，CPU以外での処理時間は昔に比べてかなり速くなりました。しかし，近年問題に挙げられることが多いのはCPUでの処理に要する時間の問題です。その中でもどうにもならないのがグラフィックスドライバーやAPI（Application Programing Interface）によるCPUのボトルネックで，ここはアプリケーション開発者が手を出すことはできない領域です。

そこでここ数年注目を浴びているのが，Direct3D 12やVulkan，Metalといったローレベルグラフィックス APIの登場です。これまでのDirect3DやOpenGLなどといったグラフィックスAPIは「パフォーマンスに影響があるGPUのキャッシュフラッシュを適切なタイミングで行う」など，グラフィックスドライバーが複雑な部分を隠蔽し開発しやすいようにしてくれていました。ただし，どのようなデータがどのタイミングでくるかはアプリケーションによって異なりグラフィックスドライバー側では全くわからないため，どのような場合でもキャッシュフラッシュできるように汎用性を重視した実装にする必要があります。そのため，やむをえないとはいえ無駄な処理が発生していました。

しかし，ローレベルグラフィックスAPIはこれまでドライバーがやっていた処理の一部をアプリケーション開発者側に担ってもらうことで，アプリケーションに合わせた適切なタイミングで処理を実行することが可能となり，CPUの処理時間の高速化につながる可能性が高くなります。一方で，開発者側の実装によってはこれまでもよりも遅くなる可能があるということに注意しなくてなりません。しかし，速くなる可能性があるのであれば，手を出したくなるのが人情というもの。本書籍ではローレベルグラフィックスAPIのひとつであるDirect3D 12を題材とし，コンピュータゲームなどに使われる"リアルタイムグラフィックスプログラミング"を中心にグラフィックスプログラミングを学んでいきます。

## 1.2 Direct3D概論

Direct3Dは，以下のようなWindows向けのマルチメディア処理用アプリケーションプログラミングインターフェイス（API）の集合である「DirectX」のひとつです。この時，DirectXはバージョンによって仕様が変わるため，バージョン12であればDirectX 12のようにどのバージョンであるかが分かるように末尾に明記するのが通例です。最近ではDirect3D以外の大きな更新がなく，更新されるのがDirect3Dのみということが多いので，"DirectX = Direct3D" として解釈されることもあります。

**表 1.1** DirectX

| 名称 | 用途 |
|---|---|
| Direct2D | 2次元描画用グラフィックスライブラリ |
| DirectWrite | フォント描画ライブラリ |
| Direct3D | 3次元描画用グラフィックスライブラリ |
| XAudio | オーディオライブラリ |
| XInput | XBox360コントローラーの入力を受け取るためのライブラリ |

Direct3Dはグラフィックスハードウェアを制御するグラフィックス描画処理を行うためのAPIです。DirectX 2（1996年）からWindows上での3Dグラフィックスがサポートされるようになりました。

そして，DirectX 8（2000年）からプログラマブルシェーダが登場します。DirectX 8以前までは固定

パイプラインと呼ばれる決められた動作しか行えませんでしたが，プログラマブルシェーダの登場により頂点シェーダやピクセルシェーダといった細かい範囲でGPUに命令を与えて様々な表現が実現できるようになりました。

初期のプログラマブルシェーダには，アセンブリ言語であったり，ピクセルに対して操作を行うピクセルシェーダが貧弱だったり，アルゴリズムによっては命令数の問題により実現できなかったりといくつかの制限がありましたが，現在では使える命令数の増加などにより大幅に軽減されています。その中でも，C言語ライクにシェーダを書けるようになる上位レベルシェーダ言語（HLSL：High Level Shader Language）の導入により，シェーダをコーディングしやすい環境が整いました。

シェーダには使用可能な命令数や機能が定められており，グラフィックスハードウェアの世代等やバージョンによって制限が異なります（この仕様のことをシェーダモデルといいます）。Direct3D 9（2002年）までのシェーダは，ポリゴンの各頂点に対して操作を行う頂点シェーダと，ピクセルに対して操作を行うピクセルシェーダの2つのシェーダステージ（シェーダによる処理段階）で構成されていました。また，それまでグラフィックスチップの仕様は製造元のメーカーによって異なり，場合によっては同じメーカーの機種でも上位モデルの機種と下位モデルでシェーダモデルが違ったり，演算精度にばらつきがあったり機能が異なったりということもあったようです。そのため，ハードウェア機能が異なる場合にプログラムをどう構成するかという問題（A社にはあるハードウェア機能の一部がB社のハードウェアでは使えない，B社ではサポートされているハードウェア機能がA社だと使えない，etc.）が発生していました[1]。

製品を作るうえではこうした問題は非常に厄介ですし，ハードウェア機能がサポートされていないためにせっかくお金を出して買ってくれたユーザーが不利益を被るようなことはあってはならない事態です。そのため，開発をする際は特定ハードウェアに依存せずにプログラムを実行できる，全社でサポートされている機能のみを使う，といった消極的選択をすることが増えました。ハードウェア機能によるパフォーマンス向上などのメリットなどが生まれた反面でせっかくの最新機能をユーザーが体験する機会を逃すというデメリットがあったといえるでしょう。

こうした問題を排除するため，Direct3D 10（2007年）では，どのDirect3D 10世代のグラフィックスチップを使っても全ての機能を利用でき，仕様の一貫性を保証する方針が打ち出されました。Direct3D 10では，第3のシェーダステージであるジオメトリシェーダの仕様追加によりシェーダモデルが4.0となり，これまでは実現できなかったポリゴン頂点の増減とプリミティブの増減が可能になりました。

そして，2009年にDirect3D 11が登場します。Direct3D 11での主な目玉はポリゴンの詳細度制御を行う「テッセレーション」を実現するためのハルシェーダとドメインシェーダが追加されたこと，GPUコンピューティングを実現するコンピュートシェーダの追加（DirectComputeの追加）によってシェーダの仕様変更（シェーダモデルが5.0になった）があったことなどが挙げられます。マルチスレッド処理対応やHDR（ハイダイナミックレンジ）圧縮といった新機能も追加されたDirect3D 11は，2020年の現在もいまだ幅広く使われています。

GPUの処理能力は昔と比べると非常に向上しました。DirectX 8が登場する前のNVIDIAが開発したGeForceシリーズ初の製品であるGeForce 256は，計算能力が50GFlopsでしたが，現在市場に投入されている最新版のGetForce RTX 3090の計算能力は35.7TFlopsとなっており，およそ714倍です。この数値からも分かるようにGPUの性能は年々どんどん向上しつつある一方で，CPU1コアあたりの性能向上は鈍くなってきており，現在はCPUのコア数が増える方向に向かっています。

DirectXが誕生した1990年代中盤は，各メーカー間の独自のGPUの機能差を吸収するための抽象化レイヤーを厚くする必要がありました。しかし，現在はハードウェアメーカーも片手で数えられる程度まで減っており，かつてのような分厚い抽象化レイヤーは不要になってきています。また，GPUの処理時間が短くなり処理数が増える一方で，CPU処理時間のボトルネックが問題になってきています。Direct3D 11では，マルチスレッド処理のためにDeferredContextというものが用意されています[2]。

こうした様々な時代背景から，2014年3月12日のGame Developers Conference（GDC）においてDirect3D 12が発表されました。

1　これの一昔前の例として，頂点シェーダで画像のピクセルデータを取り出す頂点テクスチャフェッチ（Vertex Texture Fetching）が挙げられます。

2　ただし，個人的な見解としては，色々と制限が多く，苦労の割には劇的に速くならず，マルチスレッドによる処理の高速化の期待に応えるものではありませんでした。



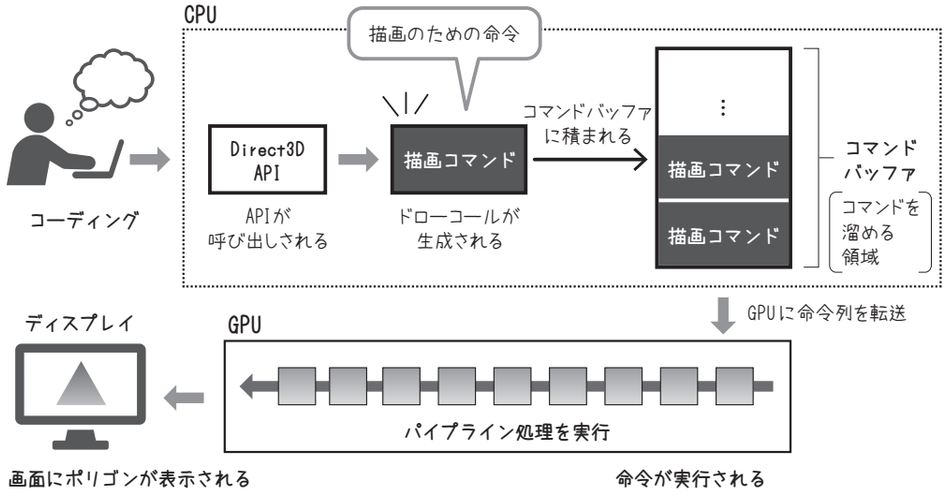


## 1.3 画面にポリゴンが表示されるまでの工程

Direct3D 12で何処が変わったかを具体的に説明する前に，まずは，ポリゴンが画面に表示されるまでの一連の処理工程（パイプライン）について確認し，理解を深めておきましょう。

### 1.3.1 大まかな流れ

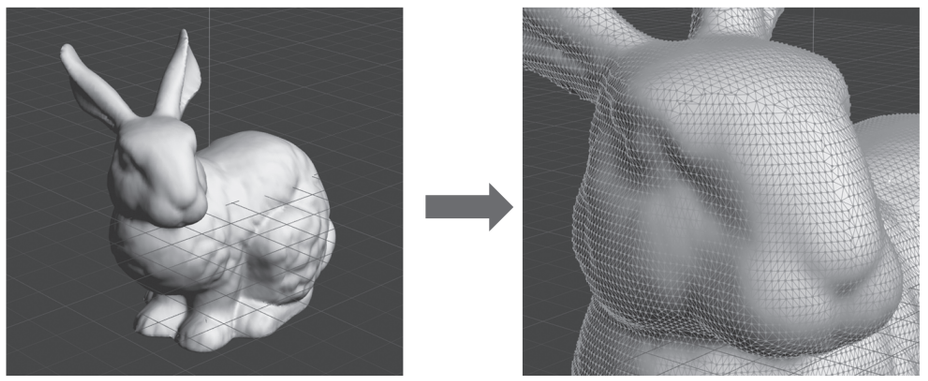
画面にポリゴンが表示されるまでの大まかな流れは図1.1のようになります。




▲ 図1.1　描画フロー

まずは，プログラマーがDirect3D 12のメソッドを呼び出すようにコーディングします。呼び出されたメソッドは，描画コマンドと呼ばれる形でコマンドバッファに一時的に格納され，ハードウェア（GPU）が解釈できる形の命令列に変換されます。そして，適切な実行タイミングでGPU側にコマンドバッファからの命令列が転送され，GPU側での処理が実行されます。このとき，GPU側は工場のベルトコンベアーのように決められた一連の処理を実行します。この一連の処理のことをパイプラインといいます（具体的なパイプライン処理については後述します）。パイプライン処理を経てPCディスプレイ上のピクセルの色が決まることで，最終的に我々が三角形のポリゴンを視認できるようになります。

なお，リアルタイム3Dコンピュータグラフィックスでは，描画の構成要素として三角形を使うことが多いです。これは，平面を構成できる最小の要素が三角形であることに由来します。複雑な幾何形状も三角形の集合体として表現します。



複雑なモデルも実は三角形ポリゴンとして表現されている

▲ 図1.2　ポリゴン表現

### 1.3.2 描画コマンドの作成

実際の描画コマンドは16進数で表現される命令になり，ハードウェアによって取り決めが異なります。例えば，「描画先を指定色で塗りつぶして，描画範囲を指定する」という描画コマンドを作る場合は下記のような感じになります。

```
// コマンドバッファ上の現在位置を取得
uint32_t* pCommandPtr = pCommandBufferHead;

// コマンドバッファに積んでいく
*pCommandPtr++ = 0x00041d91;
*pCommandPtr++ = 0xff112233;
*pCommandPtr++ = 0x00080a02;
*pCommandPtr++ = 0x07800000;
*pCommandPtr++ = 0x07080000;
```

こうした描画コマンドをコーディングする場合は，上記のように直接16進数を打ち込まずに，メソッドを利用して作成するのが一般的です。例えば，先ほどの「描画先を指定色で塗りつぶして，描画範囲を指定する」コマンドを作成するメソッドは下記のようになります。

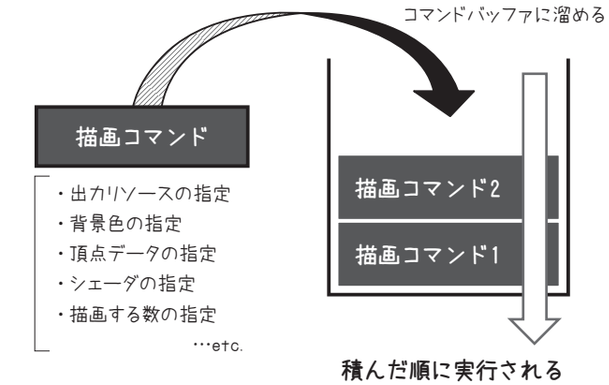
```
#define COMMAND_KIND_CLEAR_COLOR  0x00001d91
#define COMMAND_KIND_VIEWPORT     0x00000a02

// 背景色を塗りつぶす描画コマンドを生成
void MakeClearColorCommand(ICommandBuffer* pCommandBuffer, uint8_t r, uint8_t g, uint8_t b, uint8_t a)
{
    uint32_t* pCommandPtr = pCommandBuffer->GetHead();
    *pCommandPtr++ = COMMAND_KIND_CLEAR_COLOR | 1 << 18;
    *pCommandPtr++ = ((a & ff) << 24) | ((r & 0xff) << 16) | ((g & 0xff) << 8) | (b & 0xff);
    pCommandBuffer->SetHead(pCommandPtr);
}

// 描画範囲を指定する描画コマンドを生成
void MakeSetViewport(ICommandBuffer* pCommandBuffer, uint16_t width, uint16_t height)
{
    uint32_t* pCommandPtr = pCommandBuffer->GetHead();
    *pCommandPtr++ = COMMAND_KIND_VIEWPORT | 2 << 18;
    *pCommandPtr++ = width  << 16;
    *pCommandPtr++ = height << 16;
    pCommandBuffer->SetHead(pCommandPtr);
}
```

このようにアプリケーション作成のため用意されている窓口として動作するメソッドをAPI（Application Programing Interface）と呼びます。PC向けのグラフィックスAPIではDirect3Dをはじめ OpenGL，Vulkan，Metalなどが，スマートフォンや組み込み機器向けにはOpenGL ESなどが存在し，PS4やPS5，Switchのようなコンソールゲームのハードウェアには各社独自のグラフィックスAPIが存在します（それぞれできることはほぼ同じですが，APIが異なり方言のような癖があります）。実際のアプリケーションでは，上述したような背景色の指定や，描画先となる出力リソースの指定，ポリゴン描画のための頂点データの指定などといった描画コマンドをコマンドバッファに積んでいきます。

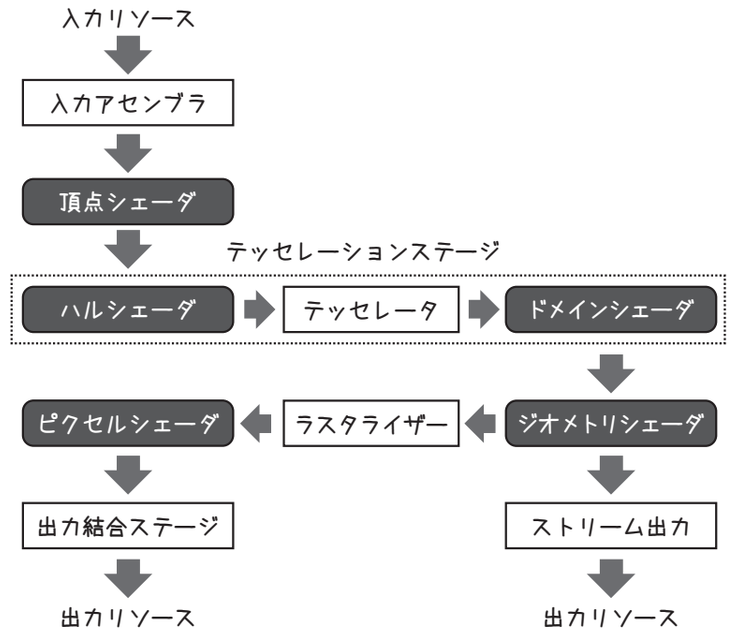





▲ 図 1.3　描画コマンド

### 1.3.3 レンダリングパイプライン

描画コマンドがGPU解釈可能な命令列として渡されると，GPU側での処理が実行されます。実際に積まれる描画コマンドによって多少動作が異なる場合がありますが，一般的には図1.4のように一連の流れによって処理されます。この各工程を通じて描画されるまでの流れを"レンダリングパイプライン"と呼びます。



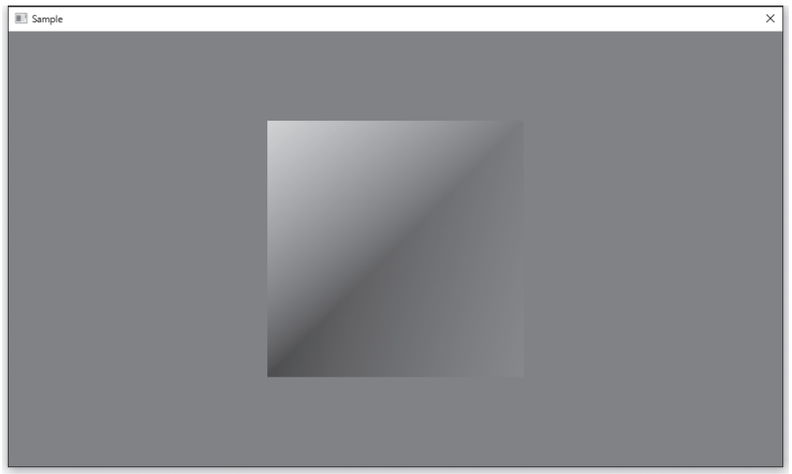

▲ 図 1.4　レンダリングパイプライン

このレンダリングパイプラインの中身は，固定機能のものとプログラマブルに処理できるものの2つに分けることができます。図1.4の白地に黒字で書かれているものが固定機能，黒地に白字で書かれているものがプログラマブルに処理できるものであり，これらの工程をGPU内の演算器（シェーダユニット）が並列計算することによってパイプライン処理が実行されます。それでは，各工程について個別に見ていきましょう。

#### 1.3.3.1 入力アセンブラ

入力アセンブラでは，入力リソースからデータを読み込み，頂点シェーダにデータを渡します。入力アセンブラに設定できるものは，頂点バッファ，インデックスバッファ，プリミティブトポロジーの3つで

きちんとデータが設定できていれば，次のような四角形が描画されるはずです。




▲ 図 3.12　四角形の描画

これで，インデックスバッファを使った描画についても学びました。

## 3.9　複数図形の描画

これまでは，1つの図形データの表示しかしませんでした。さらに，もう一歩ステップアップして2つ以上の図形を表示できるようにしてみましょう。ここでは簡単なサンプルとして，同一形状の四角形を別々の位置に2つ描画するサンプルを作ってみます。ポリゴンデータは同一なので，プログラムで変更する必要があるのは定数バッファのデータです。2つの四角形を描画したいので，ダブルバッファ分を含めて4つのディスクリプタを用意する必要があります。そのため，下記のようにNumDescriptorsを"2 * FrameCount"に変更して4つにします。

● リスト 3.23　App.cpp ／ App::OnInit() メソッドを修正

```
0587    // 定数バッファ用ディスクリプタヒープの生成.
0588    {
0589        D3D12_DESCRIPTOR_HEAP_DESC desc = {};
0590        desc.Type           = D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV;
0591        desc.NumDescriptors = 2 * FrameCount; // ★変更
0592        desc.Flags          = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;
0593        desc.NodeMask       = 0;
0594
0595        auto hr = m_pDevice->CreateDescriptorHeap(
0596            &desc,
0597            IID_PPV_ARGS(m_pHeapCBV.GetAddressOf()));
0598        if ( FAILED(hr) )
0599        { return false; }
0600    }
```

ディスクリプタ数の変更に伴い，定数バッファ用のリソース生成処理や定数バッファビューの設定処理を行うループ処理のループカウントを修正します。以下のように，2つの図形を描くため2倍に変更しておきます。

● リスト 3.24　App.cpp ／ App::OnInit() メソッドを修正

```
0628        for(auto i=0; i<FrameCount * 2; ++i)
```

四角形はそれぞれ別の場所に描画したいので，ワールド行列を次のように変更します。





● リスト 3.25　App.cpp ／ App::Render() メソッドを修正

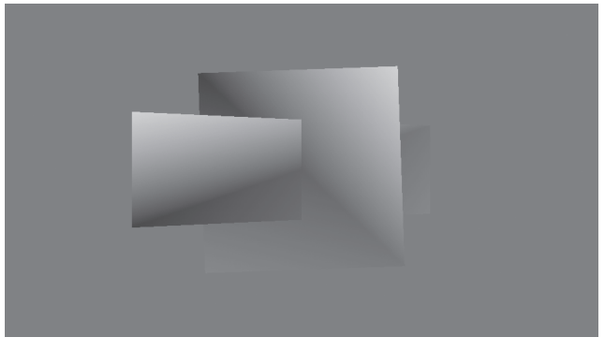
```
0880    // 更新処理.
0881    {
0882        m_RotateAngle += 0.025f;
0883        m_CBV[m_FrameIndex * 2 + 0].pBuffer->World =
0884            DirectX::XMMatrixRotationZ(m_RotateAngle + DirectX::XMConvertToRadians(45.0f));
0885        m_CBV[m_FrameIndex * 2 + 1].pBuffer->World =
0886            DirectX::XMMatrixRotationY(m_RotateAngle) * DirectX::XMMatrixScaling(2.0f, 0.5f, 1.0f);
0887    }
```

カメラの手前側の四角形はY軸周りに回転し，カメラの奥側の四角形はX軸周りに2倍の速度で回転するように設定してみました。あとは，描画する前にそれぞれ定数バッファビューをコマンドリストに設定すればよいです。実装は下記の通りです。

● リスト 3.26　App.cpp ／ App::Render() メソッドを修正

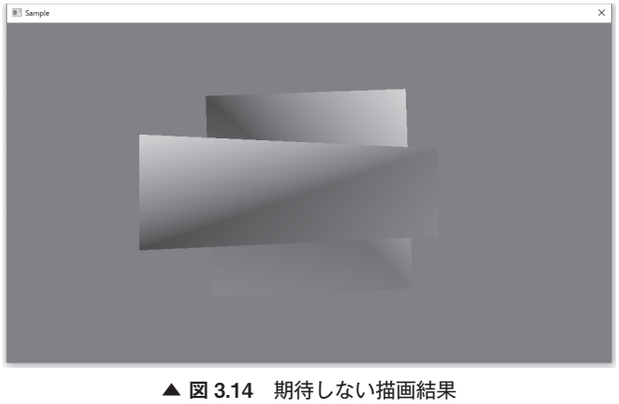
```
0929        // 手前側の三角形を描画.
0930        m_pCmdList->SetGraphicsRootConstantBufferView(0, m_CBV[m_FrameIndex * 2 + 0].Desc.BufferLocation);
0931        m_pCmdList->DrawIndexedInstanced(6, 1, 0, 0, 0);
0932
0933        // 奥側の三角形を描画.
0934        m_pCmdList->SetGraphicsRootConstantBufferView(0, m_CBV[m_FrameIndex * 2 + 1].Desc.BufferLocation);
0935        m_pCmdList->DrawIndexedInstanced(6, 1, 0, 0, 0);
```

期待する描画結果は，手前側がY軸周りに回転で，奥側がX軸周りに回転し，ポリゴンが突き刺さって見えるものです。



▲ 図 3.13　期待する描画結果

実際に画面に表示させてみましょう。



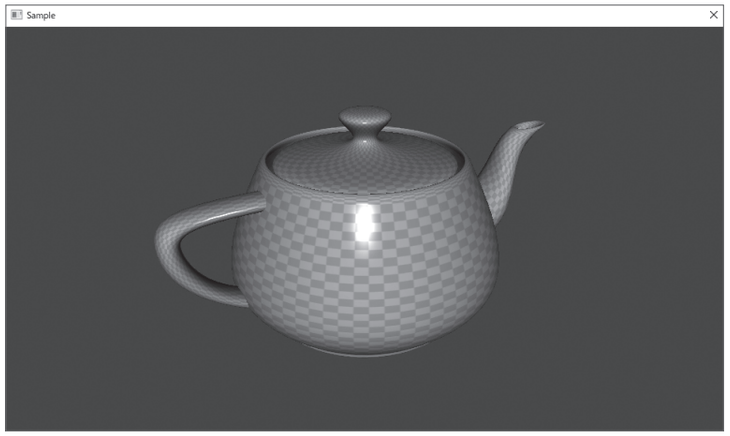

▲ 図 3.14　期待しない描画結果

Chapter 6

# 基本的なライティング

前章ではメッシュの表示を扱いましたが，メッシュ表示だけでは濃淡がないせいか全く立体的に見えないという問題があります。そこで，この章では単純な照明モデルを実装してライティングしてみます。ライティングすることで陰影付けが行われ，次のような見た目の変化が得られ，より魅力的な絵を作ることができるようになります。

▲ 図 6.1　ライティング無し




▲ 図 6.2　ライティング有り

## 6.1 光と材質

我々は昼間は太陽光の夜間は蛍光灯などのライトの下で，明かりを頼りにして暮らしています。夜に明かりを消すとどうなるでしょう？薄暗くなりますね。では，昼間に赤色に見えるものを，明かりが全くない真っ暗闇の状況で見たら赤く見えるでしょうか？真っ暗で色がほとんど分からないでしょう。つまり，色を知覚するには光が必要になります。物質が光を反射や吸収し，その結果の可視光が眼に届くことで我々は色を知覚できています。

光の反射や吸収の仕方というのは，材質の特性によって変わります。例えば，石膏や白墨（チョーク）などは，鏡や金属と比べると方向性を持った反射があまりなく，あらゆる方向へ反射します。このような反射のことを"拡散反射（Diffuse Reflection）"と呼びます。一方プラスチックや金属，鏡などは指向性を持った"鏡面反射（Specular Reflection）"と呼ばれる反射を行います。鏡面反射の特徴として，物質表面で光源が反射することにより明るく光る「ハイライト」が生じることが挙げられます。拡散反射の表現に用いられるもののひとつとしてLambertモデルが，鏡面反射の表現に用いられるものとしてPhongモデルがあります。それではそれぞれのモデルによるライティングを実装してみましょう。

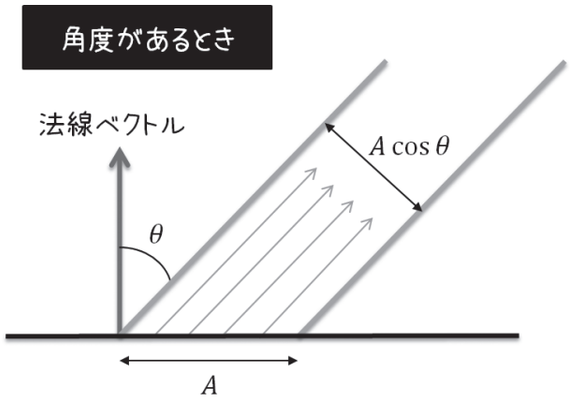




## 6.2 Lambertモデル

拡散反射の中で最も簡単なモデルは，完全拡散面（Lambert面）における反射モデルです。あらゆる方向への輝度が一様であるので，完全拡散面ではなく，均等拡散面として呼ぶこともあります。

さて，完全拡散面ではあらゆる方向に光が反射するので，光の入射方向と面の法線ベクトルのなす角度を $\theta$ として，面積に対してどれぐらい光が通過するかを考えてみます。下図のようにある面積 $A$ に対して，入射方向に垂直な面の面積は $A\cos\theta$ となります。当然ながら，面積が大きければ大きいほど光が通過する量が多くなります。つまり，光が通過する量は余弦に比例することになります。このように光の強さが入射角の余弦に比例することを「Lambertの余弦則」と言います。




▲ 図 6.3　Lambert の余弦則

この法則は1760年に出版されたJohann Heinrich Lambert著作の"Photometria"[Lambert 1760]という書籍に記載されているそうで，著者の名前から"Lambertの余弦則"と名付けられています（ちなみに"Photometria"はラテン語で書かれており，その後ドイツ語へ翻訳された要約版が1892年，フランス語への翻訳が1997年，英語への翻訳が2000年に行われているそうです。英語への翻訳は結構最近というのはとても意外です）。

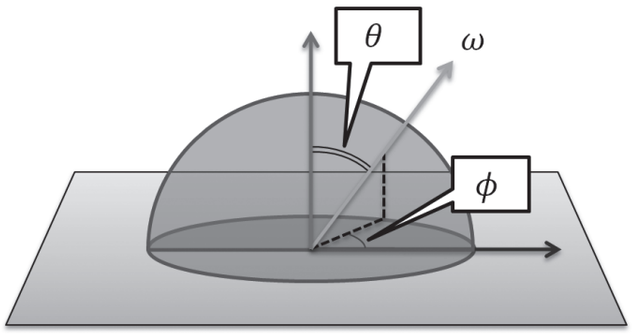
Lambertの余弦則は光の輝度についての法則です。拡散反射の強さが入射方向と面の法線ベクトルのなす角に比例すると仮定する反射モデルをLambertモデルと呼びます。Lambertモデルによる光の輝度を再現できれば，CGプログラムとして画面に表示できそうです。物質が光をどれくらい反射するかについては，反射率を定めてあげれば分かりそうです。我々がディスプレイを見たときに感じる色，つまり眼に届く光というのは，これまでのプログラムではRGB成分として指定してきました。そのため，光の反射率もRGB成分としてそれぞれ決めてあれば，反射光が求まりそうです。以上を踏まえて，Lambertモデルを数式として表すと次のようになります。

$$I = I_d k_d \cos\theta$$

$I_d$ は入射光の強さ， $k_d$ は物質の拡散光の反射率， $\theta$ は法線ベクトルと光の入射方向を示すベクトルのなす角とします。この数式を使ってプログラムを実装すればよいのですが，一点気を付けるべき所があります。それは $\cos\theta$ の値です。余弦の値は-1から1の範囲で，負の値は物質の裏側から光が入射したことを意味します。ただしゲームなどのリアルタイムレンダリングでは裏側からの入射は無視して表側からの入射のみを取り扱うことが多く見受けられるため，本書でもそれに従い余弦の値を[0, 1]の範囲に制限することで裏側からの入射を無視することにします。この点に注意して，ピクセルシェーダで拡散反射を

ところで，立体角 $\omega$ は球面座標の方位角 $\theta$ と天頂角 $\phi$ を用いて，次式で表すことができます（図8.6参照）。

$$d\omega = \sin\theta d\theta d\phi \tag{8.9}$$




▲ 図 8.6　$\omega$ をパラメータ表記する

半球なので $\theta$ は $[0, \frac{\pi}{2}]$ の範囲，$\phi$ は一周分なので，$[0, 2\pi]$ となりますから，式（8.8）と式（8.9）より以下のように表せます。

$$K\int_{\Omega}\cos\theta d\omega' = K\int_0^{2\pi}\int_0^{\frac{\pi}{2}}\cos\theta\sin\theta d\theta d\phi \tag{8.10}$$

$$= K\int_0^{2\pi}\left(\int_0^{\frac{\pi}{2}}\cos\theta\sin\theta d\theta\right)d\phi \tag{8.11}$$

$\phi$ の部分は定積分として表すことができるので，

$$\int_0^{2\pi} d\phi = [\phi]_0^{2\pi} = 2\pi - 0 = 2\pi \tag{8.12}$$

となるため，以下のように示すことができます。

$$K\int_0^{2\pi}\left(\int_0^{\frac{\pi}{2}}\cos\theta\sin\theta d\theta\right)d\phi = 2\pi K\int_0^{\frac{\pi}{2}}\cos\theta\sin\theta d\theta \tag{8.13}$$

ここで，$\cos\theta\sin\theta$ 部分に注目して，2倍角の公式を適用して解くと，次のように求まります。

$$\begin{aligned}2\pi K\int_0^{\frac{\pi}{2}}\cos\theta\sin\theta d\theta &= 2\pi K\int_0^{\frac{\pi}{2}}\frac{\sin 2\theta}{2}d\theta \\ &= \pi K\int_0^{\frac{\pi}{2}}\sin 2\theta d\theta \\ &= \pi K\left[-\frac{\cos 2\theta}{2}\right]_0^{\frac{\pi}{2}} \\ &= \frac{\pi K}{2}(-\cos\pi - (-\cos 0)) \\ &= \frac{\pi K}{2}\cdot 2 \\ &= \pi K\end{aligned} \tag{8.14}$$

計算してみると結果は $\pi K$ となりました。

ここで式（8.6）より，解いた結果が1未満である必要があります。このとき，物質表面に衝突した際にエネルギー欠損がなく，入射した光が全て反射したと仮定すると，これを満たす $K$ の値は以下のようになります。

$$K = \frac{1}{\pi} \tag{8.15}$$



以上より，$f_r = \frac{1}{\pi}$ となるので，エネルギー保存を考慮した正規化Lambertは次式で与えられます。

$$\frac{1}{\pi}k_d \tag{8.16}$$

式（8.7）では $\cos\theta$ を記載する形にしていましたが，この $\cos\theta$ はレンダリング方程式（式（8.2）参照）における $\cos\theta$ を表しているため，BRDFには含まれません。そのため，式（8.16）では $\cos\theta$ 項が無い点に注意してください。また $I_d$ に当たる部分は，レンダリング方式中の $L_i(x,\omega')$ 項に含まれるため省略していることにも注意してください。

では早速，式（8.16）を使って実際にライティングしてみましょう。まずは，使用するテクスチャとレンダーターゲットのフォーマットをsRGBフォーマットに変更しておきます（変更する理由については9章で説明します）。テクスチャはDirectXTKを使っているため，DDSファイルに変換する際に以下のようにsRGBフォーマットを指定します。

```
texconv.exe -f BC1_UNORM_SRGB SampleTexture.bmp
```

レンダーターゲットのフォーマット変更ですが，テクスチャを生成する際はSRGBなしで作成し，レンダーターゲットビューを作成する際にSRGB付きフォーマットで作成を行います。実装例は次のとおりです。

```
//--------------------------------------------------------------------------------
//        SRGBフォーマットへの変換.
//--------------------------------------------------------------------------------
DXGI_FORMAT ConvertToSRGB(DXGI_FORMAT format)
{
    DXGI_FORMAT result = format;
    switch(format)
    {
case DXGI_FORMAT_R8G8B8A8_UNORM:
    { result = DXGI_FORMAT_R8G8B8A8_UNORM_SRGB; }
    break;

case DXGI_FORMAT_BC1_UNORM:
    { result = DXGI_FORMAT_BC1_UNORM_SRGB; }
    break;

case DXGI_FORMAT_BC2_UNORM:
    { result = DXGI_FORMAT_BC2_UNORM_SRGB; }
    break;

case DXGI_FORMAT_BC3_UNORM:
    { result = DXGI_FORMAT_BC3_UNORM_SRGB; }
    break;

case DXGI_FORMAT_B8G8R8A8_UNORM:
    { result = DXGI_FORMAT_B8G8R8A8_UNORM_SRGB; }
    break;

case DXGI_FORMAT_B8G8R8X8_UNORM:
    { result = DXGI_FORMAT_B8G8R8X8_UNORM_SRGB; }
    break;

case DXGI_FORMAT_BC7_UNORM:
    { result = DXGI_FORMAT_BC7_UNORM_SRGB; }
    break;

default:
    break;
}
```