ESライブラリ＆＆
ゲームプログラミング

○　３Ｄ編 - 第１５回　パフォーマンスの最適化　○

パフォーマンスの最適化

・プログラムやデータを工夫することにより、動作をより高速に効率よくできる
・基本は無駄を省くこと。もっとも高速なのは「処理しない」「描画しない」こと

概要

アプリケーションの開発者にとって、パフォーマンス(動作速度、メモリ使用量など)を最適化することは、重要な部分といえます。より簡潔または高速なアルゴリズムを採用することにより、処理速度を向上させることができます。ゲームでは、描画処理の占める時間が大きいため、データや描画方法に工夫をすることによって、より綺麗な、または、より多くのオブジェクトを描画できるようになります。

開発初期段階からパフォーマンスの最適化に傾きすぎると、制限が多くなり開発が困難になってしまうため、方針のみ決めておき、最終段階で調整していくのが望ましいでしょう。

以下に、パフォーマンス最適化のためのガイドラインを紹介します。

Direct3D

・クリア処理は必要なときだけ行う
・ｚバッファとステンシルバッファが存在する場合は、両方を同時にクリアする
画面のクリア処理は、画面全体の描画処理のため、それなりのコストがかかります。クリアは、フレーム構築前に行いますが、画面全体を覆う画像が描画される場合は、クリアする必要がありません(ただし、ｚバッファやステンシルバッファのクリアは必要です)。
さらに、ｚバッファとステンシルバッファを同時に用いることが多くなっていますが、この２つは混合された同じ領域に存在するバッファのため、片方だけをクリアする処理はコストが高くつきます。特別なエフェクトでない限りは、両方同時にクリアするようにします。

・レンダリングステートの変更をなるべく行わない。同じステートのものは、グループ化して描画する
Direct3Dでは、レンダリングステートと呼ばれる描画に関わる設定を変更できますが、パイプラインの流れを変えるもののため、コストがかかります。１つのオブジェクトごとにステートを設定するよりは、同じステートを利用するものをグループ化してまとめて描画し、変更回数を減らすようにします。

・ライトの数は必要最小限にする
・ライトの速度は、ディレクショナル、スポット、ポイントライトの順に速い
・ライトの範囲外にあるものは処理しないようにする
ライトの数が増えるほど、比例して頂点色の計算式が増えていきます。見えない部分にあるライトや遠すぎてほとんど影響を与えないライトは、レンダリング時に使用しないように設定した方が効率がよくなります。

スプライト描画

・画像のサイズを２の乗数にする
・画像のサイズをなるべく小さくする
・画像を１枚にまとめる(テクスチャの切り替えを少なくする)
・不透明の画像は手前から描画する
Direct3Dでは、画像をテクスチャで扱います。テクスチャは、ほとんどの環境で２の乗数のサイズしか扱えません。また、サイズが大きくなるとピクセルの転送量が増えます。はじめから２の乗数のなるべく小さなサイズで画像を作成しておきます。また、テクスチャの切り換えは、レンダリングステートの切り換えと同じようにコストがかかる処理なので、１枚の画像に詰め込んだ方が、切り替えが少なくなり、より早く処理できるようになります。また、「早期ｚカリング」という奥のピクセル処理が行われなくなる機能があるので、不透明なスプライトは手前から描画するようにします。

バーテックスバッファ

・バーテックスバッファの切り替えをできるだけ減らす
・非システムメモリのバーテックスバッファの書き換えをできるだけ行わない
・非システムメモリのバーテックスバッファのロックを少なくする
・同じフォーマットの頂点データで変形不要なものは、１つのバーテックスバッファにまとめて扱う
・１回の描画で多くの頂点を描画した方がパフォーマンスは向上しやすいが、多すぎると逆に頂点キャッシュの効率が悪くなりパフォーマンスが低下するので、ある程度のサイズで分割する
・三角形ストリップを用いる。離れた位置の三角形は、縮退三角形を定義する

頂点はできるだけ多く一度に描画した方が高速です。ただし、頂点キャッシュを超えたサイズを一度に描画すると、キャッシュ効率が低下してしまう場合があります。よく使う頂点はバッファの前の方におく、バーテックスバッファをキャッシュを考慮したサイズにする、など工夫をすると効率が上がります。また、ほとんどの環境では頂点の再利用性を上げるため、内部で三角形を「三角形ストリップ」に変換しています。はじめから三角形ストリップで作成しておけば、変換処理を省くことができるので、描画速度が向上する場合があります。ストリップは連続した三角形になりますが、離れた位置にある三角形は、２つの頂点が同じ位置にある面積０の「縮退三角形」を用いると定義できます。

また、グラフィックボード上のメモリにあるバーテックスバッファは、CPUが直接アクセスできない領域になります。そのため、ロックのような直接的なアクセスをする関数は、システムメモリにコピーを作る、書き換えたデータをGPUに転送するなどといったオーバーヘッドが発生します。書き換えを多くしなければならないデータは、システムメモリに作成するようにします。

メッシュ

・もっとも高速なのは「描画しない」こと
　不透明なものは手前から描画する
　背面カリングを行っても破綻しないモデルは背面カリングを行う
　視点からの距離によってモデルのポリゴン数を削減する
　明らかに視界からはずれているモデルは描画しない
・メッシュの最適化を行うと、効率よく描画できるように向上するようにデータを再設定する
・非システムメモリに作成されたメッシュのロックを少なくする

メッシュは内部でインデックス付きバーテックスバッファを使用しているため、バーテックスバッファと同じ制限を受けます。D3DXIntersect関数は、内部でロックを行っているため、多用すると動作が重くなる場合があります。レイとの判定を多用する場合は、地形判定用の簡略化されたモデルを用意し、システムメモリに読み込むようにします。また、スプライトと同様に、不透明なモデルは手前から描画していきます。そうすれば、奥のモデルのピクセルは描画そのものがされないため、パフォーマンス向上につながります。

もっとも高速なのは「描画しない」ことなので、視界外のモデルは描画処理を省く、描画するものはLODや背面カリングなどで描画数を減らす工夫をします。

ESライブラリでの対応

ESライブラリでは、パフォーマンスの改善に役立つ以下の機能を提供しています。

SPRITE型(ISpriteクラス)	描画プール機能により、スプライトの描画を１回にまとめてできます
MODEL 型(IModel クラス)	システムメモリにメッシュの読み込みができます また、Optimize関数を呼び出すと、効率よく描画できるように、頂点の再構築を行います。さらにSetLOD関数により、面の数を削減できます
ANIMATIONMODEL型 (IAnimationModelクラス)	MODEL型と同じく、Optimize関数とSetLOD関数がサポートされます
ViewFrustum構造体	視錐台構造体です。点および境界球との衝突判定ができます