高浦 二三康

2013年3月

EClay

～CUDAを用いたオリジナル３Dレンダリングエンジン～

ソースコードは https://github.com/fumiyasutakaura/EClay よりダウンロード出来ます。

✤EClayとは？

✦コンセプト

近年、主要な3Dレンダリングライブラリとして OpenGLやDirectXが挙げられる。これらには最適化されて高速かつ高機能な機能が多数含まれており、その進化のスピードはハードウェアの進化のスピードを上回っているように思える。

3Dをよりリアルにレンダリングする手法に関しても様々に考案され、現在のハードウェア上で実行するには計算スピードが追いつかないようなものまである。

高速に動き、実用に耐えうるアプリケーションを作成するためのライブラリにするためには、その時点での主流なハードウェアの計算能力を超えるような高度なアイディアは制限する必要があり、例えばOpenGLES1.0ではシェーダーは使えず、固定機能パイプラインを利用する。OpenGLES2.0ではバーテックスシェーダとフラグメントシェーダを使うことはできるが、ジオメトリシェーダの利用は見送られている。

このようにアイディアがあり、プログラムとして実装可能なものであっても、処理速度が追いつかないものであれば実装は見送られてしまう。よって高度な次世代の3Dレンダリングのアルゴリズムを先取りするためには、実用的で広く利用されることを目的の一つとしたライブラリには頼らず、１からレンダリングの仕組みを作る必要がある。

EClayでは頂点データの読み込みからジオメトリパプラインをたどり、最終的にピクセルデータに出力されるまでのすべてを独自の実装で実現した。CUDAの動くハードウェア・OSであればどんな環境でも実行することができる。

✦名前の由来

E:電子 と Clay:粘土 の組み合わせで、「電子粘土」という意味の造語。「イークレイ」と読む。粘土のように自由に形作れるようにとの思いが込められている。

✦独自性

ジオメトリーパイプラインに関わる演算は既存の3Dレンダリングライブラリに頼らずに全て独自で行なっている。そのため、最新のレンダリングアルゴリズムの導入をライブラリの制限無しに行える。

✦クロスプラットフォーム

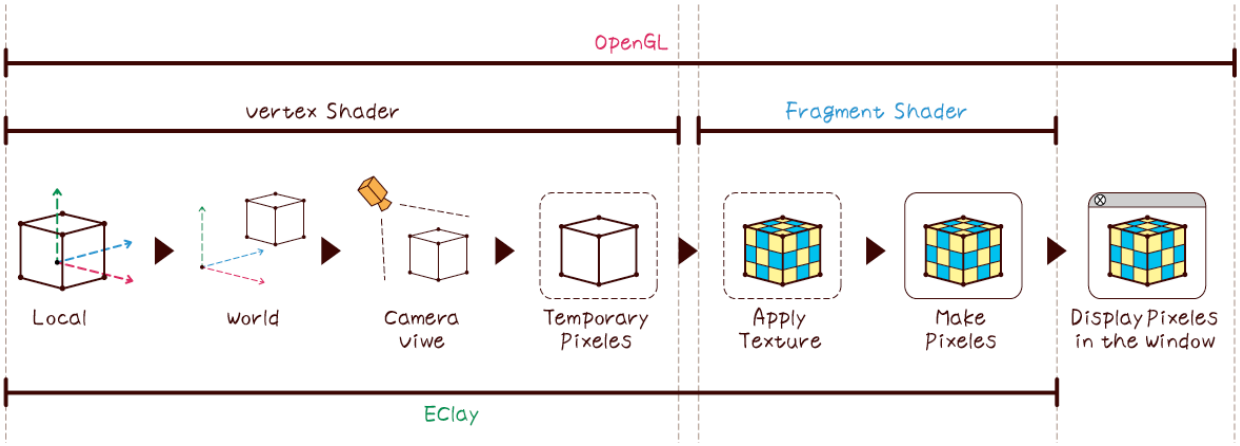
SDKの開発はMacを中心に、IDEはXcodeを使って開発している。

ただし、個別のOSに依存したライブラリ等の使用は避けており、Mac・Linux・Windowsその他CUDAの動く環境であればどのプラットフォームでも実行可能。今後CUDAの動くAndroidデバイスが登場することにも期待している。

専用のMakefileをXcodeにバインドする形でビルドしている。MakefileはOSを判定してビルドロジックを個別に設定できる。

2013年3月時点では Yellow Dog Linux for CUDA 6.3 にて初期設定のままビルドが成功することを確認している。(動作未確認)

✦OpenGLとEClayのジオメトリパイプラインの比較



頂点データの読み込みからピクセルへの出力まですべて独自の実装を行なっている。ただし、出力されたピクセルのウインドウへの表示はOpenGLのglDrawPixels()関数を使用している。この関数の呼び出しはSDK側でラップされているのでユーザーが直接触れることはない。

✤SDK構成

Application

ユーザーがアプリケーションを作成するための手助けをするクラス群。モデルファイルの読み込み・操作に必要なECModelクラスと、それをレンダリングするためのECCanvasクラスが含まれる。

Smart Pointer

アロケートしたメモリの解放を自動的に行うスマートポインタクラス。

C++のdelete演算子やCUDAのcudaFree()関数の呼び出しを自動で行うのでメモリリークを防ぐことができる。

CUDA Helper

ホスト側から呼び出すCUDAの関数をラップした関数群。

ホスト側のCUDA関数を呼び出す際は必ずここを経由する。

Mathematics

3D演算のための数学クラス群。2D・3D・4Dベクトル、4x4マトリックス(行列)、クオータニオン クラスが含まれる。

それぞれCPU、GPU用クラスが用意されている。

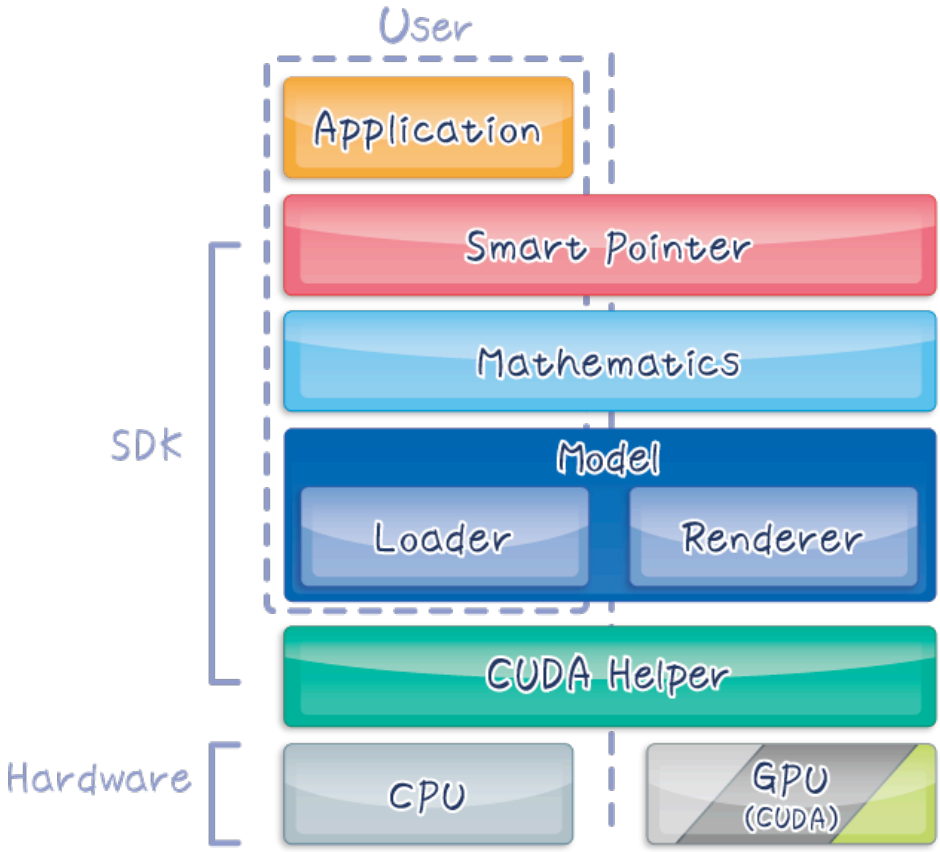
Renderer

モデルファイルに定義された頂点群を、ジオメトリーパイプラインを通じ、ピクセル化するためのレンダリングを担当する。

etc

ユーティリティー関数等が含まれる。

✤SDKレイヤー



✤SDK詳細

✦Application

3Dアプリケーションを作成するために、モデルファイルの読み込み機能と、複雑なレンダリングの仕組みをラップして、容易な手順でプログラミングを行えるようにする必要がある。それを実現するため、モデルファイルの読み

込み・操作に必要なECModelクラスと、モデルをレンダリングするための
ECCanvasクラスを用意した。

•ECModel
　モデルファイルを読み込み、頂点群をGPUメモリ(CUDAのグローバル
メモリ)に格納する。ECCanvasを継承したアプリケーションコード内で
扱うもので、ECModel内のレンダリング命令を呼び出すと自動的に
ECPixelBufferのインスタンス上にレンダリングされる。実際にはモデル
ファイルの形式ごと(WaveFrontObject・DirectXModel・fbx・collada等)に
継承して実装されたものを使用する。2013年3月時点でWaveFrontObject
形式に対応したECWaveFrontObjectが実装され、DirectXModel形式には
2013年3月時点で対応中である。その他fbx・collada等は今後の課題とな
る。ECModel.cu内にはモデルをレンダリングするためのカーネル関数が
実装されているが、頂点を処理するフェーズと処理された頂点をピクセル
化するフェーズにわかれている。これはOpenGLやDirectXのバーテックス
シェーダ・ピクセル(フラグメント)シェーダとよく似ており、同じような
レンダリングパイプラインの概念が導入されている。

•ECCanvas
　ユーザーが3Dオブジェクトを描くためのキャンバスとなるクラス。ア
プリケーションを作成する場合はこのECCanvasクラスを継承してプログ
ラミングすることになる。最終的にウインドウに表示されるピクセル群を
管理する ECPixelBuffer へのスマートポインタをメンバ変数に持ち、レン
ダリングに関する操作はここで行うことができる。具体的には純粋仮想関
数であるonInit()・onUpdate()・onDraw()の３つの関数を実装することで
アプリを作成する。
　•onInit()
　　初期化時に呼ばれる。このメソッド内でECModelクラスのイン
　スタンス等、個々のパラメータの初期化はここで行う。

　•onUpdate()
　　フレーム更新時に呼ばれる。
　　モデルの回転・拡大縮小・座標移動、カメラの回転・移動等は
　ここで行う。

　•onDraw()
　　ピクセルがレンダリングされる際に呼ばれる。
　　onUpdate()とは別スレッドで呼ばれるので、レンダリングに関わ
　る処理が思い時には単位時間あたりに呼ばれる回数は減る。
　　レンダリングしたいモデルインスタンスに対してレンダリング命
　令はここで呼び出す。

✦Smart Pointer

　参照カウント方式のスマートポインタを用意した。メモリアロケーション
や、代入演算子等で自身のポインタへの参照元が増えた時に１づつ増やし、ブ

ロックを抜けてスタックメモリが開放された時・オーナーインスタンスが解放された時等、自身のポインタへの参照が切れた時に参照カウントを１づつ減らす。参照カウントが０になった時にメモリの解放を行う。これにより、メモリ管理が容易になり、メモリリークのミスを防ぐことができる。

また、関数内で返り値としてアロケートしたオブジェクトの解放命令を、関数の外でユーザーが書く必要がないので、SDKの簡潔さを保つことができる。ユーザー自身が明示的に解放しなくてはいけない生ポインタを返す関数はSDK内に存在しない。

•C++用スマートポインタ : ECSmtPtr

通常、C++においてヒープ領域にint型のインスタンスを確保するには、

int* a = new int(123);

のようなコードを書く。これは、関数内のブロック内であれば、そのブロックを抜ける前に

delete a;

と書かなくてはいけない。

また、int* a;をクラス内のメンバ変数として持っているならば、そのクラスのデストラクタ内に同様のコードを書かなくてはいけない。

ECSmtPtrを使えば、

ECSmtPtr<int> a = new int(123);

と書くことで、上記と同様にヒープ領域にint型のインスタンスが確保され、参照カウンタによって自動的にメモリ管理が行われるので、delete演算子をアプリケーションコードの中に書く必要はない。

•CUDA用スマートポインタ : ECSmtDevPtr

ECSmtPtrをCUDA用にカスタマイズしたもの。ECSmtPtrではヒープ領域にインスタンスを生成するところを、ECSmtDevPtrではGPU内のグローバルメモリにインスタンスを生成する。ECSmtDevPtrクラスの持つ参照カウンタ変数はホスト上に確保されるため、メモリ管理はホスト上でECSmtPtrと同様に行われる。cudaMalloc()、cudaFree()、cudaMemcpy()などのCUDA上に実装されたメモリ関連の関数はECCudaOperator.cu(CUDA Helper参照)でラップされ、ECSmtDevPtr内部で呼び出されるため、ユーザーがアプリケーションコード内で直接呼び出す必要はない。(コードの簡潔化のため推奨されない。) ECSmtDevPtrによって生成したインスタンスのポインタをCUDAのカーネル関数に渡したい場合は、getPtr()メソッドを使用する。

✦CUDA Helper

CUDA上で用意されたホスト側の関数は ECCudaOperator.cu 内でラップしてある。CUDAの関数の呼び出しを一箇所にまとめることで、CUDAへの依存

が多岐に渡ることを防いだ。これによりSDKコード・アプリケーションコードの汎用性を高く保つことができる。

cudaMalloc()・cudaFree()・cudaMemcpy()などの関数はSDKコード・アプリケーションコード内で直接呼び出すことは避け、すべてECCudaOperator.cu内の実装を経由するようにしている。

役割を明確にするため、ECCudaOperator.cu 内のラッパー関数の頭には "gpu" と付けるよう命名規則を設けている。

✦Mathematics

3D演算に必要な、2D・3D・4Dベクトル、4x4マトリックス(行列)、クオータニオンのそれぞれのクラスを実装した。CPU上・GPU上の演算に対応している。

•ベクトル

2D・3D・4Dに対応している。2Dは主にテクスチャのuv座標に、3Dは頂点の位置・法線に、4Dは頂点カラーを表すRGBA値にそれぞれ利用される。

マトリックス・クオータニオンとの乗算により線形変換がなされる。

•マトリックス

4x4マトリックス。（3D演算においてほとんどのマトリックス演算は4x4マトリックスで表現できるため、4以外のnxnマトリックスやnxmマトリックスは用意していない。）

平行移動・拡大縮小・回転を表し、ベクトルと乗算することにより線形変換を行うことができる。

OpenGLと同様に右手系の座標を採用している。（DirectXは左手系）ただし、OpenGL形式のマトリックスの転置マトリックスを採用しているため、ベクトルとの乗算においては掛ける順番がOpenGL形式とは異なる。

$$\begin{pmatrix} x' \\ y' \\ z' \\ l \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ l \end{pmatrix} \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix}$$

OpenGLのMatrix

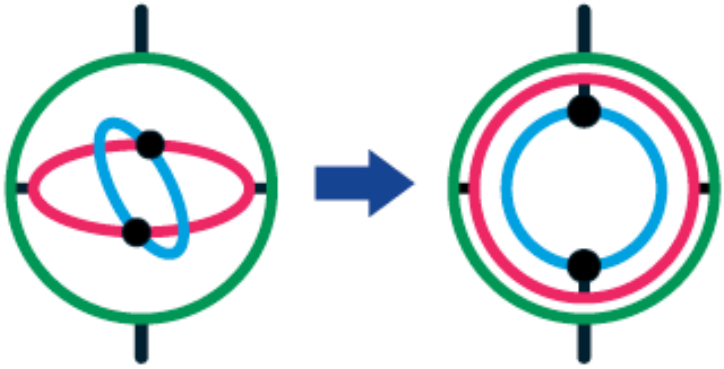
$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} a & e & i & m \\ b & f & j & n \\ c & g & k & o \\ d & h & l & p \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

EClayのMatrix

•クオータニオン

回転を表し、ベクトルやマトリックスと乗算することができる。マトリックスは16個のfloatパラメータを持つが、クオータニオンは4個のfloatパラメータで回転を表現するため、メモリの節約になる。

また、マトリックスの回転はx・y・z軸の３つのベクトルそれぞれを軸に何度回転するかを表すため、ジンバルロック(下図)が起こることがあるが、クオータニオンの場合は１つのベクトルを中心に何度回転するかを表すため、ジンバルロックは起こらず、マトリックスよりも正確に回転させることができる。



※ジンバルロックとは

マトリックスのように回転の軸が複数ある場合に、軸同士が重なることで起こる現象。例えばXYZ軸の順で回転させる場合に、Y軸を90度回転させるとX軸とZ軸が重なる。よって事実上2軸のみの回転しかできず、3軸の回転として表現できない姿勢が発生してしまう。クオータニオンは回転の軸が1本のみであるのでこの現象は起こらない。

✦Renderer

モデルファイルに定義された頂点群をジオメトリーパイプラインを通じてピクセル化するためのレンダリングを担当するクラス。

•ECRenderState

カメラの視点を表すビューマトリックス、各頂点を遠小近大に線形変換して3Dらしさを引き出すためのプロジェクションマトリックス、最終的にスクリーン座標に変換するためのスクリーンマトリックスを一元管理する。

•ECPixelBuffer

ウインドウに表示するためのピクセル群を管理するクラス。各ピクセルはRGBA値を持ち、0.0f~1.0fの値のfloat型で表現される。

✤レンダリングの仕組み

✦シェーダライクのカーネル関数

3Dプログラミングの特性上、大量の頂点とピクセルを処理する必要がある。同じ型のデータを一度に大量に処理できることを強みとするCUDAにおいては、頂点(バーテックス)用カーネル関数とピクセル(フラグメント)用カーネル関数に分けるとスマートかつ高速なコードを書くことができる。それぞれのカーネル関数は、OpenGLやDirectXにおけるバーテックスシェーダ・ピクセルシェーダのコードと非常によく似た構造になり、ジオメトリパイプラインの概念をそのまま踏襲することができる。

✦頂点の概念

EClaySDKには頂点を扱うクラスとしてECDevVertexが用意されている。

EClaySDKにおける頂点の要素には位置だけではなく、法線・UV座標・頂点カラーが含まれている。これらはジオメトリパイプライン中のピクセルシェーディングのフェーズにおいて、各ピクセルの最終的な色を決定する際に必要となるものである。

この他ボーンアニメーション(スキンメッシュアニメーションとも呼ばれる。階層的に連なったマトリックスと、各頂点に割り振られた0.0f~1.0fのウエイトの値を用いて頂点の位置を演算する手法。人間や動物のモデルのアニメーションを表現するのに用いられる。)を用いる場合はウエイトの値を追加する事になる。

•位置

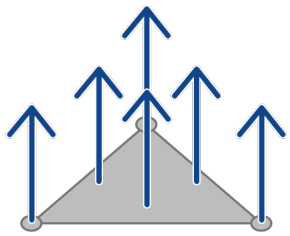
頂点の位置を3Dベクトルで表している。

ただし、CUDAのメモリコアレッシングを発生させてパフォーマンスアップを図るため、実際にはfloat4型が使われている。法線・UV座標・頂点カラーも同様である。

•法線

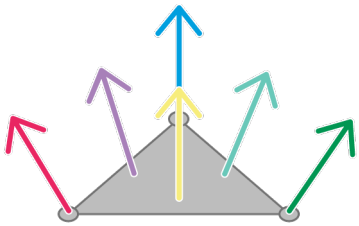
各頂点の垂直方向の単位ベクトルを表す値である。

プリミティブ(三角形)を3D空間上にレンダリングする際、そのプリミティブを含む平面の法線を各ピクセルにおいて採用(下図)し、ライトの方向ベクトルと掛けあわせて明度を計算する方法をグーローシェーディングと呼ぶ。



この場合、そのプリミティブの各ピクセルの明度はすべて同じ値になる。

これを改良し、プリミティブの各頂点のそれぞれ異なった方向の法線ベクトルを採用し、頂点以外のピクセルにおいては補間計算(「ピクセル(フラグメント)シェーディング」の「プリミティブ補間処理」にて説明)したものを法線として採用(下図)することによって、そのプリミティブの各ピクセルは滑らかにグラデーションされた明度となり、よりリアルな表現が可能となる。これをフォンシェーディングと呼ぶ。



同じプリミティブ上で各頂点ごとに異なる法線を与えるのはフォンシェーディングを用いるためである。

•UV座標

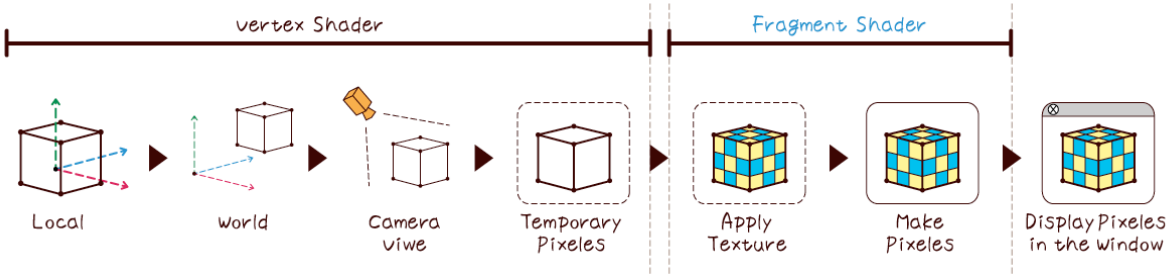
2Dのテクスチャ空間上からRGBA値を拾うための2D座標である。float4のうち、xはu、yはvを表すが、zとwはパースペクティブコレクション(「バーテックスシェーディング」の「パースペクティブコレクション」で説明)の演算に使われる。

•頂点カラー

カラーテクスチャが存在しない場合は頂点カラーを採用する。float4のxyzwの値はそれぞれrgbaに相当する。

✦ジオメトリパイプライン

頂点群のデータをピクセル化するまでの流れ。頂点を処理するバーテックスシェーディングとピクセルを処理するピクセルシェーディングから成る。

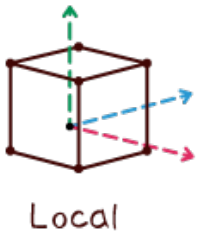


✦バーテックスシェーディング

各頂点ごとの処理。頂点の位置・法線の方向の変換等を主に行う。

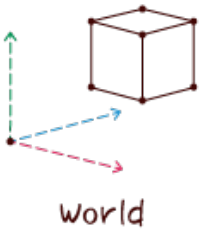
•ローカル座標系における処理

ローカル座標系における頂点の位置の移動などを行う。ボーンアニメーションはここで演算する。



•ワールド座標系に変換

ローカル座標系での処理後、線形変換用のマトリックスを用いて任意の座標・姿勢・大きさへの平行移動・回転・拡大縮小の処理をする。



$$\begin{pmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
平行移動
マトリックス

$$\begin{pmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
拡大縮小
マトリックス

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
X軸回転マトリックス

$$\begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
Y軸回転マトリックス

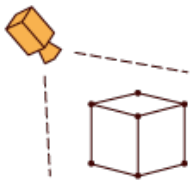
$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
Z軸回転マトリックス

回転はマトリックスだけでなく、クオータニオンを用いることでも演算できる。

$$Q(\theta, \overrightarrow{Axis}) = \left[\cos\frac{\theta}{2}, \sin\frac{\theta}{2} \times \frac{\overrightarrow{Axis}}{|\overrightarrow{Axis}|} \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \sin\frac{\theta}{2} \times \frac{\overrightarrow{Axis}}{|\overrightarrow{Axis}|} \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \sin\frac{\theta}{2} \times \frac{\overrightarrow{Axis}}{|\overrightarrow{Axis}|} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \right]$$

•ビュー座標系に変換

各モデルの頂点をワールド座標系に変換後、カメラから見た座標系に変換する。カメラの位置を原点、方向を右手系座標における(0,0,-1)方向に変換し、相対的に各頂点を移動・回転する。



Camera
viwe

•プロジェクション座標変換

ビュー座標系に変換した頂点群を見てもそのままでは平行投影となり、2Dの映像にしか見えない。そこで遠近法を各頂点に適用することによって3Dらしさを引き出す。遠近法はプロジェクションマトリックスを頂点に掛け合わせることで実現できる。通常、視野角θは人間の目に近い60度という値が採用される。

$$\begin{pmatrix} \frac{screenHeight}{screenWidth} \times \frac{-1}{\tan\frac{\theta}{2}} & 0 & 0 & 0 \\ 0 & \frac{-1}{\tan\frac{\theta}{2}} & 0 & 0 \\ 0 & 0 & \frac{far}{far-near} & \frac{-far \times near}{far-near} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

プロジェクションマトリックス

•スクリーン座標変換

スクリーンマトリックスを適用することによって実際にウインドウにレンダリングされるピクセルの座標系に変換する。

$$\begin{pmatrix} \frac{-screenWidth}{2} & 0 & 0 & \frac{screenWidth}{2} \\ 0 & \frac{-screenHeight}{2} & 0 & \frac{screenHeight}{2} \\ 0 & 0 & -(far-near) & near \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

スクリーンマトリックス

•パースペクティブコレクション

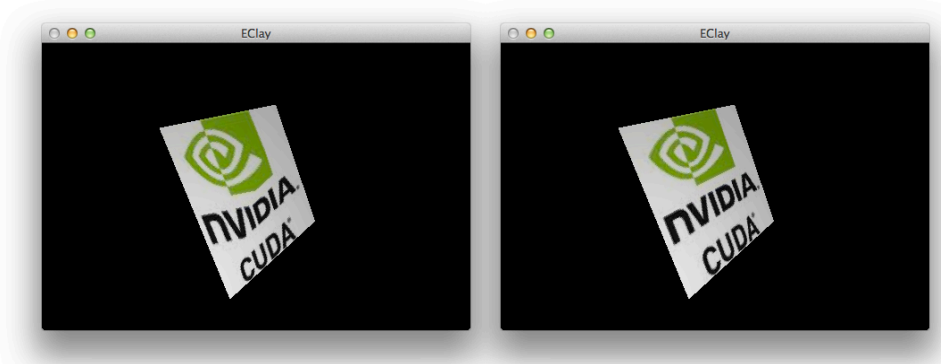
UV座標は2D座標で表現されるが、そのままでは頂点の奥行きが考慮されない値になってしまいテクスチャ適用後に歪んだ表示になるので、プロジェクション座標変換で発生したwの値を用いて奥行きの影響を付加する。

ECModel.cuファイルの中でプリプロセッサ#defineで設定しているPERSPECTIVE_CORRECTION でパースペクティブコレクションを適用するかどうかの設定が可能である。

モデル中の各プリミティブが全体のピクセルに対して大きいほど、適用しなかった時の影響が大きい。

逆に各プリミティブが小さければ、パースペクティブコレクションの計算を省くことにより多少のパフォーマンスアップを図ることができる。

下図は左がパースペクティブコレクション非適用時、右が適用後である。



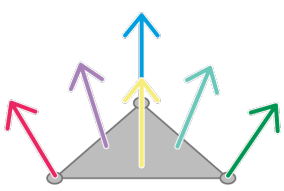
✦ピクセル(フラグメント)シェーディング

頂点処理後の各ピクセルごとの処理。

変換された頂点の座標等を受け取り、頂点カラーやテクスチャ・ライト等を適用していき、最終的なピクセルの色を決定する。

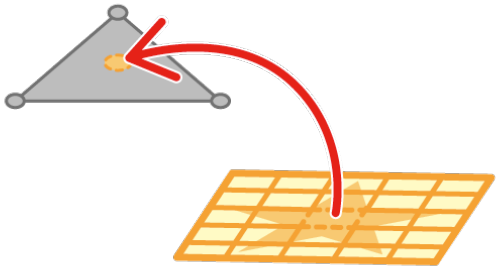
•プリミティブ補間処理

プリミティブ単位ではなくピクセル単位で処理をしたいため、プリミティブをマッピングしたピクセルのうち、頂点に当たらないピクセル部分の法線・UV座標・頂点カラーを各頂点の補間計算により求める。

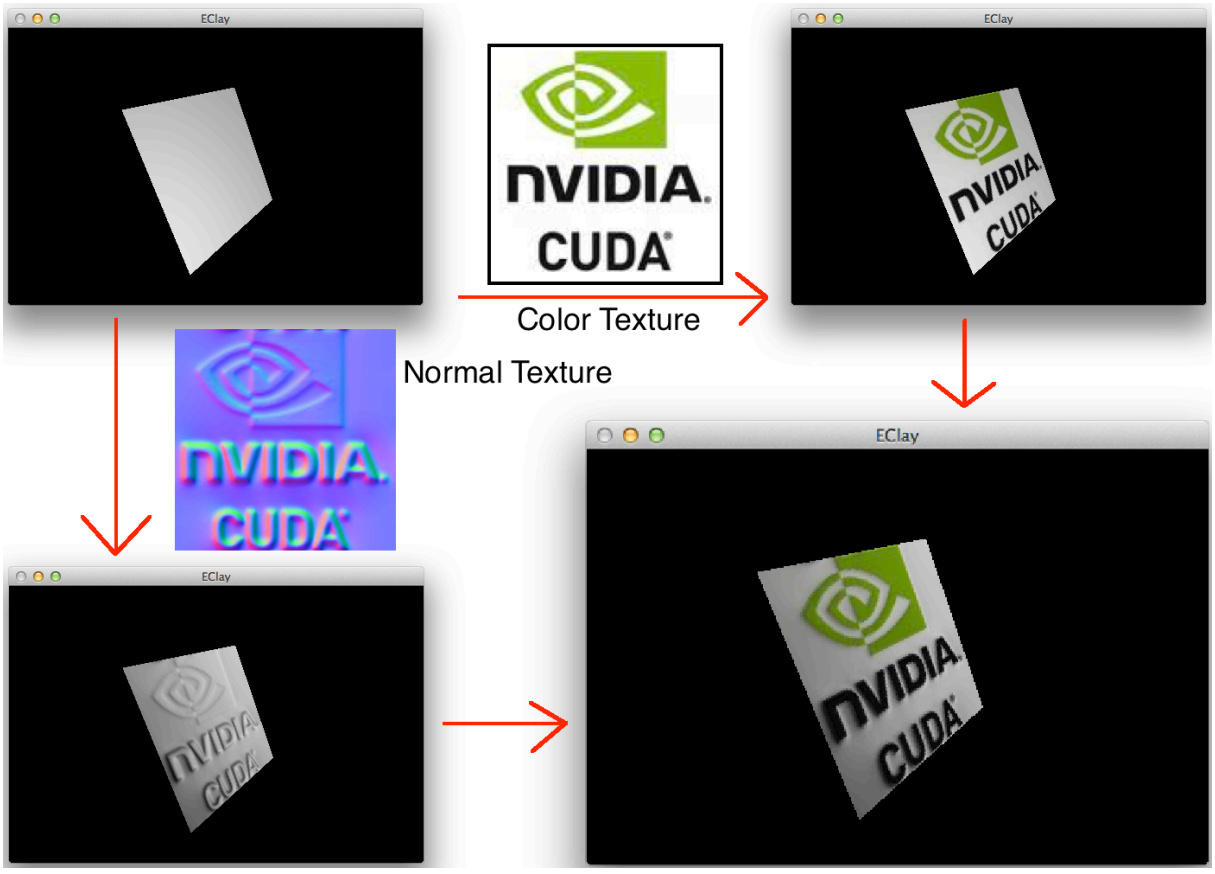


•テクスチャの適用

3Dモデルの各箇所の色を記録した画像を読み込み、UV座標を使ってピクセルに適用する(これがカラーテクスチャ)。

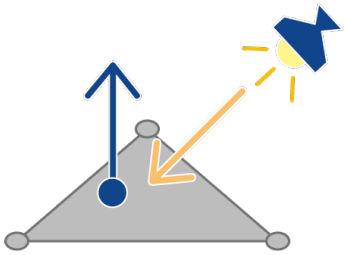


その他、法線の方向ベクトルをRGB値で表現して記録したノーマルテクスチャなどのテクニックも用いることができる。プリミティブの頂点情報を補間してピクセル単位の法線を求めるのではなく、予め計算してテクスチャにマッピングされた情報を拾うことで表現の幅を広げ、よりリアルなレンダリングを行うことができる。ノーマルマッピングはレンガなど凹凸のある物体の表現に対して有効である。



•ライティング

明るさや色などを示すパラメータと方向を示すベクトルを持ち、頂点の法線ベクトルと掛け合わせることにより色に明るさ・暗さを付加することができる。



以上の手順で最終的なピクセルの色が決定され、windowに表示される。

✤今後の課題

✦fbx、collada等の各種フォーマットへの対応

✦ピクセルシェーディングのための各種画像処理関数の対応

✦(固体) 物理エンジンの開発

✦流体力学エンジンの開発