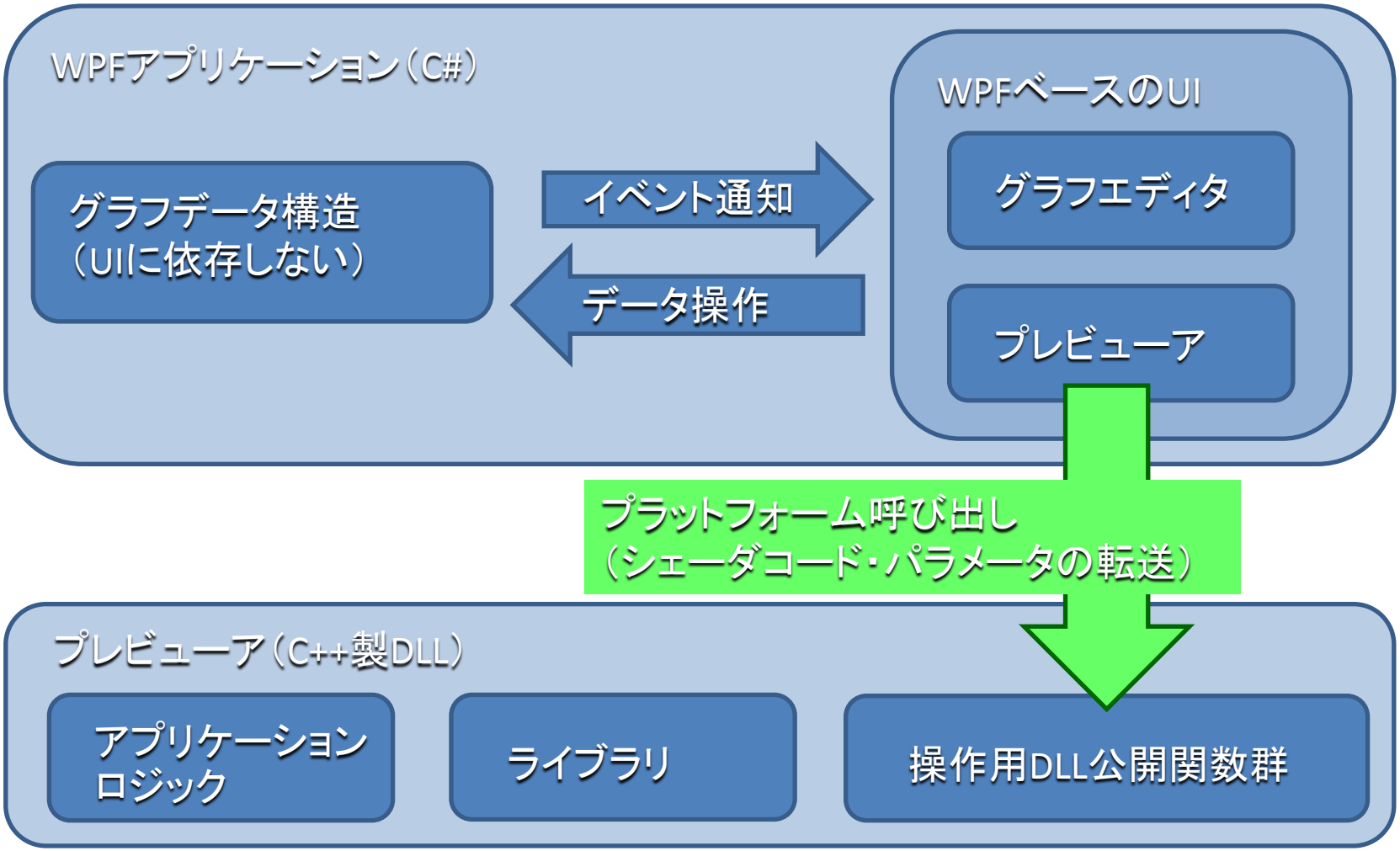
メタシェーダ設計概要

全体構成

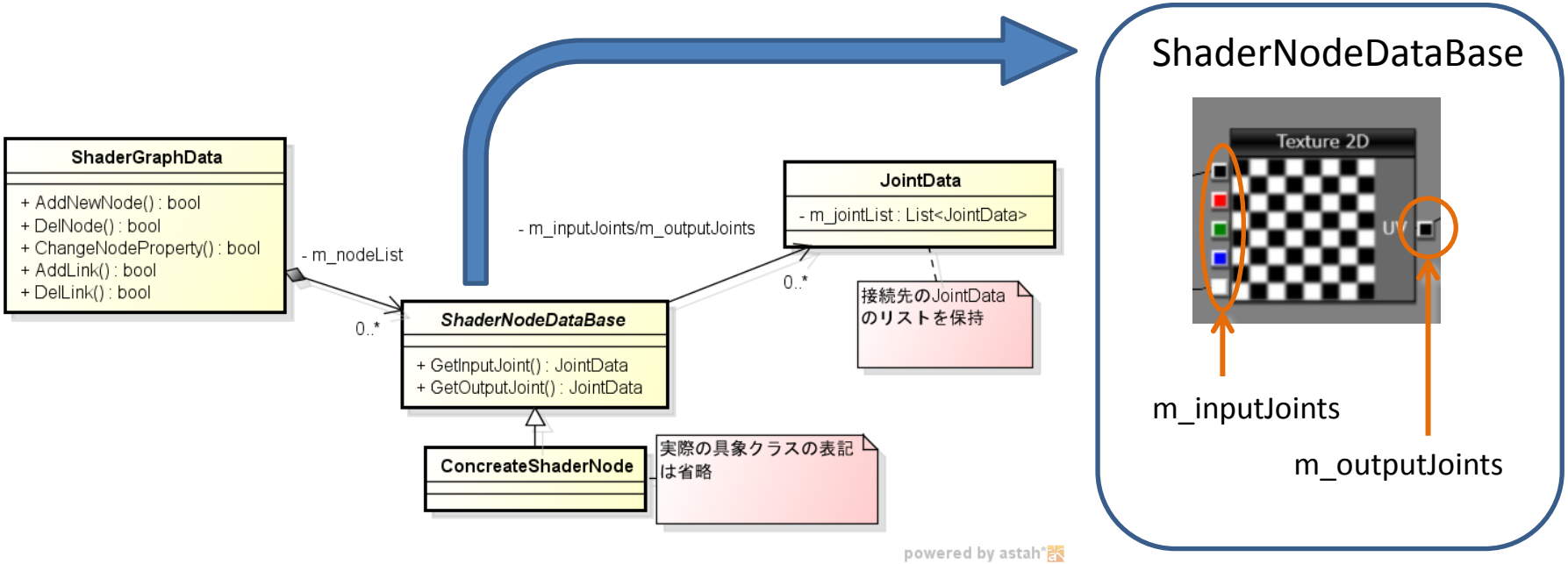

WPFアプリケーション（C#）
グラフデータ構造
（UIに依存しない）
イベント通知
データ操作
WPFベースのUI
グラフエディタ
プレビューア
プラットフォーム呼び出し
（シェーダコード・パラメータの転送）
プレビューア（C++製DLL）
アプリケーション
ロジック
ライブラリ
操作用DLL公開関数群

「WPF」＋「C++製DLL」の理由

- ゲームエディタと同じ構成を想定
  - C++製ゲームとそれにアクセスするC#製UI
- 何故WPF？
  - 「Windows Form」よりも柔軟なレイアウトが可能
    - スタック状や比率ベースの配置など、座標を意識しないレイアウト
      - 動的なUI部品の追加が容易
    - 3D空間へのUI部品の配置をサポート
      - 射影変換後の画面上でも、2D同様マウス操作が可能
      - メインウィンドウのグラフ編集画面に利用
  - WPFを用いての業務用アプリケーションの開発経験があったため
    - UIイベントのルーティングや、ドラッグ＆ドロップといったフレームワーク固有の機能の調査、学習期間の削減
- 何故C++？
  - HLSL（DirectX）ベースの描画
  - 本来は、ゲーム側のAPIを使用してレンダリングすることを想定

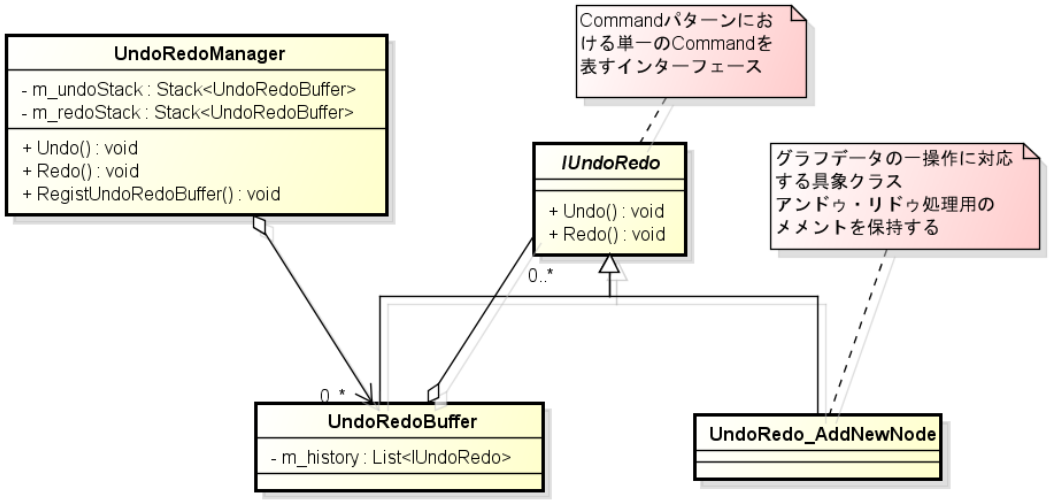
グラフデータ構造

- ShaderGraphDataクラスからグラフを操作
  - ノードの追加＆削除、リンクの追加＆削除、プロパティの変更等
- ノード間の接続は、ノードが保持するJointDataを経由
  - グラフ内のトラバースもJointDataを介して行う



アンドゥ・リドゥ処理

- CommandパターンとMementパターンで実装
  - データ構造に対する一処理をIUndoRedoの具象クラスとして実装
    - 例）下図のUndoRedo_AddNewNodeは新規ノードの追加に対応
    - 復元対象はデータ構造のみとし、UIへの反映はイベントで通知
  - UIによる一回の操作を複数の具象クラスで表し、UndoRedoBufferにまとめる
  - UndoRedoManagerがスタックでUndoRedoBufferを管理
    - アンドゥ・リドゥの実行はUndoRedoManagerのUndo,Redoメソッドを呼ぶだけ

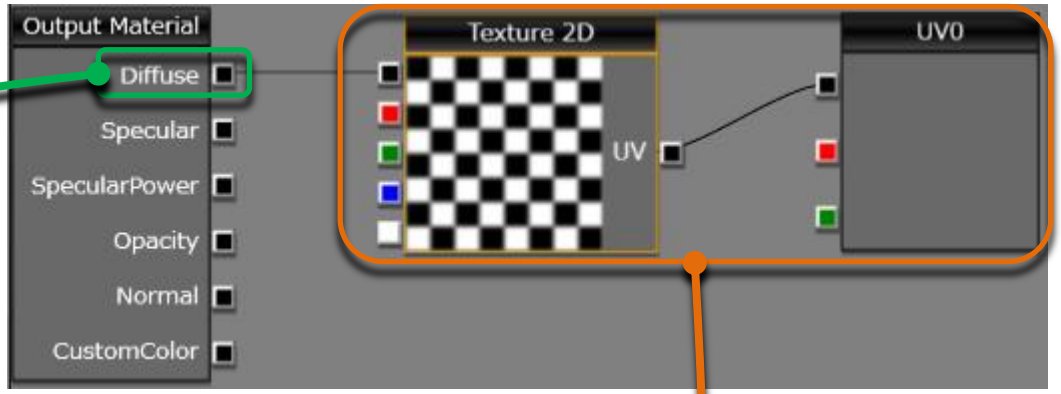


リフレクションによるコードの削減

- リフレクションとは？
  - 実行時に型にアクセスできる機能
    - メソッドやプロパティを名前指定で動的に呼び出すことが可能
      - プロパティの取得イメージ)ValueType value = object.GetValue(“PropertyName”);
        - » 実際はもう少し複雑
- 一方、データ構造のアンドゥ・リドゥ処理はノードのプロパティ変更（＆巻き戻し）がほとんど
  - 全てのプロパティ変更処理にIUndoRedoの具象クラスが必要
    - 特定の型のパラメータをメメントとしてもつ
    - Undo()、Redo()メソッド内でノードのプロパティにメメントの値を設定
  - プロパティ変更用のIUndoRedoの汎用具象クラスを構築
    (実装はParameterUndoRedo.csを参照)
    - 「特定の型」の指定⇒ジェネリックス(型パラメータ)で指定
    - プロパティ変更⇒指定したプロパティ名に応じてリフレクションで変更

シェーダコードの生成①

- 基本的なアイデア(実装はShaderCodeGenerator.csを参照)
  1. 「OutputMaterial」の各ジョイントに入力する部分グラフをコード化
  2. テンプレートコード内の対応箇所を「1」で生成したコードで置換



テンプレートコード

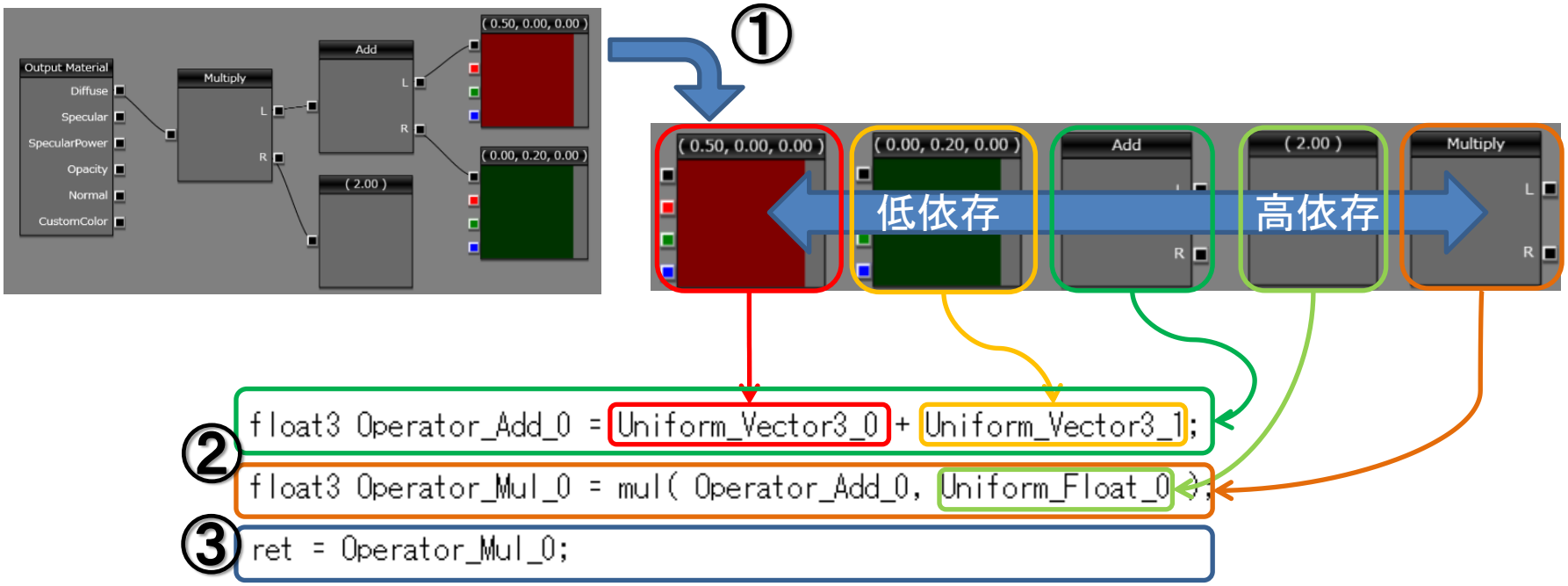
```
// get diffuse parameter
float3 GetDiffuse( PARAMETERS In )
{
#ifdef FUNC_Diffuse
    float3 ret;
%Diffuse%
    return ret;
#else
    // return default value
    return float3( 0.0f, 0.0f, 0.0f );
#endif
}
```

Diffuseへ接続する部分グラフのコード化

```
float4 Color_Texture2D_0 = tex2D( Uniform_Texture2D_0, In.Texcoord0 );
ret = Color_Texture2D_0.xyz;
```

シェーダコードの生成②

- 部分グラフのコード生成手順
  - ① 部分グラフに含まれるノードを依存度の低い順にソート
  - ② 各ノードから逐次的にコード生成
  - ③ 最後のノードをret(returnする値)に設定

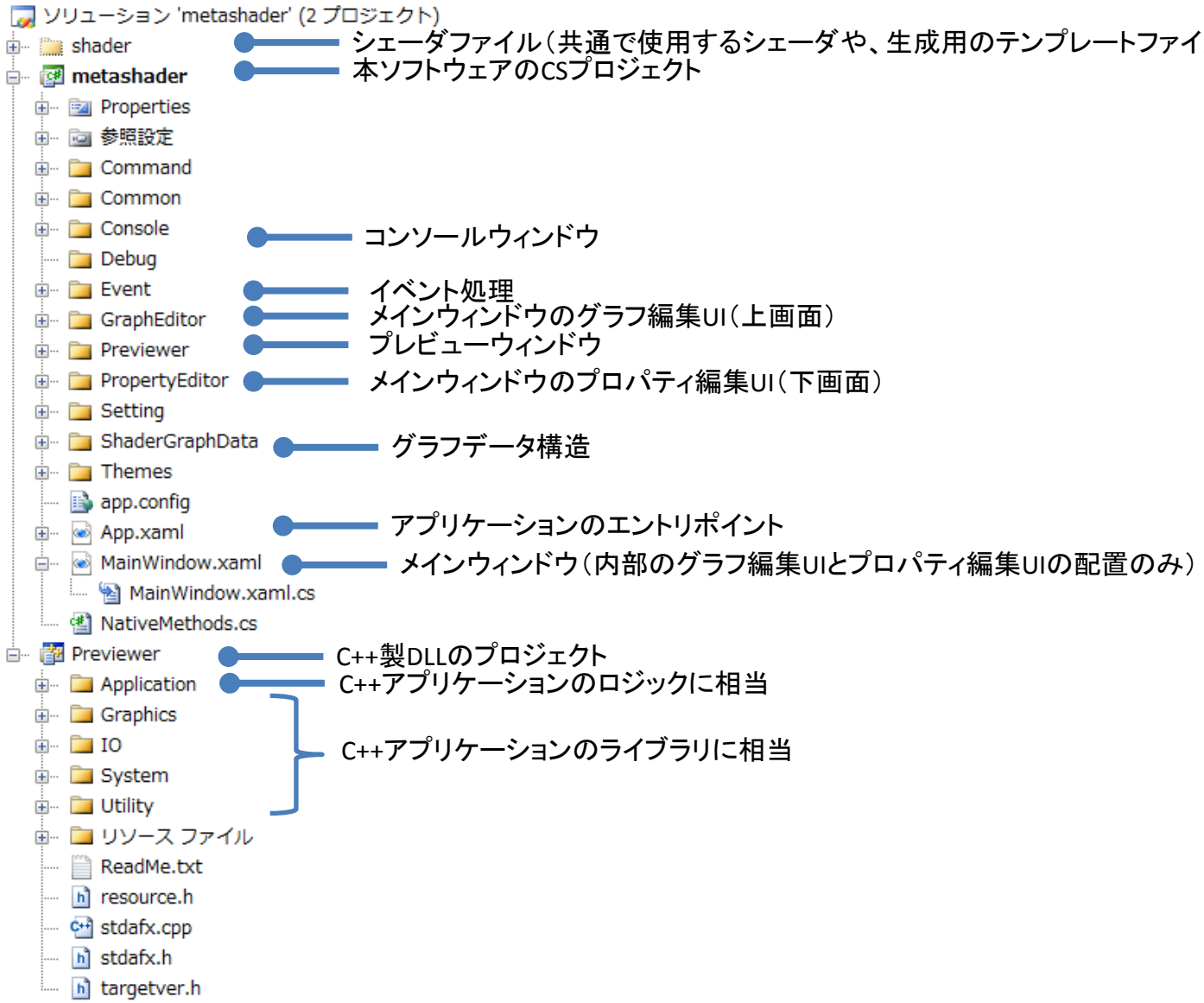


展望

- ツールの改良
  - データドリブン化
    - ノード定義のデータドリブン（ユーザー定義のノードの利用）
      - **一部対応済み「Fresnel」はユーザ定義のノード(EXEの再コンパイルの必要がない)**
    - マテリアル定義のデータドリブン（ユーザー定義のマテリアルの利用）
- ゲームへの組み込みに向けて
  - 頂点シェーダ側も動的に生成
  - 影描画への対応
  - マテリアルのインスタンス化
    - uniform変数の抽出、変更可能なデータとして外部データ化
  - リソースマネージャへ統合
    - マテリアルをリソースとして管理
    - リソースマネージャからテクスチャリソースの参照
    - ジオメトリに対するインスタンス化したマテリアルの適用

付録

VisualStudio上でのソリューション構成



参考文献

- Epic Games Inc., Unreal Development Kit
  - http://www.udk.com/
- tri-Ace Inc., "STAR OCEAN 4 : Flexible Shader Management and Post-processing" , GDC 2009
- tri-Ace Inc., "Shader Kanrijirei - Jiyudoto Hikikaeni - (Postmortem of Shader Management)“, CEDEC 2008