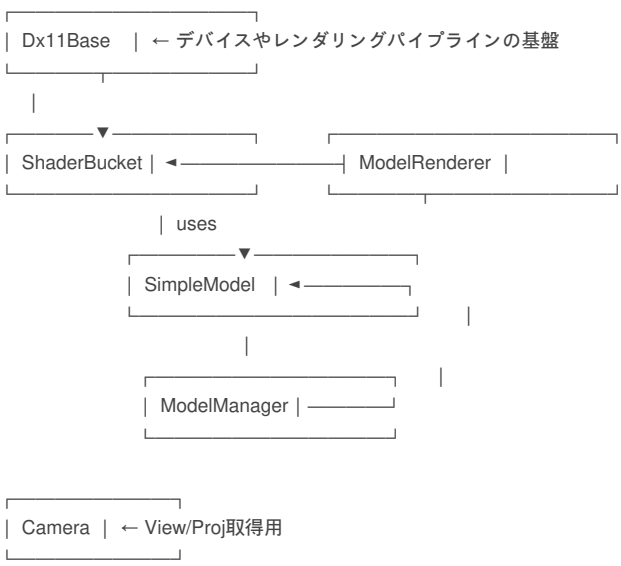
新しいDx11base設計方針（今進んでいるところまで）

☒ 主要クラスの役割一覧

クラス名	役割	主な責務	他クラスとの関係
`Dx11Base`	DirectX初期化＆基盤	デバイス・コンテキスト・スワップチェイン・バックバッファなどの管理 ビューポートやラスタライザ設定もここで一元化	全クラスが依存。最下層の基盤クラス
`ShaderBucket`	シェーダーの一括管理	`.hlsl` を読み込んで、VS/PS/IL を生成・キャッシュ キーでシェーダー切り替えを容易にする	`ModelRenderer` や `Sprite` など描画系から利用される
`Camera`	カメラ情報の管理	ビュー行列・プロジェクション行列を提供 LookAtや移動処理も持つ	`ModelRenderer` などが View/Proj を取得する
`SimpleModel`	モデルの純粋なデータ保持	頂点・インデックス・テクスチャ・サンプラ・名前など 必要なバッファを持ち描画準備済みの状態	`ModelManager` から生成され、`ModelRenderer` に使われる
`ModelManager`	モデルの読み込み・キャッシュ	ファイルから `SimpleModel` を生成 ハンドル付きでモデルを管理・再利用	`ModelRenderer` からハンドル経由で取得される
`ModelRenderer`	モデル描画の実行役	`SimpleModel` + `Transform` 情報を使って描画 内部で `Dx11Base` , `Camera` , `ShaderBucket` を使って描画する	`SimpleModel` を渡されて描画処理を行う
`Transform`（補助）	モデルの変換行列生成	SRTからWorld行列生成、法線変換行列も作成可	`ModelRenderer` が使用

☒ クラス関係図（概略）



☒ 役割分離の利点

設計	メリット
`SimpleModel` に描画コードを持たせない	同じモデルを複数箇所で描画可能（インスタンス化）
`ModelRenderer` に描画処理を集中	ステート設定、シェーダー設定、描画の変更が一元管理できる
`ModelManager` に読み込みとキャッシュを集中	同じファイルを何度も読み込む無駄を防止
`ShaderBucket` のキー管理	`"Basic3D"`、`"Sprite"`、`"ShadowMap"` など切り替えが簡単
`Dx11Base` に初期化・基盤処理を集中	DirectX設定の複雑さを隠蔽し、他クラスをシンプルに保てる

☒ まとめ

このような分離設計は、
☒ **単体テストや差し替え**がしやすく、
☒ **描画方式の拡張（PBR、影、インスタンス描画など）**にも柔軟に対応できます。

必要であれば、`ModelRenderer::Draw()` や `ModelManager::GetHandle()` など、具象的なコード実装例も提供できます。

✔ モデル関係責任分割

クラス	責任
`SimpleModel`	**データ所有者** ：頂点・インデックス・テクスチャ等のDirectXリソース保持。 ※レンダリングコードは持たない
`ModelRenderer`	**描画担当** ：`SimpleModel` のデータを使って描画処理を実行。Transformやシェーダーとの連携もここで処理
`ModelManager`	**管理担当** ：`SimpleModel` をファイルからロードし、キャッシュとして保持。ハンドル管理なども行う

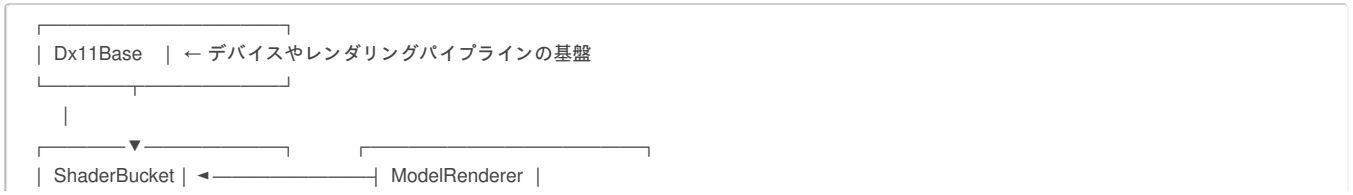
今迷っているところ

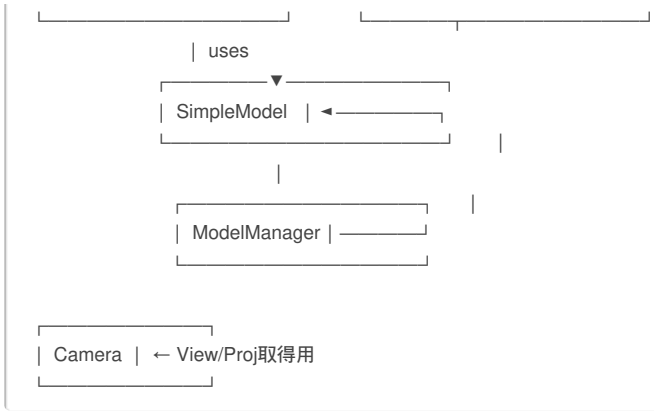
- modelにvertexbuffer、indexbuffer、textureを直接もたせるのは良いか悪いかよくわからない
  - （現在のエンジンはFBX=モデル＝全部入り）
  - これだと、インスタンスごとにレンダリングを切り替えたりとかが色々やりづらいとおもう
- modelManagerにレンダリングリソースとして情報をもたせる方法を考えている
  - ChatGPTによるとアニメーションとか拡張しようとするとめんどくさくなりそうな予感がする
- とりあえず、modelとFBXは完全に切り分けることにする（他のモデル形式とかも考えることもあるかもだから）
- model、manager、rendererによって責任を分割したけど、誰が何を持つかも分割されて困った

６月１７日現在の様子

☒ DirectX11 モデル描画エンジン：構造と設計レビュー

☒ DirectX11 モデル描画エンジン クラス構成概要





☒ メリット

1. 役割分離された設計

- `ModelManager`, `ModelRenderer`, `FBXLoader` が明確に責務を分担
- モデルデータと描画用GPUリソースを分離しており拡張が容易

2. マテリアル単位の柔軟な管理

- `SimpleModel` -> `MaterialMesh` の構造によりマルチマテリアル対応が容易
- マテリアルごとにテクスチャ、ポリゴンの分割が可能

3. シェーダーステート集中管理

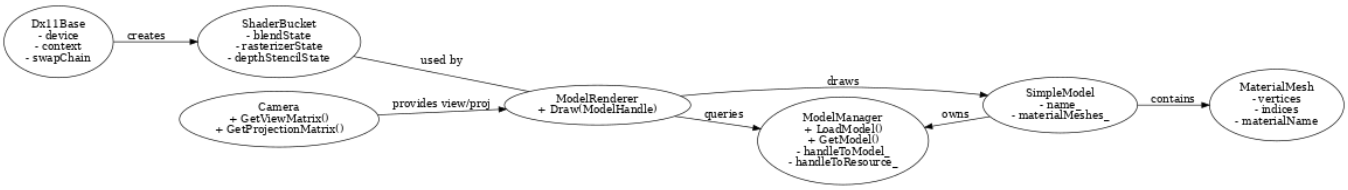
- `ShaderBucket` によりブレンド/Zステート管理を一元化
- アルファブレンドやZWrite切替も柔軟に対応

4. リソース共有対応

- モデルパスごとに `ModelHandle` を発行、リソースの使いまわしが可能
- 将来的に `MaterialPool`, `TextureManager` にも応用可能

5. アニメーション拡張の布石あり

- `Vertex` に `controlPointIndex` を保持済みで、スキンメッシュ対応の余地あり



⚠ デメリット・今後の課題

1. スキンアニメーション未対応

- Bone構造、SkinWeight、Clip、Poseデータ未実装
- `Vertex` に `boneWeight` / `boneIndex` 配列が未導入

2. 階層構造の保持なし

- FBXのノード親子構造がSimpleModel内に保持されていない
- 階層的なTransform更新や描画順の制御が難しい

3. インスタンシング未対応

- 同一モデルの多重描画でリソース効率化ができていない

4. 描画順制御がアプリ依存

- Zソートや描画カテゴリ分離が `ModelRenderer` 側に組み込まれていない

5. グローバル依存の可能性

- `ShaderBucket`, `Dx11Base` などがグローバルに依存しやすく、テスト性が低下する懸念

☒ 改善ロードマップ

ステージ	改善内容
☒ 現在	非アニメモデル・マテリアル描画まで実装済み
☒ 次	Bone構造・スキンウェイト・アニメーションクリップ導入
☒ 拡張	GPUスキニング、Shaderの構造整理、AnimationClip管理クラスの追加
☒ 拡張	Zソート処理、半透明オブジェクトの描画順制御の導入
☒ 拡張	階層Transform・ノードアニメーションの導入、カメラアニメ対応

Revision #5
★Created 31 May 2025 15:13:35 by youe2
Updated 12 June 2026 05:15:54 by youe2