

はじめに

ドット絵アニメーションや端末コンソールを使ったアニメーションには、不思議な魅力がある。筆者にとって、それらは特別楽しい存在だ。git fetch コマンドや npm コマンドを叩いていると、アスキー文字で構成されたアニメーションがプリントされることがあるが、忠犬のように、いつもつつい眺めてしまう。ただのアスキー文字ですら、表示位置、表示時間のバランス次第では、何らかキャラクター性を演出できることを示唆している。

一方で、プログラマーとしてこれを実装する立場で考えてみよう。

プログレスバーに代表されるように、1行の中でのアニメーションであれば苦勞はないだろう。1行に限定されない場合でも、もちろんそれぞれの表示位置、表示時間を仕様書に落とし込み、ハードコード (*hard coding*) することは可能である。可能ではあるのだが、生産性に欠ける。はたしてそれはプログラマーの作業なのだろうか、という思いがよぎるに違いない。きっと何か足りてないのである。「画面上のどこに、何を、どれくらいの時間表示するか」を上手に扱う**何か**が。

本書は何者か？

本書は、C++ を習得したいプログラマー向けに書いた。エンジニアアーキテクチャをひも解きながら、比較的大きなプログラムをコーディングする手法やそこに秘められた意図について言及する。題材にアニメーションシステムを選んだのは、筆者の興味もあるが、読者の理解を助けるためでもある。われわれの身の回りを埋め尽くすスマートフォンゲームや Web サイト内の実例を通して、「そのアニメーションを裏で制御している方法」をあれこれ想像してみてほしいのだ。「例示は理解の試金石」*1なのである。

Adobe Flash というソリューションをご存じだろうか。Flash は jQuery や HTML5 登場以前に一時代を築いた Web アニメーションエンジンである*2*3。Flash のコンテンツ制作には専用の有償ツールを使用するが、画像ファイル、音声ファイルなどの素材を準備して「画面上のどこに、何を、どのように表示するか」といった詳細を直感的に編集できたため、デザイナーやアーティストの支持を多く集めた。進化の過程で、ブラウザゲームなどのインタラクションにも幅広く採用された。

本書では、そのような**非プログラマー向けのスプライトアニメーション環境を実現するエンジン**を仮想した。

*1 "数学ガール 結城浩 試金石" で検索

*2 "wikipedia adobe flash" で検索

*3 本書に掲載されている製品名は、一般に各社の登録商標または商標です

目次

はじめに	i
第 1 章 特定のクラスインスタンスのツリー化	1
1.1 MTree クラスの要求	2
1.2 全体構造	3
1.3 子インスタンスの関連付け	5
1.4 インスタンスの一括削除	6
1.5 子孫要素の列挙	7
第 2 章 リソースのロード／アンロード手順のカプセル化	9
2.1 Asset クラスの要求	9
2.2 コンストラクターとデストラクター	10
2.3 リソースのカスタマイズ	11
2.4 ハンドルの提供	12
Column: std::unique_ptr を使った後処理	12
2.5 メタデータの保持	13
Column: boost::property_tree::ptree と JSON パーサー	14
2.6 パスとツリー化	15
Column: std::experimental::filesystem::v1::path の演算	17
第 3 章 リソースタイプの規定	18
3.1 AssetHandler クラスの要求	18
3.2 コンストラクターとデストラクター	19
3.3 リソースタイプの登録	20
3.4 リソースタイプの判別	20
3.5 既定のディレクトリ用初期化ハンドラー	20
第 4 章 定期処理のカプセル化	25
4.1 Actor クラスの要求	26
4.2 コンストラクターとデストラクター	28

4.3	定期処理の追加と実行	28
4.4	ツリー化	29
4.5	ローカル変数/イベントの保持	29
4.6	ユースケース	30
第 5 章	複数のクラスインスタンスのリスト化	32
5.1	CList クラスの要求	33
5.2	全体構造	33
5.3	コンポーネントリストへのインスタンスの追加	34
5.4	インスタンスの個別削除	35
5.5	インスタンスの検索	35
5.6	Actor へのコンポーネントの追加	36
5.7	Actor からのコンポーネントの一括削除	37
第 6 章	アクターとリソースの関連付け	38
6.1	ユースケース	38
6.2	リソースのロードとアンロード	39
6.3	リソースの検索	39
6.4	リソースの関連付け	40
第 7 章	アクターと文字列の関連付け	42
7.1	ユースケース	42
7.2	検索インデクスの生成	43
7.3	アクターの検索	44
7.4	検索インデクスの関連付け	45
第 8 章	アクターと座標変換の関連付け	46
8.1	ユースケース	46
8.2	座標変換成分の生成	47
8.3	3D 座標変換	47
	Column: boost::qvm	49
8.4	座標変換の関連付け	49
第 9 章	アクターとテクスチャの関連付け	52
9.1	ユースケース	52
9.2	マテリアルパラメーターの生成	53
9.3	マテリアルの関連付け	53
第 10 章	アクターとアニメーションの関連付け	56

10.1	アニメーションデータソースの指定	56
10.2	アニメーションの再生開始	56
10.3	アニメーションのデータ構造	57
10.4	アニメーションの関連付け	58
10.5	ユースケース	59
10.6	アニメーションカーブによるデータ補間	61
第 11 章	アクターと入力デバイスの関連付け	65
11.1	入力データ保持領域の生成	65
11.2	ユースケース	66
11.3	入力フレームの更新と入力データの反映	68
11.4	デジタルスイッチの立ち上がり判定	68
11.5	ビット数の数え上げ	69
第 12 章	さまざまなアクションとトリガー	71
12.1	アクションとトリガーとオペレーター	72
12.2	コンポーネントを編集するアクション	73
12.3	プロパティツリーを編集するアクション	73
12.4	状態を変更するアクション	74
12.5	イベントを通知するアクション	74
12.6	状態による制約	74
12.7	イベント発生による制約	74
12.8	入力による制約	75
第 13 章	既存コードベースとの融合	77
13.1	App クラスの要求	77
13.2	SDL2 の導入と相互運用性	79
13.3	Adapter クラスの要求	80
13.4	SDL2 ライブラリの初期化/終了	81
13.5	メインウィンドウの生成	82
13.6	入力デバイスのハンドリング	83
13.7	2D レンダリングのハンドリング	83
13.8	2D レンダリングコンポーネント	85
おわりに		86

第2章

リソースのロード／アンロード手順のカプセル化

アニメーションシステム構築において、安全なリソースハンドリングは重要な課題の1つである。アプリが画像を表示したり、音声を再生したり、何らかのローカルファイルをアクセスしようとするには、先にオープンする必要がある、いずれ使用済みになれば後処理としてそれらをクローズしなければならない。ファイルが対象ならば、期待される後処理はハンドルクローズであるが、他にも COM オブジェクト、動的メモリなど、後処理の方法はそれぞれである。これらの処理対象を一般化してリソースと捉え、1つ1つ個別に状態を持ち、複数を束ねてロード（前処理）／アンロード（後処理）できると、リソースの数が爆発的に増加したとしても全体として扱いやすさを維持できる。

2.1 Asset クラスの要求

ここで Asset クラスを導入して、ローカルファイルとロード／アンロード手順をカプセル化する方法を検討してみよう。Asset クラスの主な要求は以下の通りとする。

- 1つの Asset インスタンスは、リソースとして扱うことのできる1つのローカルファイルを表す: `ctor`
 - リソースであればどんな種類でも表現できる
- 個々の Asset インスタンスはロード／アンロードができる: `load()`、`unload()`
 - 複数回ロードも可能、ただし2度目とそれ以降のロードは実質的に何もしない
- 個々の Asset インスタンスはリソースハンドルとメタデータを保持できる: `handle()`、`props()`
- Asset インスタンス間には 1 vs N の親子関係がある
 - 子インスタンスは親インスタンスの `delete` に伴って `delete` される
- 個々の Asset インスタンスは1つのパスを持ち、検索対象とすることができる: `path()`、`find()`

実用的なコードではないが、ユースケースを以下に示す。3行目に注目してほしい。後述する `AssetHandler` クラスと組み合わせて使うことで、このようにシンプルなアセットハンドリングを実現したい。

リスト 2.1: `Tutorial1::test1()` (`examples/tutorial1.cpp` より)

```

1: void test1() {
2:     cout << __FUNCTION__ << endl;
3:     Asset asset1(L"asset/tutorial1.json", AssetHandler::initializerJson);
4:     cout << "--- load" << endl;
5:     asset1.load();
6:     cout << "--- unload" << endl;
7:     asset1.unload();
8:     cout << "--- dtor" << endl;
9: }

```

Tutorial1::test1() の出力

```

Tutorial1::test1
tte::AssetHandler::initializerJson: asset\tutorial1.json
--- load
{
  "contentType": "image\png",
  "tag1": "123",
  "array": [
    {
      "value": "3"
    },
    {
      "value": "2"
    },
    {
      "value": "1"
    }
  ]
}
Asset::load: asset\tutorial1.json
--- unload
Asset::unload: asset\tutorial1.json
--- dtor
Asset::dtor

```

このユースケースでは、1つの JSON ファイルを1つの `Asset` インスタンスで表現しており、ロードによってファイル内のコンテンツの解析 (*parse*) を実施している。

さあ、始めよう

ここまでで説明した背景と要求とユースケースから、あんならなどのようなコードを実装するだろうか。どのように API を宣言するだろうか。以降のリファレンス実装解説を読み進めるにあたって、ぜひ一度考えてみてほしい。

2.2 コンストラクターとデストラクター

では実装を見ていこう。コンストラクターとデストラクターの実装を以下に示す。ここで Path は `std::experimental::filesystem::v1::path` のエイリアス (*alias*) である。

リスト 2.2: `Asset::ctor/dtor` (engine/asset.h より)

第 12 章

さまざまなアクションとトリガー

以下のサンプルコードを見て欲しい。

リスト 12.1: Tutorial10::test1() (examples/tutorial10.cpp より)

```
1: void test1() {
2:     cout << __FUNCTION__ << endl;
3:     AssetHandler::clear();
4:     AssetHandler::appendInitializer({ L"unk", AssetHandler::initializerUnknown, });
5:     AssetHandler::appendInitializer({ L".json", AssetHandler::initializerJson, });
6:     AssetHandler::appendInitializer({ L"", AssetHandler::initializerDir, });
7:     auto assetRoot = make_unique<Asset>(
8:         L"asset/tutorial10", AssetHandler::factory(L"")
9:     );
10:    cout << "--- ctor" << endl;
11:    auto actor1 = make_unique<Actor>(
12:        [](Actor &a) {
13:            a.findComponent<Input>([&a](auto &input) {
14:                if (input.buttons("down").pressed()) {
15:                    a.findComponent<Animator>([](auto &animator) {
16:                        animator.play("0");
17:                    });
18:                }
19:            });
20:        },
21:        [&assetRoot](Actor &a) {
22:            Resource::append(*assetRoot)(a);
23:            a.importProps(assetRoot->find(L"tutorial10.json").props());
24:            Animator::append(assetRoot->find(L"tutorial10.anim.json"))(a);
25:        }
26:    );
27: }
```

ここまでの成果を組み合わせて、L"asset/tutorial10" 以下のリソースをロードし（2行目）、入力デバイスから "down" の押下があったら（14行目）、アニメーション "0" を再生するアクターを作成している（16行目）。

これはこれで解説可能なコードであり、かつデータ駆動化も容易に実現しそうなコードではあるが、もう少し記述量を減らせないだろうか。何か記述量を減らすための手立てやルールを、新たに導入できないだろうか。

さあ、始めよう

ここまでに説明した背景から、あならならどのようなコードを実装するだろうか。どのように API を宣言するだろうか。以降のリファレンス実装解説を読み進めるにあたって、ぜひ一度考えてみてほしい。