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、いつもついつい眺めてしまう。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労はないだろう。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。きっと何かが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。エンジン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、ブラウザー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

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れば、先にオープンする必要があり、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トの解析（*parse*）を実施している。

さあ、始めよう

ここまでに説明した背景と要求とユースケースから、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。

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（２２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)。

これはこれで解読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。