# 情報可視化やインタラクションのためのライブラリ Siv3D の機能強化と C++17, C++20 への対応

鈴木 遼　上田 和紀　坂井 滋和

情報可視化やインタラクションのための C++ライブラリ Siv3D は 2016 年に OSS 化されて以降，現在までに 5,000 回（コミット）を超える機能追加と改良が公開リポジトリ上で行われている．これらには近年のハードウェア・開発ツール・言語仕様の進化，ユーザの利用事例の検証等を踏まえた，様々な新機能や API 設計の見直し，C++17/C++20 への対応などの興味深い事例が含まれている．本稿では Siv3D の OSS 化以降 4 年間にわたる設計の変遷と取捨選択，および普及のための取り組みを紹介し，情報可視化やインタラクションに関するライブラリおよび現代的な C++ライブラリの設計と運用に役立つ知見を共有する．

## 1　はじめに

### 1.1　Siv3D の概要

Siv3D はグラフィカルでインタラクティブなアプリケーションの開発に便利な Application Programming Interface (API) を提供する C++ライブラリである．Siv3D を導入することで，ユーザは次のような操作を組み合わせたアプリケーションプログラムを非常に短いコードで記述できる [18].

- 図形や画像，テキスト，動画，3D モデルなど，グラフィックスの描画
- マウスやキーボード，Web カメラ，マイク，ゲームパッドなど，ヒューマンインタフェースデバイス (HID) の使用
- ウィンドウ処理，ファイルシステム，ネットワーク
- 画像処理や音声処理
- 物理演算や経路探索，幾何などの計算
- データ構造やアルゴリズム

Siv3D はゲームやメディアアートのようなインタラクティブなアプリケーションを，C++で効率的にコーディングすることを目的に開発されている．2021 年 8 月時点での直近 1 年間のダウンロード数は約 1 万回である．動作環境は Windows，macOS，Linux，Web で，コードリポジトリ `https://github.com/Siv3D/OpenSiv3D` において MIT ライセンスで配布されている．約 2,000 ファイル（コメントや空行を除き 205 KLOC）のオリジナルのソースコードと，さらに 90 のサードパーティ・オープンソースソフトウェア (OSS) から構成される．

Siv3D のアプリケーションコードのほとんどが，Siv3D により提供される型や関数を使って記述されるため，一般的な C++ライブラリよりも domain-specific language (DSL) としての性格が強い．テキストや画像，図形をインタラクティブに表示する Siv3D アプリケーションのサンプルコードを次に示す．

```
#include <Siv3D.hpp>

void Main()
{
  // 背景を水色に
  Scene::SetBackground(ColorF{0.8,0.9,1.0});
  // フォントを作成
  const Font font{ 60 };
  // 画像ファイルからテクスチャを作成
  const Texture texture{ U"image.png" };
```

Enhancements to Siv3D, a Library for Information Visualization and Interaction, Including Adopting C++17/20

Ryo Suzuki, Kazunori Ueda, Shigekazu Sakai, 早稲田大学, Waseda University.

This is an unrefereed paper. Copyrights belong to the Author(s).

```
  // テクスチャを描画する座標
  Vec2 pos{ 200,200 };

  while (System::Update())
  {
    // テクスチャを描く
    texture.draw(pos);

    // 画面の中心にテキストを表示
    font(U"Hello, Siv3D!").drawAt(
      Scene::Center(), Palette::Black);

    // マウスカーソルを中心に
    // 半透明の赤い円を描く
    Circle{ Cursor::Pos(), 40 }
      .draw(ColorF{ 1, 0, 0, 0.5 });

    // キーボードの[A] が押されたら
    if (KeyA.down())
    {
      // メッセージを出力
      Print << U"Hello!";
    }

    // 画面上に"Move" と書かれた
    // ボタンを表示し，それが押されたら
    if (SimpleGUI::Button(U"Move",
      {640,40}))
    {
      // 画面上のランダムな位置に座標を移動
      pos = RandomVec2(Scene::Rect());
    }
  }
}
```

プログラムのエントリポイント `main()`（Windows では `WinMain()`）はライブラリ側にあり，ウィンドウやレンダリングエンジンなど必要なサブシステムの初期化が終わった段階で，ライブラリ側がユーザの実装した `Main()`関数を呼び出す形でアプリケーションが実行される．`Main()`関数が制御を返すと，サブシステムのリソースの解放など後片付けの処理が行われ，プログラムが終了する．ユーザがサブシステムのための定型的なコードを記述せずに済むこのような構成は，openFrameworks や Cinder など，C++のアプリケーション開発フレームワークでよく用いられる．

```
// 説明のための簡易的なコード
// （ライブラリ側）
int main() // エントリポイント
{
  SubSystems s;
  if (!s.startUp())
    return -1;
  Main(); // ユーザのプログラムを実行
  s.shutDown();
}
```

### 1.2 Siv3D の特徴

Siv3D は最新の C++仕様を積極的に活用しつつ，いくつかのテクニカルな API 設計により，生産性が高く，間違えにくく，学習しやすいユーザコードを実現している．また，2016 年のオープンソース化以降，5,000 回以上のコードの追加や変更（コミット）が公開リポジトリ上で行われ，20 人以上がコミッタとしてライブラリの開発に参加するなど，活発な開発とユーザコミュニティ運営が続いている．

Siv3D の API 設計技法や，開発生産性の向上，ユーザコミュニティの継続に関する経験は，新しい C++ライブラリの開発者のための再利用可能な知見として共有できると考える．

### 1.3 関連記事

C++のライブラリや API の設計について，Reddy が基本的な作法を広く論じている [11]．Gregory は，C++によるゲームエンジンの構成を解説するなかで C++ API 設計についても述べている [7]．Nystrom は C++によるゲームプログラミングに有益なデザインパターンを複数取り上げ，現代的なコードによる実装と用例を紹介している [9]．よりプリミティブで一般的な C++のコーディングルールとしては C++ Core Guidelines [16] が充実している．

本稿ではこれらの記事で扱っている汎用的なルールや技法については繰り返さず，代わりに Siv3D の API 設計で特定の課題を突破するために実装した技巧的なコードや，最新の C++仕様の用例のうち，他のプロジェクトでも導入しやすい事例を紹介する．

### 1.4 本稿の構成

本稿ではまず，Siv3D で実際に使われている API やコーディングスタイルをピックアップし，使い勝手の良い C++ライブラリの実現のために C++から「もっと引き出す」応用的な設計技法について論じる．とくに，Siv3D のオープンソース化当初の論文 [20] 以降に改良が進んだ部分に比重を置く．二番目に，Siv3D のオープンソースプロジェクトの継続とユーザコミュニティの発展のための取り組みについて述べる．三番目に，Siv3D のような規模の開発をどのよ

うなツールの助けによって実現したかを紹介する．最後に，これからの Siv3D で解決したい課題と，C++や開発ツールが将来どのように進化・整備されると，Siv3D のようなライブラリの開発者にとって嬉しいかという意見を報告する．

## 2 C++ライブラリ設計

本節では，Siv3D の使い勝手を改善するために取り組んだ API 設計の工夫，C++17/C++20 への対応など，ライブラリ設計の参考となる事例を紹介する．

### 2.1 柔軟な図形定義のためのコンストラクタオーバーロード

図形を扱う C++ ライブラリは，通常，図形をいくつかのクラスに分類し，それぞれのクラスのコンストラクタについて関数オーバーロード (function overloading) によって，目的の図形を最小限の入力から柔軟に定義できる方法を提供する．ここでは長方形のクラス Rect を考える．

```
struct Point
{
  int x, y;
};

struct Rect
{
  int x, y, w, h;

  // 左上を (0,0)におく正方形の定義に便利
  explicit Rect(int size);

  // 左上を (0,0)におく長方形の定義に便利
  Rect(int w, int h);

  // 正方形の定義に便利
  Rect(int x, int y, int size);

  // 長方形の定義に便利
  Rect(int x, int y, int w, int h);

  // Point 型の値から構築
  Rect(Point pos, int size);

  // Point 型の値から構築
  Rect(Point pos, int w, int h);

  // ...
};

int main()
{
  // 左上の座標が (40,20)
  // 一辺が50 の正方形
  Rect rect{ 40, 20, 50 };
}
```

長方形の定義においては，中心座標を基準とするほうが都合が良いケースもあるし，左下や右下を基準にするほうが便利な場合もある．しかし，C++のオーバーロードの解決は引数の型と個数の対応関係によって行われるため，左上座標を基準とするコンストラクタと中心座標を基準とするコンストラクタは単純には共存できない．このようなとき，オフセットの計算をラップした静的関数を提供するのが 1 つの解決方法であるが，コンストラクタではないため，new 式や，一時オブジェクトの作成を介さずコンテナに要素を追加する emplacement が使えないなど，一貫性に欠く．

```
struct Rect
{
  // 中心座標と幅，高さからRect を作成
  static Rect FromCenter(
    Point center, int w, int h);

  // ...
};

int main()
{
  Point pos{ 10, 20 };

  std::vector<Rect> rects;

  rects.emplace_back(pos, 100);

  rects.emplace_back(pos, 100, 50);

  // emplacement で構築できない
  rects.push_back(
    Rect::FromCenter(pos, 100, 50));
}
```

コンストラクタのオーバーロードの文脈でこの問題を解決する方法として，タグ・ディスパッチング (tag dispatching) がある．オーバーロードを増やすために，タグとして機能するだけの，メンバを持たないクラスを登場させる．

```
struct TopLeft{};
struct Center{};

struct Rect
{
  Rect(TopLeft, Point topLeft,
    int w, int h);

  Rect(Center, Point center, int w, int h);
```

```
10
11   // ...
12 };
13
14 int main()
15 {
16   Point pos{ 10, 20 };
17   Rect a{ TopLeft{}, pos, 30, 40 };
18   Rect b{ Center{}, pos, 30, 40 };
19 }
```

列挙子によって動的に分岐する方法でも，コンパイル時定数に基づく最適化でゼロオーバーヘッドにできる可能性がある．図形が複数種類ある場合，その性質に合わせた選択肢を提供する列挙型を都度用意する必要があり，ユーザコードはタグ・ディスパッチングと比較して少し冗長で文字数も増える．

```
1  struct Rect
2  {
3    enum class Alignment
4    {
5      Center,
6      TopLeft,
7    };
8
9    // Alignment:: を省略するため
10   static constexpr auto
11     Center = Alignment::Center;
12   static constexpr auto
13     TopLeft = Alignment::TopLeft;
14
15   Rect(Alignment a,
16     Point pos, int w, int h);
17
18   // ...
19 };
20
21 struct Circle
22 {
23   enum class Alignment
24   {
25     Center,
26     TopCenter, // TopLeft は無い
27   };
28 };
29
30 int main()
31 {
32   Rect a{ Rect::TopLeft,
33     {10, 20}, 30, 40 };
34
35   Rect b{ Rect::Center,
36     {10, 20}, 30, 40 };
37 }
```

これ以外に，座標クラスとそのコンストラクタを継承した，型情報に基準位置を持つクラスを作ることも考えられる．この方法は実引数の個数と記述を少なくできる．

```
1  struct TopLeft : Point
2  {
3    // コンストラクタの継承
4    using Point::Point;
5  };
6
7  struct Center : Point
8  {
9    // コンストラクタの継承
10   using Point::Point;
11 };
12
13 struct Rect
14 {
15   Rect(TopLeft topLeft, int w, int h);
16
17   Rect(Center center, int w, int h);
18
19   // ...
20 };
21
22 int main()
23 {
24   Rect a{ TopLeft{10,20}, 30, 40 };
25   Rect b{ Center{10,20}, 30, 40 };
26 }
```

しかし，そうしたクラスをジェネリックに扱うためにクラステンプレート化すると，次のようにユーザコードで型を明示的に記述する必要が生じる．

```
1  struct Point
2  {
3    int x, y;
4    Point() = default;
5    Point(int _x, int _y);
6  };
7
8  struct PointF
9  {
10   float x, y;
11 };
12
13 template <class PointType>
14 struct TopLeft : PointType
15 {
16   using PointType::PointType;
17 };
18
19 template <class PointType>
20 struct Center : PointType
21 {
22   using PointType::PointType;
23 };
24
25 struct Rect
26 {
27   Rect(TopLeft<Point> topLeft,
28     int w, int h);
29
30   Rect(Center<Point> center, int w, int h);
```

```
31
32    // ...
33 };
34
35 int main()
36 {
37    Rect a{ TopLeft<Point>{10,20}, 30, 40 };
38    Rect b{ Center<Point>{10,20}, 30, 40 };
39 }
```

Siv3D では，ユーザのコストを抑えつつコンストラクタのバリエーションを増やすという課題に対し，呼び出し側がコンストラクタ引数とタグをセットにしたオブジェクトを簡易に構築できる仕組みを考案した [19]．その構文の見た目から「名前付き引数」と呼ぶ．当然 C++のコア言語機能には本当の名前付き引数の仕組みはないため，疑似的に再現したものである．ソースコードを GitHub から入手できる [17]．

```
1  struct Rect
2  {
3    Rect(TopLeft_<Point> topLeft,
4      int w, int h) {}
5
6    Rect(Center_<Point> center,
7      int w, int h) {}
8
9    // ...
10 };
11
12 int main()
13 {
14   Point pos{ 11, 22 };
15   Rect a{ TopLeft = pos, 30, 40 };
16   Rect b{ Center = pos, 30, 40 };
17   Rect c{ TopLeft(10,20), 30, 40 };
18   Rect d{ Center(10,20), 30, 40 };
19 }
```

名前付き引数によるオーバーロードにより，Siv3D の長方形クラスや円クラスのコンストラクタは，様々な位置を基準にして図形を定義できるようになった．線分クラスは，2 点の座標の組からだけでなく「始点と，始点から終点へのベクトル」や「始点と，始点から終点への角度と距離」といった引数での構築も可能になった．

```
1  void Main()
2  {
3    using namespace Arg;
4    Point pos1{ 11, 22 }, pos2{ 33, 44 };
5    Rect a{ topLeft = pos1, 30 };
6    Circle b{ topCenter = pos2, 30 };
7    Line c{ pos1, pos2 };
8    Line d{ pos1, direction(40, 50) };
9  }
```

### 2.2 コードの表現力の向上

関数やパラメータの意味を推測しにくいコードは，間違いを生みやすい．次のコードは昔の Siv3D のコードであるが，挙動を正しく読み当てるのは難しい．

```
1 Timer timer{ 10, true };
```

正解は，構築と同時に「カウントがスタート」する 10「秒」のタイマーであるが，時間の単位や第二引数の意味は，コメント無しでは伝わらない．このような問題が Siv3D の API の各所に見られたため，

- ユーザ定義リテラルを積極的に活用し，数値リテラルに単位の情報を付与する
- boolean literal を型安全で説明的な定数に置き換える

という方針でコードの表現力を向上させた，現在の Siv3D では次のように記述しないとコンパイルできない．

```
1 Timer timer{ 10s, StartImmediately::Yes };
```

#### 2.2.1 ユーザ定義リテラルの活用

ユーザ定義リテラルは C++11 で導入されたコア言語機能で，指定したサフィックスの付いた数値・文字列リテラルについて，その値を引数とする関数の糖衣構文として機能する特殊な文法である．例えば，次のコードでは，_piサフィックスによって，円周率の X 倍を乗算演算子を使わずに得られる．

```
1  double operator ""_pi(long double x)
2  {
3    return (x * std::numbers::pi);
4  }
5
6  int main()
7  {
8    double a = 0.5_pi; // a == 1.5707...
9  }
```

C++標準ライブラリにおいても，数値リテラルを時間型に変換する sサフィックス，hサフィックスなどが定義されている．これらを活用することで，リテラルに単位が付与され，コードの情報量が増えるほか，関数呼び出しよりも短いコードで何らかの処理を行うことができる．Siv3D が提供するユーザ定義リテラルを表 1 にまとめ，実際のコードでの用法を次に示す．

```
1 void Main()
```

表 1　Siv3D が提供するユーザ定義リテラル

| サフィックス | 戻り値の型 | 引数の型 | 説明 |
|---|---|---|---|
| _pi | double | long double<br>unsigned long long | π |
| _piF | float | long double<br>unsigned long long | π |
| _deg | double | long double<br>unsigned long long | 度数法からラジアン角へ変換 |
| _degF | float | long double<br>unsigned long long | 度数法からラジアン角へ変換 |
| _big | BigInt | const char* | 多倍長整数 |
| _bigF | BigFloat | const char* | 多倍長浮動小数点数 |
| _dp | DecimalPlaces | unsigned long long | 小数点以下の桁数を指定するマニピュレータ |
| _fmt | FormatHelper | const char32* | Python スタイルの文字列フォーマットの適用 |
| _re | RegExp | const char32* | 文字列を正規表現オブジェクトへ変換 |
| _emoji | Emoji | const char32* | 文字列を絵文字オブジェクトに変換 |
| _icon | Icon | unsigned long long | 整数をアイコン ID に変換 |

```
{
  // 正方形を45 度回転させた図形を作成
  Rect{ 10 }.rotated(45_deg);

  // 文字列を画像ファイルパスとみなし，
  // 画像ファイルをオープン
  Texture a{ U"image.png" };

  // 文字列を絵文字とみなし，
  // 絵文字データを読み込む
  Texture b{ U" 任意の絵文字"_emoji };

  // 多倍長浮動小数点数
  // （リテラル部分は文字列と見なされる）
  auto c = 9.99999999999999999999_bigF;

  // Python 風の文字列フォーマット
  Print << U"{} {}"_fmt(U"Hello", 20);

  // 正規表現のマッチ
  U"(\\d+)-(\\d+)-(\\d+)"_re
    .match(U"2021-09-01");
}
```

### 2.2.2　強く型付けされたブーリアン型

boolean literal を関数の実引数で使うと，引数の取り違えやコードの可読性低下の原因となる．名前の付いた定数を用意することで意図を読み取りやすくなるが，強い型付けは無く，取り違えの危険性は残る．

```
void G(bool);

void F(bool formatDrive, bool enableLog)
{
  if (enableLog) { }

  G(formatDrive);
}

int main()
{
  constexpr bool EnableLog = true;

  F(EnableLog, false); // 本当に正しい？
}
```

C++11 以降では，型付けのある enum class を使うことで安全にできるが，値を暗黙に bool 型に変換できないため，ライブラリの実装側の手間が増える．

```
enum class FormatDrive { No, Yes };
enum class EnableLog { No, Yes };

void G(bool);

void F(FormatDrive formatDrive,
  EnableLog enableLog)
{
  if (enableLog) { } // エラー

  // 面倒
  if (enableLog == EnableLog::Yes) { }

  // 面倒
  G(formatDrive == FormatDrive::Yes);
```

```
}

int main()
{
  F(FormatDrive::No, EnableLog::Yes);

  // エラー
  //F(EnableLog::Yes, FormatDrive::No);
}
```

Siv3D では，このギャップを埋める次のようなユーティリティクラスを実装した．説明のために実際のコードから一部を簡略化している．

```
// 異なるタグのYesNo 間の変換は不可
template <class Tag>
struct YesNo
{
  struct Helper { bool yesNo; };

  explicit constexpr YesNo(bool yesNo)
      noexcept
    : m_yesNo{ yesNo } {}

  constexpr YesNo(Helper helper) noexcept
    : m_yesNo{ helper.yesNo } {}

  // contextually converted to bool
  // の文脈でbool 型の値としてふるまう
  explicit constexpr operator bool() const
      noexcept
  {
    return m_yesNo;
  }

  // それ以外の文脈ではこの関数を使う
  constexpr bool getBool() const noexcept
  {
    return m_yesNo;
  }

  // ○○::Yes と記述できるように
  static constexpr Helper Yes{ true };

  // ○○::No と記述できるように
  static constexpr Helper No{ false };

private:

  bool m_yesNo;
};
```

if ()のような contextually converted to bool の文脈で bool 型の値のようにふるまうため，enum class を使う方法と比較し，値を使用する箇所の変更が最小限で済む．主要なコンパイラの最適化を有効にしたビルドにおいて，単純に bool 型の値を使用したときと同じアセンブリが生成されることを確認した．

```
// タグ名は自由に決められる
using FormatDrive
  = YesNo<struct FormatDrive_tag>;
using EnableLog
  = YesNo<struct EnableLog_tag>;

void G(bool) {}

void F(FormatDrive formatDrive,
  EnableLog enableLog)
{
  if (enableLog) { }

  G(formatDrive.getBool());
}

int main()
{
  F(FormatDrive::No, EnableLog::Yes);

  // エラー
  //F(EnableLog::Yes, FormatDrive::No);
}
```

### 2.3 C++17, C++20 規格の活用

Siv3D は動作要件のコンパイラを新しいバージョンに絞っているため，C++の新しい規格を活用する API を，対応コンパイラの登場とほぼ同時に実装できる．C++17 や C++20 の機能の中で，特に Siv3D への恩恵が大きかったものを紹介する．

#### 2.3.1 コンストラクタへの nodiscard 属性

Siv3D では，一時オブジェクトを作成したのちにメンバ関数で必要な操作を行うというパターンを多用するため，次の例のようにオブジェクトを作成しただけで何もしない誤ったコードをしばしば書いてしまう問題があった．

```
void Main()
{
  // 無意味だが警告されなかった
  Rect{ 100, 50, 40, 30 };

  Rect{ 100, 50, 40, 30 }.draw();
}
```

C++17/20 でコンストラクタへの [[nodiscard]] 属性 [14] が有効になったため，Siv3D では原則としてすべてのコンストラクタに [[nodiscard]]属性を付与し，前述のようなケースでコンパイラが適切な警告メッセージを発するよう改善した．

### 2.3.2 Designated Initialization

C++20 から，集成体 (aggregates) に対してメンバ変数名を指定した初期化 (designated initialization) が可能になった [13]．これまでは，メンバ変数の多い構造体を初期化する場合，集成体初期化で一斉にパラメータを記述するか，初期化後に名前を使って値を代入するかの二択であった．前者は同じ型の値を取り違えても気付かない，後者は構造体を const にできず，また，一部のメンバ変数の初期化を忘れる可能性があるという欠点があった．こうした問題が designated initialization によって解決された．

```
struct BoundingBox
{
  double xMin = Math::Inf;
  double yMin = Math::Inf;
  double xMax = -Math::Inf;
  double yMax = -Math::Inf;
};

// - 通常の集成体初期化
// どの値がどのメンバ変数にセットされるか，
// この局所的なコードだけではわからない
BoundingBox A()
{
  return{ 0.1, 0.2, 0.3, 0.4 };
}

// - あとで値を代入
// 値をセットし忘れたメンバ変数があっても
// 気付かない
BoundingBox B()
{
  BoundingBox b;
  b.xMin = 0.1; b.yMin = 0.2;
  b.xMax = 0.3; b.yMax = 0.4;
  return b;
}

// Designated Initialization
BoundingBox C()
{
  return{
    .xMin = 0.1, .yMin = 0.2,
    .xMax = 0.3, .yMax = 0.4
  };
}
```

### 2.3.3 Concepts

C++20 の Concepts の登場まで，テンプレートの型に制約を加えるには，SFINAE と呼ばれる複雑なイディオムを使う必要があった．以前より Siv3D では，型制約を用いたオーバーロードが必須でない単純なテンプレートにおいても，ユーザが誤った型を引数に渡した際に生じる難解なエラーメッセージを回避するために SFINAE を用いていたが，Concepts によりコードが単純化された．C++20 への移行期で Concepts が未サポートのコンパイラ向けのコードと，Concepts 対応コードをライブラリに共存させるため，Siv3D では次のようなマクロを定義して利用した．

```
#if __cpp_lib_concepts

// Concepts をサポートするコンパイラ用
# define CONCEPT_INTEGRAL template <std::
    integral Integral>
# define CONCEPT_INTEGRAL_ CONCEPT_INTEGRAL

#else

// Concepts をサポートしないコンパイラ用
// - SFINAE イディオムを使用
# define CONCEPT_INTEGRAL template <class
    Integral, std::enable_if_t<std::
    is_integral_v<Integral>>* = nullptr>
# define CONCEPT_INTEGRAL_ template <class
    Integral, std::enable_if_t<std::
    is_integral_v<Integral>>*>

#endif

CONCEPT_INTEGRAL
auto Square(Integral n);

// _ 付きは，SFINAE 版での
// デフォルトテンプレート引数の再定義回避用
CONCEPT_INTEGRAL_
auto Square(Integral n) { return n * n; }

int main()
{
  // 読みやすいエラーメッセージが出力
  //Square(0.5);

  Square(10);
}
```

## 2.4 サブシステムの初期化と終了処理

Siv3D のように多くのサブシステムから構成されるライブラリの一般的なデザインパターンの 1 つは，各サブシステムをシングルトンクラスとすることである．サブシステムには依存関係があり，正しい順番で初期化と終了処理が行われる必要があるため，工夫が必要である [1] [7]．

Siv3D では，サブシステム管理の設計にあたって，サブシステムを追加する際のコードの変更を最小にするようなクラスを設計した．すべてのサブシステム

を単一の std::tupleにまとめて格納することが特徴で，std::get()を個別の getter の代わりにでき，また，サブシステムの初期化と終了処理のタイミングを短いコードで一括して制御できるという利点がある．

次に示すコードでは，例として IRendererと IAudio というサブシステムを管理している．ここに新しくサブシステムを追加したい場合，

- Manager.hpp に前方宣言を追加
- サブシステムを tuple の要素に追加
- Manager.cpp でヘッダをインクルード

という 3 行の変更だけで済む．

```
// SubSystem.hpp
#pragma once
#include <cassert>

template <class ISubSystem>
class SubSystem
{
public:

  ~SubSystem()
  {
    // Manager::~Manager() によって破棄済み
    assert(pSubSystem == nullptr);
  }

  ISubSystem* get() noexcept
  {
    return pSubSystem;
  }

  void release()
  {
    delete pSubSystem;
    pSubSystem = nullptr;
  }

private:

  // ファクトリ関数でサブシステムを生成
  ISubSystem* pSubSystem
    = ISubSystem::Create();
};
```

```
// Manager.hpp
#pragma once
#include <tuple>
#include "SubSystem.hpp"

// 前方宣言だけで OK.
// サブシステムのコード変更の影響を受けない
class IRenderer;
class IAudio;

class Manager
{
public:

  Manager() noexcept;

  ~Manager();

  template <class Interface>
  static auto* Get() noexcept
  {
    return std::get<SubSystem<Interface>>(
        pManager->m_subsystems).get();
  }

private:

  inline static Manager* pManager = nullptr;

  // サブシステムをこのtuple に追加
  std::tuple<
    SubSystem<IRenderer>,
    SubSystem<IAudio>>
    m_subsystems;
};

// 短いコードでサブシステムに
// アクセスできる便利なマクロ
#define GET(SUBSYSTEM) Manager::Get<I##
     SUBSYSTEM>()
```

```
// Manager.cpp
#include "Manager.hpp"

// サブシステムのヘッダ
#include "IRenderer.hpp"
#include "IAudio.hpp"

template <size_t I, class Tuple>
static void Release(Tuple&& t)
{
  std::get<I>(t).release();
  if constexpr (I > 0)
    Release<I - 1>(t);
}

template <class Tuple>
static void ReleaseAll(Tuple&& t)
{
  return Release<std::tuple_size_v
    <std::remove_reference_t<Tuple>>-1>
    (std::forward<Tuple>(t));
}

Manager::Manager() noexcept
{
  pManager = this;
}

Manager::~Manager()
{
  // サブシステムを初期化と逆順ですべて終了
  ReleaseAll(m_subsystems);
  pManager = nullptr;
}
```

```cpp
// IRenderer.hpp
#pragma once

class IRenderer // サブシステムの例
{
public:

  static IRenderer* Create();

  virtual ~IRenderer() = default;

  virtual void draw() = 0;
};
```

```cpp
// IAudio.hpp
#pragma once

class IAudio // サブシステムの例
{
public:

  static IAudio* Create();

  virtual ~IAudio() = default;

  virtual void play() = 0;
};
```

```cpp
// Library.cpp
#include "Manager.hpp"
#include "IRenderer.hpp"

void DrawSomething()
{
  // IRenderer に直接アクセス
  GET(Renderer)->draw();
}

int main()
{
  // シングルトンオブジェクトを作成
  Manager mgr;

  DrawSomething();
} // デストラクタでサブシステムを順番に破棄
```

### 2.5 RAII を利用したレンダーステート管理

Resource Acquisition Is Initialization (RAII) は，オブジェクトの初期化と破棄にリソースの確保と解放を関連付ける，C++ の基本的なイディオムである．標準ライブラリでは `std::fstream`や，`std::lock_guard` で活用されていてる．Siv3D においても，テクスチャ，音声データなど，リソースを管理するクラスで必ず使われている．

RAII における管理対象をリソースからステートに拡張すると，グラフィックスを描画するプログラムを組む際に頻繁に登場する，レンダーステート（描画の方法）や使用するシェーダプログラム（GPU 用のプログラム）の切り替え，座標変換行列のスタック操作をコードに記述するうえでも便利である．Siv3D では，レンダーステートを，そのオブジェクトのスコープが有効な間だけ適用する `ScopedRenderStates2D`クラスや，コンストラクタで新しい座標変換行列を現在の行列スタックに push し，デストラクタで pop するような `Transformer2D`といったクラスを提供する．グローバルなレンダーステートに行った変更がデストラクタで自動的にリセットされるため，関数の呼び出し先で行われた変更が呼び出し元に影響することが無くなり，プログラムの見通しが良くなる．これらのクラスの変数はコード中では明示的に使われず無意味に見えるが，実際はデストラクタが仕事をする．

```cpp
void A()
{
  ScopedRenderStates2D s{
    BlendState::Additive,
    SamplerState::ClampNearest };

  Draw();
} // s のデストラクタが
  // レンダーステートをリセット

void B()
{
  A();

  // ここではA() の中での
  // レンダーステート変更の影響を受けない
  Draw();
}

void C()
{
  // スケールをpush
  Transformer2D t1{ Mat3x2::Scale(2.0) };

  // 2 倍のスケールで描画される
  Draw();
  {
    // スケールをpush
    Transformer2D t2{ Mat3x2::Scale(1.5) };

    // 3 倍のスケールで描画される
    Draw();
  } // t2 のデストラクタがスケールをpop

  // 2 倍のスケールで描画される
  Draw();
} // t1 のデストラクタがスケールをpop
```

### 2.6 ヘッダファイルのドキュメント性の向上

C++の IDE は，関数や定数の宣言箇所に記述されたドキュメントを認識し，コーディングのヒントとしてコードエディタ上にツールチップ等で表示する機能を有する．また，コードエディタ上でプログラマが関数や値をクリックして，実際の宣言箇所へジャンプできる機能もある．こうしたことから，ライブラリのヘッダファイルにはユーザのための情報が適切に記述されていることが望ましい．Siv3D のヘッダファイルの書き方について採用した方針を説明する．

#### 2.6.1 ドキュメンテーションタグ

2021 年現在の主要なコードエディタで最も広くサポートされている C++ ドキュメンテーションタグの記述方式は Doxygen である（そのほかに，一部のコードエディタでサポートされる記述方式として XML コメント [8] がある）．

Siv3D では，Doxygen タグのうち，関数やクラスの機能の概要を説明する`@brief`タグ，引数を説明する`@param`タグ，テンプレート引数を説明する`@tparam`タグ，戻り値がある場合にその意味を説明する`@return`タグをすべての API に記述することを目標とした．必要に応じて`@remark`タグで，関数の副作用や使い方の注意など補足の説明を行った．例えば，`std::basic_string::size()`と`std::basic_string::length()`がそうであるように，同じ挙動をする別名の関数については，ユーザが混乱しないよう，お互いが同じものであることを明記した．

#### 2.6.2 public メンバと private メンバの順序

ヘッダに記述するクラスの public メンバを private メンバよりも前に配置することで，ヘッダファイルを閲覧したユーザが，必要とするドキュメントに早くたどり着けるようにした．

#### 2.6.3 .ipp ファイルの導入

ヘッダファイルに記述されるインライン関数の実装は，ヘッダファイルの行数を増加させる．ユーザの関心が高いインタフェースやドキュメンテーションが画面に出現する頻度が低下し，一覧性や検索性が損なわれる．Siv3D では当初，.hpp ファイル + .cpp ファイルの構成でこのような問題があったことから，ファイル構成を見直し，インライン関数の定義の記述に特化したヘッダファイル（拡張子 .ipp）を，インライン関数を持つほとんどのヘッダファイルに追加した．導入後，.hpp ファイルには基本的にドキュメントとインタフェースのみが記述されるようになった．

ただし，インライン関数の実装のすべてを .ipp ファイルに移動することはできない．Hidden Friedns [2] の関数の実装をヘッダ内のクラス定義の外に記述すると，そのヘッダをインクルードした翻訳単位では Hidden Friends としての効力が失われる．部分的な解決方法として，Hidden Friends の関数の実装を .cpp ファイルに記述することができる．その場合，関数を実装した翻訳単位（通常はユーザに公開されないライブラリの内部コード）においては Hidden Friends としての効力を失うが，ヘッダをインクルードしたソースファイルでは Hidden Friends としてふるまう．なお，インライン化が必要な constexpr 関数ではこの解決方法が使えないという制約があるため，必要に応じて一部の Hidden Friends であるメンバ関数の定義は .hpp ファイルに残されたままである．

.ipp ファイルの導入前後で，各ファイルに含まれるコードがどのように変化するか，説明用のクラス `A`を用いたサンプルコードを示す．差分に注目されたい．

**.ipp ファイル導入前:**

```
// A.hpp
/// @brief ドキュメント
class A
{
public:

  /// @brief ドキュメント
  A() = default;

  /// @brief ドキュメント
  constexpr A(int a, int b)
    : m_data{ a * b } {}

  /// @brief ドキュメント
  constexpr int getData() const noexcept
  {
    return m_data;
  }

  /// @brief ドキュメント
  constexpr void increment() noexcept
  {
    ++m_data;
  }

  /// @brief ドキュメント
```

```
  void update();

  /// @brief ドキュメント
  friend constexpr bool operator ==(int lhs, const A& rhs) noexcept
  {
    return (lhs == rhs.getData());
  }

  /// @brief ドキュメント
  friend std::ostream& operator << (std::ostream& os, const A& a);

private:

  int m_data;
};
```

```
// A.cpp
#include "A.hpp"

void A::update()
{
  // ...
}

std::ostream& operator << (std::ostream& os, const A& a)
{
  return os << a.getData();
}
```

**.ipp ファイル導入後:**

```
// A.hpp
/// @brief ドキュメント
class A
{
public:

  /// @brief ドキュメント
  A() = default;

  /// @brief ドキュメント
  constexpr A(int a, int b);

  /// @brief ドキュメント
  constexpr int getData() const noexcept;

  /// @brief ドキュメント
  constexpr void increment() noexcept;

  /// @brief ドキュメント
  void update();

  /// @brief ドキュメント
  friend constexpr bool operator ==(int lhs, const A& rhs) noexcept
  {
    return (lhs == rhs.getData());
  }

  /// @brief ドキュメント
  friend std::ostream& operator << (std::ostream& os, const A& a);

private:

  int m_data;
};

#include "detail/A.ipp"
```

```
// A.ipp
constexpr A::A(const int a, const int b)
  : m_data{ a * b } {}

constexpr int A::getData() const noexcept
{
  return m_data;
}

constexpr void A::increment() noexcept
{
  ++m_data;
}
```

```
// A.cpp
#include "A.hpp"

void A::update()
{
  // ...
}

std::ostream& operator << (std::ostream& os, const A& a)
{
  return os << a.getData();
}
```

.ipp ファイルの導入はライブラリコードの総量を増大させるが，ユーザ視点から見るとヘッダの一覧性が改善される．またクラス A のインライン関数が，別の同様に書かれたクラス B に依存していた場合，

- A.hpp → B.hpp

の順でのインクルードはできないが，.ipp ファイルの導入後は

- A.hpp → B.hpp → A.ipp → B.ipp

の順でインクルードすることにより問題なくコンパイルできる．依存関係のためにヘッダ内で constexpr 化できなかったメンバ関数を constexpr 化できるようになり，ライブラリ設計面でのメリットがある．

## 3 ユーザコミュニティの運営

本節では，Siv3D のユーザコミュニティを運営，拡大するためにとった施策と，その成果を報告する．

### 3.1 イベントの実施

Siv3D の普及とユーザコミュニティの発展のために，次のイベントを継続して開催している．

**勉強会** Siv3D の開発環境のセットアップ方法やチュートリアルを解説する，2〜6 時間のイベント．数名〜数十名が参加する．年に数回の頻度で開催．

**実装会** Siv3D のアクティブユーザおよびコミッタを対象に，コードレビューや技術的な助言，開発ツールの使用法指導などを行う．数名〜十数名が参加する．毎月開催．

**チャレンジ** Siv3D に実装したい新機能を課題形式にしてまとめ，ユーザコミュニティに解決をしてもらう，オンラインで数カ月間にわたって開催するイベント．年に 1 回の頻度で開催．

地方各地で開催している勉強会は，現地の大学のサークルやプログラミング教室など，ローカルのコミュニティと協力して開催することで，会場の手配や集客を手伝ってもらっている．実装会は，参加人数を絞る代わりに一人あたりに多くの時間を割いて，技術サポートや高度な使い方の講習を行う．現時点で Siv3D のコードの執筆者は 23 名いるが，そのうち実装会参加後にコミッタとなったメンバーは 16 名と，効果を上げている．どちらも 2020 年のパンデミック以降は，オンラインでの開催にシフトしている．

「チャレンジ」は，Google Summer of Code を参考にしたイベントで，Siv3D に実装されると便利である様々な機能の実装課題をリストアップし，ユーザコミュニティの有志に挑戦してもらうという取り組みである．参加者どうしがコミュニケーションできる専用のグループチャットを用意したり，技術的なサポートやコードレビューを頻繁に提供したりすることで参加者をサポートした．課題の発表時点では具体的な解決法の見当がついていないものもあったが，参加者の調査や研究によっていくつもの課題が達成された．例えば「世界地図や日本地図を表示できるようにしたい」という目標は，地理空間情報を記述するテキストフォーマット GeoJSON のパーサと，読み取ったデータを Siv3D の基本図形クラス Polygon 型に変換する関数の実装によって実現された．

### 3.2 コミュニティ運営に利用したサービス

Siv3D のユーザコミュニティの発展に役立ったツールを紹介する．

**Slack** 登録しているメンバーとメッセージやファイルをやり取りできるグループウェア．ユーザが質問を投稿すると誰かが回答してくれる質問チャンネルや，特定の新機能の開発に関心のあるメンバーが進捗報告や技術的な相談をするチャンネルなどが活発である．

**Discord** 無料で利用できるビデオ通話サービス．若い世代にはコンピュータゲームプレイ時のボイスチャットサービスとして普及している．画面共有機能を持つため，遠隔で動作を確認しながらのサポートにも活用できる．Siv3D コミュニティでは，2020 年のパンデミック以降，オンラインの勉強会や実装会を開催する場合のプラットフォームとして活用機会が増えた．

**Whereby** Web ブラウザ上で追加のアプリのインストールや，ホスト以外のアカウント登録が不要で使えるビデオ通話サービス．画面共有機能を持つ．1〜2 人との小規模な会議で用いた．

**GitHub Sponsors** ソースコードホスティングサービス GitHub 上で 2019 年に始まった，OSS 開発者を金銭的に支援できるサービス．支援の対価として氏名の掲載や個別のユーザサポートを設定するケースが多い．Siv3D はこのサービスを通して，最初の 2 年間で約 2,000 米ドルの資金を得た．

## 4 開発ツール

### 4.1 C++開発ツール

2000 年代後半以降，C++は他のプログラミング言語と比較して，開発支援ツールの発展が見劣りする状況であったが，近年はいくつかのツールの登場により状況が改善している [15]．

大規模な C++ プロジェクトである Siv3D においても，一般的な IDE (Visual Studio や Xcode) の機能に加え，C++コーディングの生産性を高めるための様々なツールを活用した．それらの機能と用途を紹介する．

**Wandbox**[23] 数十種類以上の C++ コンパイラによるコンパイルと，コードの実行結果を確認できるオンラインコンパイラ．コンパイラごとの C++コードの解釈の差異の検証と，エラーや警告メッセージの確認のために使用した．

**Compiler Explorer**[3] 様々なバージョンのコンパイラやコンパイルオプション設定で C++コードをコンパイルし，そのアセンブリ出力を確認できるオンラインコンパイラ．抽象度の高い C++コードを書く際，各コンパイラでオーバーヘッドが生じないかを確認するために使用した．

**Quick C++ Benchmark**[21] 複数の異なるコードの実行時間を相対的に比較して可視化する，ベンチマーク機能を備えたオンラインコンパイラ．バックエンドに Google Benchmark[5] を使用し，その API である，コード中の未使用変数の最適化を抑止する命令などが提供される．Siv3D のコードに近似計算やアルゴリズムの変更を導入する際のプロファイリングに使用した．

**PVS-Studio**[10] C++ や C#, Java を対象とした静的コード解析ツール．実行時性能に影響するコード，冗長なコード，タイプミス，バグが疑われる箇所等を検出できる．

こうしたツールにより Siv3D のコードの問題が検出された事例を紹介する．次のコードでは，`add()`の第一引数で move されているオブジェクトの use-after-move が，`_fmt()`内で発生する．PVS-Studio によって検出された．

```
m_fonts.add(std::move(font),
  U"({0} {1} size: {2})"_fmt(
    font->getFamilyName(),
    font->getStyleName(), fontSize));
```

次の例は目視で気付きにくいミスである．6 行目のリストの最初の要素は`bezier.p0.y`が正しい．PVS-Studio により，疑わしいコードとして検出された．

```
const double minX = std::min(
  {bezier.p0.x, bezier.p1.x, bezier.p2.x});
const double maxX = std::max(
  {bezier.p0.x, bezier.p1.x, bezier.p2.x});
const double minY = std::min(
  {bezier.p0.x, bezier.p1.y, bezier.p2.y});
const double maxY = std::max(
  {bezier.p0.y, bezier.p1.y, bezier.p2.y});
```

### 4.2 Web プラットフォームへの対応

Siv3D の C++コードを入力にして，Web ブラウザで動作する JavaScript コードを得るツールチェーンを，Emscripten などを利用して整備した．詳細な構成は Siv3D のコードリポジトリのビルド手順から確認できる．これにより，Siv3D のプログラムが従来の PC だけでなく，モバイルを含むより広いプラットフォームで動作するようになった．現在 Google Chrome, Microsoft Edge, Mozilla Firefox などの主要 Web ブラウザの最新版での動作を確認している．ネイティブアプリケーション専用の命令を使用していないなど，条件が良い場合，既存のプログラムを 2〜3 行変更するだけで Web ブラウザで動作するようになる．過去に Siv3D で制作したゲーム作品を Web ブラウザでの動作に対応させて再公開するユーザも見られた．

Web 対応にあたっては，従来の Linux 版のコードをベースに OS ネイティブの命令を Web ブラウザで実行可能なコードに置き換える作業や，既存の OpenGL4 レンダリングエンジンをベースに WebGL 2.0 レンダリングエンジンを新しく実装・追加するなど，70 のソースファイルの追加が発生したが，全体から見れば少ない割合の変更で Web への移植が実現したことは特筆すべきである．

## 5 今後の課題と，C++ やツールへの要望

本節では，これからの Siv3D で解決したい課題と，C++や開発ツールが将来どのように進化・整備されると望ましいかについて，Siv3D の開発・運用経験を踏まえた提言を行う．

### 5.1 Siv3D の今後の目標・課題

#### 5.1.1 Web 版のサイズ圧縮

現在の Web 版 Siv3D は，PC 版とほぼ共通のコードやアセットファイルを使用するため，実行時に必要となるファイルサイズについてはあまり考慮されていない．ネットワーク速度に依存するが，現在の Web 版の Siv3D アプリケーションは，Web ブラウザでページにアクセスしてから実行までに平均で 5〜15 秒前後かかり，その大半はファイルのダウンロー

ドに要している．

これらの改善のために，機能が制限されるが必要なファイルサイズがコンパクトになるようなビルドオプション，単一のファイルで複数のウェイトや属性のフォントを生成できる variable fonts の使用によるフォントサイズ圧縮などを検討している．

#### 5.1.2 サブセットライブラリ

Siv3D はプログラムのエントリポイントのほか，サブシステムの初期化や状態更新，連携をライブラリ側のコア部分が担っている．各種 API とコア部分の依存関係が密接であるため，Siv3D が提供する C++のユーティリティクラスや文字列，画像処理など，一部だけをその機構から独立に抜き出して再利用することが難しい．つまり，Siv3D を使っていない既存のプロジェクトにおいて，Siv3D の一部の機能だけを使いたい場合，Siv3D がエントリポイントを握るプログラムのほうに既存のコードを持ち込むという主従関係が要求される．

この制約は Siv3D のコード資産が多くのプロジェクトで活用される機会を狭めている．Siv3D が管理するサブシステムに依存しないスタンドアローンな API だけを切り出したサブセットライブラリを提供できると望ましい．サブセットライブラリの利用者が増えることで，その部分の改良やテストが加速し，それが本体に反映される効果も期待される．

### 5.2 C++ への要望

#### 5.2.1 移植性のある乱数分布，シャッフルの実装

C++の規格では，標準ライブラリの乱数分布や範囲要素のシャッフルに使われる細かいアルゴリズムを規定していないため，疑似乱数生成エンジンに同じシードを与えて乱数ソースの再現性を確保しても，処理系によって異なる実装を持つ乱数分布やシャッフルの操作を通過することで実行結果が一致せず，プログラムの移植が妨げられる問題がある．

標準の C++では，次のような単純なコードでさえ処理系によって実行結果が異なる．

```
#include <iostream>
#include <random>

int main()
{
  std::mt19937 rng{ 12345 };

  std::uniform_int_distribution<int>
    ud{ 1, 6 };

  std::cout << ud(rng) << '\n';
}
```

これらの機能に関して，Siv3D では標準ライブラリに依存せず，代替となる共通の実装を用意した．乱数分布については専門的な数学知識が要求されるため，自前での実装は行わず，ヘッダオンリーで必要な機能がそろう Abseil C++ library [4] を用いた．

この問題は標準ライブラリの範疇で解決できればメリットが大きい．`std::portable`といった名前空間に，移植性が保証された代替関数を提供することが考えられるだろう．

#### 5.2.2 constexpr 数学関数

平面の法線，光源の方向などの 3 次元ベクトルについて，軸に平行でない単位ベクトルを，次のようなコードでコンパイル時定数にしたいが，ベクトル正規化の計算で使われる平方根関数 `std::sqrt()`は constexpr でないためコンパイルできない．

```
constexpr Vec3 v
  = Vec3{ 1.0, 1.0, 1.0 }.normalized();
```

3DCG に関するライブラリ API は，Siv3D に限らず，計算量の節約のため，正規化されているベクトルを入力の前提条件としている場合が多い．ベクトル正規化の計算を constexpr 化できれば，より多くの計算をコンパイル時に評価でき，実行時性能やコードの記述性の向上につながる．

C++23 以降に向けて，標準ライブラリの一部の数学関数の constexpr 化が提案されている [12] ため，進展に期待したい．

#### 5.2.3 UTF-16, UTF-32 サポートの拡充

ひらがなや漢字を含む文字列について，各文字へのランダムアクセスをシンプルなコードで記述できるよう，現在の Siv3D では UTF-32 を基本の文字コードとしている．当初の Siv3D は Windows 版から開発が始まった歴史的経緯で，Windows のネイティブ API と互換性のある `wchar_t`型 (Windows では UTF-16) を採用していたが，`wchar_t`型は処理系により文字

コードおよびサイズが異なるため移植性に難があり，途中からは，文字コードが規定されている型に移行することにした．UTF-8 は空間効率に優れるが，ひらがなや漢字，さらに複数のコードポイントからなる絵文字を含む場合の取り扱いが難しかったため UTF-32 を選んだ．Siv3D のプログラムの文字列リテラルに使われるプレフィックス `U`は `char32_t`型の文字列リテラルであることを示す．

残念ながら，C++標準ライブラリの文字列処理 API は，`std::basic_string`や `std::basic_string_view`を除き，`char`型，よくても `wchar_t`型までのサポートしか保証されていない．標準ライブラリ API のクラステンプレート引数にこれら以外の型を指定してもコンパイルに問題が生じるケースがほとんどである．正規表現，`std::format`, 標準入出力などで，`char16_t`, `char32_t`がサポートされると，文字列に UTF-8 以外を採用するライブラリ開発者の負担が減る．

### 5.3 開発・整備が望まれるツールの提言

#### 5.3.1 テスト用ファイルのコレクション

各種ファイル形式の読み込みの検証に使える，あらゆる種類・内部形式のテストファイルが一か所に集約されていると便利である．例えば JPEG ファイルであれば，画像の大きさが最小のもの，最大のもの，画像データの内部フォーマットが特殊なもの，Exif データを含むものなど，ユーザ環境で登場しうるあらゆるバリエーションのファイルが用意され，プログラムがそれらを適切に扱えるかベンチマークできるファイル群である．こうしたものを，画像以外も含む数百種類以上のファイル形式について網羅し，開発者がアクセスしやすい形で提供されると，Siv3D のように，様々なファイル形式を扱うライブラリ開発者の助けになる．

#### 5.3.2 軽量なカラー絵文字フォント

Siv3D では移植性のために，あらゆるプラットフォームで，共通のカラー絵文字の TrueType フォントファイルをアプリケーションに同梱している．ファイルには現時点で約 3,600 文字が収録され，サイズは 8.6 MB（Zstandard 圧縮後）と，アプリケーションを構成するファイル群の中でも一番の容量を占めている．

**表 2　Siv3D に登場する用語とその言い換え例**

| 用語 | 言い換え例 |
|---|---|
| 描画 | 表示，描く，出す，出力 |
| 音声 | オーディオ，サウンド，音楽，BGM (background music)，効果音，SE (sound effect) |
| 透過 | 半透明，透ける |
| 座標 | 位置，場所，地点 |
| 動画 | ビデオ, ムービー |
| テキスト | 文字列，文章，メッセージ |
| フレームレート | FPS (frames per second) |

Unicode Emoji は例年 100〜300 文字前後のペースで追加されており [22]，フォントファイルのサイズもますます大きくなっていくだろう．カラー絵文字を現在よりも小さいサイズで保存できるベクター画像形式 [6] の普及や，ファイルサイズを優先してデザインされた絵文字の登場が望まれる．

#### 5.3.3 言い換え表現に強いドキュメント検索

ユーザが特定の操作や機能の実現方法をドキュメントやライブラリヘッダ，Web で検索する際，言い換え表現がヒットせず，別の用語を使っていれば得られていたはずの情報にたどり着けないことがある．Siv3D プログラミングで頻繁に使われる用語の中から，言い換え表現が存在する例を表 2 にまとめた．

こうした言い換え表現を検索時のヒントとして登録し，検索結果に反映してくれるドキュメンテーションジェネレータや検索エンジンがあると便利である．

## 6 おわりに

本稿では，Siv3D の実例を通して，C++ライブラリ API 設計のテクニックと，開発や普及に関する経験を説明した．C++の言語の進化は，C++14 から C++20 にかけての規格書のページ増分だけで測っても 495 ページと膨大である．ライブラリ開発者がユーザの期待や，言語の要求する水準に応えていくためには，設計技法や開発生産性向上に関する議論と知見の共有がますます必要になっていくだろう．本稿の事例がそのような流れの一助になれば幸いである．

## 参 考 文 献


[ 1 ] Bilas, S.: An Automatic Singleton Utility, *Game Programming Gems. Mass: Charles River Media, Inc*, (2000).

[ 2 ] Brown, E. W. and Sunderland, D.: Recommendations for Specifying "Hidden Friends", Technical report, March 2019. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1601R0.

[ 3 ] Godbolt, M.: Compiler Explorer (website). `https://godbolt.org/`, (also at Internet Archive `https://web.archive.org/web/20210803150815/https://godbolt.org/`).

[ 4 ] Google: Abseil C++ library (code repository). `https://github.com/abseil/abseil-cpp`.

[ 5 ] Google: Benchmark (code repository). `https://github.com/google/benchmark`.

[ 6 ] Google: IconVG (code repository). `https://github.com/google/iconvg`.

[ 7 ] Gregory, J.: *Game engine architecture*, crc Press, 2018.

[ 8 ] Microsoft: XML Documentation. `https://docs.microsoft.com/en-us/cpp/build/reference/xml-documentation-visual-cpp?view=msvc-160`.

[ 9 ] Nystrom, R.: *Game programming patterns*, Genever Benning, 2014.

[10] PVS-Studio: PVS-Studio (website). `https://pvs-studio.com/en/pvs-studio/`, (also at Internet Archive `https://web.archive.org/web/20210724220841/https://pvs-studio.com/en/pvs-studio/`).

[11] Reddy, M.: *API Design for C++*, Elsevier, 2011.

[12] Rosten, E. J. and Rosten, O. J.: constexpr for <cmath>and <cstdlib>, Technical report, June 2021. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0533R8.

[13] Shen, T. and Smith, R.: Designated Initialization Wording, Technical report, July 2017. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0329R4.

[14] Sommerlad, P.: [[nodiscard]] for constructors, Technical report, July 2019. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1771R1.

[15] Stroustrup, B.: Thriving in a crowded and changing world: C++ 2006–2020, *Proceedings of the ACM on Programming Languages*, Vol. 4, No. HOPL(2020), pp. 1–168.

[16] Stroustrup, B. and Sutter, H.: C++ Core Guidelines (website). `https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines`, (also at Internet Archive `https://web.archive.org/web/20210803200203/https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines`).

[17] Suzuki, R.: siv::NamedParameter (code repository). `https://github.com/Reputeless/NamedParameter`.

[18] Suzuki, R.: ちょっと変わったプログラミング教室-プログラミングで絵, 音, 3D 物体を描こう, 動かそう-: 2. 音や画像で遊ぼう-インタラクティブアプリケーションための C++ フレームワーク「Siv3D」, 情報処理, Vol. 58, No. 6(2017), pp. 474–480.

[19] Suzuki, R. and Choh, I.: C++14 における名前付き引数の実装, 情報処理学会第 *79* 回全国大会講演論文集, Vol. 2017(2017).

[20] Suzuki, R. and Choh, I.: Siv3D: インタラクティブアプリケーションのための C++ フレームワーク (サイバー増大号)–(特集 ソフトウェア論文), コンピュータソフトウェア= *Computer software*, Vol. 34, No. 4(2017), pp. 17–38.

[21] Tingaud, F.: Quick C++ Benchmark (website). `https://quick-bench.com/`, (also at Internet Archive `https://web.archive.org/web/20210620011638/https://quick-bench.com/`).

[22] Unicode: Emoji Versions (website). `https://unicode.org/emoji/charts/emoji-versions.html`, (also at Internet Archive `https://web.archive.org/web/20210729211342/https://unicode.org/emoji/charts/emoji-versions.html`).

[23] Wandbox: Wandbox (website). `https://wandbox.org/`, (also at Internet Archive `https://web.archive.org/web/20210708011025/https://wandbox.org/`).