2025 年 1 月　臨地実務実習レポート

椎野 泰成

目次

今回の臨地実務実習の内容

2025 年 1 月の臨地実務実習では、課題を与えられその課題のコードレビューを担当や、代表に行ってもらいながら進めるものだった。Git のリポジドリを作成し、その中に課題を置きブランチを切りながら課題を進めた。出された課題は以下の 4 種類だった。

1. 画像データの入力と出力及び、画像ファイル形式の相互変換プログラムの作成
2. Direct3D11 を使用し、obj ファイル（Stanford Bunny） の描画と、シェイディング処理の作成
3. コマンドコンソールにおける、コマンドデザインパターンを使用した計算器の作成と、それを使用した単体テストの実行
4. 静的メモリに割り当てる関数オブジェクト（inplace_function）

一つ目の課題では、**TGA**、**BMP**、**DDS** ファイルの読み込み、書き出し、相互変換を行った。また、**TGA** ファイルは RLE 圧縮を使用していることがあるため、その解凍及び、圧縮処理の作成を行った。この課題ではポインタを使用したメモリアクセス及び、変数型を使用しキャストすることでデータを読み込むことが課題の主題だったと思われる。

二つ目の課題は、ImGui 、Direct3D11 を使用し以下のことを順に行った。ImGui はデバッグのために使用した。

1. 三角形の描画
2. 四角形の描画
3. テクスチャ付きの四角形の描画
4. **セピア色フィルター**の適用
5. **グレースケール**の適用
6. obj ファイル（Stanford Bunny）の描画
7. **Phong shading** の適用
8. **Bloom** の適用
9. obj ファイルの読み込み処理の速度を向上させる

詳細については別で記述するが、課題の主題としては Direct3D11 の全体の仕組みを理解することが主題だったと思われる。

三つ目の課題は、コマンドデザインパターンを使用した計算器の開発だった。コンソールへ数値、演算子を入力しそれを計算するプログラムの作成を行った。このプログラムでは

Undo、Redo 処理もコマンドデザインパターンにより作成し、また逆ポーランド記法に変換し計算、Google Test を使用した単一テストも行った。

最後に 4 つめの課題は、静的メモリに割り当てる関数オブジェクト（inplace_function）の作成だった。臨地実務実習の最終日から始めたため触りのみだったが、静的メモリに関数オブジェクトを割り当て、呼び出すところまでの作成を行った。

臨地実務実習の課題としては以上の 4 点を行った。またすべての課題を通して設計レベルでの改善も平行して行った。具体的にはデザインパターンや結合度、凝縮度などを意識し設計することをコードレビューを通じて行った。
以下ではそれぞれの課題を通じて得た知識、技術等や課題の詳細について記述する。

課題 1　画像ファイル形式の相互変換プログラム

この課題ではコマンドライン引数に以下のように値を入力し、プログラムを実行することで変換を行うものを作成した。

C++
`test.exe /i 入力ファイルパス /o 出力ファイルパス`

/i と**/o** は順不同で入力できるようにし、入力ファイルパスのファイルを読み込み、出力ファイルパスのファイルの拡張子で出力するのが課題だった。
担当の方から出された課題の狙いとしては以下の 3 点。

1. ポインタの扱いを理解する
2. **C++11**、**C++17** などのモダンな書き方を理解する
3. デザインパターンを理解する

実際これらはこの課題を通じて身についたものだった。また、命名規則のみのものではあったがコーディングルールに従いながら開発を行った。
他にあった指定としてはソースファイルを **UTF-8 BOM 付き**で作成するというものだった。これは Git の Web ページ上において、日本語でソースファイルにコメントが記述されていても、文字化けせず読めるようにするためのものだった。
また、Git の作業の流れとして以下を説明された。

1. ローカルで main ブランチから作業**ブランチを作成**する
2. 作業ブランチに**切り替え**する
3. 作業ブランチに**コミット**する
4. 作業ブランチを**プッシュ**する
5. GitHub で**プルリクエスト**を作成する
6. レビューをもらったら**マージ**する
7. ローカルで main ブランチに**切り替え**てから**プル**をして GitHub の更新を取得する

この作業の流れは実際に一か月を通じて行った。また、以下で記述する内容の順番などはプルリクエストの順に記述している。

コマンドライン引数を処理し、ファイルパスを取得

C++の main 関数でコマンドライン引数を受け取るには以下のようにする必要がある。

C++

```
int main(int argc, char* argv[])
{
    return 0;
}
```

argc は **argv** の要素数であり、**argv** 内の **char*** データから入力を受け取る。

以下が最終的に出来上がったコマンドライン引数を処理するプログラムである。

C++

```
// 引数の数が合わない場合、エラーを出力して終了
if (argc != 5)
{
    cout
    << "引数の数が合いません。以下の例のように実行してください。"
    << endl;

    cout
    << "image_format_converter.exe /i ファイルパス /o 出力フォルダ"
    << endl;

    return ERROR_INVALID_ARGUMENTS;
}

// /i、/o がそれぞれ一つずつ指定されているか確認
int isArgCorrect = 1;
isArgCorrect *=
(
    string(argv[1]) == "/i" && string(argv[3]) == "/i"
) ? 0 : 1;

isArgCorrect *=
(
    string(argv[1]) == "/o" && string(argv[3]) == "/o"
) ? 0 : 1;

if (isArgCorrect != 1)
```

```
{
    cout << "引数が不正です。" << endl;
    return ERROR_INVALID_ARGUMENTS;
}

string importPath;
string exportPath;
for (int i = 1; i <= 3; i += 2)
{
    // 入力ファイルパスを取得
    if (string(argv[i]) == "/i") importPath = argv[i+1];

    // 出力フォルダパスを取得
    else if (string(argv[i]) == "/o") exportPath = argv[i+1];
}

// 引数が正しく取得できているか確認
if (importPath.empty() || exportPath.empty())
{
    cout << "読み込めるファイル形式が見つかりませんでした。" << endl;
    return ERROR_FILE_LOAD_FAILED;
}
```

これらは単純な文字列操作の一つである。注意点としては、**argv** は **char*** 型であるため以下のようには比較できないことだ。

C++
`argv[1] == "/i"`

char*型は文字列の先頭のポインタであるため、このようには比較できない。

C++での定数

C++では**#define** は使わず以下のように **constexpr** や **eumu class** を使用する。

C++

```
constexpr unsigned int SUCCESS = 0;


enum class ERROR
{
    INVALID_ARGUMENTS,
    FILE_LOAD_FAILED
};
```

C の従来の**#define** だと型を持たないため、型チェックが行われない。しかし、**constexpr** は型を持つためコンパイル時にチェックされ、より安全である。また、**enum class** と **enum** の違いは以下のような点であり、**enum** ではなく **enum class** を使うことが C++では推奨される。

特徴	enum	enum class
スコープの有無	なし	あり
暗黙の整数変換	可能	不可能
名前衝突	しやすい	しにくい
型の安全性	低い	高い

文字列の受け渡し時のコピー

C++17 では std::string_view が存在し、これは文字列を関数などに渡す際に元の文字列をコピーせず渡せるものである。コピーする必要がないときは基本的にこれを使用するべきだと知った。使い方としては以下のようになる。

C++

```
FileAnalysis(string_view importPath);
```

また、**string&** で行う手段もあるが、C++である以上標準ライブラリを使用したものの方が好ましい。

標準ライブラリらのプリコンパイルヘッダーを作成して使用

プリコンパイル済みヘッダー とは多数の**#include** 文を含むヘッダーファイルをプリコンパイルすることで、時間とリソースを節約できるものである。基本的にプログラム開発では使用するべき。Visual Studio では以下の設定を行うことでプリコンパイル済みヘッダーをしようすることが可能になる。

1. **pch.h** 、**pch.cpp** をプロジェクトへ追加
2. プロジェクト → プロパティ → C/C++ → プリコンパイル済みヘッダー → プリコンパイル済みヘッダーを**使用（/Yu）**に変更
3. プロジェクト → プロパティ → C/C++ → 詳細設定 → 必ず使用されるインクルードファイルを **pch.h** に変更
4. **pch.cpp** → プロパティ → C/C++ → プリコンパイル済みヘッダー → プリコンパイル済みヘッダーを**作成(/Yc)**に変更

C++

```
// pch.h
#pragma once

#include <vector>
#include <iostream>
#include <memory>
#include <string>
#include <string_view>

// pch.cpp
#include "pch.h"
```

プリコンパイル済みヘッダーは別のヘッダーファイルではインクルードせず、必ずソースファイルのみでインクルードする。ヘッダーファイルで標準ライブラリを使用する場合はその都度標準ライブラリをインクルードする。これにより、Visual Studio の設定が変わろうともプリコンパイル済みヘッダーが使用されないだけで、ビルドは通るようにできる。

std::map、std::vector を使用した、変換対象ファイル形式の管理

今回のプログラムでは取得したファイルパスの拡張子から、読み込むファイル形式と、出力するファイル形式を取得し処理を行う必要がある。その際、以下のように処理を記述してしまうと、対応する画像ファイルが増えるたびに分岐処理が増え、大量の修正が必要になる。

C++

```
if (ext == "tga")
{
    cout << "TGA ファイル処理" << endl;
}
else if (ext == "bmp")
{
    cout << "BMP ファイル処理" << endl;
}
```

そこで以下のように **std::map** を使用し、拡張子から処理を分岐させるようにすることで、分岐処理を対応する画像ファイル形式が増えるたびに追加する必要がなくなり、その画像ファイル形式の処理を行うクラスのインスタンスを要素に追加することで処理を簡単に追加できる。

C++

```
std::map<std::string, std::unique_ptr<IConverter>> converters_;
converters_["bmp"] = std::make_unique<Bmp>();
converters_["tga"] = std::make_unique<TGA>();

converters_[ext]->analysis();
```

また、**IConverter クラス**を以下のようにすることで **std::map** ではなく、**std::vector** などでも管理、分岐処理することができる。

C++

```
class IConverter
{
public :
    IConverter() = default;
    virtual ~IConverter() = default;

    bool judgeExt(std::string_view importPath);
```

```
    virtual std::unique_ptr<FileData> analysis
    (
        std::unique_ptr<u8[]>& importData
    ) = 0;
};

std::vector<std::unique_ptr<IConverter>> converters_;
converters_.emplace_back(std::make_unique<BMP>());
converters_.emplace_back(std::make_unique<TGA>());

for (int i = 0; i < converters_.size(); i++)
{
    if (converters_[i]->judgeExt(importPath))
    {
        converters_[i]->analysis(importData);
    }
}
```

以上のどちらかのようにすることで、**IConverter クラス**を継承するそれぞれのファイル形式の処理を持つクラスを作成し、**std::map** 、**std::vector** に追加することで分岐処理を都度追加することなく、新たな画像ファイル形式に対応できる。

凝集度を意識することで、単一で明確に定義された責任を持たせる

凝集度とは、モジュールやコンポーネント内の要素がどれだけ関連し、単一で明確な目的に集中しているかを示す指標。高い**凝縮度**を持つモジュールは、単一で明確に定義された責任を持つため、理解しやすく保守しやすい、良いプログラムになる。

凝縮度には以下の 7 つのレベルがある

レベル	名称	説明
7（最高）	機能的凝集度	モジュール内の要素が単一の明確に定義された機能や責任によって関連している状態
6	順序的凝縮度	モジュール内の要素が特定の順序で実行されることによって関連付けられている状態
5	通信的凝縮度	モジュール内の要素が操作するデータや通信するデータによって関連している状態
4	手続き的凝縮度	モジュール内の要素が特定のタスクを達成するため実行する順序によって関連している状態
3	時間的凝縮度	モジュール内の要素が特定の時間によって関連している状態
2	論理的凝縮度	モジュール内の要素が論理や機能によって関連しているが、単一の明確な目的を持たない状態
1（最低）	偶発的凝縮度	モジュール内の要素が意味のある関連性や共通の目的を持たない状態

以下の例ではファイルの書き込みを行っている。しかし、この実装だとファイルを書き込む関数に、ファイル名からファイルパスを作る処理が入ってしまっていることがわかる。

C++

```
bool FileWrite
(
    std::string_view folderPath, std::string_view writeFileName,
    std::unique_ptr<unsigned char[]>& fileData,
    const unsigned int dataSize
){
    FILE* fp = nullptr;
    errno_t error;

    string writeFilePath = folderPath.data();
    if (writeFileName.find_first_of("¥¥") != writeFileName.size())
```

```
    {
        writeFilePath += "¥¥";
    }

    writeFilePath += writeFileName.data();

    error = fopen_s(&fp, writeFilePath.data(), "wb");
    if (error != 0)
    {
            printf("ファイルが読み込めませんでした");
            return false;
    }

    fwrite(fileData.get(), sizeof(unsigned char), dataSize, fp);
    printf("ファイルを書き込みました");
    fclose(fp);

    return true;
}
```

この実装だと、FileWrite がファイルパスの作成と、ファイルの書き出しの二つの責任を持ってしまっている。そのため以下のようにすることで、機能的凝縮になる。

C++

```
std::string SetFilePath
(
    std::string_view folderPath, std::string_view fileName
){
    string path = folderPath.data();
    if (path.find_first_of("¥¥") != path.size())
    {
        path += "¥¥";
    }

    path += fileName.data();
    return path;
}
```

```
bool FileWrite
(
    std::string_view path,
    std::unique_ptr<unsigned char[]>& fileData, unsigned int dataSize
){
    FILE* fp = nullptr;
    errno_t error;

    error = fopen_s(&fp, writeFilePath.data(), "wb");
    if (error != 0)
    {
            printf("ファイルが読み込めませんでした");
            return false;
    }

    fwrite(fileData.get(), sizeof(u8), dataSize, fp);
    printf("ファイルを書き込みました");
    fclose(fp);

    return true;
}

std::string path = SetFilePath(folderPath, fileName);
bool isSuccess = FileWrite(path, fileData, dataSize);
```

このように、**機能的凝集度**を目指して高い**凝集度**を維持することができれば、関数やクラスが行うことや責任が明確になり、保守しやすく、理解しやすいコードになる。そのため**凝縮度**は、ソフトウェア開発において非常に重要である。

結合度を意識することで、不具合が起きた際に調べる範囲を限定する

結合度とは、利用者またはメンテナンスをする者にとって対象を利用、保守しやすいように対象の内容が整理、分割できているかどうかを、その状態によって段階に分けて表現するソフトウェア測定法の一種。

結合度は以下の 7 つのレベルで表せる。

レベル	名前	内容
1（最高）	メッセージ結合	引数のないメソッドの呼び出しを行うパターン
2	データ結合	単純なデータの受け渡しのみを行うパターン
3	スタンプ結合	引数で構造体などのオブジェクトを渡すパターン
4	制御結合	引数の種類によって、メソッドの内の処理が変わるパターン
5	外部結合	単一のグローバルデータを参照しているパターン
6	共通結合	複数のグローバルデータを参照しているパターン
7（最低）	内容結合	他のオブジェクトの内部を参照しているパターン

以下の例では、ファイルの解析及び、変換を行うクラスである **Converter クラス**が、ファイルのデータを持っている。

C++

```
class Converter
{
private:
    int width = 0;
    int height = 0;
    std::unique_ptr<unsigned char[]> pixels = nullptr;

public:
    Converter() = default;
    ~Converter() = default;

    void analysis(unique_ptr<unsigned char[]> src, unsigned int size);
    std::unique_ptr<unsigned char[]> convert();
};
```

この例では、ファイルの解析及び、変換を行うクラスであるのにも関わらず、ファイルのデータを持ってしまっているため凝縮度としても問題はあるが、結合度としての問題は、**画像データを持つ変数に対して、複数の関数事が関わっていること**である。

これはそれらの関数が、今回の場合、画像データを持つ変数に対して**依存**しているということになる。こうなってしまうと何らかの不具合が生まれた際に、関数がその変数に依存していることで、その変数が関わっているすべての処理で不具合が生まれる可能性が出てくるだけでなく、変数に対して処理が行われる順番にも何の保証もないことでプログラム全体に不具合が生まれる可能性が出てきてしまう。こうなると不具合を修正することがとても困難になる。

これは**結合度**のレベルで言うと**外部結合**や**共通結合**、**内部結合**にあたり、悪い設計にあたる。そのため、以下のように修正することで**データ結合**になり、良い設計になる。

C++

```
struct FileData
{
    int width = 0;
    int height = 0;
    std::unique_ptr<unsigned char[]> pixels = nullptr;
};

class Converter
{
public:
    Converter() = default;
    ~Converter() = default;

    std::unique_ptr<FileData> analysis
    (
        unique_ptr<unsigned char[]> src,
        unsigned int size
    );

    std::unique_ptr<unsigned char[]> convert
    (
        std::unique_ptr<FileData>& fileData
    );
};

Converter converter;
std::unique_ptr<FileData> fileData = converter.analysis(src, size);
```

```
std::unique_ptr<unsigned char[]> exportFileData
= converter.convert(fileData);
```

この修正でデータ結合になり、変数に対して処理が行われる順番が保証されるだけでなく、データのライフサイクルが明確になることで不具合が生まれる可能性のある範囲が限定される。そのため不具合が生まれる原因の発見は容易になりやすい。また、凝縮度の観点から見ても **FileData**、**Converter** がそれぞれ単一の責任を持っていることで良い設計になっている。
このように、**結合度**を意識した開発により、不具合が起きた時に調べる範囲が限定されることで保守性の高いプログラムになる。また、結果的に凝縮度の観点から見ても良い設計になることが多いため理解しやすくもなり、良いプログラムにすることができる。

std::unique_ptr と std::shared_ptr のどちらを使うか

std::unique_ptr と std::shared_ptr は所有権の扱い方に違いがあり、どちらの方が好ましいかは状況によって異なる。それぞれのメリットとデメリットは以下のようになる。

種類	メリット	デメリット
unique_ptr	■ 所有権が明確 ■ 軽量で効率的 ■ カスタムデリータの指定が可能 ■ スコープを外れると自動解放	■ 複数所有が不可能 ■ コピー操作が禁止（ムーブのみ可能）
shared_ptr	■ 複数箇所でリソースの共有が可能 ■ 最後の参照が消えると自動解放 ■ コピー操作が可能	■ 参照カウント管理によるオーバーヘッド ■ 循環参照の可能性 ■ カスタムデリータの実装が複雑

このように、**unique_ptr** は単一の所有権を持つケースで効率的に使用でき、**shared_ptr** は複数の所有者が必要なケースで使用することが適切である。しかしここで注意するべきこととして複数の所有権がどういうものなのかということだ。所有権とは物を全面的に支配する物件、目的物を自由に使用・処分できる権利ということであり、要素に対してアクセスして使用するというだけなら所有しているわけではない。このようなケースは以下のような場合である。

C++

```
class FileContainer
{
private:
    std::shared_ptr<FileData> data;
public:
    void createData();
    void clearData();
    std::shared_ptr<FileData> getData();
};

class Converter
{
private:
    std::shared_ptr<FileData> data;
public:
    void setData(std::shared_ptr<FileData>);
    void analysis(unique_ptr<unsigned char[]> src, unsigned int size);
};
```

FileContainer 、**Converter** らがともにファイルのデータを所有していることになっているが、この場合だとファイルデータの**所有権**を持っているのは作成の削除を行う **FileContainer クラス**である。**Converter クラス**はこのデータに対してアクセスし内容を **analysis 関数**で変更するが、作成や削除など所有権を持つと言える処理は何も持っていない。ならどうすれば所有権を明確にできるかというと、**unique_ptr<FileData>&** とし参照を使用するもしくは、関数の引数で渡すようにするかだ。

C++

```
class FileContainer
{
private:
    std::unique_ptr<FileData> data;
public:
    void createData();
    void clearData();
    std::unique_ptr<FileData>& getData();
};
```

```
class Converter
{
private:
    std::unique_ptr<FileData>& data;
public:
    void setData(std::unique_ptr<FileData>& data);
    void analysis(unique_ptr<unsigned char[]> src, unsigned int size);
};
```

これは結合度の観点から見れば良くない設計ではあるが、所有権は明確になっている。

このように、所有権という言葉の意味からしても **shared_ptr** の使用は慎重になるべきことを知った。多くの場合で一つの物の所有権を持つオブジェクトは一つであり、**unique_ptr&** を使用することが良い場合が多い。今回のような小さいプログラムの場合はなおさら **shared_ptr** の使用は控えるべきである。

Include-what-you-use を意識したインクルード

C++ではヘッダーファイルで別のヘッダーファイルをインクルードする。その際に適切にインクルードされていないと多重定義によるエラーや、未定義シンボルエラー、コンパイル時間の増加を引き起こす。そうならないために適切にヘッダーファイルをインクルードする考え方が **include-what-you-use** である。以下の例を用いて説明する。

C++

```
// file_data.h
class FileData
{
    int width = 0;
    int height = 0;
    std::unique_ptr<unsigned char[]> pixels = nullptr;
};

// file_container.h
#pragma once

#include "file_data.h"
```

```
class FileContainer
{
private:
    std::unique_ptr<FileData> data;
public:
    void createData();
    void clearData();
    std::unique_ptr<FileData>& getData();
};
```

この例では **FileContainer クラス**で **FileData クラス**を使用している。しかしこの例では以下のようにすることでヘッダーファイルではインクルードせず、ソースファイルでのみインクルードすることができる。

C++

```
// file_container.h
#pragma once

class FileData;

class FileContainer
{
private:
    std::unique_ptr<FileData> data;
public:
    void createData();
    void clearData();
    std::unique_ptr<FileData>& getData();
};

// file_container.cpp
#include "file_container.h"
#include "file_data.h"
```

このようにヘッダーファイルで対象のクラスなどの定義が必要ない場合は前方宣言にすることで、適切なインクルードを行うことができる。これによりインクルードが原因で起こる不具合を避けることに繋がるため、積極的に前方宣言にすることはできないか、本当にインクルードの必要性があるかは考え続ける必要がある。

reinterpret_cast を使用したバイナリデータの解析

fopen_s(&fp, importPath.data(), "rb"); で読み込んだファイルのバイナリデータを解析するには、以下のように読み込む先頭のポインタを動かしながらキャストし、構造体にデータを読み込む。

C++

```
#pragma pack(push, 1)
struct BmpFileHeader
{
    unsigned short fileType;
    unsigned int fileSize;
    unsigned short fileReserved1;
    unsigned short fileReserved2;
    unsigned int fileOffBits;
};
#pragma pack(pop)

std::unique_ptr<unsigned char[]> importData = load(filePath);
BmpFileHeader* fileHeader = reinterpret_cast<BmpFileHeader*>
(
    importData.get()
);
```

ここで注意しなければいけないことは以下の 2 点である。

1. reinterpret_cast はポインタを動かさないため、次のデータを読み取るには再度そのデータの先頭にポインタを設定しなければいけない
2. キャストする先の型は必ず pragma push 、pack をしなければいけない

一つ目は reinterpret_cast の使用方法についてのものであるが、二つ目はメモリのアラインメントにより正しいサイズの構造体にバイナリのデータをキャストできなくなることを防ぐものである。
コンパイラは構造体メンバ間にパディングバイトを挿入し、構造体のサイズやレイアウトを変える可能性があるため、このように#pragma pack(push, 1) を使用することで、構造体のメンバが 1 バイト単位で詰められ、パディングが入らなくなる。そのため BMP ファイルのヘッダーフォーマットと一致し、正しくキャストできるようになる。#pragma pack(pop) はアラインメントの設定をもとに戻すものであり、基本的に#pragma pack(push, n) をしたら行う必要がある。

画像ファイル形式変換プログラムにおける、適切な設計デザイン

「std::map、std::vector を使用した、変換対象ファイル形式の管理」でも触れたが、後々様々な画像ファイル形式を追加することが想定される中で、保守性が高く、理解しやすいプログラムを作成するために考えなければいけないことは以下の 3 点である。

1. 読み込み、書き込むファイル形式が違う以上、画像ファイルの何のデータを読み込み、保存する必要があるか
2. 画像ファイル形式ごとに異なる処理を行わなければいけない処理は何か
3. どうやって、ファイルの拡張子から画像ファイル形式ごとに異なる処理を呼び出すのか

まず一つ目の「読み込み、書き込むファイル形式が違う以上、画像ファイルの何のデータを読み込み、保存する必要があるか」だが、これを考える上で必要なことは入力ファイルパスと、出力ファイルパスが持っている情報、異なる画像ファイルにおいて共通して持っているデータが何かということである。
入力ファイルパスには読み込むファイルの名前、その拡張子の情報があり、出力ファイルパスには出力する際に使用するファイル名、変換先拡張子の情報を持っている。また、異なる画像ファイルにおいて共通して持っているデータは以下の通り。

- 画像の幅
- 画像の高さ
- 画像のピクセルデータ（RGBA）

画像の幅と高さは、画像ファイルのヘッダーとして記述するだけでなく、その画像の非圧縮時のピクセルデータのサイズを求める際も使用する。また、画像のピクセルデータは RGB の場合は 24bit のカラーデータになり、RGBA の場合は 32bit のカラーデータになり画像ファイルのピクセルデータとしては 32bit カラー以上に対応する必要性はないため RGBA（32bit カラー）で保存すればよい。
次に二つ目の「画像ファイル形式ごとに異なる処理を行わなければいけない処理は何か」は以下の 3 つ。

- 読み込むファイルの拡張子と、その変換処理を持つクラスのインスタンス種類が同じ画像ファイル形式かどうか判断する処理
- 読み込んだバイナリデータを解析し、画像データを取得する処理
- 保存している画像データから、出力するファイルのバイナリデータを作成する処理

読み込むファイルの拡張子と、その変換処理を持つクラスのインスタンスの種類が同じどうか判断する処理を画像ファイル形式ごとの異なる処理とすることで、入力ファイルの拡張子から処理を行うかどうかの分岐処理の際の文字列比較を別で用意することができ、これにより画像ファイル形式ごとの異なる処理を持つクラスを追加するだけで、追加の分岐処理を記述する必要はなくなる。
また、これ以外の残りの二つではバイナリデータから解析、変換しバイナリデータで出力とすることで、すべての画像ファイル形式で同じ入力元のデータと出力するデータの作成が行える。これによりバイナリデータを読み込む処理と、書き込む処理とデータの受け渡しを行えばよくなるため、結合度または凝縮度の観点から見ても良い設計になる。
以下が実装例である。

C++

```
class FileData
{
public :
    int width = 0;
    int height = 0;
    std::unique_ptr<unsigned char[]> pixels = nullptr;
};

class IConverter
{
public :
    IConverter() = default;
    virtual ~IConverter() = default;

    bool judgeExt(std::string_view importPath) = 0;

    virtual std::unique_ptr<FileData> analysis
    (
        std::unique_ptr<unsigned char[]>& importData
    ) = 0;

    virtual std::unique_ptr<unsigned char[]> convert
    (
        std::unique_ptr<FileData>& fileData,
        unsigned int& rtDataSize
```

```
    ) = 0;
};

class BMP : public IConverter
{
public:
    BMP() = default;
    ~BMP() override = default;

    bool judgeExt(std::string_view importPath) final;

    std::unique_ptr<FileData> analysis
    (
        std::unique_ptr<unsigned char[]>& importData
    ) final;

    std::unique_ptr<unsigned char[]> convert
    (
        std::unique_ptr<FileData>& fileData,
        unsigned int& rtDataSize
    ) final;
};

class TGA : public IConverter
{
public:
    BMP() = default;
    ~BMP() override = default;

    bool judgeExt(std::string_view importPath) final;

    std::unique_ptr<FileData> analysis
    (
        std::unique_ptr<unsigned char[]>& importData
    ) final;
```

```
    std::unique_ptr<unsigned char[]> convert
    (
        std::unique_ptr<FileData>& fileData,
        unsigned int& rtDataSize
    ) final;
};

class Converter
{
private :
    std::vector<std::unique_ptr<IConverter>> converters_;

public :
    Converter() = default;
    ~Converter() = default;

    void add(std::unique_ptr<IConverter> converter);

    std::unique_ptr<FileData> fileAnalysis
    (
        std::string_view importPath
    );

    void fileConvert
    (
        std::string_view exportPath,
        std::unique_ptr<FileData> &fileData
    );
};

std::unique_ptr<FileData> Converter::fileAnalysis
(
    string_view importPath
){
    for (int i = 0; i < converters_.size(); i++)
    {
```

```
        if (converters_[i]->judgeExt(importPath))
        {
            // ファイルのバイナリ読み込み
            std::unique_ptr<unsigned char[]> importFileBinary
            = FileLoad(importPath);

            // バイナリデータから解析
            std::unique_ptr<FileData>
            = converters_[i]->analysis(importFileBinary);
        }
    }
}

void fileConvert
(
    std::string_view exportPath, std::unique_ptr<FileData> &fileData
){
    for (int i = 0; i < converters_.size(); i++)
    {
        if (converters_[i]->judgeExt(exportPath))
        {
            unsigned int binarySize = 0;

            // 画像データから書き出すバイナリデータの作成
            std::unique_ptr<unsigned char[]> exportFileBinary
            = converters_[i]->convert(fileData, binarySize);

            // バイナリデータからファイルを書き出す
            FileWrite(exportPath, exportFileBinary.get(), binarySize);
        }
    }
}

// main.cpp
Converter converter;
converter.add(make_unique<BMP>());
```

```
converter.add(make_unique<TGA>());

unique_ptr<FileData> fileData = converter.fileAnalysis(importPath);
converter.fileConvert(exportPath, fileData);
```

このようにすることで、結合度または凝縮度の両方の観点から見ても良い設計になり、新しい画像ファイル形式に対応するために必要なことは以下の 2 つのみになり、良い設計と言える。

- **IConverter クラス**を継承する追加する画像ファイル形式の処理を行うクラスの作成
- **Converter クラス**に **add(std::unique_ptr<IConverter> converter)** で追加する

これがオブジェクト指向プログラミングの利点の一つであり、インタフェースを使用した良いクラスの設計である。

assert を使用したエラーの防止と、コードの意図の表明

assert とは開発時に除去できるバグを見つけるために使用するものであり、「関数の引数がある範囲内の値でなければならない」「ある状態でこの関数を呼び出してはならない」といったことを表明することで、その関数を呼び出すユーザーに対する要件として設定ものである。
ここで注意しなければいけないことが **assert** とは意図を表明するものであって、通常のエラーチェックに使うものではないことだ。コードを記述する際に記述者が想定している条件に合致しているかどうかなどのときに使用する。これにより、記述者が想定していた条件が分かることでコードの意図がコードを読んだ他人にも伝わり、理解がしやすくなる。また **assert** は以下のようにすることで、デバッグ時以外では **assert** 処理を無効化するようにする。

C++

```
#define NDEBUG
#include <cassert>
```

適切な命名

クラスや構造体、変数の命名はそのクラスなどが何をしているかを表し、「名は体を表す」とも言うように設計において大事な点の一つである。以下の例は設計そのものも良くないが良くない命名の一つでもある。

C++

```
class FileManager
{
private :
    std::unique_ptr<FileData> fileData_;

public :
    FileManager();
    ~FileManager();

    std::string_view getPathExt(std::string_view path) const;
    std::string getFileNameFromPath(std::string path) const;
    bool setFolderPath(std::string& folderPath);

    std::unique_ptr<u8[]> fileLoad(std::string_view path);
    bool fileAnalysis(std::unique_ptr<u8[]>& srcBuff);
    u32 fileWrite
    (
        std::string_view filePath, u8* fileData, const u32 dataSize
    );
};
```

FileManager クラスは名前の通りならばファイルのマネージャーであり、マネージャーという言葉の通りであればファイルデータを管理しなければいけないがこのクラスでは管理していない。また、管理という言葉自体の意味合いが広すぎるためクラスが何を行っているか明確にできていないためより詳しくそのクラスは何をしているかを考え命名するべきである。

ならどう命名するかというかだが、まずクラスが何をしているかを考えることが重要である。今回の場合は **fileAnalysis** 以外は全てユーティリティ関数を集めただけのクラスなため、そもそもこのクラスが行う単一のことが存在しないためクラス化せず、関数で処理を記述するべきであるが、クラスが何をしているか、単一の何をしているかまで考えることでより良い命名をすることできるだけでなく、後述する単一責任を果たすことにも繋がる。

クラスが単一の何をしているかを考えることが出来た後、ファイルパスの管理などのように意味合いが広い言葉ではなく、ファイルパスの操作のようにより意味合いの狭い言葉に修正していくことでより良い命名をすることができる。

単一責任原則を守ることで、不具合の防止

単一責任原則とはクラスが担う責任は、たったひとつに限定すべきとする設計原則であり、これを果たしたクラスは**単一の関心事について正常動作に責任を負ったクラス**と言える。これが果たせないと「適切な命名」でも記述したようにクラスが何をしているかが明確にならず適切な命名ができないだけでなく、不具合が起きた際にその不具合が関連する処理に関わっているクラスがどれかがはっきりしなくなり、不具合の修正が困難になる。
また単一責任原則を守れていると、クラスの使用用途が明確になるため、適切に使用できかつ、他者から見ても理解しやすくなる。以下が単一責任原則を守れていないクラスの例だ。

C++

```
class Converter
{
public:
    Converter() = default;
    ~Converter() = default;

    std::string getFileExt(std::string_view path);
    std::string getFileName(std::string_view path);

    std::unique_ptr<FileData> fileAnalysis
    (
        std::string_view importPath
    );

    void fileConvert
    (
        std::string_view exportPath,
        std::unique_ptr<FileData> &fileData
    );
};
```

Converter クラスは以下の二つの責任を持ってしまっている。こうなると **Converter** という命名も適切でない。

1. ファイルの変換
2. ファイルパスから文字列を作成

そこで以下のように修正することで**単一責任原則**を果たしかつ、適切な命名のできているクラスにすることができる。

C++

```
std::string GetFileExt(std::string_view path);
std::string GetFileName(std::string_view path);

class Converter
{
public:
    Converter() = default;
    ~Converter() = default;

    std::unique_ptr<FileData> fileAnalysis
    (
        std::string_view importPath
    );

    void fileConvert
    (
        std::string_view exportPath,
        std::unique_ptr<FileData> &fileData
    );
};
```

GetFileExt、 GetFileName 関数はユーティリティ関数であり、わざわざクラス化する必要のないためこのようにすることで単一責任原則を果たすことができる。この例は単純な場合の話であり、単一責任原則を果たすためにはどのように設計すれば良いか悩むこともある。その際はまず適切に命名することを考えることで、自然とそのクラスが行うことが明確になりその結果、そのクラスが果たすべき責任の対象が分かるため、単一責任原則を果たしたクラスにすることができる。

ソースファイルのローカル関数や変数の使用方法

単一責任原則を果たすためにユーティリティ関数をクラス外で宣言、定義する際は以下のようにする。これはソースファイルで行っているとする。

C++

```
namespace
{
    int INTERNAL_VARIABLE = 0;

    std::string GetFileExt(std::string_view path)
    {
        // ファイルパスの拡張子の取得処理
    }
}
```

このようにすることの利点は以下の 3 点。

1. 宣言されたファイル内でのみ有効なため、他のファイルに影響を与えない
2. 異なるファイルで同じ名前の識別子を定義しても、それらが競合することを防げる
3. 外部に公開しない変数や関数をファイル内に隠蔽することができる

以上のようにとても便利であり、適切に使用することでコードのカプセル化と安全性を向上させることができる。ただし、過度の使用はコードの可読性、保守性を下げる可能性があるため避けるべきである。

リファクタリング時の注意点

リファクタリング時に注意することは特に以下の二つ。

1. デザインパターンがもたらす効果を理解し使用すること
2. 共通化して良いか、良くないかを正しく判断すること

一つ目にもあるように、デザインパターンがもたらす効果を理解し使用しなければ、かえって良くない設計のプログラムになってしまう場合がある。勘違いしてはいけないこととして、デザインパターンにそうように記述すれば良い設計になるのではなく、デザインパターンがもたらす効果を理解し記述すれば良い設計になることである。
また共通化することで、処理の重複を防げる場合があるが、無理に共通化してしまうと返って依存度が増してしまい、変更容易性が低下してしまう。そのため、共通化する際は結合度や単一責任原則の観点から一度考えて行うべきである。

連長圧縮（RLE - Run Length Encoding）の圧縮と解凍

TGA 画像ファイルではピクセルデータの格納に、連長圧縮（RLE - Run Length Encoding）を使用する場合があり、画像ファイル形式の変換を行う際に RLE の解凍と、RLE への圧縮は行う必要がある。TGA の RLE ではピクセルデータを以下の 2 種類のパケットに分けて記録する。

1. **RLE パケット**
    - 同じピクセル値が連続する場合に使用
    - **ヘッダー1 バイト**と**1 つのピクセルデータ**で構成される
        - ヘッダー：上位 1 ビットが **1**（RLE パケットを示す）、下位 7 ビットが「連続するピクセル数 - 1」
        - ピクセルデータ：1 ピクセル分のデータ（ビット深度に応じてサイズが異なる）
    - 例：24 ビット RGB データ（**255, 0, 0**）が 4 回連続の場合：
    - **83 FF 00 00**
        - **83**：RLE パケッ
        - **FF 00 00**：ピクセルデータ。

2. **Raw パケット**
    - 異なるピクセルが連続する場合に使用。
    - **ヘッダー1 バイト**と**複数のピクセルデータ**で構成される。
        - ヘッダー：上位 1 ビットが **0**（Raw パケットを示す）、下位 7 ビットが「含まれるピクセル数 - 1」。
        - ピクセルデータ：指定された数のピクセル分のデータ。

これらのパケットを読み込みピクセルデータの先頭から順にカラーデータを取り出していくことで RLE の解凍を、順に格納していくことで圧縮を行える。
DirectXTexTGA.cpp でも解凍処理があるが、以下も解凍処理の一つの例であり、ピクセルデータを BGRA の順の色データで取得している。

C++

```
using u8  = unsigned char;
using u16 = unsigned short;
using u32 = unsigned int;
using s32 = signed int;
```

```
struct BGRA
{
    u8 b;
    u8 g;
    u8 r;
    u8 a;
};

std::unique_ptr<u8[]> TGA::uncompress
(
    std::unique_ptr<u8[]> &importData,
    u32 dataOffset, s32 width, s32 height, u16 pixelDepth
){
    u32 maxAlpha = 255;
    std::unique_ptr<u8[]> pixels = std::make_unique<u8[]>
    (
        width * height * 4
    );

    u8* src = importData.get() + dataOffset;
    u16 clrWidth = pixelDepth / 8;

    for (u32 y = 0; y < height; y++)
    {
        for (u32 x = 0; x < width;)
        {
            if (*src & 0x80) // Repeat
            {
                u32 count = (*src & 0x7F) + 1;
                src++;

                BGRA* pixel = reinterpret_cast<BGRA*>(src);
                for (u32 i = 0; i < count; i++)
                {
                    assert(y*width*4 + x + 3 < width*height*4);
```

```
                pixels[GetIndex(x, y, width, height) ]
                = pixel->b;

                pixels[GetIndex(x, y, width, height) + 1]
                = pixel->g;

                pixels[GetIndex(x, y, width, height) + 2]
                = pixel->r;

                pixels[GetIndex(x, y, width, height) + 3]
                = (clrWidth == 4) ? pixel->a : 0xff;

                x++;
            }

            src += clrWidth;
        }
        else // Literal
        {
            u32 count = (*src & 0x7F) + 1;
            src++;

            for (u32 i = 0; i < count; i++)
            {
                assert(y*width*4 + x + 3 < width*height*4);

                BGRA* pixel = reinterpret_cast<BGRA*>(src);
                pixels[GetIndex(x, y, width, height)]
                = pixel->b;

                pixels[GetIndex(x, y, width, height) + 1]
                = pixel->g;

                pixels[GetIndex(x, y, width, height) + 2]
                = pixel->r;
```

```
                    pixels[GetIndex(x, y, width, height) + 3]
                    = (clrWidth == 4) ? pixel->a : 0xff;

                    src += clrWidth;
                    x++;
                }
            }

            if (x >= width) break;
        }
    }

    return pixels;
}
```

また以下が RLE への圧縮処理の例。

C++

```
using u8  = unsigned char;
using u16 = unsigned short;
using u32 = unsigned int;
using s32 = signed int;

struct BGRA
{
    u8 b;
    u8 g;
    u8 r;
    u8 a;
};

class FileData
{
public :
    s32 width = 0;
    s32 height = 0;
    std::unique_ptr<u8[]> pixels = nullptr;
```

```
};

bool IsSameBGRA(BGRA* a, BGRA* b)
{
    return a->b == b->b && a->g == b->g
    && a->r == b->r && a->a == b->a;
}

std::vector<u8> TGA::compress(std::unique_ptr<FileData> &fileData)
{
    std::vector<u8> compressData;

    u32 runMaxLen = 128;
    for (u32 y = 0; y < fileData->height; y++)
    {
        for (u32 x = 0; x < fileData->width;)
        {
            u8* thisPixelStart

            = &fileData->pixels[y*fileData->width*4 + x*4];
            BGRA* thisPixel = reinterpret_cast<BGRA*>(thisPixelStart);

            // 判定するべき次のピクセルがない場合は Literal
            if (x + 1 == fileData->width)
            {
                u32 count = 1;
                compressData.push_back(count - 1);
                compressData.push_back(thisPixel->b);
                compressData.push_back(thisPixel->g);
                compressData.push_back(thisPixel->r);
                compressData.push_back(thisPixel->a);

                break;
            }

            u8* nextPixelStart = thisPixelStart + 4;
```

```
            BGRA* nextPixel = reinterpret_cast<BGRA*>(nextPixelStart);

            if (IsSameBGRA(thisPixel, nextPixel)) // Repeat
            {
                u32 count = 1;
                for (u32 i = 0; i < runMaxLen; i++)
                {
                    if (x + i >= fileData->width) break;

                    BGRA* judgePixel
                    = reinterpret_cast<BGRA*>(nextPixelStart + i*4);

                    if (IsSameBGRA(thisPixel, judgePixel)) count++;
                    else break;
                }

                compressData.push_back(0x80 | (count - 1));
                compressData.push_back(thisPixel->b);
                compressData.push_back(thisPixel->g);
                compressData.push_back(thisPixel->r);
                compressData.push_back(thisPixel->a);

                x += count;
            }
            else // Literal
            {
                u32 count = 0;
                for (u32 i = 0; i < runMaxLen; i++)
                {
                    if (x + i >= fileData->width) break;

                    BGRA* judgePixel
                    = reinterpret_cast<BGRA*>(nextPixelStart + i*4);

                    if (!IsSameBGRA(thisPixel, judgePixel))
                    {
```

```
                    count++;
                    thisPixel = judgePixel;
                }
                else break;
            }

            compressData.push_back(count - 1);
            for (u32 i = 0; i < count; i++)
            {
                BGRA* pixel = reinterpret_cast<BGRA*>
                (
                    thisPixelStart + i*4
                );

                compressData.push_back(pixel->b);
                compressData.push_back(pixel->g);
                compressData.push_back(pixel->r);
                compressData.push_back(pixel->a);
            }

            x += count;
        }

        if (x >= fileData->width) break;
    }
}

return compressData;
}
```

課題 2　Direct3D11、ImGui を使用した描画プログラム

この課題では Direct3D11 を使用した各種描画処理の作成と、ImGui を使用したその各種描画処理のデバッグ機能の作成を行った。課題は以下の順で作成していくように指示され、一つずつ行っていった。

1. 三角形の描画
2. 四角形の描画
3. テクスチャ付きの四角形の描画
4. **セピア色フィルター**の適用
5. **グレースケール**の適用
6. obj ファイル（Stanford Bunny）の描画
7. **Phong shading** の適用
8. **Bloom** の適用
9. obj ファイルの読み込み処理の速度を向上させる

この課題の狙いは Direct3D11 の使い方を理解し、Direct3D11 そのものへの理解も深めることが狙いだったと思われる。

ImGui の導入

ImGui の導入は以下の手順で行った。

1. master ブランチを「Download ZIP」でローカル環境に移動
2. ImGui の example_win32_directx11 の example_win32_directx11.vcxproj を Visual Studio で開く
3. example_win32_directx11 をスタートアッププロジェクトに設定

以上を行い、**main.cpp** へ変更を加え、**main.cpp** と同じ階層のディレクトリに新規ファイルを作成することで Direct3D11 を使用した描画プログラムの作成を行った。

Direct3D11 の初期化処理の作成において、ID3D11Device の扱い方

ImGui の **example** の **main.cpp** では以下のように D3D11 の各種変数を宣言している。

C++

```
static ID3D11Device* g_pd3dDevice = nullptr;
static ID3D11DeviceContext* g_pd3dDeviceContext = nullptr;
static IDXGISwapChain* g_pSwapChain = nullptr;
static bool g_SwapChainOccluded = false;
static UINT g_ResizeWidth = 0, g_ResizeHeight = 0;
static ID3D11RenderTargetView* g_mainRenderTargetView = nullptr;
```

今回は ImGui の **main.cpp** に既にある初期化は利用しながら処理の作成を行った。その中で、**ID3D11Device** のみは初期化処理を行う関数に引数として渡すのではなくグローバル変数としてアクセスするようにした。グローバル変数を複数のファイルから参照するには以下のようにすることで可能になる。

C++

```
// helpers.h
Microsoft::WRL::ComPtr<ID3D11Device>& D3DDevice();

// helpers.cpp
ComPtr<ID3D11Device> &D3DDevice()
{
    static ComPtr<ID3D11Device> device;
    return device;
}
```

このようにすると、**helpers.h** をインクルードすることで、どんなファイルからでも同じグローバルインスタンスにアクセスすることができるようになる。

Direct3D11 の初期化の流れ

以下の流れで Direct3D11 は初期化することができる。

1. **Device** の作成
2. **SwapChain** の作成
3. **SwapChain** のバッファの **RenderTargetView** の作成
4. **DepthStencil** 用の 2D テクスチャの作成
5. **DepthStencil** 用の 2D テクスチャのビューである、**DepthStencilView** の作成
6. **World**、 **View**、 **Projection** 行列の定数バッファの作成
7. **World**、 **View**、 **Projection** 行列定数バッファを **DeviceContext** を使用し登録
8. **BlendState** の作成
9. **SamplerState** の作成

Direct3D11 での三角形、四角形の描画

三角形の描画には Direct3D11 の初期化処理以外に以下の処理が必要になる

1. 三角形の表示位置を決める**アフィン行列**の定数バッファの作成
2. **頂点シェーダー**及び**ピクセルシェーダー**のコンパイル済みデータ（CSO ファイル）のバイナリの読み込み
3. **頂点シェーダー**及び**ピクセルシェーダー**の作成
4. 頂点データの **InputLayout** の作成
5. **InputLayout** で指定したのと同じ構造体を使用し頂点データの作成
6. 頂点データから**頂点バッファ**と**インデックスバッファ**の作成
7. **ViewPort** の設定
8. **InputLayout** をセット
9. **PrimitiveTopology** をセット
10. **アフィン行列**、 **View** 、**Projection** らの定数バッファをセット
11. **DepthStencilView** をクリア
12. **RenderTarget** をセット
13. **RenderTargetView** をクリア
14. **頂点シェーダー**及び**ピクセルシェーダー**をセット
15. **頂点バッファ**及び**インデックスバッファ**をセット
16. **アフィン行列**を **Map** により更新
17. **drawIndex** を実行し、描画処理を実行

また四角形の描画は、三角形の描画からただ頂点数が増えるだけであるため、6「頂点データから**頂点バッファ**と**インデックスバッファ**の作成」と 、17「**drawIndex** を実行し、描画処理を実行」のみ変更すれば良い。それぞれ以下を変更する。

- 6：作製する頂点バッファの元データに頂点を追加し、インデックスバッファもそれに対応するように追加する
- 17 : **drawIndex** の描画するインデックスを増やす

テクスチャの作成

Direct3D11 においてテクスチャは以下の手順で作成する。

1. **サブリソースデータ**にピクセルデータと、テクスチャの **SysMemPitch** （ピクセルデータの 1 行あたりのデータサイズ）を設定する
2. **2D テクスチャ**を**サブリソースデータ**から作成
3. この **2D テクスチャ**の **ShaderResourceView** を作成

この手順により作成されたテクスチャは画像ファイルのピクセルデータから作成されているため、同じ画像ファイルのテクスチャを使用したい場合、作成した 2D テクスチャを使いまわすことで、重複して作成することなく使用するメモリ量を最低限にできる。
以下の例では同じ画像ファイルのデータから複数の 2D テクスチャを作成することが無いように制御する **Container クラス**である。

C++

```
struct TextureAttr
{
    std::string path = "";
    Microsoft::WRL::ComPtr<ID3D11Texture2D> texture = nullptr;
    Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> view = nullptr;
};

class TextureContainer
{
private:
    std::vector<std::unique_ptr<TextureAttr> textures_;
```

```
public:
    TextureContainer() = default;
    ~TextureContainer() = default;

    // コピーコンストラクタとコピー代入演算子を削除
    TextureContainer(const TextureContainer&) = delete;
    TextureContainer& operator=(const TextureContainer&) = delete;

    unsigned int addTexture(std::string path);
    std::unique_ptr<TextureAttr>& getTexture(unsigned int id);

    unsigned int getContainerSize();
}
```

これの **addTexture** で同じパスの画像ファイルの場合追加しないとすることで、重複しないようにすることができる。
またここで注意しなければいけない点として、**unique_ptr** データを **vector** で格納している点である。**unique_ptr** はコピーが禁止されており、**vector<std::unique_ptr<TextureAttr>>** **をメンバに持つ TextureContainer** もコピー操作ができないようにする必要があるため、コピーコンストラクタとコピー代入演算子を削除する必要がある。

テクスチャの適用

作成した **2D テクスチャ**を適用するには、以下のようにリソースを GPU 側に登録する必要がある。

C++

```
d3dDeviceContext->PSSetShaderResources
(
    0, 1, textureView.GetAddressOf()
);

// Pixel Shader
Texture2D txDiffuse : register(t0);
```

ここでは、**PSSetShaderResources** で **ShaderResource** の 0 番目に 1 つ、**textureView** の作成元の **2D テクスチャ** を登録している。これにより以下のように**ピクセルシェーダー**内で

使用することができる

```C++
float4 color = txDiffuse.Sample(samplerState, input.Tex);
```

ここで **samplerState** を使用する必要があり、**samplerState** も登録する必要があるが登録は **samplerState** の作成時に行われるため **samplerState** を一つのみ使用する場合は手動で登録する必要はない。ただし、使用するには以下のようにピクセルシェーダー内で宣言する必要がある。

```C++
SamplerState samplerState : register(s0);
```

シェーダー内の分岐処理を定数バッファで減らす

GPU での処理において、if 文やループ数の違う for 文などが、オブジェクトやポリゴンごとの処理で存在すると、ワープダイバージェンスと呼ばれることが起き、処理速度が低下する。詳しくは GPGPU でのプログラミングについて知る必要があるが、大まかに言うと GPU での処理ではたくさんのスレッドが同じ処理を同時にし、そのスレッドはワープ（CUDA の用語）と呼ばれるもので束ねられており、そのワープ内の束ねられたスレッドらが全く同じ処理の流れをたどらない場合起きるのがワープダイバージェンスである。
ここで注意しなければいけないことは同じ処理の流れをたどれば良いだけで、同じ値になる必要はない。また、同じ処理の流れをたどる場合でも、同じ処理をたどらない場合がある可能性が存在すると同時に実行されるスレッド間での同期が必須になることで、処理速度が低下する。以下の例のような場合で起こる

C++

```
if (isUsingTexture)
{
    // Texture を使用するポリゴンの場合の処理
}
else
{
    // Texture を使用しない場合の処理
}
```

その時描画するオブジェクトらすべてがテクスチャを持っている場合でも、この処理がシェーダー内にある場合、前述したとおり処理速度が低下する。そこでテクスチャを使用する場合としない場合で処理の流れを同じにするためには、以下のような定数バッファを作成し、その値をテクスチャカラーと入力カラー（頂点カラー）に掛ける。

C++

```
cbuffer psColorAttr : register(b3)
{
    float colorScale
    float3 padding[3];
};

float4 output = txDiffuse.Sample
(
    samplerState, input.Tex
) * colorScale + input.Col * (1 - colorScale);
```

colorScale の値が **1** の際は、**input.Tex** （テクスチャカラー）に **1** が、**input.Col** （頂点カラー）に **0** が掛けられることで、頂点カラーの値はテクスチャカラーに足されることなく、テクスチャカラーのみが反映される。逆に **colorScale** の値が **0** の際はテクスチャカラーに **0** が、頂点カラーに **1** が掛けられることで、頂点カラーのみが反映される。これにより分岐処理を使用せず、テクスチャがあるオブジェクトとないオブジェクトで同じ処理の流れをたどることができる。
今回は定数バッファで行ったが、頂点バッファに **colorScale** の情報を持たせバッファサイズは大きくなってしまうが同じことができる。また頂点カラーが必要ないなら、ダミーテクスチャ（無色テクスチャなど）をテクスチャとして使用することでも同じ処理の流れをたどることができる。
また、**padding** は定数バッファを作成するために必要であり、説明は次の「定数バッファと ImGui を使用した三角形、四角形の移動」で説明する。

定数バッファと ImGui を使用した三角形、四角形の移動

Direct3D11 において、定数バッファを作成しシェーダー内で使用するには以下の手順を踏む必要がある。

1. **D3D11_BUFFER_DESC** に値を設定
2. **D3D11_SUBRESOURCE_DATA** で **pSysMem** を使用することで値の初期化をする
3. **CreateBuffer** によりバッファを作成
4. **PSSetConstantBuffers** で対応するスロットに登録
5. **cbuffer** 及び **register(b 登録したスロット番号)** により、シェーダー側で受け取る宣言をする
6. 各シェーダーで使用

シェーダー内で何らかの値を使用したい場合は必ずこの手順を踏み、定数バッファを作成する必要がある。**World**、 **View**、 **Projection**、 **アフィン 行列**らもすべてこの定数バッファである。またここで注意する必要があることが **padding** である。**D3D11_BUFFER_DESC** の **ByteWidth** は常に 16 の倍数である必要があるため以下のような構造体の場合、**float padding[3]** を構造体の末尾に追加する必要がある。

C++

```
struct ColorAttr
{
    int colorScale;
};
```

そしてこれらの値を **Main ループ** 内で更新するには以下の手順が必要になる。

1. 空の **D3D11_MAPPED_SUBRESOURCE** を作成
2. **D3DDeviceContext** の **Map** により GPU 側にあるデータに対して書き込めるようにする
3. **D3D11_MAPPED_SUBRESOURCE** の **pData** に対して書き込む
4. **D3DDeviceContext** の **Unmap** により書き込みを終了する

これらにより、情報をシェーダーに渡しオブジェクトの移動などの編集が可能になる。ここで使用するのが ImGui である。ImGui を使用するには example_win32_directx11/main.cpp にもあるが、以下のようにまず初期化する必要がある。

C++

```
// Setup Dear ImGui context
IMGUI_CHECKVERSION();
ImGui::CreateContext();
ImGuiIO& io = ImGui::GetIO(); (void)io;

// Enable Keyboard Controls
io.ConfigFlags |= ImGuiConfigFlags_NavEnableKeyboard;

// Enable Gamepad Controls
io.ConfigFlags |= ImGuiConfigFlags_NavEnableGamepad;

// Setup Dear ImGui style
ImGui::StyleColorsDark();
//ImGui::StyleColorsLight();

// Setup Platform/Renderer backends
ImGui_ImplWin32_Init(hwnd);
ImGui_ImplDX11_Init(g_pd3dDevice, g_pd3dDeviceContext);
```

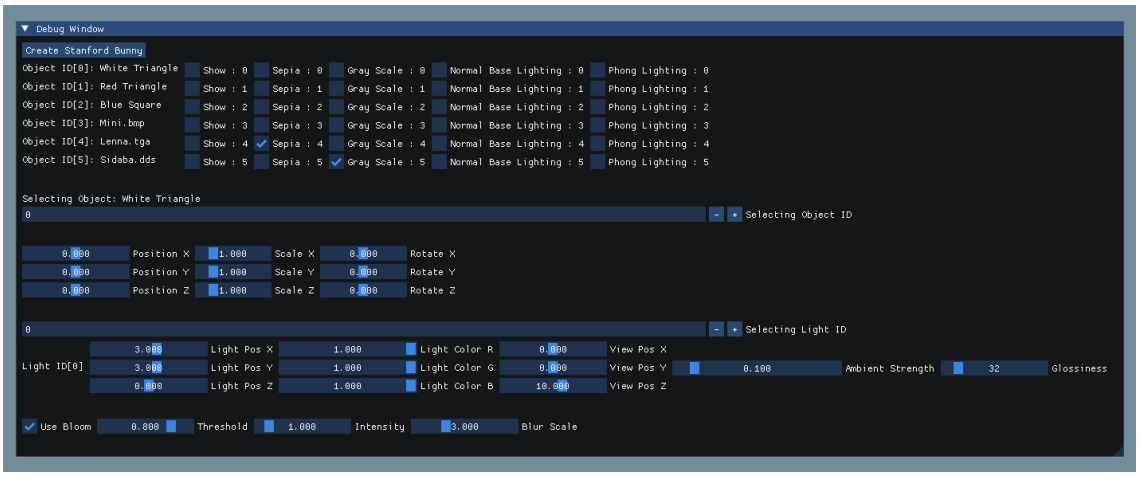
そして ImGui のウィンドウを表示するには **main ループ** 内に以下のような処理を記述する必要がある。

C++

```
// Start the Dear ImGui frame
ImGui_ImplDX11_NewFrame();
```

```
ImGui_ImplWin32_NewFrame();
ImGui::NewFrame();

// ---- Window を表示する処理 -----

// Rendering
ImGui::Render();

ImGui_ImplDX11_RenderDrawData(ImGui::GetDrawData());

// Present with vsync
gSwapChain->Present(1, 0);
```

ImGui のウィンドウのデザインは「**// ---- Window を表示する処理 -----**」 の部分で決定し、以下のようなウィンドウを画面に表示することができる。



このウィンドウでは以下の要素を使用している。

1. **Button**
2. **Table**
3. **Text**
4. **Checkbox**
5. **Dummy**
6. **InputInt**
7. **SliderFloat**、**SliderInt**

また、**SetNextWindowPos** によりウィンドウの表示位置を指定し、**Begin** と **End** でウィンドウの要素を編集している。

C++

```
// ImGuiCond_FirstUseEver で最初に一度のみ実行
// これにより、ウィンドウが動かせなくなることがないように
ImGui::SetNextWindowPos(ImVec2(20, 20), ImGuiCond_FirstUseEver);

if (ImGui::Begin("Debug Window"))
{
    // 要素を編集
}

ImGui::End();
```

以下ではそれぞれの要素の使い方について簡単に説明する。

■ Button

C++

```
if (ImGui::Button("Button Text"))
{
    // ボタンが押された際、実行される処理を記述
}
```

■ Table

C++

```
int tableColumns = 3

// テーブルの列の幅をその要素に合わせて固定
ImGui::BeginTable
(
    "Table Name", tableColumns,
    ImGuiTableFlags_SizingFixedFit
);

ImGui::TableNextRow(); // 次の行へ

ImGui::TableNextColumn(); // 次の列へ
// 要素の作成
```

```
ImGui::TableNextColumn(); // 次の列へ
// 要素の作成

ImGui::TableNextColumn(); // 次の列へ
// 要素の作成

ImGui::EndTable();
```

■ Text

C++

```
ImGui::Text("Text");
```

■ Checkbox

C++

```
bool flag = false;
if (ImGui::Checkbox(("Check Bot Text", &flag))
{
    // flag が変更された際に行われる処理
}
```

■ Dummy

C++

```
// 要素同士に隙間を開けるために使用
ImGui::Dummy(ImVec2(0.0f, 20.0f));
```

■ InputInt

C++

```
int num = 0;
if (ImGui::InputInt("Text", &num))
{
    // 数値が入力された場合、実行される処理
}
```

■ SliderFloat、SliderInt

C++

```
float fNum = 0.0f;
```

```
float fMin = -2.0f;
float fMax = 2.0f;
bool isFloatChanged = ImGui::SliderFloat("Text", &fNum, fMin, fMax);
if (isFloatChanged) // スライダーの値が変更された場合実行される処理

int iNum = 0;
int iMin = -2;
int iMax = 2;
bool isIntChanged = ImGui::SliderInt("Text", &iNum, iMin, iMax);
if (isIntChanged ) // スライダーの値が変更された場合実行される処理
```

頂点、インデックスバッファを作成した元のデータについて

頂点、インデックスバッファらの作成は以下のように、**D3D11_BUFFER_DESC** の **ByteWidth** にデータサイズを、**D3D11_SUBRESOURCE_DATA** の **pSysMem** に作成元データの先頭のポインタを指定することで作成する。

C++

```
D3D11_BUFFER_DESC bd = {};
bd.Usage = D3D11_USAGE_DEFAULT;
bd.ByteWidth = sizeof(Vertex) * vertexSize;
bd.BindFlags = D3D11_BIND_VERTEX_BUFFER;
bd.CPUAccessFlags = 0;
bd.MiscFlags = 0;
bd.StructureByteStride = sizeof(Vertex);

D3D11_SUBRESOURCE_DATA InitData = {};
InitData.pSysMem = vertices;

D3DDevice()->CreateBuffer
(
    &bd,
    &InitData,
    &vertexBuff
);
```

この処理をした後は、元データの **vertices** は今後使用することはないため、解放する必要があることに注意が必要である。

継承は使用せず、インタフェースで実装することでクラスの肥大化を防ぐ

C++において継承は非常に重要な機能であることは間違いない。しかし、継承を正しく使用できない場合親クラスに機能が集中し、単一責任を果たせなくなるだけでなく、凝縮度の観点から見ても良くない設計となる。ならどうすれば良いかというとインタフェースを使用すればよい。デザインパターンでもインタフェースを積極的に使用している。インタフェースは以下のように使用する。

C++

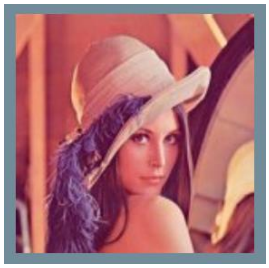
```
class IObjectDrawer
{
public:
    virtual ~IObjectDrawer() = default;

    virtual void setVertexBuff
    (
        ID3D11DeviceContext* d3dDeviceContext
    ) = 0;
    virtual void setIndexBuff
    (
        ID3D11DeviceContext* d3dDeviceContext
    ) = 0;

    virtual HRESULT updateAffineMat
    (
        ID3D11DeviceContext* d3dDeviceContext,
        Microsoft::WRL::ComPtr<ID3D11Buffer> affineMatBuff
    ) = 0;
    virtual HRESULT updateCBScene
    (
        ID3D11DeviceContext* d3dDeviceContext,
        Microsoft::WRL::ComPtr<ID3D11Buffer> cbScene
    ) = 0;

    virtual HRESULT setTexture
    (
        ID3D11DeviceContext* d3dDeviceContext,
```

```
        TextureContainer& textureContainer, unsigned int slot
    ) = 0;

    virtual void drawIndex(ID3D11DeviceContext* d3dDeviceContext) = 0;
};

class VisualObject : public IObjectDrawer
{
public:
    ~VisualObject() final = default;
    void setVertexBuff(ID3D11DeviceContext* d3dDeviceContext) final;
    void setIndexBuff(ID3D11DeviceContext* d3dDeviceContext) final;

    virtual HRESULT updateAffineMat
    (
        ID3D11DeviceContext* d3dDeviceContext,
        Microsoft::WRL::ComPtr<ID3D11Buffer> affineMatBuff
    ) final;
    virtual HRESULT updateCBScene
    (
        ID3D11DeviceContext* d3dDeviceContext,
        Microsoft::WRL::ComPtr<ID3D11Buffer> cbScene
    ) final;

    virtual HRESULT setTexture
    (
        ID3D11DeviceContext* d3dDeviceContext,
        TextureContainer& textureContainer, unsigned int slot
    ) final;

    void drawIndex(ID3D11DeviceContext* d3dDeviceContext) final;
}
```

このようにすることで、適切に継承を使用することができる。機能を追加する際は新たにインタフェースを作成し、それを VisualObject で継承することで処理を追加することができる。これにより IObjectDrawer には処理が追加されることなくクラスが膨大になることはない。

ピクセルシェーダーでのセピア色カラーフィルターの作成

セピア色のカラーフィルターとは、以下の左の画像を、右の画像のように色合いを変化させるフィルターのことである。




元のカラーデータをこのように変化させるには以下のような処理が必要になる。

C++

```
float4 srcColor = output;

output.r
= (srcColor.r * 0.393) + (srcColor.g * 0.769) + (srcColor.b * 0.189);

output.g
= (srcColor.r * 0.349) + (srcColor.g * 0.686) + (srcColor.b * 0.168);

output.b
= (srcColor.r * 0.272) + (srcColor.g * 0.534) + (srcColor.b * 0.131);
```

この処理により色合いをセピア調へ変えることができ、ピクセルシェーダー内で行うことでフィルターを追加することができる。
このとき、セピア色フィルターを使用するかしないかを指定するために定数バッファを使用し、分岐処理をピクセルシェーダー内で行う必要がある。もしくは、セピア色フィルターを適用する別のピクセルシェーダーを作成し、セピア色フィルターを適用するオブジェクトはそのピクセルシェーダーで処理を行うようにする。

ピクセルシェーダーでのグレースケールの適用

グレースケールは以下の式で色合いを元のカラーから変更することができる。この処理をセピア色フィルターと同様にピクセルシェーダー内で記述することで、グレースケールをオブジェクトに適用することができる。

C++

```
float gray
= (output.r * 0.299) + (output.g * 0.587) + (output.b * 0.114);

output.r = gray;
output.g = gray;
output.b = gray;
```

obj ファイルの高速な読み込み

obj ファイルは以下の要素から構成される。

1. 頂点座標
2. 法線ベクトル
3. UV 座標
4. ポリゴンを成す頂点の「頂点座標番号/法線ベクトル番号/UV 座標番号」の組み合わせ

今回読み込む対象のファイルは Stanford Bunny と呼ばれるポリゴン数は 144046、頂点数は 72378 のウサギである。13419KB ものファイルサイズなため、読み込みを適切にし、高速で読み込む必要がある。
まずファイルの読み込みだが、以下のように **getline** での読み込みは一行ずつ読んでいくため、非常に低速。

C++

```
std::ifstream file("example.obj");

while (std::getline(file, line))
{
    // データを格納する処理
}
```

そこで以下のようにバイナリで読み込むことにより、一度で読み込みが行えるため高速で行える。

C++

```
std::unique_ptr<unsigned char[]> LoadFile
(
    std::string path, fpos_t& size
){
    FILE* fp = nullptr;
    fopen_s(&fp, path.data(), "rb");

    //ファイルの末尾へ移動して、サイズを計算
    fseek(fp, 0L, SEEK_END);
    fgetpos(fp, &size);

    //ファイルの最初に移動
    fseek(fp, 0L, SEEK_SET);

    std::unique_ptr<unsigned char[]> buff
    = std::make_unique<unsigned char[]>(size);

    //サイズ分のデータを読み込む
    fread(buff.get(), 1, size, fp);
    fclose(fp);

    return buff;
}
```

バイナリで読み込んだデータから頂点、インデックスバッファを作成する必要があるため、バイナリデータを解析する必要がある。この際バイナリを先頭から順に解析するには、バイナリデータの先頭をポインタに格納し、それをインクリメントしながら解析していくことで高速に解析を行える。

C++

```
unsigned char* ptr = reinterpret_cast<unsigned char*>(binary.get());
unsigned char* end = ptr + binarySize;
while (ptr < end)
{
    ptr++;
}
```

バイナリデータを **unsigned char*** で操作している中で、obj ファイルの数値データは **unsigned char** でアクセスしているため、以下のように一文字ずつ配列に格納されていることになる。

ptr	ptr + 1	ptr + 2	ptr + 3	ptr + 4	ptr + 5	ptr + 6	ptr + 7
1	.	2	4	5	3	8	2

これから **float** の値を取得するには、以下のように文字列に変換しそれを **float** に変換することで行える。これを駆使することでバイナリデータから頂点座標、法線ベクトル、UV 座標を **float** 型で取得する。

C++

```
std::string str(ptr, ptr + size);
float num = std::stof(str);
```

このとき、obj ファイルの座標系は右手座標系、DirectX は左手座標系なため以下のように変換する必要がある。また UV 座標も DirectX では原点が左上であり、obj ファイルは左下にあるためこちらも変換する必要がある。

C++

```
vertex.z *= -1;
normal.z *= -1;

uv.y = 1 - uv.y;
```

以上から値を取得し、次はポリゴンを成す頂点の各「頂点座標番号/法線ベクトル番号/UV 座標番号」の組み合わせから、頂点、インデックスバッファを作成する必要があるがこの時注意しなければいけないこととして、同じ頂点座標、UV 座標を持つ頂点は頂点バッファに必ず一つのみ存在するようにしなければ、バッファサイズが肥大化することである。一つの頂点が異なる法線ベクトルを持つ場合は無いため、頂点の固有の情報となるのは頂点座標と UV 座標の組み合わせになる。

これらの組み合わせは「頂点座標番号/法線ベクトル番号/UV 座標番号」の**頂点座標番号**と**UV 座標番号**の組み合わせと等しくなるため、これらの番号をペアのキーとし、**map** を作成することで、同じペアの頂点データは一つしか存在しないことが保証されるようにすることができる。
また、この map の値として保持する必要があるのは、キーのペアのそれぞれの番号に対応する頂点座標、UV 座標を持つ頂点バッファ上のインデックス番号である。これを値とすることで、「頂点座標番号/法線ベクトル番号/UV 座標番号」からその頂点の頂点バッファ上

のインデックス番号が取得できることで、インデックスバッファを作成することができる。

この時使用するのは **std::map** ではなく **std::unordered_map** である。理由は **std::map** はアクセスするために木構造を辿って要素を追加し、木のバランスを保つためにそれぞれ O(log n)の時間がかかるが、**std::unorderd_map** はアクセス、要素を追加するために平均的に O(1)の時間で済み、圧倒的に高速なため **std::unordered_map** を使用する。また、**std::unordered_map** でキーとして **std::pair** を使用するためにはカスタムハッシュ関数が必要なため、以下のようにハッシュ関数も作成する必要がある。

C++

```
struct PairHash
{
    template <typename T1, typename T2>
    std::size_t operator()(const std::pair<T1, T2>& t) const
    {
        return
        std::hash<T1>{}(t.first) ^ (std::hash<T2>{}(t.second) << 1);
    }
};

std::unordered_map
<
    std::pair<unsigned int, unsigned int>,
    unsigned int, PairHash
> vertexIdMap;
```

そして以下のように **map** に要素を追加し、頂点、インデックスバッファらの元データを作成することができる。これにより頂点バッファのサイズを最小にすることができる。

C++

```
std::vector<Vertex> dstVertices; // 頂点バッファの元データ
std::vector<unsigned int> dstIndices; // インデックスバッファの元データ
if (vertexIdMap.try_emplace
(
    std::make_pair(v1_VID, v1_UVID),
    dstVertices.size())
){
    dstVertices.emplace_back
```

```
    (
        // v1_VID、v1_UVID に対応する各頂点座標、UV 座標を持つ頂点データ
    )
}

if (vertexIdMap.try_emplace
(
    std::make_pair(v2_VID, v2_UVID),
    dstVertices.size()
){
    dstVertices.emplace_back
    (
        // v2_VID、v2_UVID に対応する各頂点座標、UV 座標を持つ頂点データ
    )
}

if (vertexIdMap.try_emplace
(
    std::make_pair(v3_VID, v3_UVID),
    dstVertices.size())
){
    dstVertices.emplace_back
    (
        // v3_VID、v3_UVID に対応する各頂点座標、UV 座標を持つ頂点データ
    )
}

dstIndices.emplace_back(vertexIdMap[std::make_pair(v1_VID, v1_UVID)]);
dstIndices.emplace_back(vertexIdMap[std::make_pair(v3_VID, v3_UVID)]);
dstIndices.emplace_back(vertexIdMap[std::make_pair(v2_VID, v2_UVID)]);
```

push_back と emplace_back の違い

push_back と **emplace_back** には以下の特徴がある。

- **push_back** は要素をコピーもしくは、ムーブして追加
- **emplace_back** はコンテナの末尾に要素を直接構築

emplace_back は直接構築するため、**push_back** と違いコピーやムーブを避けるため特に複雑なオブジェクトが要素の場合はパフォーマンスの向上が期待できる。そのため、末尾に要素を追加するのなら **emplace_back** を使用することが推奨される。

インデックスバッファのデータサイズでの注意点

インデックスバッファの型を以下のようにしている場合、注意が必要である。

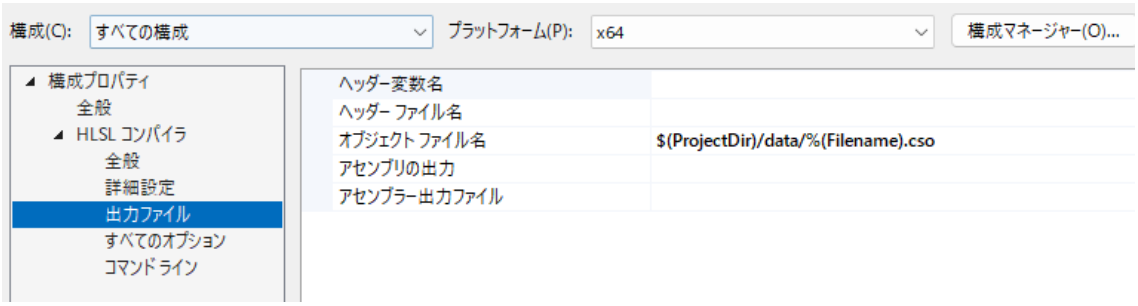
C++

```
// インデックスバッファの元データ
std::vector<unsigned short> srcIndices;

D3D11_BUFFER_DESC bd = {};
bd.ByteWidth = sizeof(unsigned short) * srcIndices.size();

D3D11_SUBRESOURCE_DATA InitData = {};
InitData.pSysMem = indices;
```

unsigned short の最大値は **65,535 (2^16 - 1)**なため、もし頂点バッファの要素数がこの最大値より多い場合、当然インデックスバッファにはこの頂点バッファのインデックス番号が入るため、65535 より大きな値が入力される可能性がある。そうなってしまうと正常に値を受け取ることができず、不具合が生じる。そのため **unsigned short** ではなく、**unsigned int** などより大きなサイズにするようにしなければいけない。

コンパイル済みシェーダーの使用

シェーダーはコンパイルすると **cso** ファイルになり、そのファイルをバイナリ読み込みすることでシェーダーとして使用することができる。Visual Studio でシェーダーを実行前にコンパイルするには以下のように、コンパイルしたシェーダーファイルの出力先の設定を変更する。



cso ファイルは **data** フォルダを作成し、その中に置いておくのが基本だと思われる。

ランバート反射による陰影付け

頂点バッファに頂点の法線ベクトルを含め実行し、ピクセルシェーダーに以下の処理を追加することで、ランバート反射モデルを使用した陰影付けを行うことができる。この場合ライト用の定数バッファを作成、設定する必要がないため容易にオブジェクトに陰影を付けることが可能である。

HLSL

```hlsl
// 法線を正規化
float3 normal = normalize(input.Normal);

// 光の方向（カメラ方向に固定）
float3 lightDir = float3(0.0, 0.0, -1.0);

// ドット積を計算
float intensity = max(dot(normal, lightDir), 0.0);

// 色を陰影に基づいて変化
float3 baseColor = output.rgb;
float3 shadedColor = baseColor * intensity;

output.rgb = shadedColor;
```

法線ベクトルと光ベクトルの内積が 1 に近いほど明るくなり、0 より小さい場合は真っ暗になるようになっている。これにより陰影付けを行える。

Direct3D11 でのライトの使用

ライトを使用すると言ってもそれ用の何らかの関数が必要というわけではなく、ランバート反射で行ったようにシェーダー内でライトのパラメータを使用するだけでできるため、ライトのパラメータの定数バッファを作成、設定すれば後はシェーダーの処理に変更を加えるだけで良い。

Phong Shading の適用

Phong Shading に使用するライトのパラメータ定数バッファは以下のようになる。

HLSL

```
cbuffer cbLight : register(b4)
{
    float3 LightPos;
    float padding0;

    float3 LightColor;
    float padding1;

    float3 ViewPos;
    float padding2;

    float AmbientStrength;
    int Glossiness;
    float padding3[2];
};
```

また、**Phong Shading** ではピクセルシェーダー内で入力されている頂点のワールド座標が必要なため、頂点シェーダー内で World 行列から頂点のワールド座標を計算し、ピクセルシェーダーに渡す必要がある。

HLSL

```
// 頂点シェーダーの出力するデータであり、ピクセルシェーダーに渡される
struct VS_OUTPUT
```

```
{
    float4 Pos : SV_POSITION;
    float2 Tex : TEXCOORD0;
    float3 WorldPos : POSITION;
    float3 Normal : NORMAL0;
    float4 Col : COLOR0;
};

// 頂点シェーダーの戻り値を変更
VS_OUTPUT main(VS_INPUT input)
{
    // 頂点のワールド座標を計算
    output.WorldPos = mul(input.Pos, World);

    // 頂点シェーダーの処理

    return output;
}

// ピクセルシェーダーで頂点シェーダーの出力を受け取る
float4 main(VS_OUTPUT input) : SV_TARGET
{
    // ピクセルシェーダーの処理
}
```

以上で Phong Shading に必要なデータは全てシェーダーに渡せたため、以下のようにピクセルシェーダー内で処理を行うことで Phong Shading を適用することができる。

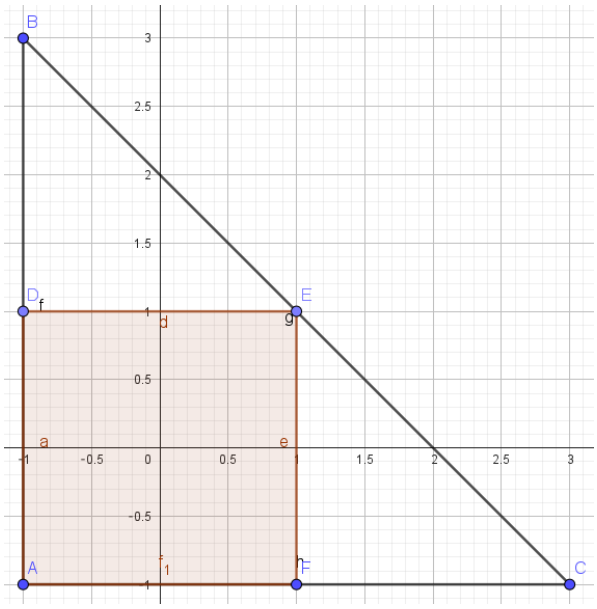
HLSL

```
// 法線を正規化
float3 normal = normalize(input.Normal);

// ライト方向ベクトルを正規化
float3 lightDirVec = normalize(LightPos - input.WorldPos);
float3 viewDirVec = normalize(ViewPos - input.WorldPos);

// 環境光の設定
```

```
float3 ambient = AmbientStrength * LightColor;

// 拡散反射光の計算
float diff = max(dot(normal, lightDirVec), 0.0);
float3 diffuse = diff * LightColor;

// 鏡面反射光の計算
float3 reflectDir = reflect(lightDirVec, normal);
float spec = pow(max(dot(viewDirVec, reflectDir), 0.0f), Glossiness);
float3 specular = spec * LightColor;

// 最終的な色の計算
float3 result = (ambient + diffuse + specular) * output.rgb;
output.rgb = result;

return output;
```

Full Screen Triangle を使用した、テクスチャへの描画

Full Screen Triangle とは Direct3D のレンダリング処理により、テクスチャの全てのピクセルに対して処理を行うために使用するものであり、以下の画像の黒線の三角形のことである。橙色の四角形(1)が**NDC 座標空間**をカメラ方向から Z 軸と平行に見た際のものであり、描画範囲にあたる。



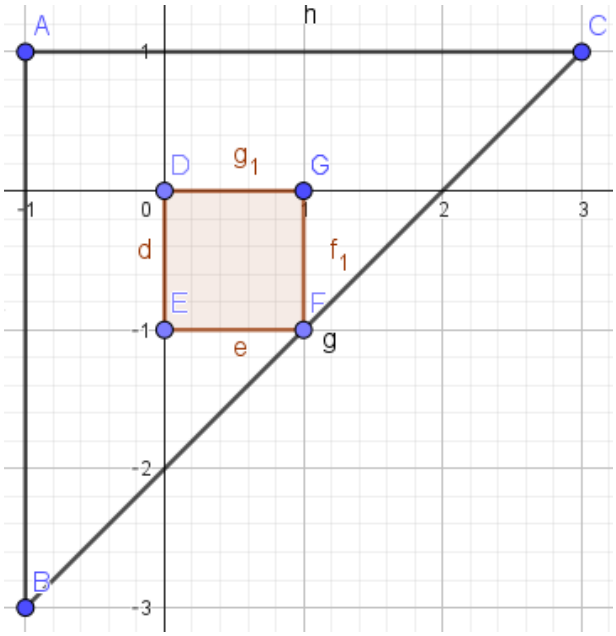
NDC 座標空間とは、カメラ座標変換されている座標を以下の範囲に正規化したもの。

- **X 軸** ： 左端が-1、右端が 1
- **Y 軸** ： 下端が-1、上端が 1
- Z 軸 ：NearZ 平面が 0、FarZ 平面が 1

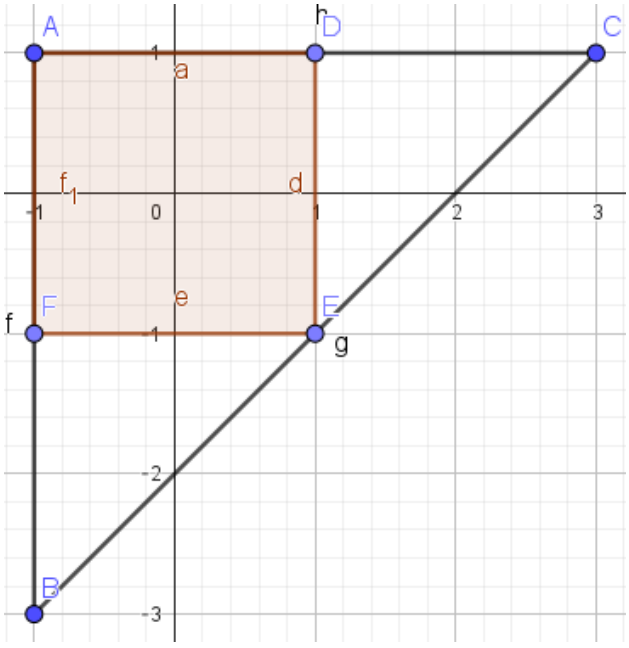
そのため、**Full Screen Triangle** の 3 頂点の座標はこの画像のように以下になる。

- (-1, -1, 1)
- (-1, 3, 1)
- (3, -1, 1)

これにより **NCD 座標空間**上の四角形(1)すべてを覆うことができる。またこのとき UV 座標を正確に設定しなければいけない。現在の **NDC 座標空間**上の頂点をそのまま UV 値として使用すると、DirectX では UV 座標原点は左上であり、U 値は右へ行くほど増加し、V 値は下へ行くほど増加するため以下のようになる。以下の画像では Y 軸は下から上へ向かってしまっているが、符号が反転しているとする。



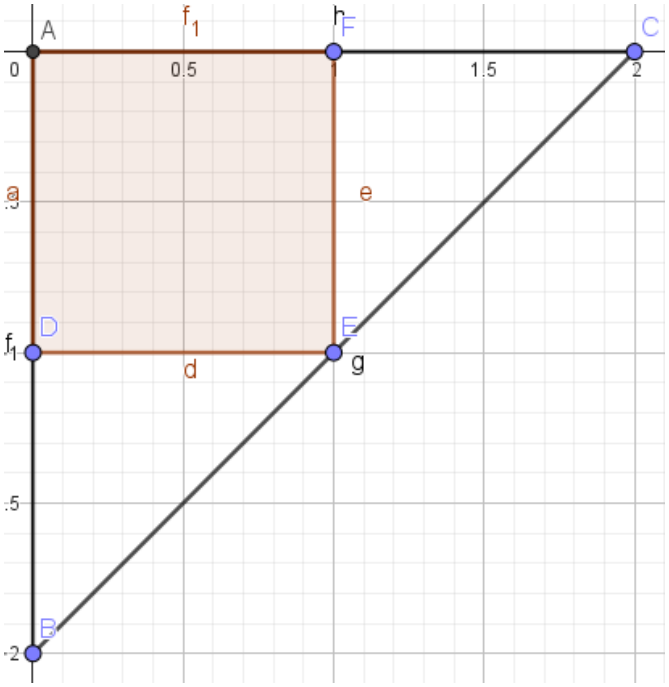
この画像の橙色の四角形(2)の範囲にテクスチャがあり、NDC 座標空間の四角形(1)の範囲が重なるようにすることで、描画範囲全体にテクスチャが適用されるようにすることができる。UV 座標空間上での四角形(1)は以下の画像のようになる。



この範囲をテクスチャの範囲である四角形(2)に、重ねるように変形すればよい。また、テクスチャは左上から右下にデータが格納されているため、三角形の向きは NDC 座標空間での向きと等しくある必要がある。よって Y 軸を反転させる必要もある。以下が手順ごとの変形処理になる。

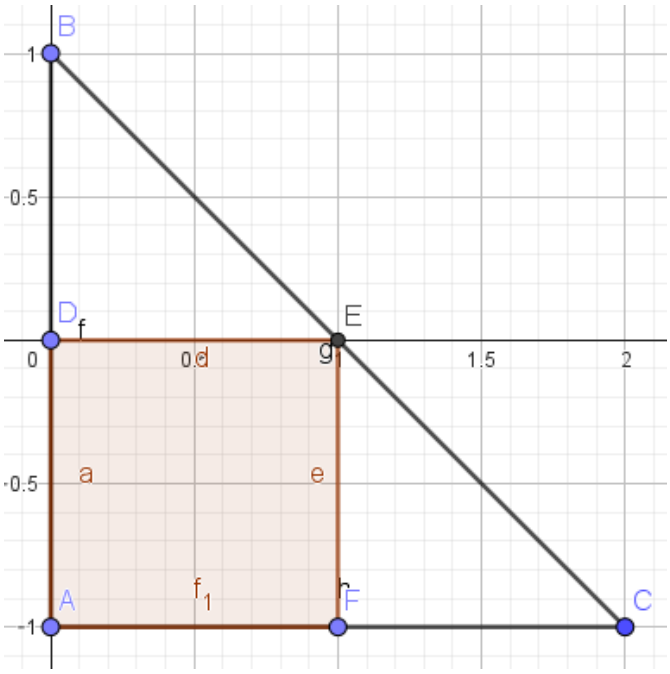
1. **NDC 座標空間**での四角形範囲を、テクスチャ範囲に重ねる。以下の式で行える

C++
`// input.Pos は NDC 座標空間上での三角形の頂点座標` `output.Tex = (input.Pos + 1.0) * 0.5;`



2. 三角形の向きを、**NDC 座標空間**での三角形の向きに揃える。以下の式で行える。

C++
`output.Tex.y = 1.0 - output.Tex.y;`



以上の処理により、描画される範囲にテクスチャの範囲が重なることで、正常にテクスチャマッピングすることができる。これにより、テクスチャの範囲全体に何らかのシェーダーの処理を行いながら、その実行結果をテクスチャとして出力することができる。

複数パスを使用したレンダリング

複数パスを使用したレンダリングとは、複数の RenderTarget で実行結果を 2D テクスチャとして次のレンダリングに渡しながら、描画処理を重ね掛けしていくものである。RenderTarget としてレンダリングをするには以下の構造体が持つ値らが必要になる。そのため一つのパスごとにこの構造体が必要になる。

C++

```
struct RenderTarget
{
    Microsoft::WRL::ComPtr<ID3D11Texture2D> texture;
    Microsoft::WRL::ComPtr<ID3D11RenderTargetView> view;
    Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> srv;
};
```

Texture2D を D3D11_TEXTURE2D_DESC から作成したのち、その 2D テクスチャの RenderTargetView を D3D11_RENDER_TARGET_VIEW_DESC 、ShaderResourceView、を D3D11_SHADER_RESOURCE_VIEW_DESC から作成する。

C++

```
RenderTarget renderTarget;

// テクスチャ作成
D3D11_TEXTURE2D_DESC textureDesc = {};

// パラメータを設定
d3DDevice->CreateTexture2D
(
    &textureDesc, nullptr, renderTarget.texture.GetAddressOf()
);

// RenderTargetView 作成
D3D11_RENDER_TARGET_VIEW_DESC rtvDesc = {};
```

```
// パラメータを設定
d3DDevice->CreateRenderTargetView
(
    renderTarget.texture.Get(), &rtvDesc,
    renderTarget.view.GetAddressOf()
);

// ShaderResourceView 作成
D3D11_SHADER_RESOURCE_VIEW_DESC srvDesc = {};

// パラメータを設定
d3DDevice->CreateShaderResourceView
(
    renderTarget.texture.Get(), &srvDesc,
    renderTarget.srv.GetAddressOf()
);
```

RenderTargetView を OMSetRenderTargets で以下のようにレンダリング対象に指定し、処理を実行した場合その実行結果が 2D テクスチャに入れられ、その実行結果の View が ShaderResourceView であるため、これを次のパスにテクスチャとして渡すことで複数パスでのレンダリングを行うことができる。
また、前述した Full Screen Triangle を使用することで、実行結果の 2D テクスチャに対し、ピクセルシェーダーを重ね掛けするように実行する。

C++

```
d3dDeviceContext->OMSetRenderTargets
(
    1, sceneRT.view.GetAddressOf(), depthStencilView.Get()
);
gD3dDeviceContext->ClearRenderTargetView
(
    firstRT.view.Get(), clearColor
);

gD3dDeviceContext->VSSetShader(firstVS.Get(), nullptr, 0);
gD3dDeviceContext->PSSetShader(firstPS.Get(), nullptr, 0);
```

```
// 処理を実行
// 画面全体を覆う三角形の頂点、インデックスバッファを設定
u32 stride = sizeof(Vertex);
u32 offset = 0;
d3dDeviceContext->IASetVertexBuffers
(
    0, 1, fullScreenTriangleVertexBuff.GetAddressOf(),
    &stride, &offset
);

d3dDeviceContext->IASetIndexBuffer
(
    fullScreenTriangleIndexBuff.Get(), DXGI_FORMAT_R32_UINT, 0
);

// 次のパスのレンダリング
gD3dDeviceContext->OMSetRenderTargets
(
    1, nextRT.view.GetAddressOf(), nullptr
);

// Full Screen Triangle 用の頂点シェーダーを設定
gD3dDeviceContext->VSSetShader
(
    FullScreenTriangleVS.Get(), nullptr, 0
);

// 重ね掛けするピクセルシェーダーを設定
gD3dDeviceContext->PSSetShader(nextPS.Get(), nullptr, 0);

// 1 度目の実行結果をテクスチャとして設定
gD3dDeviceContext->PSSetShaderResources
(
    0, 1, firstRT.srv.GetAddressOf()
);
// 処理を実行
```

ここで注意する点として、**ClearRenderTargetView** は **firstRT** に対してのみ実行するということだ。**Full Screen Triangle** を使用した描画では書き込む 2D テクスチャの全てのピクセルを更新するため、**ClearRenderTargetView** を実行する必要性がない。

Bloom の適用

Bloom をピクセルシェーダーにより適用するために必要なデータを、定数バッファでライトなどと同様に送る必要がある。送るデータは以下の構造体のようになる。

HLSL

```
cbuffer cbBlurParams : register(b5)
{
    float Threshold; // 閾値
    float2 TexSize; // (1 / textureWidth, 1 / textureHeight)
    float BloomIntensity;

    float BlurScale;
    float padding[3];
}
```

Bloom は以下の順で処理を行うことで適用することができる。

1. 一定の値（**Threshold**）より明るい部分のみ摘出。
2. 水平方向にブラーを掛ける
3. 垂直方向にブラーを掛ける
4. シーンのレンダリング結果（**Bloom** 処理を行わない場合のオブジェクトの描画結果）と垂直方向にブラーまで掛けた結果を合成する

ブラー処理ではガウス分布の重み{ 0.252035, 0.215441, 0.134143, 0.059103, 0.017278 }を使用する。
これらの順に処理結果を渡していくことで処理を行う。それぞれのピクセルシェーダーでの処理は以下のようになる。

■ 一定の値（**Threshold**）より明るい部分のみ摘出

HLSL

```
Texture2D sceneTexture : register(t0);
SamplerState samplerState : register(s0);
```

```
float4 main(VS_FSQOutput input) : SV_TARGET
{
    float3 color = sceneTexture.Sample(samplerState , input.Tex).rgb;

    // 輝度計算
    float brightness = dot(color, float3(0.2126, 0.7152, 0.0722));

    float3 bloomColor
    = (brightness > Threshold) ? color : float3(0.0f, 0.0f, 0.0f);

    return float4(bloomColor, 1.0f);
}
```

■ 水平方向にブラーを掛ける

HLSL

```
Texture2D bloomTexture : register(t0);
SamplerState samplerState : register(s0);

float4 main(VS_OUTPUT input) : SV_TARGET
{
    float weights[5]
    = { 0.252035, 0.215441, 0.134143, 0.059103, 0.017278 };

    float2 texelSize = 1.0 / TexSize;

    float3 color = bloomTexture.Sample
    (
        samplerState, input.Tex
    ).rgb * weights[0];

    for (int i = 1; i < 5; i++)
    {
        float2 offset = float2(texelSize.x * i * BlurScale, 0);

        color += bloomTexture.Sample
```

```
        (
            samLinear, input.Tex + offset
        ).rgb * weights[i];

        color += bloomTexture.Sample
        (
            samLinear, input.Tex - offset
        ).rgb * weights[i];
    }

    return float4(color, 1.0f);
}
```

■ 垂直方向にブラーを掛ける

HLSL

```
// Constant Buffer Variables
Texture2D horizBlurTexture : register(t0);
SamplerState samplerState : register(s0);

float4 main(VS_OUTPUT input) : SV_TARGET
{
    float weights[5]
    = { 0.252035, 0.215441, 0.134143, 0.059103, 0.017278 };

    float2 texelSize = 1.0 / TexSize;

    float3 color = horizBlurTexture.Sample
    (
        samplerState, input.Tex
    ).rgb * weights[0];

    for (int i = 1; i < 5; i++)
    {
        float2 offset = float2(0, texelSize.y * i * BlurScale);

        color += horizBlurTexture.Sample
```

```
        (
            samLinear, input.Tex + offset
        ).rgb * weights[i];

        color += horizBlurTexture.Sample
        (
            samLinear, input.Tex - offset
        ).rgb * weights[i];
    }

    return float4(color, 1.0f);
}
```

■ 結果を合成

HLSL

```
Texture2D sceneTexture : register(t0);
Texture2D vertBlurTexture : register(t1);
SamplerState samplerState : register(s0);

float4 main(VS_FSQOutput input) : SV_TARGET
{
    float3 sceneColor = sceneTexture.Sample
    (
        samplerState, input.Tex
    ).rgb;

    float3 bloomColor = vertBlurTexture.Sample
    (
        samplerState, input.Tex
    ).rgb;
    return float4(sceneColor + bloomColor * BloomIntensity, 1.0f);
}
```

以上の処理の結果を最後に SwapChain のバッファにレンダリングすることで、Bloom の結果を画面に反映することができる。

画面サイズ変更によるバッファサイズの更新

画面サイズが何らかの手段で変更された際、更新するべきものは以下の 4 つ。

1. **DepthStencil**
2. **RenderTarget**
3. **SwapChain**
4. ウィンドウサイズが関わっている**定数バッファ**

SwapChain は **ResizeBuffers** によりサイズ変更が行えるが、それ以外は一度解放して新しく作り直さなければならない。

バージョン管理が必要のないファイルは、コミットしない

コンパイル済みシェーダーである **cso** ファイルは、Git に上げる必要はないため gitignore に追加するなどして、コミットしないようにする。

while 文を使用したポーリング処理の注意点

以下のように **Condition** が **true** になった際のみ処理を行う処理があったとき、CPU 使用率に注意する必要がある。

C++

```
while (true)
{
    if (checkCondition())
    {
        handleEvent();
    }
}
```

このようにしてしまうと、常にループが実行されてしまうことにより一つの CPU コアの使用率が何も処理を行っていないのにも関わらず 100%になってしまうため非常に良くない。そこでループの実行される間隔を制御する必要がある。以下はその一つの例になる。

C++

```
while (true)
{
```

```
    if (checkCondition())
    {
        handleEvent();
    }

    std::this_thread::sleep_for(std::chrono::milliseconds(100));
}
```

このようにすることで、CPU 使用率を抑えることができる。
今回はポーリング処理だったが、C++ならおそらく std::condition_variable を使用し、スレッドを待機させ、処理を指定のタイミングで実行するようにするべきだと思われる。

真偽値を返す関数の命名

真偽値を返す関数の命名をする際に、先頭に **is** を付けようとすることはよくあると思われる。しかし、**is** で始めることが重要なのではなく、真偽値を返す関数は if 文で使用される場合が多いため、先頭に if を置いて最もしっくりくる表現にすることが重要である。以下のような構文に当てはめた考えることで、より良い命名になる場合が多い。

■ **if オブジェクト名 関数名**

一つの例として、**Item** クラスのメンバ変数に **name** という文字列があり、その文字列が空かどうか判断する場合は **isNameEmpty()** という名前を付ける場合があるかもしれない。しかし、この構文に当てはめてみるとおかしいことに気づく。

■ if this is name empty

そこで構文に当てはまるようにするには以下のように has を使用する。

■ if this has empty name

これから関数名にすると **hasEmptyName**()となり良い命名になる。
また、**is** 以外を先頭に持たせる場合は、動詞か助動詞を使うべきである。以下はその例。

■ 動詞で始まる場合
- exists (存在するか)
- contains (含まれているか)

- has（持っているか）
- needs（必要か）

■ 助動詞で始まる場合

- can（できるか）
- should（すべきか）
- need（する必要があるか）

protected を使用する必要がある実装は、おかしい場合が多い

protected は継承先のクラスでも基底クラスのメンバ関数や変数を使用するためのものだが、「継承は使用せず、インタフェースで実装することでクラスの肥大化を防ぐ」でも記述したが、これは親クラスに継承先で使用する関数や変数を追加していくため、クラスの肥大化に繋がる。よって **protected** は使用せずインタフェースで実装するようにするべきである。

データを格納する際は、文字列で比較するのではなく数値で

テクスチャのデータのコンテナを作る際に以下のようにする場合がある。

C++

```
class TextureContainer
{
private:
    std::unorederd_map
    <
        std::string,
        std::unique_ptr<TextureData>
    > textures_

public:
    TextureContainer() = default;
    ~TextureContainer() = default;

    TextureContainer(const TextureContainer&) = delete;
    TextureContainer& operator=(const TextureContainer&) = delete;

    void addTexture(std::string path);
```

```
    std::unique_ptr<TextureData>& getTexture(std::string);
};
```

こうしてしまうと、テクスチャを取得する際にそのテクスチャがコンテナに含まれているかを確かめるには、std::unordered_map にそのテクスチャのパスがあるかどうかを文字列比較により確かめなければいけなくなる。getTexture が頻繁に使用されるものでないならそこまで問題はないかもしれないがそうでない場合、これは問題になる。
そこで以下のように数値で比較することで、配列サイズとその数値を比較するだけで済むため問題が解消される。

C++

```
class TextureContainer
{
private:
    std::vector<std::unique_ptr<TextureData>> textures_;
public:
    TextureContainer() = default;
    ~TextureContainer() = default;

    TextureContainer(const TextureContainer&) = delete;
    TextureContainer& operator=(const TextureContainer&) = delete;

    unsigned int addTexture(std::string path);
    std::unique_ptr<TextureData>& getTexture(unsigned int id);
};

std::unique_ptr<TextureData>& TextureContainer::getTexture
(
    unsigned int id
){
    if (id >= textures_.size()) return nullptr;
    return textures_[id].get();
}
```

このように、データの固有情報としては文字列より数値である方が良い場面が多いため、積極的に数値で管理するようにするべき。

課題 3　コマンドデザインパターンを使用した、コンソール計算器

この課題では、コマンドデザインパターンを使用しコンソールに数値や演算子を入力することで計算を行う、コンソール計算機の開発を行った。また、完成したプログラムを使用し単体テストも行った。この課題で作成するべき処理は以下のもの。

- ユーザーに数値を入力させ、コンソール上で数式を表示する
- コマンドらを格納していき、計算が実行されるとその格納されたコマンドらから計算
- コマンドらから逆ポーランド記法を使用し、計算順序を考慮した計算
- 入力の Undo と Redo

この課題の狙いは、コマンドパターンなどのデザインパターンの理解や逆ポーランド記法の理解、単体テストのやり方を理解することだったと思われる。

コマンドデザインパターンとは

コマンドデザインパターンとは要求をオブジェクトとしてカプセル化し、呼び出し元と実際の処理オブジェクトを分離するデザインパターンである。ここで注意するべきことは以下のクラスらのような構成にし、処理することがコマンドデザインパターンを使用するということではなく、要求の呼び出し元と、実際の処理オブジェクトを分離することが本質ということである。

- **Command**
- **ConcreteCommand**
- **Receiver**
- **Invoker**

Invoker はクライアントに対して、アクション実行のトリガーを提供するものであり、直接にはコマンドデザインパターンの本質に関わっていない。そのためプロジェクトの規模や状況によっては **Invoker** を使わない設計にすることも考慮するべきである。

計算式の要素のコマンド化

今回の計算器では以下の処理に対応するようにした。

- 四則演算での計算
- 括弧を使用した計算

これらをコマンドパターンを使用し行うには、計算式に出てくる要素をコマンドとし、コマンドを格納、そのコマンドらから計算とするようにする。そのため以下の計算式に出てくる要素らをコマンドとする必要がある。

- 数値
- 四則演算子
- イコール
- 括弧

STL コンテナを使用した、異なるコマンドインスタンスの作成

コマンドを、ユーザーのコンソールへの入力から作成する際に文字列比較を行う必要があるが、以下のように **std::map** や **std::vector** などの STL コンテナを使用した分岐処理にすることで、後からコマンドの追加を行ったとしても分岐処理として追加するべき箇所はこのコンテナのみにすることができるため、以下のようにする。

C++

```
std::unordered_map<char, std::unique_ptr<Command>> cmdTable;
cmdTable[CMD_NUMBER] = std::make_unique<NumberCmd>();
cmdTable['+'] = std::make_unique<AddCmd>();
cmdTable['-'] = std::make_unique<SubtractCmd>();
cmdTable['*'] = std::make_unique<MultiplyCmd>();
cmdTable['/'] = std::make_unique<DivideCmd>();
cmdTable['('] = std::make_unique<LeftParenCmd>();
cmdTable[')'] = std::make_unique<RightParenCmd>();
```

Command クラスは各コマンドクラスのインタフェースになり、**cmdTable** に各数値、演算子の文字列から作成するべきコマンドクラスのインスタンスを設定している。

cmdTable からコマンドインスタンスを作成し、計算が実行されるまでコマンドらを格納す

る変数に追加するためには、**cmdTable** 上のインスタンスを複製して渡す必要がある。通常 **std::unique_ptr** 型の変数を複製するには以下のようにする。

C++

```
std::unique_ptr<ExampleClass> src
= std::make_unique<ExampleClass>();

std::unique_ptr<ExampleClass> copy
= std::make_unique<ExampleClass>(*src);
```

しかし、**cmdTable** には同じインタフェースは継承しているが、異なるクラスのインスタンスが追加されているため、以上のようなコピーは行おうとすると以下のようになる。

C++

```
if (input == '+')
{
    std::unique_ptr<Command> copy
    = std::make_unique<AddCmd>(*cmdTable[input]);
}
else if (input == '-')
{
    std::unique_ptr<Command> copy
    = std::make_unique<SubtractCmd>(*cmdTable[input]);
}
...
```

これだと、**STL コンテナ**を使用してもコマンドが追加されるたびに、マップへ要素を追加するだけでなく、分岐処理の追加記述も必要になってしまう。これではむしろ **STL コンテナ**を使用してしまっていることで良くない設計になってしまっている。

そこで以下のように **clone メソッド**をコマンドクラスらが持つようにすることで、各 **Command インタフェース**を継承するコマンドらのインスタンスを、**Command インタフェース**として格納したマップから、様々な種類の **Command インタフェース**を継承するコマンドクラスらのインスタンスを作成することができる。

C++

```
class Command
{
public:
```

```
    virtual ~Command() = default;
    virtual std::unique_ptr<Command> clone() = 0;
};

class AddCmd : public Command
{
public:
    AddCmd() = default;
    ~AddCmd() override = default;

    std::unique_ptr<Command> clone() override
    {
        return std::make_unique<AddCmd>(*this);
    }
};

class SubtractCmd: public Command
{
public:
    SubtractCmd() = default;
    ~SubtractCmd() override = default;

    std::unique_ptr<Command> clone() override
    {
        return std::make_unique<SubtractCmd>(*this);
    }
};

std::unique_ptr<Command> copy = cmdTable[input]->clone();
```

これにより、コンソールへの入力から異なる種類だが、同じインタフェースを継承しているクラスらのインスタンスを、文字列から作成することができる。

コンソールでの数式表示

コンソールで入力された値から数式を表示するためには、値が入力されるたびにコンソールの入力をクリアし、数式の更新を行う必要がある。そのため以下のような処理をコンソールへの入力が行われるたびに行えばよい。

C++

```
system("cls"); // コンソールのクリア

std::cout << "Console Calculator" << std::endl;
std::cout << expression << std::endl;
std::cout << "------------------------------" << std::endl;
```

expression を数式の文字列とし、この変数の値を更新することで式の表示を行える。

式へコマンドが追加できるかできないかの分岐

式へコマンドを追加できるかは、式の最後に追加されたコマンドによる。各コマンドクラスらの次に追加できるコマンドは以下のようになる。

最後に追加したコマンド	追加可能なコマンド
なし	数値、左括弧
数値	四則演算子、右括弧
四則演算子	数値、左括弧
左括弧	数値、左括弧
右括弧	演算子、右括弧

このように、各コマンドの次に追加することができるコマンドは、コマンドごとに異なるため何らかの関数内で分岐処理を行うか、式へ自身を追加する関数を各コマンドに持たせなければならない。
前者では、コマンドを追加するたびに追加するコマンドへの処理の記述だけでなく、何らかの関数内の分岐処理も追加で記述しなくてはいけなくなるため、後者の式へ自身を追加する関数を各コマンドに持たせることで、コマンドを追加する際は追加するコマンドクラスの処理のみ記述すれば良くなるため、より良い設計になる。以下のようなクラス構成にすれば可能になる。

C++

```
class Command
{
```

```
public:
    virtual ~Command() = default;
    virtual bool append
    (
        Calculator* calculator,
        std::vector<std::unique_ptr<Command>>& dst
    ) = 0;
};

class NumberCmd : public Command
{
private:
    double num_ = 0.0;

public:
    NumberCmd() = default;
    ~NumberCmd() override = default;
    virtual bool append
    (
        Calculator* calculator,
        std::vector<std::unique_ptr<Command>>& dst
    ) override;
};

class AddCmd : public Command
{
private:
    double num_ = 0.0;

public:
    AddCmd() = default;
    ~AddCmd() override = default;
    virtual bool append
    (
        Calculator* calculator,
        std::vector<std::unique_ptr<Command>>& dst
```

```
    ) override;
};

class LeftParenCmd : public Command
{
private:
    double num_ = 0.0;

public:
    LeftParenCmd() = default;
    ~LeftParenCmd() override = default;
    virtual bool append
    (
        Calculator* calculator,
        std::vector<std::unique_ptr<Command>>& dst
    ) override;
};
```

コマンドデザインパターンでは要求をオブジェクトとしてカプセル化し、呼び出し元と実際の処理オブジェクトを分離するため、式にコマンドを追加するという要求もオブジェクトとしてカプセル化するため、そのオブジェクトを持つ **Calculator クラス**を引数で渡す必要がある。
式にコマンドを追加するという要求の種類としては以下のようになる。

- 数値を式に追加
- 数値、左括弧、右括弧以外を式に追加
- 左括弧を式に追加
- 右括弧を式に追加

演算子は今回の場合、数値、左括弧、右括弧以外のすべてであるためこのようになる。これらの処理をそれぞれのコマンドクラスの **append メソッド**で呼び出すことで、自身を式へ追加する関数を各コマンドに持たせることができる。

コンソールへの入力が文字か数値かの判断

コンソールへ入力させ、それを受け取るためには **std::cin** を使うが今回のような場合、入力は数値化文字列か入力されるまで分からないため、どちらでも受け取れるように **std::string 型**で受け取る必要がある。このとき **char 型**では、大きな数値がコンソールへ入力されると受け取れないため、**std::string 型**でなければいけない。
そしてその受け取った文字列が数値かどうか判別するためには std::isdigit を使用し以下のようにすることで行える。

C++

```
std::string input;
std::cin >> input;

if (std::isdigit(input[0]) || (input[0] == '-' && input.size() > 1))
{
    // 数値の場合
}
else
{
    // 数値以外の場合
}
```

このとき注意しなければいけないことは、数値の先頭に「-」が来る可能性があり、「-」は引き算としても使用するため、負の数値の場合も考慮するようにする。

逆ポーランド記法への変換

今回の場合、計算式を逆ポーランド記法へ変換するとは、コンソールへの入力から作成したコマンドらを格納した std::vector 型の変数から、**逆ポーランド記法**へ変換した std::queue 型の変数を作成するということになる。これを行う関数を作成し、使用することで変換を行える。
逆ポーランド記法へ変換するにあたって各演算子の計算優先度が必要になるため、以下のように **Command クラス**に追加することで、各コマンドクラスで異なる値を返せるようにする。

C++

```
class Command
{
```

```
public:
    virtual ~Command() = default;
    virtual int priority() const { return 0; }
};

class AddCmd : public Command
{
public:
    AddCmd() = default;
    ~AddCmd() override = default;
    int priority() const override { return 1; }
};

class MultiplyCmd : public Command
{
public:
    MultiplyCmd() = default;
    ~MultiplyCmd() override = default;
    int priority() const override { return 2; }
};
```

数式を逆ポーランド記法へ変換するために今回は Shunting-Yard アルゴリズムを使用した。
実装例としては以下のようになる。

C++

```
std::queue<std::unique_ptr<Command>> RPN::ToRPN
(
    std::vector<std::unique_ptr<Command>>::iterator start,
    std::vector<std::unique_ptr<Command>>::iterator end
){
    // 逆ポーランド記法のコマンド列
    std::queue<std::unique_ptr<Command>> rpnCmds;

    // 演算子スタック
    std::stack<std::unique_ptr<Command>> opeStack;

    std::vector<std::unique_ptr<Command>>::iterator it = start;
```

```
    while (it != end)
    {
        // 数値の場合、キューにプッシュ
        if (PtrAs<NumberCmd>(it->get()))
        {
            rpnCmds.emplace(std::move(*it));
        }
        else
        {
            // 数値以外の場合
            // 左括弧
            if (PtrAs<LeftParenCmd>(it->get()))
            {
                opeStack.emplace(std::move(*it));
                it++;
                continue;
            }

            if (opeStack.empty()) opeStack.emplace(std::move(*it));
            else
            {
                // 右括弧
                if (PtrAs<RightParenCmd>(it->get()))
                {
                    // 左括弧までスタックから取り出す
                    while
                    (
                        !opeStack.empty() &&
                        !PtrAs<LeftParenCmd>(opeStack.top().get())
                    ){
                        rpnCmds.emplace(std::move(opeStack.top()));
                        opeStack.pop();
                    }
                    opeStack.pop(); // 左括弧を取り除く
                    it++;
```

```
                    continue;
                }

                // 括弧内の演算子の場合、スタックにプッシュ
                if
                (
                    it->get()->priority() > opeStack.top()->priority()
                ){
                    opeStack.emplace(std::move(*it));
                }
                else
                {
                    while
                    (
                        !opeStack.empty() &&
                        it->get()->priority()
                        <= opeStack.top()->priority()
                    ){
                        rpnCmds.emplace(std::move(opeStack.top()));
                        opeStack.pop();
                    }

                    opeStack.emplace(std::move(*it));
                }
            }
        }
        it++;
    }

    while (!opeStack.empty())
    {
        rpnCmds.emplace(std::move(opeStack.top()));
        opeStack.pop();
    }
    return rpnCmds;
}
```

この実装例では変換する対象の式の配列の最初及び、最後のイテレーターを渡し逆ポーランド記法変換済みの配列を取得する。**Command** らは **std::unique_ptr** で管理しているため、元の配列から変換済み配列へ **move** するようになっている。

逆ポーランド記法変換済み Queue から計算を行う

逆ポーランド記法変換済み **Queue** から計算を行うためには、**std::queue** から値を取り出していきながら計算する必要がある。この際 **Command** クラスに必要なメソッドは以下のようになる。

C++

```
class Command
{
public:
    virtual ~Command() = default;
    virtual std::unique_ptr<Command> execute
    (
        Calculator* calculator,
        std::unique_ptr<Command>& left, std::unique_ptr<Command>& right
    ) = 0;
};

class AddCmd : public Command
{
public:
    AddCmd() = default;
    ~AddCmd() override = default;
    std::unique_ptr<Command> execute
    (
        Calculator* calculator,
        std::unique_ptr<Command>& left, std::unique_ptr<Command>& right
    ) override;
};
```

取り出し、計算する処理としては以下のようになる。

C++

```
// スタックにキューからコマンドを取得しながら計算
std::stack<std::unique_ptr<Command>> calcCmdStack;

while(!rpnCmdQueue.empty()) // キューが空になるまで繰り返す
{
        // キューから取り出す値が数値である限りスタックにプッシュ
        while(PtrAs<NumberCmd>(rpnCmdQueue.front().get()))
    {
        calcCmdStack.emplace(std::move(rpnCmdQueue.front()));
        rpnCmdQueue.pop();
    }

    // キューから取り出す値が演算子の場合

    // スタックから計算する値らを取得
    std::unique_ptr<Command> leftNumCmd
    = std::move(calcCmdStack.top());
    calcCmdStack.pop();

    std::unique_ptr<Command> rightNumCmd
    = std::move(calcCmdStack.top());
    calcCmdStack.pop();

    std::unique_ptr<Command> ope = std::move(rpnCmdQueue.front());
    rpnCmdQueue.pop();

    // 計算実行
    std::unique_ptr<Command> result
    = ope->execute(calculator, leftNumCmd, rightNumCmd);
    if (!result) return;

    calcCmdStack.emplace(std::move(result));
}
```

この処理の **while ループ**が全て終了したとき、**calcCmdStack** のサイズは 1 になりその一つの要素が計算結果となる。

スマートポインタでのやり取り時の生ポインタの使用

std::unique_ptr や **std::shared_ptr** は **get** メソッドを使用することで生ポインタにより受け取ることができるが、このように扱うのは極力さけるべきである。
理由としてはスマートポインタであることが分からなくなってしまうからである。もし他人が作成した関数の引数に実はスマートポインタであり、get メソッドを使用し生ポインタを渡しているのにも関わらず、それが **new** を使用して作成したものだと考えてしまい、**delete** などを行ってしまい不具合が起きることが想定されるからである。

このことからも、スマートポインタでのやり取りしているのにも関わらず、生ポインタで受け渡しは行うべきでない。ただし自身のインスタンスを渡したいため **this** を使用したい場合など、やむを得ない場合もあるため、その場合は他人からその生ポインタ変数がスマートポインタであると分かるようにするべきである。

メンバ変数で別クラスのポインタや参照を持つことの設計上の問題

以下のように、クラスのメンバ関数内で別のクラスのポインタや参照を使用し、処理を行ってしまう場合がある。

C++

```
class AddCmd : public Command
{
private:
    Calculator* calculator;
public:
    AddCmd() = default;
    ~AddCmd() override = default;

    std::unique_ptr<Command> execute
    (
        std::unique_ptr<Command>& left,
        std::unique_ptr<Command>& right
    ) override;
};
```

```
std::unique_ptr<Command> AddCmd::execute
(
    std::unique_ptr<Command>& left,
    std::unique_ptr<Command>& right
){
    return calculator->add(left, right);
}
```

このようにしてしまうと、結合度の観点で見ると、**calculator** の値が状況に異なるためハイブリット結合（外部結合）となってしまい悪い設計になってしまう。そのため、クラスのメンバ関数内で別のクラスのポインタや参照を持たないようにメンバ関数の引数としてその別のクラスを渡すようにすることで、データ結合としこれよりは良い設計となる。

C++

```
class AddCmd : public Command
{
public:
    AddCmd() = default;
    ~AddCmd() override = default;
    std::unique_ptr<Command> execute
    (
        Calculator* calculator,
        std::unique_ptr<Command>& left,
        std::unique_ptr<Command>& right
    ) override;
};

std::unique_ptr<Command> AddCmd::execute
(
    Calculator* calculator,
    std::unique_ptr<Command>& left,
    std::unique_ptr<Command>& right
){
    return calculator->add(left, right);
}
```

しかしこの場合だと受け渡しているデータが、クラスインスタンスのポインタであり単純でないため、**execute 関数**内で例として **Calculator クラス**のメンバ変数の値を変えるなど、**Calculator クラス**と **Command クラス**の依存度が高まる場合がある。
そのため **Calculator クラス**が継承する以下のようなインタフェースを作成することで、**Command クラス**が呼び出せる関数を制限できる。これにより受け渡すデータを単純にし、依存度が高まることが無いようにすることができる。

C++

```
class ICommandExecutable
{
public:
    virtual ~ICommandExecutable() = default;

    std::unique_ptr<Command> add
    (
        std::unique_ptr<Command>& left,
        std::unique_ptr<Command>& right
    ) = 0;
};

class Calculator : public ICommandExecutable
{
public:
    Calculator() = default;
    ~Calculator() override = default;

    std::unique_ptr<Command> add
    (
        std::unique_ptr<Command>& left,
        std::unique_ptr<Command>& right
    ) override;
};

class AddCmd : public Command
{
public:
    AddCmd() = default;
```

```
    ~AddCmd() override = default;

    std::unique_ptr<Command> execute
    (
        ICommandExecutable* executableFunc,
        std::unique_ptr<Command>& left,
        std::unique_ptr<Command>& right
    ) override;
};

std::unique_ptr<Command> AddCmd::execute
(
    ICommandExecutable* executableFunc,
    std::unique_ptr<Command>& left,
    std::unique_ptr<Command>& right
){
    return executableFunc->add(left, right);
}
```

履歴を使用した式の入力の Undo と Redo

コンソールへの入力からコマンドを作成しそれを式に追加する際、履歴にもコマンドを追加することで **Undo** と **Redo** を行うことができる。
Undo では現在の式に追加されているコマンドを一つ削除し、**Undo** した回数を 1 増やす。また **Redo** では履歴のサイズから **Undo** した回数を引き、その値をインデックスとした履歴のコマンドを現在の式に追加するだけでよい。**append** では自分を式へ追加している。

C++

```
class Command
{
public:
    virtual ~Command() = default;
    virtual bool append
    (
        Calculator* calculator,
        std::vector<std::unique_ptr<Command>>& dst
    ) = 0;
```

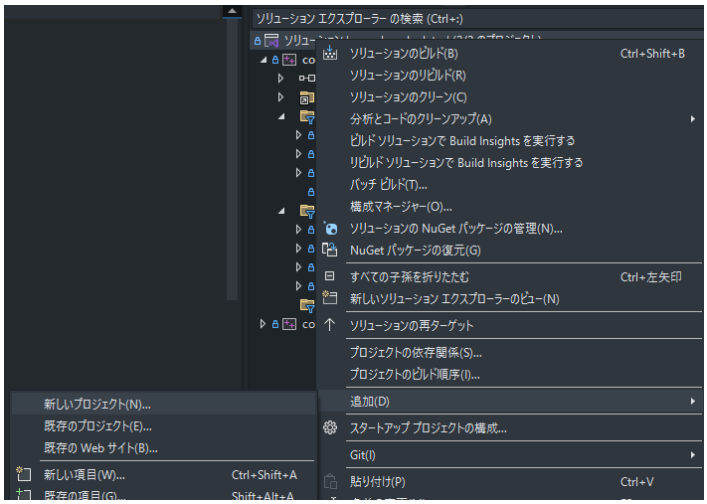
```
};

void Calculator::undo()
{
    // これ以上 Undo できない
    if (expressionCmds_.size() == 0) return;

    expressionCmds_.pop_back();
    undoTimes_++;
}

void Calculator::redo()
{
     // これ以上 Redo できない
    if (undoTimes_ == 0) return;

    historyCmds_[historyCmds_.size() - undoTimes_].append
    (
        this, expressionCmds_
    );

    undoTimes_--;
}
```

これだけの処理で **Undo** と **Redo** を実行できるのはコマンドデザインパターンの要求をオブジェクトとしてカプセル化し、呼び出し元と実際の処理オブジェクトを分離していることによるものであり、これがコマンドデザインパターンの利点の一つである。

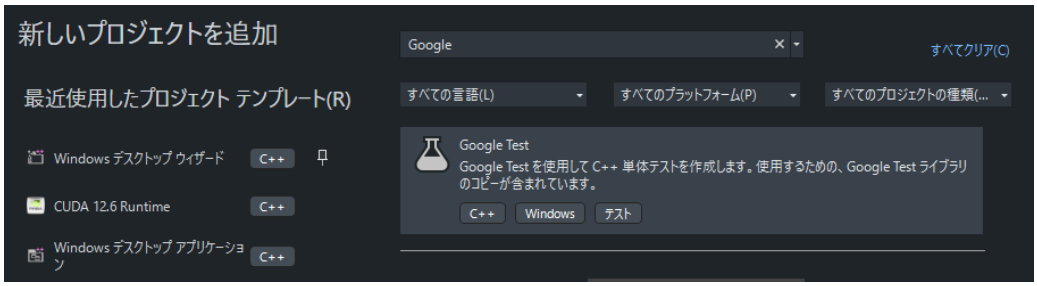
Visual Studio での Google Test を使った単体テストの行い方

現在の Visual Studio では C++のデスクトップ開発をダウンロードしている場合、Google Test は既にダウンロードされているため、新しいプロジェクトとして追加することで使用することができる。使用するまでの手順は以下のようになる。

1. ソリューションに Google Test のプロジェクトを追加



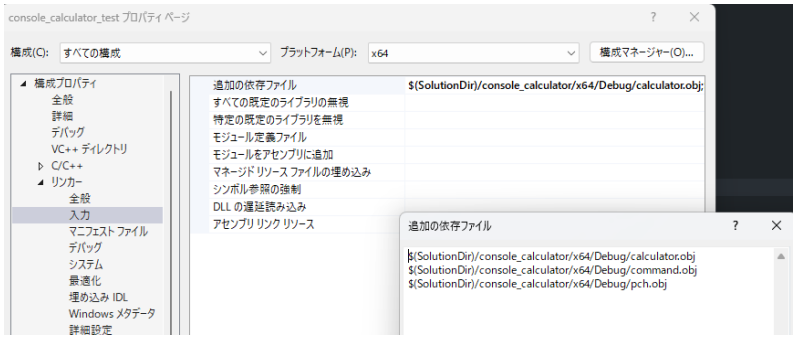
ソリューションにこの画像のようにすることで、プロジェクトを新規作成し追加できる。Google Test のプロジェクトはこれのことである。



テストプロジェクトを作成しようとすると、作成する際の設定を決めるウィンドウが出てくるが、基本的に何も変更する必要はない。

2. テスト対象プログラムのコンパイル済み obj ファイルを参照する

Google Test プロジェクトのプロパティ -> リンカー -> 入力 -> 追加の依存ファイルに、テスト対象のプロジェクトのコンパイル結果である、各 C++ファイルの obj ファイルを追加する。これによりビルドが通り、テストを実行することができる。



3. test.cpp に記述しテストを実行する

Google Test を使用した単一テストの実行方法

Google Test には **Test 関数**を定義するための以下のようなマクロが提供されており、このマクロを使用し、**Test 関数**らを作成後それらの Test を main 関数内で実行する。

C++

```
TEST(testCaseName, testName)
{

}

int main(int argc, char **argv)
{
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

そして Test 関数にテストが成功したか、失敗したかを判断するための以下の関数を追加することで、実行するとコンソールにそのテストが成功したか、失敗したかが出力される。

C++

```
EXPECT_EQ(expected, actual);
```

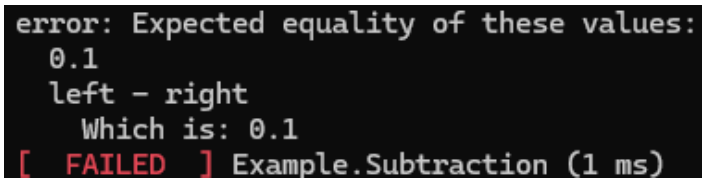
アサーションマクロの適切な選択

アサーションマクロは、テストの結果を確認するために使用される。
EXPECT_EQ は期待される値（expected）と実際の値（actual）が等しいとき成功とし、等しくないとき失敗と返すものである。アサーションマクロは様々なのもがあるため、用途に応じて適切なものを選ぶ必要がある。
例として double 型変数の値が期待している値と等しいかテストする場合、以下では正常に動作しない。

C++

```
TEST(Example, Subtraction)
{
    double left = 1.0;
    double right = 0.9;

    EXPECT_EQ(0.1, left - right);
}
```

これの実行結果は以下のようになる。



この処理が失敗となる理由は浮動小数点型における誤差のためである。EXPECT_EQ では double 型などの浮動小数点型の場合、誤差に対応できないことで失敗となってしまう。
そこで以下のように EXPECT_DOUBLE_EQ に変更することで誤差に対応することができ、浮動小数点型であっても正常にテストを行うことができる。

C++

```
TEST(Example, Subtraction)
{
    double left = 1.0;
    double right = 0.9;

    EXPECT_DOUBLE_EQ(0.1, left - right);
}
```

このように、**アサーションマクロ**をテスト結果の値から適切に選択する必要がある。

単体テストにより分かる、単一責任及び結合度の重要性

単体テストを行うためには、テスト対象のクラスのインスタンスを Test 関数内で作成、そのクラスのメンバ関数を実行し、その実行結果をアサーションマクロで成功か失敗か出力する場合がある。このような場合のときに単一責任や結合度が重要になる。以下がそれぞれなぜ単体テストを実行するために大事になるのかである。

■　単一責任

クラスが単一責任を果たしているということは、他のクラスらと依存していない、もしくは最低限にできているということである。

単体テストを実行するにあたってもしテスト対象のクラスが複数の別のクラスらと依存している場合、それらのクラスのインスタンスの作成、データの構築、受け渡しを Test 関数内で行う必要ができてしまい、単体のテストを行いたいのにも関わらず、プロジェクト上でエラーハンドリングすることと何も変わらないようなことになってしまう。また、テストを実行するために必要なクラスが増えるということは、リンクさせる必要のあるソースファイルをコンパイルした obj ファイルも増えるということであり、設定するべきものが増えてしまう。

このように、単一責任を果たせていない場合単体テストの実行が困難になってしまう。これはつまり単体テストの実行が容易になるようなクラスにすることで、単一責任を果たしたクラスにすることができるということである。

■　結合度

クラスが良い結合度（データ結合など）になっている場合、そのクラスのメンバ関数に対して何らかのデータを渡すことで、そのメンバ関数を動作させることができる。これはつまり Test 関数内で作成するべきデータが限定されるということであり、単体テストを容易に実行できる。

以上のように、単体テストを実行するにあたってクラスの単一責任や、結合度は重要になる。またこれは単体テストを容易に実行できるようにクラスを設計することで、単一責任を果たし、良い結合度のクラスにできるということである。そのためクラスの作成段階から、単体テストをどのようにして行うかを想像しながら作成するべきであり、単体テストを容易に行うためにも、単一責任や結合度は重要になる。

課題 4　静的メモリに割り当てる関数オブジェクト

4 つめの課題は、静的メモリに割り当てる関数オブジェクト（inplace_function）の作成だった。臨地実務実習の最終日から始めたため触りのみだったが、静的メモリに関数オブジェクトを割り当て、呼び出すところまでの作成を行った。

関数オブジェクトとは

関数オブジェクトとは、任意の引数を持つ **operator()** メンバ関数を定義することで、そのオブジェクトのインスタンスを指す変数の名前が、あたかも関数名（関数指示子）であるかのような構文で、定義した関数を呼ぶことができる構造体やクラスのことである。以下のようなものになる。

C++

```cpp
struct Functor
{
    void operator()(int x) const
    {
        std::cout << "Functor called with " << x << std::endl;
    }
};
```

静的メモリと動的メモリ

静的メモリとは変数に **[]** を使用することで、配列を作成するものである。それに対して動的メモリとは **new** などを使用し、配列を作成するものである。

C++

```cpp
// 静的メモリ
int memoryStatic[size];

// 動的メモリ
int* memoryDynamic = new int[size];
```

メンバ関数の Const 修飾と mutable

C++においてメンバ関数のクラスを以下のように **const** にすると、そのメンバ関数内で自身のメンバ変数の値を変更できなくなりまた、別のメンバ関数を呼び出せなくすることができる。

C++

```
class ExampleClass
{
private:
    int num_;
public:
    void func();

    int ConstMethod() const;
};

int ExampleClass::ConstMethod() const
{
    // num_ = 0; // コンパイルエラー
    // func(); // コンパイルエラー

    std::cout << num_ << std::endl; // OK
    return num_; // OK
}
```

このようにすることで、**const メンバ関数**がクラスデータに対して影響を及ぼさないことが保証され、単一責任を果たし、凝縮度を高めることになる。
また、**const メンバ関数内**で自身の特定のメンバ変数の値を変えたい場合、mutable を使用することで、値の変更を **const メンバ関数内**でも許可することができる。

C++

```
class ExampleClass
{
private:
    mutable int num_;
public:
    void func();
```

```
    void ConstMethod() const;
};

void ExampleClass::ConstMethod() const
{
    num_ = 10; // OK
}
```

これにより、**const 修飾子**を使用することでクラスのメンバ変数、関数へのアクセスを制限しながら、特定のメンバ変数は値の変更を行えるようにできることで、結合度が悪くなることを防ぐことができる。

また、**const 修飾子**と参照を以下のように使用することで、外部から参照することは可能だが、値を変えることはできないようにすることができる。

C++

```
class ExampleClass
{
private:
    int num_;
public:
    const int& GetNum() const { return num_; }
};

const int& num = obj.GetNum(); // Reference
int num2 = obj.GetNum(); // Copy
```

これにより、適切にメンバ変数の値を外部で使用することができる。

バイトデータでの関数オブジェクトの保存

関数オブジェクトは型ごとに異なることが想定されるため、コピーや移動をしようとする際、関数オブジェクトを保存しているデータの型が異なる場合、余計な処理が増えることになる。そのため **std::byte** を使用し、ビット列としてのバイトデータに関数オブジェクトを保存することでデータ上は違うが、保存しているデータの型は等しくなり、コピーや移動が容易になる。バイトデータは以下のようにする。

C++

```
mutable alignas(alignof(void*)) std::byte storage[StorageSize];
```

ここで **StorageSize** は関数オブジェクトのサイズを表している。またここでは alignas を使用しバイトデータが特定のサイズでアラインメントされるよう指定し、そのサイズは型ごとに異なり、それを alignof から取得している。これにより **void***の適切なアラインメント要件が **storage** に適用される。

テンプレートの特殊化を使用したクラスの宣言

テンプレートの特殊化とは、テンプレートが使用されている同じクラス名、関数名のものにおいて、そのテンプレートが特定の値のときに使用されるものを指定することができる。テンプレートの特殊化ではパラメータが確定したテンプレートが必要なため、以下のように前方宣言をして使用する必要がある。

C++

```
template <typename Signature, std::size_t StorageSize>
class InplaceFunc;

template <typename R, typename... Args, std::size_t StorageSize>
class InplaceFunc<R(Args...), StorageSize>
{

};
```

この特殊化では **typename Signature** が、**typename R**、 **typename... Args** により構成される場合、下の **InplaceFunc** を使用するというものである。また、この **InplaceFunc** を実際に使用するには以下のようにする。

C++

```
struct Functor
```

```
{
    void operator()(int x) const
    {
        std::cout << "Functor called with " << x << std::endl;
    }
};

InplaceFunc<void(int), sizeof(Functor)> func = Functor();
```

ここでは R には **void**、Args には **int**、StorageSize には **Functor** のサイズが入る。これはこの **InplaceFunc** で呼び出す関数は **Functor 関数オブジェクト**の型が **void**、引数が **int** のみの**オペレーター演算子**であることを指定している。

また、**Functor()**を使用し初期化を行っており、これにより呼ばれるコンストラクタは以下である。

C++

```
template <typename R, typename... Args, std::size_t StorageSize>
class InplaceFunc<R(Args...), StorageSize>
{
public:
    template <typename F>
    InplaceFunc(F&& f)
    {

    }
}
```

このコンストラクタの「型名&& 変数名」は右辺値参照をしているということであり、リソースの **move** が可能である。

また、左辺値と右辺値とはそれぞれ以下の特徴を持つものだと理解しておくと良い。

値名	特徴
左辺値	アドレスを取得できる値、名前の付いた変数
右辺値	アドレスを取得できない値、一時オブジェクト

placement new を使用した関数オブジェクトの割り当てとその解放

配置新規化（placement new）とは以下のように **new** を使用する際に既に確保されているメモリにオブジェクトを割り当てるものである。既に確保されているメモリとは今回の場合、関数オブジェクトのサイズを持つバイトデータになる。
また std::decay_t を使用し型 **F** の標準的な形式（**F** のこと、**F&**の場合は **F** を推測する）に変換し、**f** から関数オブジェクトを右辺値で作成している。
f が右辺値の場合、型 F は既に標準的な形式のため **move** が行われ、左辺値の場合は **f** が **F&** のため標準的な形式（**F**）を **std::decay_t** で推測し、その型で元データの **f** のコピーを右辺値として作成する。

C++

```
alignas(alignof(void*)) std::byte storage[StorageSize];

template <typename F>
InplaceFunc(F&& f)
{
    new (&storage) std::decay_t<F>(f);
}
```

このように **placement new** を使用し、既に確保されたメモリ領域に関数オブジェクトを割り当てた場合、この **InplaceFunc クラスインスタンス**が解放される際に解放されるメモリは、storage であり割り当てた関数オブジェクトが解放されるのではなく、割当先のメモリが解放されることになる。この場合、関数オブジェクトのデストラクタが呼ばれない。
そのため以下のように **InplaceFunc クラス**のデストラクタで、関数オブジェクトのデストラクタを強制的に呼び出す必要がある。

しかしここで注意しなければいけない点として、**InplaceFunc** は関数オブジェクトの指定の型、引数の operator()を呼び出すものだということである。これはつまり同じ型、引数の operator()を持つ別の関数オブジェクトを同じインスタンスに割り当てる可能性があるということになるため、**InpalceFunc** のデストラクタで関数オブジェクトのデストラクタを呼ぶためには確保済みメモリであるバイトデータから割り当てられている関数オブジェクトの型にキャストし、呼び出さなければならない。また、デストラクタは暗黙的に呼び出されるためコンストラクタのように引数の値から型を予想するテンプレートの使用はできない。
ならどのようにすることで、**InpaceFunc** のデストラクタでバイトデータに格納されている関数オブジェクトのデストラクタを呼び出すかというと、関数オブジェクトをバイトデータに **placement new** を使用し割り当てた段階で、**InplaceFunc** のデストラクタでバイトデー

タのキャスト先となる関数の型を、以下のようにテンプレートとして保存、登録した関数ポインタを用意し、それを InplaceFunc のデストラクタで呼び出せばよい。

C++

```
// キャスト先となる関数オブジェクトの型をテンプレートを使用し保存。
// その保存した関数オブジェクトの型にキャストし、デストラクタを呼び出す。
template <typename F>
static void Del(void* storage)
{
    static_cast<F*>(storage)->~F();
}

void (*deleter)(void*);

InplaceFunc(F&& f)
{
    // placement new で関数オブジェクトを保存
    new (&storage) std::decay_t<F>(f);

    // デストラクタを呼び出す関数オブジェクトの型を保存
    deleter = &Del<std::decay_t<F>>;
}

~InplaceFunc()
{
    // 関数オブジェクトのデストラクタを呼び出す
    deleter(storage);
}
```

このようにすることで、InplaceFunc が解放される際に割り当てた関数オブジェクトのデストラクタが呼ばれるようにすることができる。

割り当てた関数オブジェクトのオペレーター演算子を使用した呼び出し

InplaceFunc では割り当てた関数オブジェクトの、InplaceFunc 作成時にテンプレートによって登録したオペレーター演算子を呼び出す必要がある。

C++

```
struct Functor
{
    void operator()(int x) const
    {
        std::cout << "Functor called with " << x << std::endl;
    }
};

InplaceFunc<void(int), sizeof(Functor)> func = Functor();

func(10); // 呼び出し
```

func()により呼ばれる **InplaceFunc** のメソッドは以下のもの。

C++

```
template <typename Signature, std::size_t StorageSize>
class InplaceFunc;

template <typename R, typename... Args, std::size_t StorageSize>
class InplaceFunc<R(Args...), StorageSize>
{
public:
    R operator()(Args... args) const
    {

    }
};
```

このオペレーター演算子の中でバイトデータに割り当てられた関数オブジェクトの型 R(今回の例の場合 **void**)、引数 **Args…**（今回の例の場合 **int** のみ）のオペレーター演算子を呼び出せばよい。しかし **func(10)**のように呼び出すため、デストラクタと同様に引数から関数オブジェクトの型を予測することはできない。そのためデストラクタの関数ポインタと同様、呼び出すための関数ポインタを作成する必要がある。

C++

```
template <typename F>
static R Call(void* storage, Args&&... args)
```

```
{
   return (*static_cast<F*>(storage))(std::forward<Args>(args)...);
}

R (*caller)(void*, Args&&...);

template <typename F>
InplaceFunc(F&& f)
{
    new (&storage) std::decay_t<F>(f);

    caller = &Call<std::decay_t<F>>;
}
```

ここで **Call 関数**にはテンプレートを使用し、**std::decay_t<F>**を登録しているため、バイトデータに割り当てた関数オブジェクト **storage** を **void***型から **F***型にキャストし、引数を与えることで、割り当てられた関数オブジェクトの登録されたオペレーター演算子を呼び出すことができる。

関数オブジェクトの再割り当てと明示的なデストラクタの呼び出し

以下のように一度作成した関数オブジェクトに、別の関数オブジェクトを割り当てなおす際は、明示的にデストラクタを呼び出す必要がある。

C++

```
InplaceFunc<void(int), sizeof(Functor)> func = Functor();
func(10);

func.~InplaceFunc(); // 明示的なデストラクタ呼び出し

func = AnotherFunctor(); // 再割り当て
func(20);
```

このようにしなければ、既に割り当てられているメモリ領域に新たなデータを割り当ててしまうことによって、**メモリリーク**や**未定義の動作**が発生する可能性がある。そのため再割り当てを行う際は必ず明示的に関数オブジェクトのデストラクタを呼び出す必要がある。

臨地実務実習での反省点及び、課題として残ったこと

まず反省点としては、担当の方にも何か今回の臨地実務でするべきだったこと、足りなかったことはありませんかと質問した際に指摘を受けた点でもある、課題を与えられたらまずゴールを確認し、都度確認を取りながら作業を進めるようにするべきだったことである。今回は課題を与えられ、その課題のプログラムを作成し、コードレビューを受け確認してもらう形式だったが、課題を進め一段落ついたところで初めて確認してもらうのではなく、課題のプログラムはどのようなものにするべきか逐次確認を取りながら進めるべきだった。想定しているゴールが担当者の方と、自分の中で認識の齟齬があった場合が 2、3 度あり作業効率が落ちてしまった。これが実際の業務で起きてしまった場合、違った方向で開発してしまった時間が無駄になることで、その時間に会社から出される給料も会社側からすれば無駄な出費だったということになり、会社としての損失になる。そのようなことが起こらないよう、できましたと確認をお願いするのではなく、確認を取り認識を合わせながら開発し、提出時の確認ではほとんど確認する必要性がないようにするべきだった。

次に課題として残ったことである。今回の課題ではプログラムのメモリ操作にも触れ、その中で適切にデータを参照、移動、コピーする必要があったが、コンピュータサイエンスに関する知識不足からコードレビューを受けなければ適切にデータを扱えなかった。担当者の方々に今後するべきことを質問した際にも、メモリアロケータなどを自作し、コンピュータサイエンスへの理解を深めるべきと指摘を受け、実際に臨地実務実習を通じてもその通りだと実感したためコンピュータサイエンスへの理解を深めることが課題として残った。

また良い結合度や凝縮度、単一責任を実現できているプログラムを、何を作成したいのか的確に言語化することでリファクタリング時にプログラムを変更し改善するのではなく、最初から作成できるようになることも課題になる。今回の臨地実務実習での課題を通じて、作成するプログラムが持つ機能、構造を正確かつ、詳しく言語化することにより、より良い設計にすることができると実感した。しかしまだまだこの言語化が、プログラム作成前では十分にできないのが現状である。リファクタリング時に機能や構造を言語化し、適切な設計に修正することはできるようになったが、プログラム作成前にこれを行うには慣れや経験がまだまだ足りない。そのためこれが 2 つ目の課題として残った。

以上を踏まえ、今後の学業で何らかの課題に人と取り組む際は、課題のプログラムはどのようなものにするべきか逐次確認を取りながら進めたいと思う。また学業以外でもそうだが、何らかのプログラムを作成する際は、作成前に機能、構造を正確かつ詳しく言語化し、良い結合度や凝縮度、単一責任を実現できているプログラムに、リファクタリング時に修正するのではなく、最初からすることができるようにしていきたい。

参考文献

Visual Studio で C++ プロジェクト作成後に行うプリコンパイル済みヘッダー
【C++】assert の使い方と利用シーン
assert - cpprefjp
assert を使うべき場面と使うべきでない場面
basic_string_view - cpprefjp
constexpr - C++11 の機能 - cpprefjp
eunum class
クラスの命名のアンチパターン
std::map 型を使って見通しのよいコーディング
知っておくとためになる。モジュール凝集度について解説
結合度について
unique_ptr - cpprefjp
shared_ptr - cpprefjp
pack - Microsoft C++ ドキュメント
reinterpret_cast 演算子 - Microsoft C++ ドキュメント
無名名前空間の使い方
連長圧縮 - Wikipedia
DirectXTex TGA ファイル処理実装
リファクタリング自爆奥義集
Dear ImGui - ゲーム開発用 GUI ライブラリ
コマンドデザインパターン
逆ポーランド記法 - Wikipedia
[初心者向け]破滅的なクソコードを書かないために意識したい 3 つのこと
循環参照 - Wikipedia
セピア調フィルタのうんちく
Wavefront .obj ファイル - Wikipedia
右手系 - Wikipedia
フォンシェーディング - Wikipedia
ランバート反射 - Wikipedia
Bloom (shader effect) - Wikipedia
condition_variable - cpprefjp
真偽値を返す関数のネーミング
isdigit - cpprefjp
Shunting Yard Algorithm について

Visual Studio の Google Test プロジェクトの使い方
関数オブジェクト - Wikipedia
mutable 記憶域クラス指定子 - IBM Docs
decay - cpprefjp
alignas - C++11 の機能 - cpprefjp
alignof - C++11 の機能 - cpprefjp
テンプレートの特殊化 - C++入門
左辺値、右辺値と C++のムーブと完全転送を知る
placement new でメモリを管理する