KLab

KLab Tech Book

クラブテックブック

GitHub Actions でカジュアルに OpenAI を利用する

2D Shader 試験場

mypy 再訪

shader graph で水表現

XML プログラミング言語「XSLT」の可能性

好きな C++ のコア機能発表ドラゴン

VOL. 14

全年齢対象

KLab Tech Book Vol. 14

KLab 技術書サークル　著

2024-11-02 版　　**KLab** 技術書サークル 発行

はじめに

このたびは本書をお手に取っていただきありがとうございます。本書は KLab 株式会社の有志にて作成された KLab Tech Book の第 14 弾です。

KLab では主にモバイルオンラインゲームを開発していますが、KLab Tech Book では社内有志が業務との関連によらない、新しい知識や技術に好きなようにチャレンジした内容を執筆しています。表紙のデザインも社内のデザイナーの方にご協力いただき、KLab 感溢れる一冊に仕上がっています。

今回は 6 記事を収録しました。それぞれの著者によって、仕様についての丁寧な説明や、実用的な手法の紹介などが行われています。章ごとに内容が独立しているので、気になるものから順に読み進めていただいて問題ありません。

記事で触れられている技術領域は様々ですが、あまり触れたことがない領域であっても、一読していただくことで新しい興味分野の発掘や思いもよらぬ発想の種になるかもしれません。

本書を通して、そんな知的な営みの楽しさを感じていただけたらさいわいです。

梅澤　寿史

お問い合わせ先

本書に関するお問い合わせは tech-book@support.klab.com まで。

免責事項

本書に記載された内容は、情報の提供のみを目的としています。したがって、本書を用いた開発、製作、運用は、必ずご自身の責任と判断によって行ってください。これらの情報による開発、製作、運用の結果について、著者はいかなる責任も負いません。

目次

第 1 章

GitHub Actions でカジュアルに OpenAI を利用する

Daisuke Yamaguchi / @_gutio_

1.1 はじめに

GitHub Actions は、リポジトリのイベントに応じて自動化されたワークフローを実行するためのツールです。また、OpenAI の API を利用することで、AI を活用したさまざまなタスクを自動化できます。

本章では、GitHub Actions から OpenAI の API をカジュアルにワンショット利用するために作成した、公開ワークフローを紹介します。

1.2 なぜワンショットアクションを作ったのか

GitHub を使っていると、コードレビューや文章校正のようなちょっと高度な内容を自動化したいと思うことがあります。AI を活用することで、これらのタスクを自動化することができます。AI を活用するとなると、ユーザーとのやり取りやコンテキストの管理など色々と大変そうに思えますが、渡されるテキストに対して固定でやって欲しいことを事前に決めておき、テキストを変えながら実行するだけでも便利に活用することができます。固定でやって欲しい内容を事前に決めておくことを関数の作成と考えて、渡すテキストを関数の引数と考えると、普段のコーディングと同じように AI を活用することができるのではないかと考えました。そのために、チャット API の往復をしないワンショットの形で、GitHub Actions から OpenAI の API を利用するためのアクションを作成しました。

1.3 ワンショットアクションの作り方

GitHub Actions では、JavaScript を利用した自作のアクションを簡単に作成できます。Azure OpenAI Services の API を利用するためには、@azure/openai を使います。Action 内部でやることは非常にシンプルで、Azure OpenAI の API を呼び出して変数に格納するだけです。

ワンショットアクションの作成

自作のアクションを作成するためには、主に2つのファイルが必要です。

1. `action.yml`：アクションのメタデータを定義するファイル
2. `index.js`：アクションの実行内容を定義するファイル

action.yml の作成

メタデータとしては、アクションの名前や説明、入力パラメータなどを定義します。OpenAI の API を利用するための認証情報と、リクエストするテキストを入力パラメータとして受け取ります。

入力パラメータとしては以下のようにしました。

- `endpoint`：Azure OpenAI のエンドポイント URL
- `key`：Azure OpenAI の API キー
- `model`：利用するモデル
- `system_message`：AI 定義するシステムメッセージ
- `user_message`：問いかけるユーザーメッセージ

また、出力パラメータとしては以下のようにしました。

- `result_assistant_message`：AI からの返答
- `result_json_raw`：API からの生の JSON（デバッグなど用途）

そのほかに、実行条件の指定や、実際に実行するスクリプトを指定します。

index.js の作成

実際に OpenAI の API を呼び出すスクリプトを記述します。Azure OpenAI の API を呼び出すためには、 @azure/openai を使います。action.yml で定義した入力パラメータを

OpenAI へリクエストできるように整形して、Azure OpenAI の API を呼び出します。その結果を GitHub Actions の setOutput 関数を利用して保存して返します。

1.4 利用例

実際にワンショットアクション（ai1shot）を使った例を 2 つ紹介します。

利用例 1：LGTM コメントの自動生成

このワンショットアクションを使うと、PR タイトルを元にした LGTM コメントの自動生成が簡単に作れます。PR に「LGTM」とコメントすると、Bot が PR タイトルを元に LGTM メッセージを生成し、コメントします。

ワークフローの設定

リスト 1.1 のように LgtmMessage.yml ファイルを作成し、 .github/workflows ディレクトリに配置します。

リスト 1.1: LgtmMessage.yml

```
name: LgtmMessage

on:
  issue_comment:
    types:
      - created

jobs:
  run:
    if: (github.event.issue.pull_request != null) && github.event.comment.body =→
= 'LGTM'
    runs-on: ubuntu-latest

    steps:
      - name: Run Ai1Shot
        id: run_ai1shot
        uses: gutio/ai1shot@v1.0.0
        with:
          endpoint: ${{ vars.ENDPOINT_URL }}
          key: ${{ secrets.API_KEY }}
          model: "gpt-4o"
          system_message: "あなたはコードレビュワーです。つぎのタイトルのPullRequestに→
対してLGTMに一言添えるいいメッセージを考えてください。"
          user_message: ${{ github.event.issue.title }}
```

```
- name: LGTM PR Comment
  uses: actions/github-script@v6
  env:
    PR_COMMENT: ${{ steps.run_ai1shot.outputs.result_assistant_message }}
  with:
    github-token: ${{secrets.GITHUB_TOKEN}}
    script: |
      github.rest.issues.createComment({
        issue_number: context.issue.number,
        owner: context.repo.owner,
        repo: context.repo.repo,
        body: process.env.PR_COMMENT
      })
```

実行手順

1. PR に「LGTM」とコメントします。
2. Bot が PR タイトルを元に LGTM メッセージを生成し、コメントします。

実際に使ってみた例としてはこのような形になります。

図 1.1: LGTM 利用例

何度か試してみましたが、思いのほか、PR タイトルを元にした面白い LGTM メッセージが生成されていて、適当に作られたメッセージなのに自己肯定感が高まりました。

利用例 2：校正提案

このワンショットアクションを使うと、文章の表記ゆれや Typo チェックを自動で行うことができます。私は、技術広報の業務も担当しており、ブログの typo など文章校正の手間を少しでも減らしたいと思っていたので、この機能を作りました。

ワークフローの設定

LGTM の際と基本的には同様で、リスト 1.2 のように ProofReading.yml ファイルを作成し、 .github/workflows ディレクトリに配置します。その際、テキストファイルの中身を変数に読み込むために 1 つ工夫があります。actions/github-script というアクションは、GitHub Actions の中で Javascript を利用して GitHub API を簡単に利用できるアクションです。また、GitHub API を使わなくても、ただの Javascript を簡単に動かす方法としても使えます。このアクションを使って、テキストファイルの中身全体を簡単に変数に読み込むことができます。

リスト 1.2: ProofReading.yml

```
# AiProofReading: Ai校正のアクションを実行するワークフロー
# このワークフローは、PullRequestのコメントに「校正提案」と書かれたときに実行されます。
name: AiProofReading

on:
  issue_comment:
    types:
      - created

jobs:
  run:
    # 実行トリガのコメントか判定する
    if: (github.event.issue.pull_request != null) && (github.event.comment.body →
== '校正提案')
    runs-on: ubuntu-latest

    steps:
      # Issue 番号からブランチ名を取得する
      - name: Set Branch
        uses: actions/github-script@v6
        id: set-target-branch
        with:
```

```
          github-token: ${{secrets.GITHUB_TOKEN}}
          result-encoding: string
          script: |
            const pull_request = await github.rest.pulls.get({
              owner: context.repo.owner,
              repo: context.repo.repo,
              pull_number: context.issue.number
            })
            return pull_request.data.head.ref

      # ブランチをチェックアウトする
      - name: Checkout contents
        uses: actions/checkout@v3
        with:
          ref: ${{ steps.set-target-branch.outputs.result }}

      # ブランチ名を整形する
      - name: Extract branch name (not pr)
        env:
          REF: ${{ steps.set-target-branch.outputs.result }}
        shell: bash
        run: |
          echo "BRANCH_NAME=$(echo ${REF})"
          echo "BRANCH_NAME=$(echo ${REF})" >> $GITHUB_ENV

      # README.mdの内容を読み出す
      - name: Get text
        uses: actions/github-script@v6
        id: get_text
        with:
          result-encoding: string
          script: |
            try {
              const fs = require('fs')
              const prOwnerUser = context.payload.issue.user.login
              const text = fs.readFileSync(`${process.env.BRANCH_NAME}/README.md→
`, 'utf8')
              core.setOutput("text", text);
            } catch(err) {
              core.error("Error file reading")
              core.setFailed(err)
            }

      # Ai1Shotを使って校正提案のアクションを実行する
      - name: RunAi1Shot
        id: run_ai1shot
        uses: gutio/ai1shot@v1.0.0
        with:
          endpoint: ${{ vars.ENDPOINT_URL }}
```

```
          key: ${{ secrets.API_KEY }}
          model: "gpt-4o"
          system_message: "あなたは広報記事の校正担当です。文章の表記ゆれやTypoチェック→
してください。校正箇所の列挙をしてください。"
          user_message: ${{ steps.get_text.outputs.text }}

      # 校正提案をPRコメントで入れる
      - name: ProofReading PR Comment
        uses: actions/github-script@v6
        env:
          PR_COMMENT: ${{ steps.run_ai1shot.outputs.result_assistant_message }}
        with:
          github-token: ${{secrets.GITHUB_TOKEN}}
          script: |
            // トリガーとなったコメントのユーザーを取得する
            const commentUser = context.payload.comment.user.login
            github.rest.issues.createComment({
              issue_number: context.issue.number,
              owner: context.repo.owner,
              repo: context.repo.repo,
              body: `@${commentUser}\n` + process.env.PR_COMMENT
            })
```

実行手順

1. ブログ記事の文章の PR に「校正提案」とコメントします。
2. Bot が校正提案内容をコメントします。

実際に使ってみた例としてはこのような形になります。

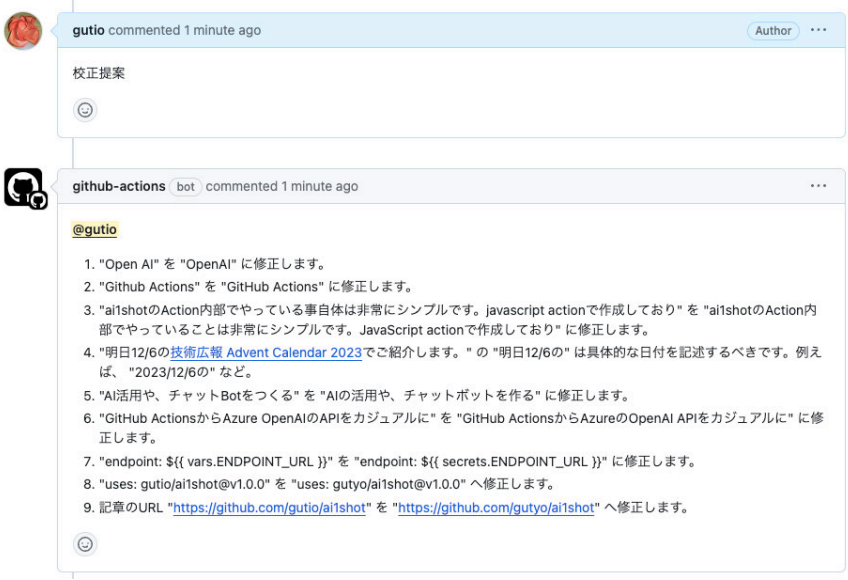

図 1.2: 校正提案利用例

Typo チェックや文体の統一など、文章校正の手間を少しは減らすことができました。

1.5 まとめ

この記事では、GitHub Actions から Azure OpenAI の API をカジュアルにワンショット利用するアクションを作成し、利用例を紹介しました。AI のチャット活用は、会話の往復が必要で敷居が高いと感じるかもしれませんが、このような簡単な使い方もあるということをご紹介しました。GitHub Actions から OpenAI の API をカジュアルにワンショット利用するアクションを使って、さまざまな用途で活用してみてください。

第 2 章

mypy 再訪

Shunsuke Ito / @fgshun

2.1 はじめに

2024 年現在。 PEP 484 - Type Hints[*1] から 10 年が経ち、typing モジュール[*2]や mypy[*3] の発展もますます進んでいます。ここらで Python 3.12 、 mypy 1.12 を用いて静的型チェックに再入門してみようと思い立ちました。得た知識を書き連ねていきます。

2.2 mypy とは

Python では変数には型はなく、コードを実行してはじめて想定外の型のオブジェクトを扱ったことによるエラーが発生します。実行前やコードを書いている最中にこのようなエラーの存在を知ることができる言語ではありません。なんとかならないものでしょうか。 mypy がこれを可能としてくれます。

mypy とは Python の静的型チェッカーです。既存のコードに対してもリテラルや組み込み型、標準ライブラリだけで書かれていれば、誤った型のオブジェクトを扱っている箇所を検出します。しかし、ダックタイピング的な、あいまいなコードでは動作する場合もしない場合もあり、動かしてみるまで確認不能な場合があります。こういったケースでも PEP 484 にのっとった型ヒントをつけることで書き手の意思を変数に反映し、意図せぬ型のオブジェクトを操作しようとしていることを事前に検知することができます。

[*1] PEP484: `https://peps.python.org/pep-0484/`

[*2] typing: `https://docs.python.org/ja/3.12/library/typing.html`

[*3] mypy: `https://mypy-lang.org/`

2.3 とりあえず動かす

mypy を使って誤ったコードのエラーを検出してみることにします。 mypy は他のサードパーティ製ライブラリ同様に PyPI に登録されており、インストールは簡単です。 pip を用いる場合は `pip install mypy` でインストールすることができます。その後、誤ったコード リスト 2.1 を用意します。

リスト 2.1: add_int_and_str.py

```
a = 1 + '2'  # int と str の加算。実行しても TypeError となる
```

このコードに対して リスト 2.2 のように mypy を実行することで、誤った加算を検知してくれます。

リスト 2.2: mypy の実行

```
$ mypy a.py
a.py:1: error: Unsupported operand types for + ("int" and "str")  [operator]
Found 1 error in 1 file (checked 1 source file)
```

2.4 型ヒントが必要となるとき

リスト 2.2 では確認対象がリテラルを用いた単純なコードであったため、検知に成功しました。いっぽう、 リスト 2.3 では `int_add` 関数自体は Python としてあり得ないコードではないため mypy はエラーとして報告してはきません。

リスト 2.3: mypy ではエラーを検知できない例

```
def int_add(a, b):
    """整数同士を足す"""
    return a + b

int_add(1, '2')
```

「この関数では int の加算をしたいのだ」という意図を mypy に伝えるには、関数 int_add の引数たちが int 型であることを示しておく必要があります。次のように記述します。

リスト 2.4: mypy ではエラーを検知できるようにするため型ヒントを追加

```
def int_add(a: int, b: int):
    """整数同士を足す"""
    return a + b

int_add(1, '2')
```

こうすることで、 int_add 関数の引数 b が int 型であり、ここに互換性のない str 型のオブジェクトを渡すのが誤りであることを mypy が検知できるようになります。

リスト 2.5: 型ヒントに反する関数呼び出しの検知

```
$ mypy a.py
a.py:5: error: Argument 2 to "int_add" has incompatible type "str"; expected "in→
t"  [arg-type]
Found 1 error in 1 file (checked 1 source file)
```

2.5 型ヒントの付け方

関数の引数に型ヒントをつける方法は リスト 2.4 にて示しました。引数以外にも戻り値、ローカル変数らに型ヒントをつけることが可能です。 リスト 2.6 のようにします。

リスト 2.6: 型ヒントの一例

```
a: int = 7

def concat(x: str, y: str) -> str:
    z: str = x + y
    return z
```

mypy は型ヒントだけをみるのではなく、推論も行います。このため、 リスト 2.7 の変数 z にはヒントをつけずとも str と認識してくれます。

リスト 2.7: 型推論してくれている例

```
def concat1(x: str, y: str) -> str:
    z = x + y
    return z + 1  # + 演算で error
def concat2(x: str, y: str) -> int:
    z = x + y
    return z  # 戻り値のヒントとあっていないため error
```

複数種の型を受け取るケースに対応するには、ユニオン型[*4]を用います。これは | で型を組み合わせることで作成します。

リスト 2.8: ユニオン型

```
def my_len0(s: str | bytes) -> int:
    return len(s)
```

なお、 mypy は Union 型に対する型チェックや早期リターンをある程度認識し、型を絞り込む方向での推論を行ってくれます[*5]。

リスト 2.9: Type narrowing

```
def my_len1(s: str | None) -> int:
    if s is None:
        return -1
    # 以後、 s の型は str と認識される
    return len(s)

def repeat(s: str | int) -> str:
    if isinstance(s, str):
        # s は str
        return s * 2
    else:
        # s は int
        return str(s) * 2
```

[*4] https://mypy.readthedocs.io/en/stable/kinds_of_types.html#alternative-union-syntax
[*5] https://mypy.readthedocs.io/en/stable/type_narrowing.html

Union 型は順序の認識を行っていません。あるコードにて `A | B` と記述し、別のコードに `B | A` と記述してもこれらは同じ型とみなされます。表記ゆれを心配する必要はありません。

リスト 2.10: Union は順序が異なっても同じ型扱い

```
def use_repeat(s: int | str) -> None:
    repeat(s)  # OK
```

何度も同じ記述をくりかえす必要がある場合、コードが長くなりがちです。その場合は別名をつけておくことができます。 `type C = A | B` のように記述します[*6][*7]。こうしてできた `C` は型ヒントとして使用することが可能となります[*8]。

リスト 2.11: 別名をつける

```
type Basestring = str | bytes
def use_my_len0(s: Basestring) -> int:
    return my_len0(s)  # OK
```

[*6] Python 3.12 からの記法です。3.11 以前では C = A | B と単なる代入文として記述します

[*7] https://docs.python.org/ja/3.12/reference/simple_stmts.html#the-type-statement

[*8] https://mypy.readthedocs.io/en/stable/kinds_of_types.html#type-aliases

2.6 自作クラスの使用

ここまで、組み込み型およびそれを組み合わせたユニオン型の説明をしてきました。型ヒントで用いることができるのはこれだけにとどまりません。自作のクラスもまた用いることができます。

リスト 2.12: 自作クラスの例 - Vector2

```
from __future__ import annotations
class Vector2:
  def __init__(self, x: float, y: float) -> None:
    self.x: float = x
    self.y: float = y
  def magnitude(self) -> float:
    return (self.x ** 2 + self.y ** 2) ** 0.5
  def __add__(self, other: Vector2) -> Vector2:
    return self.__class__(self.x + other.x, self.y + other.y)

def calc_distance(a: Vector2, b: Vector2) -> float:
  return ((a.x - b.x) ** 2 + (a.y - b.y) ** 2) ** 0.5

distance: float
distance = calc_distance(Vector2(0., 0.), Vector2(1., 1.))  # OK
distance = calc_distance(0, 1)  # Error
```

自作クラスのサブクラスもまた、型ヒントとして用いることができます。そしてとある型を要求するところにそのサブクラスを入れることは可能です。

リスト 2.13: サブクラス

```
class A: ...
class B(A): ...

x: B = B()  # OK
y: A = B()  # OK
z: B = A()  # Error
```

2.7 ダックタイピング

クラス名で型ヒントをつけることで、そのクラスのインスタンスを期待する箇所であることを示すことができます。しかし、既存の Python コードには「とある属性・メソッドを持っていること」をもって同質のものとみなす、いわゆるダックタイピングによるものが存在しています。 Python は「とある属性・メソッドを持っていること」を表現するためにプロトコル[*9]と呼ばれる仕組みを用意しました。型ヒントにはクラス名だけでなくプロトコルを記述することができます。そして mypy はプロトコルを認識し、渡された型がプロトコルに即したものであるかを確認してくれるのです。このことにより、自作クラスを関連する型のサブクラスとする必要がなくなっています。

リスト 2.14 は `Vector` という「大きさを計算できるというメソッド magnitude を持つ」というプロトコルを定義しています。そして、これで型ヒントをつけたところに同シグネチャのメソッドを持つクラス、 リスト 2.12 の `Vector2` クラスを入れることが可能であることを示しています。 Vector と Vector2 は親子関係にあるわけではないにも関わらず、です。

リスト 2.14: ダックタイピング

```
from typing import Protocol
class Vector(Protocol):
  def magnitude(self) -> float: ...

def calc_magnitude(vector: Vector) -> float:
  return vector.magnitude()

m: float = calc_magnitude(Vector2(1., 1.))  # OK
calc_magnitude(0)  # Error
```

mypy は `collections.abc` のプロトコルも同様に理解します。 リスト 2.15 の `EmptyBox` クラスは `Sized` のサブクラスではありませんが、 `Sized` プロトコルを要求するところに入れることが可能です。 `Sized` プロトコルが要求する `__len__` メソットを持っているためです。

[*9] https://mypy.readthedocs.io/en/stable/protocols.html

リスト 2.15: 組み込みの抽象基底クラスとダックタイピング

```
from collections.abc import Sized

class EmptyBox:
  def __len__(self) -> int:
    return 0

def length(collection: Sized) -> int:
  return len(collection)

result = length(EmptyBox())  # OK
```

■コラム: ... とはなにか？ - Ellipsis

リスト 2.13 や リスト 2.14 で登場した ... とはなんでしょう？ これは Ellipsis オブジェクトをさすリテラルです。つまり単独の ... は作用がない文の一種にすぎません。

Ellipsis は具体的な実装が不要なところに省略を意図しておかれます。用途のひとつがプロトコルです。プロトコルを示すためのクラスのメソッドには具体的な処理内容が必要ありません。メソッドが「存在するかどうか」にしか着目していないからです。プロトコルのメソッドは名前と型ヒントがあるだけで役割を果たせるのです。そのため、プロトコルのメソッドには ... が置かれるのが通例です。 ... は人が読むと省略記号に、Python インタプリタが読むと作用がない文に見えるというわけです。

2.8 ジェネリック型

Python のコンテナには格納したオブジェクトの型に関する情報がないため、mypy ではエラー検出ができない場合があります。こういった事態に対応するために内容物の型を記述するためのジェネリック型[*10]が存在します。そして組み込み型のコンテナたちはジェネリック型を示すための添字記法に対応しています。

リスト 2.16 は変数 values が list であり、その内容物が int であるというコードです。

[*10] https://mypy.readthedocs.io/en/stable/generics.html

リスト 2.16: int の list

```
values: list[int] = [0, 1, 2]

values = [0, 1, 2, 3]  # OK
values = 'abc'  # Error
i: int = values[0]  # OK
s: str = values[0]  # Error
for value in values:
  print(value + 1)  # OK
  print(value + 'spam')  # Error
values.append(4)  # OK
values.append('spam')  # Error
```

ジェネリック型を扱う関数は リスト 2.17 のように記述します。関数名の後ろの `[T]` は `get_first` 関数が 1 種類の型 `T` を扱うということを示すパラメータリストです。そして引数は `T` の list であり、戻り値は `T` です。これにより mypy は「`get_first` に int の list である `values` が渡された時には、戻り値が int になること」を認識します。

リスト 2.17: ジェネリック型を扱う関数

```
def get_first[T](li: list[T]) -> T:
  return li[0]

values: list[int] = [0, 1, 2]
i: int = get_first(values)  # OK
j: str = get_first(values)  # Error
```

ジェネリック型を扱うクラスは リスト 2.18 のように記述します。 クラス名の後ろの `[KT, VT]` は `MyDict` クラスが 2 種類の型 `KT` と `VT` を扱うということを示すパラメータリストです。これにより mypy は、「変数 `d` は int と str の対を記録する MyDict であり、 `d` の `get_value` メソッドの引数が int であり戻り値が str であること」を認識します。

リスト 2.18: ジェネリック型を扱うクラス

```
class MyDict[KT, VT]:
  def __init__(self, keys: list[KT], values: list[VT]) -> None:
    self.keys = keys
    self.values = values
  def get_value(self, key: KT) -> VT:
    for i, k in enumerate(self.keys):
      if k == key:
        return self.values[i]
    raise KeyError

d: MyDict[int, str] = MyDict([0, 1, 2], ['one', 'two', 'three'])
s: str = d.get_value(0)  # OK
t: str = d.get_value('spam')  # Error
i: int = d.get_value(0)  # Error
```

2.9 おわりに

最初期の Python の型ヒントと mypy ではコンテナに型ヒントをつける際には `typing` モジュールの import が必要であったり、プロトコルのサポートがなかったり、ジェネリックの記法が回りくどかったりしたものでしたが、 Python 3.12 ・ mypy 1.12 ではそんな問題が解消しています。ぜひ、使ってみてください。型ヒントがついたコードは読んでよし、 mypy に静的型チェックをさせてよしです。

第 3 章

2D Shader 試験場

Norimasa Shibata

3.1　はじめに

「2D アーティストは嘘をつく」

おいおい、何を言っているんだ急に？ と思われる方もいるかもしれませんが、作品で表現する時は、嘘をついて表現したり、品質を上げるための工夫がたくさん必要です。1 枚の 2D キャンバスに対して、立体的に描く為に、色々な遠近法を用いたり、陰影を使って立体感を出したりしています。また、「補色」「色彩心理」などを用いて「誇張表現」や「視線誘導」等、伝えたい情報を入れ込んでデザインしています。

ここで大事なのは、正確な空間描画や伝えたい情報を正確に表現することでしょうか？ 間違っているわけではなく、それも非常に重要な点であると私は思います。しかし、個人的にはもっと大事なことがあります。

それは【らしさ】という点になります。

先ほど述べたように、正確に表現するために色々な技法を使用しますが、逆に『崩す』ことをしたりもします。「デフォルメ」「漫画パース」「モノトーン化」等の『誇張表現』になります。ありふれている表現ですが、間違えるとバランスが悪く、本来伝えたい点より、違和感が目立ってしまいます。ここで、大事になってくるのが、「違和感なく落とし込む」という点です。『崩す』度合いをコントロールして、【らしさ】の範囲に落としてこんでいくのです。

今回は、スマホゲームでの負荷軽減に使えそうなものとして、3D で表現する「揺らめく旗」を、2D で製作してみました。

なぜ 2D で？ といった疑問に対して。「2D で立体的なアニメーション」「モバイル端末でも実装可能な形で」というお題目。これだけの表現であれば、実は全然新しい機能なんて要りません！

- 「動画」
- 「テクスチャの連番再生」
- 「3D アニメーション」

いくらでも手法はあると思います。

ただし、ここで問題になってくるのが、ゲーム開発で発生するトラブルです。「容量」「3D/2D 環境の影響」「メモリ負荷」等の様々なトラブルが出てきます。色々な角度からの手段を持つことで、トラブルを回避しつつ、やりたい表現を実現するために試作しています。

現状、自分に「3D」側での行うには知識が乏しいので、という理由もありましたが、未来の自分に期待しておきます。

3.2 作戦

「DCC ツール[*1]でやっている事を、ShaderGraph で再現し、2D 変形と陰影を制御すれば、3D 表現出来るのではないか？」基本的にはこの考えで行けるかなと考えました。

というのも、ShaderGraph でこれまで色々と遊んで触ってきた事もあり、次のような感覚がありました。

- 表現自体は DCC ツール側の機能にあるので、理解して ShaderGraph で再現できれば、作れるイメージがある
- 「グレースケールのイメージ」を用いて変形などを行う手法であれば、ShaderGraph の UV マップを使うことで再現可能だと思った
- デザインでも、プロシージャル[*2]な作り方は好きだったので、特に拒否反応も無く取り組めそう

[*1] Digital Content Creation の略で、ゲームや CG などのデジタルコンテンツを作るために使用するソフトの総称

[*2] 手続き型。一定のルールに基づいて段階的に制作したり、数式や処理を組み合わせ、順番に適用していくこと。デザインで使う場合は、色々な展開や差分を作りやすい手法となります

3.3 実装

内容に入る前に、制作した「ShaderGraph」の全体図を共有しておきます。

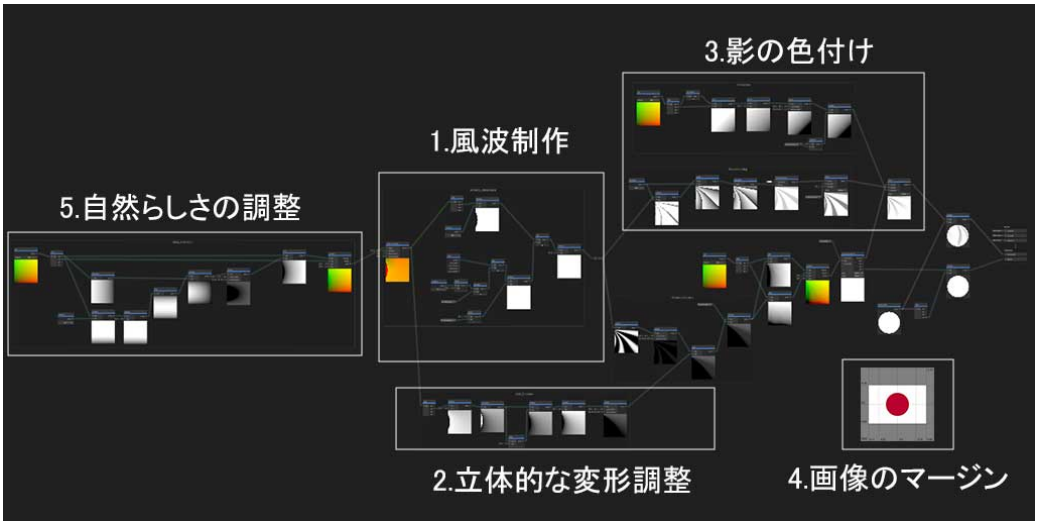

図 3.1: ShaderGraph の全体図

複雑な作りになっていますが、2D 視点での説明になるので、細かなノード調整等は、省いています。

1. 風波らしさ

まず初めに、主となる風波の形状を作ってみることにしました。波自体は、UV マップの X に対して、Sine を演算することで、横向きの波を作り、後は角度を変えて、UV をスライドさせればと、試してみました。結果としては、波が直線的過ぎて、違和感がありすぎたのでやめました。(図 3.2)

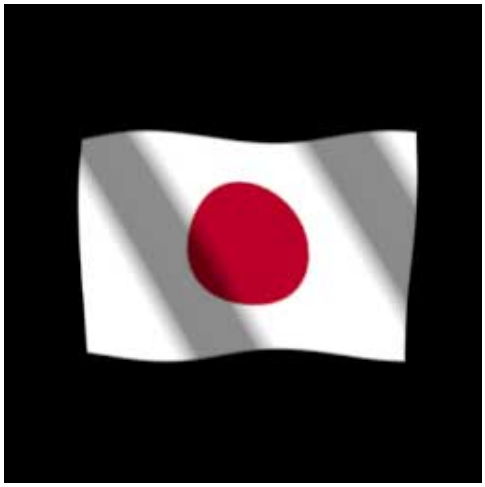

図 3.2: 規則的すぎる風波

影を外すと…アメーバ的な物には使えそうな気もしますが…
ここで、自然らしさを考慮して、繋がれた旗がなびく様を参考にしてみました。

- 力の作用する位置と範囲に差を付ける
- 波が平行直線にならないようにする

そうすれば、もっと納得感のある動きになるのでは？ と考えました。
そこで、力の作用する範囲に差を付ける為、以下の要素を考えてみました。

- 左上：影響範囲（小）
- 右下：影響範囲（大）

この条件を満たす為に用いたのが、【極座標】です。極座標は、（X,Y）＝（横軸, 縦軸）だったものを（X,Y）＝（半径, 角度）置き換えるようなものなのですが、Y の値は螺旋状に円を描いています。こちらの Y に Sine や Cosine を演算することで、0 基点から放射状に波が作られます。（図 3.3）

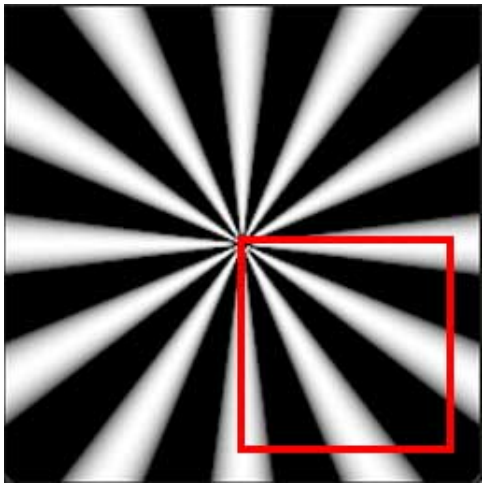

図 3.3: 極座標を利用した風波マップ

図の右下のエリアは、風波のベースとしてはちょうど良さそうなので、基点が左上になるように、UV マップを調整します。波の基礎はいい感じに出来たので、変形に流し込んでみました。ところが、波は流れているのに旗は流れていない、そんな様に見えていました。(図 3.4)

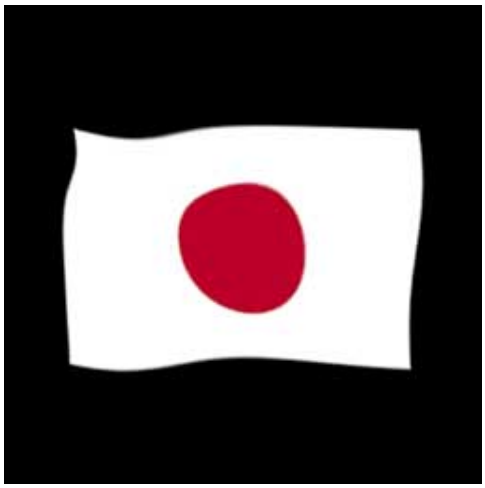

図 3.4: 風波テスト

2. 立体的らしさ

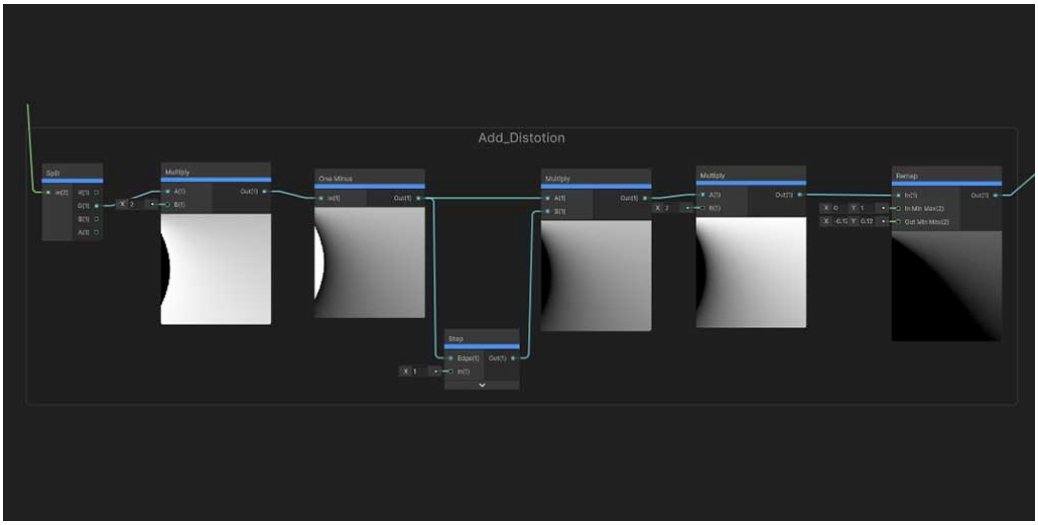

図 3.5: 立体的らしさの ShadreGraph

気を取り直して、違和感を軽減する為に、何か案を考えていました。ベースとなる波は今のところ、そんなに悪くなさそう、、、

そういえば、立体的な表現をまだ取り入れてませんでした。奥行きを取り入れることで、2Dの向きの風にも立体感を表現出来ないかな？ と考えてみました。

2D で奥行きを表現する際に、空間を表現するにあたっては、斜めのパースを使おうと思いました。よく漫画などの飛び蹴り等で勢いのある画がありますが、あれです。

斜めのパースだと、画面内に「手前」「中央」「奥」と立体的に見せる情報が多く入れられるので、奥行きがあるように見せやすいんですよね。

ということで、斜めの軸に回転したような形状、つまり左下が手前、右上が奥側になるような変形を加えることにしました。

左下の数値が小さく、右上が大きいマップ…一から作ろうとも思いましたが、ちょうど良さそうな UV マップがありました。【波】で使用した「極座標」です。極座標の Y のマップを見ると、下を基点に時計回りに数値が増えていくのが見て取れます。こちらの右下部分のエリアを使用していきます。(図 3.6)

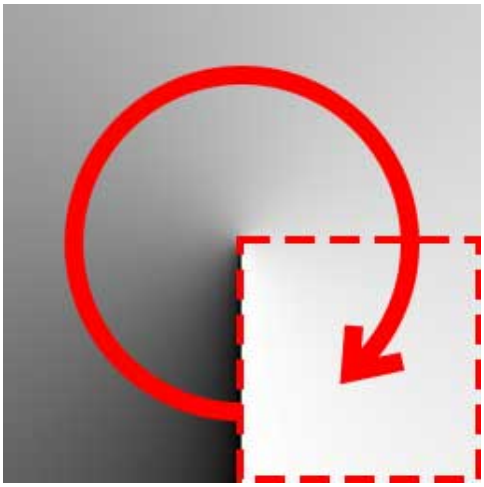

図 3.6: 極座標の Y マップ

※図は、マイナス値を可視化する為に、Y 値に「0.5」を足しています。

使用するエリアは、「0.25~0.5」の値のマップになるので、Remap を使用し数値を調整していきます。(図 3.7)

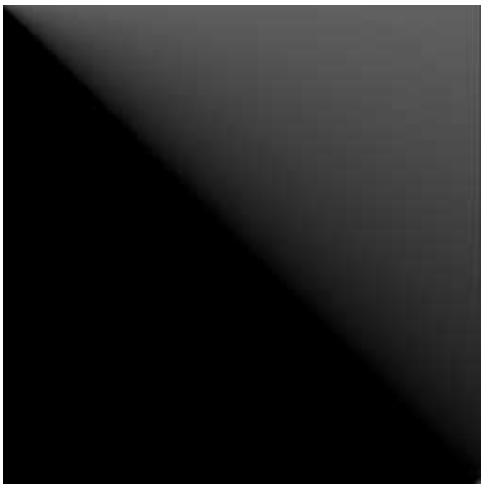

図 3.7: 変形用の±マップ

綺麗な変形にしたわけでもなかったので、ちょうど良いかなと思って試しに使ってみると、良い感じに旗が湾曲しました。(図 3.8)

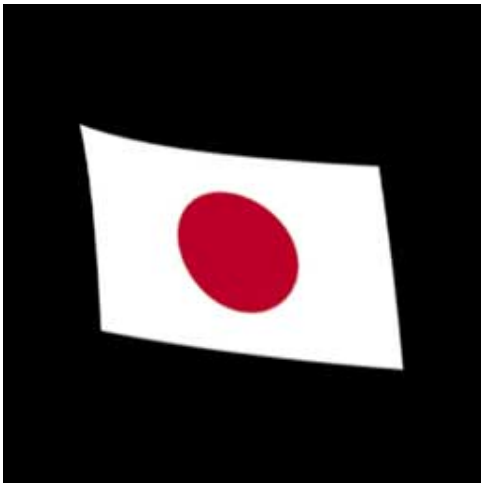

図 3.8: 旗の湾曲

唐突に、感覚的な調整をおこなったのですが、こちらには理由があります。

後述で記載している「自然らしさ」と繋がるのですが、自然の中では規則的なリズムは、殆ど生じないのです。なので、一定のルールに沿っているのであれば、それ以外の部分は、ノイズとして組み込んであげることで、自然らしさに繋がっていきます。

確認の為、波と変形を組みわせて見ると、良い…とても良い感じです。立体的になびいている感じが出てきました。(図 3.9)

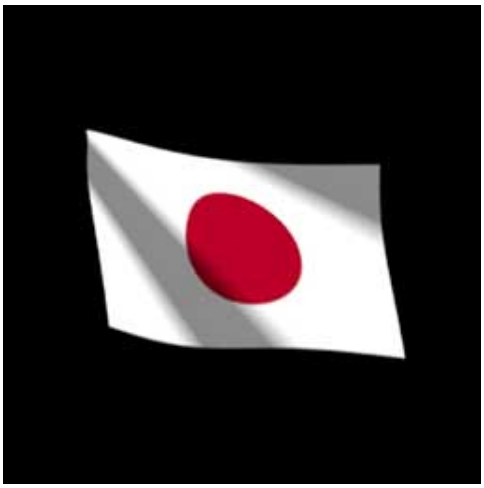

図 3.9: 旗のなびき

3. 影の色付け

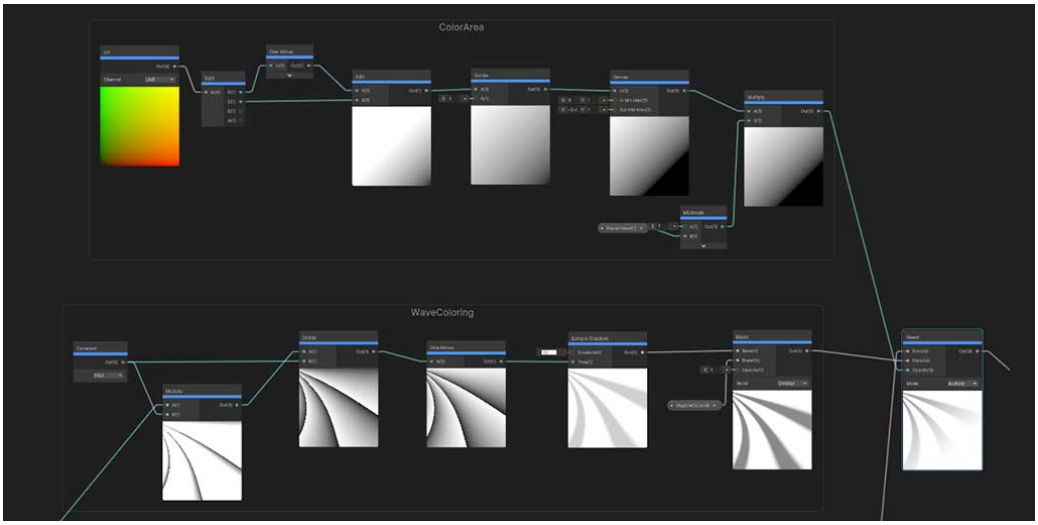

図 3.10: 影の色付けを行う ShadreGraph

初めに、風波を作った波があるので、このグラデーションマップに色を付けたら簡単に仕上がるのでは、と意気揚々に色を付けたら失敗でした。そう、高低差で使用するグラデーションマップでは、登りも下りも色がついてしまい、片側にだけ色を乗せることが出来なかったのです。

そこで、1 波分のグラデーションマップを作っていくことにしました。1 波分の周期…ちょうどいい演算ノードがありますね、「Modulo」演算[*3]です。「Modulo」演算は、「割り算をした際の余り」を求める演算子になります。

1 波は、「Cosine」で作ったので、1 周期は 2 π = TAU になります。

1. 除数「TAU」で、Y 座標の「Modulo」をとったマップ生成
2.「TAU」で除算
3.「One Minus」でマップ反転

これで、「TAU」周期で「0 → 1」に変化していくマップが作れ、変形の波の最底辺からの周期と合わせることが出来ます。

[*3] ShaderGraph 触るまで知りもしませんでした。

波を Cosine で作った理由は、こちらになります。Sine では 1 波の途中から始まるので、1 波の周期にカラーを当て込むのが面倒になるからです。（図 3.11）

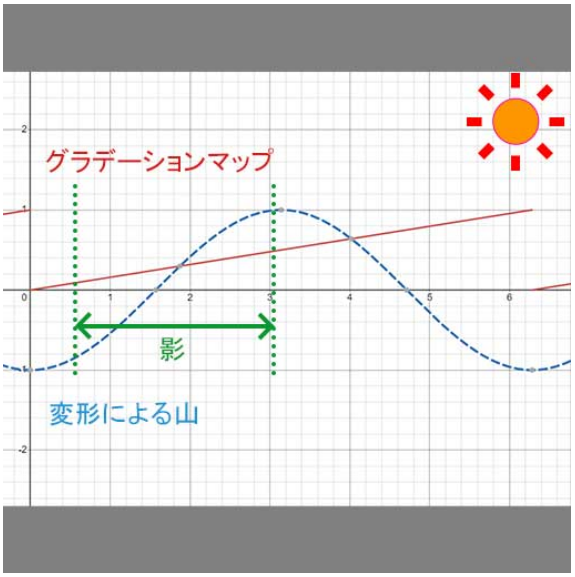

図 3.11: 周期とグラデーションマップのグラフ

あとはこのグラデーションマップにカラーを割り当てることで、波に対して陰影をつけることができました。（図 3.12）

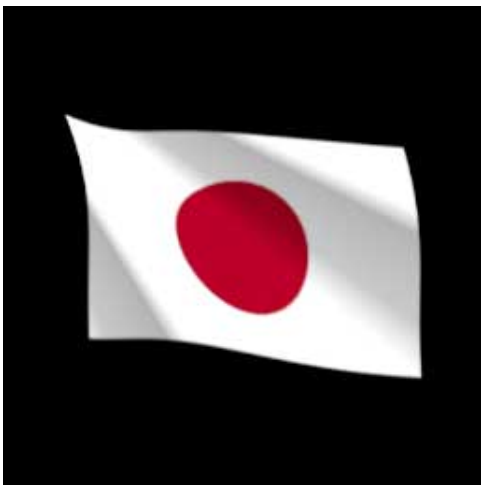

図 3.12: カラーの割り当て

完成です!と思いましたが、陰影が常に強いので、悪目立ちしています。

…そういえば、形状で奥行きをつけたことに対して、カラー側で何も対応をいれていないせいでした。

そこで、陰影に対して、環境光や反射光等の間接光を受けて、影が薄くなるようにしてみます。これは、光によって出来た物体影が、色々な間接光により、影が薄くなっていく現象になります。(図 3.13)

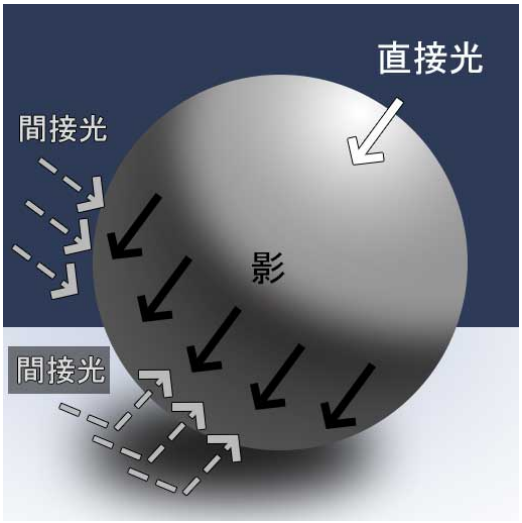

図 3.13: 陰影と間接光

こちらを満たすグラデーションマップを作り、「Blend」ノード[*4]の「Opacity」につなぎます。「Base」には、元となる画像の RGBA 値を繋ぎ、「Blend」にグラデーションマップで影の色を付けたものを繋ぎ、「Mode」は『Multiply』[*5]にします。こちらを行うことで、空間表現を取り入れ、色に立体感らしさを出すことができました。(図 3.14)

[*4] Base の RGBA 値に、Blend を合成するノードになります。Mode は演算の仕方を変更でき、Opacity は反映する領域を調整できます。

[*5] 合成形式としては乗算になります。2D で影を付けたりしたことがある人には、馴染みのある合成形式になるかと思います。

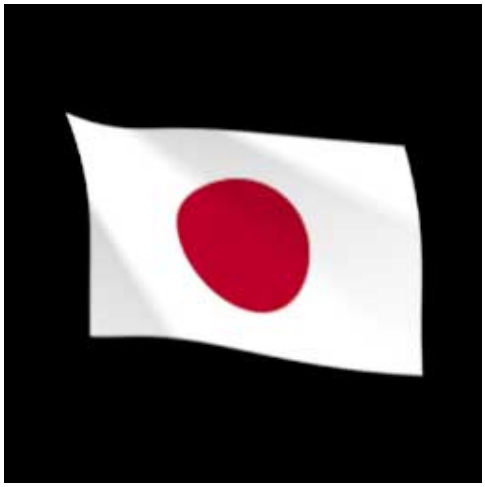

図 3.14: カラーテスト

4.Shader での数値調整と透明領域の関係

「らしさ」とは関係ないですが、説明でかなり端折った部分があります。波や変形等で作ったマップに対して、Remap 等による数値調整です。

実は今回の波打つ変形ですが、画像の周りに透明な空白領域が無いと、成立しません。3D オブジェクトでの変形は用いず、内側の UV マップで作り上げている為になります。

UV マップは、X 軸,Y 軸の「$0 \rightarrow 1$」領域を使って変形させていますので、各軸のマップに対して、数値を加減することで、以下のような効果が得られます。

- X マップ
  - ＋数値：左にずれる
  - －数値：右にずれる
- Y マップ
  - ＋数値：下にずれる
  - －数値：上にずれる

そして、その透明領域と調整に何の関係があるかというと、上下左右のマージンが変化に耐えうる、最大と最小になります。

- 上側マージン：Y の－値上限
- 下側マージン：Y の＋値上限
- 左側マージン：X の＋値上限
- 右側マージン：X の－値上限

UV マップは縦横 0 → 1 のマップになるので、マージンが画像サイズに対してどのくらいの割合かを目安にすると良いです。精密には計算していませんが、検証データだとおおよそ以下の値くらいになります。（図 3.15）

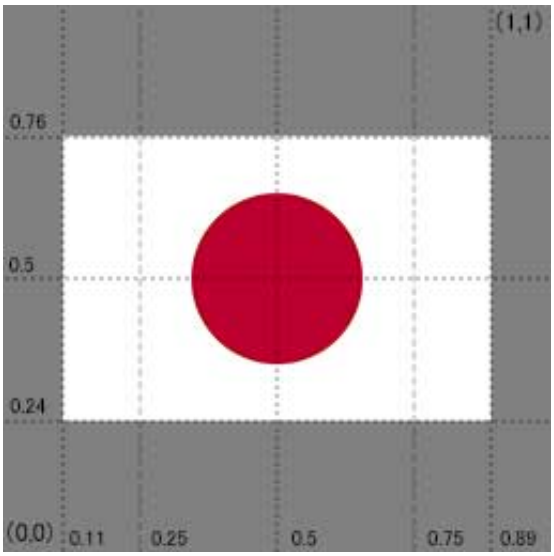

図 3.15: マージン

- 左右のマージン：± 0.11
- 上下のマージン：± 0.24

波や変化の値から必要なマージンを計算し、それ以上のマージンを用意しましょう。

5. 自然らしさ

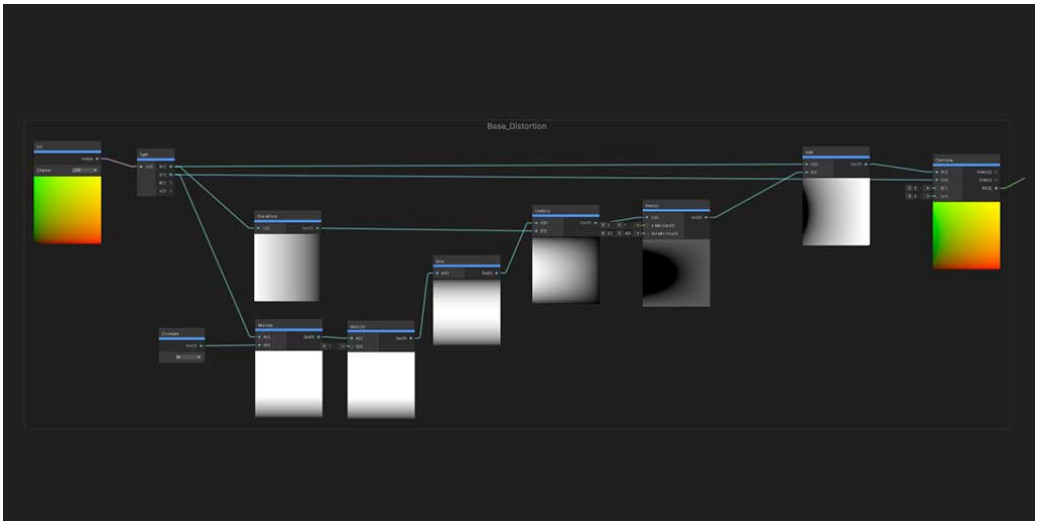

図 3.16: 自然らしさの ShadreGraph

この段階で、終わっても良かったのですが、波のパターンを分かりやすく、もう少し自然に出来ないかと考えました。全体的に動きが固い…全体か…根本の元となっている部分に変形を加えたら、今まで作っていた「波」「変形」にも変化が加えれるよなぁ…と思い、色々な変形ノードを試して試行錯誤してみました。

ShaderGraph の良いところですね。プロシージャルに作っている分、色んな案を試しやすいです。結果としては、今まで作っていた部分への影響は抑えつつ、軽い形状変化を加えることにしました。

どういった変形かというと、風を受けた際の「ふくらみ」感が出せるように、上下部分：影響無し、中央部分：右に寄せる、といった変形になります。(図 3.17)

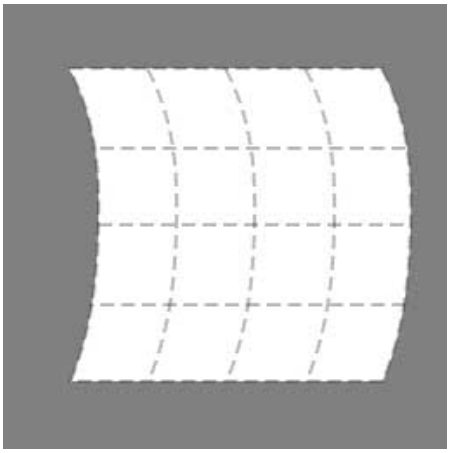

図 3.17: ふくらみ表現の変形イメージ

また、左右で影響の濃淡も加えたかったので、左側：変化（大）、右側：変化（小）、といった点も加えます。

それでは、上記の変形したい形状を、グレースケールで描いて行きます。

やり方は色々ありますが、今回は Sine 波の半周期を用いてみました。新規 UV マップで、

1. Y 値に「PI」を乗算
2. Sine 演算

こうすることで、Sine 波の半周期が、「0~1」の表示区間に収まります。（図 3.18）

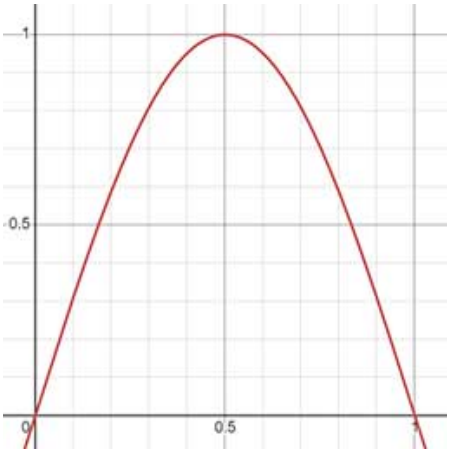

図 3.18: 半周期の山グラフ

変化の濃淡は、X 値に「OneMinus」を行うことで「1 → 0」のマップが出来ます。2 つのマップを乗算することで、左右で影響度の違う歪みを作っています。やっていることは、Photoshop でよく使う、マスクにグラデーションかける感じに似てますね。

出来上がったマップに、「Remap」度合を調整して、完成です。（図 3.19）

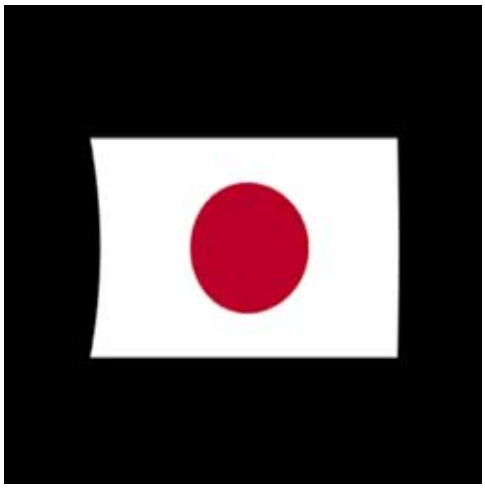

図 3.19: ベース歪みテスト

上図では少し歪んだ程度ですが、こちらはベースにかけている変形になります。ということは、後工程にある波の生成も併せて湾曲しますので、良い感じにしあがります。（図 3.20）

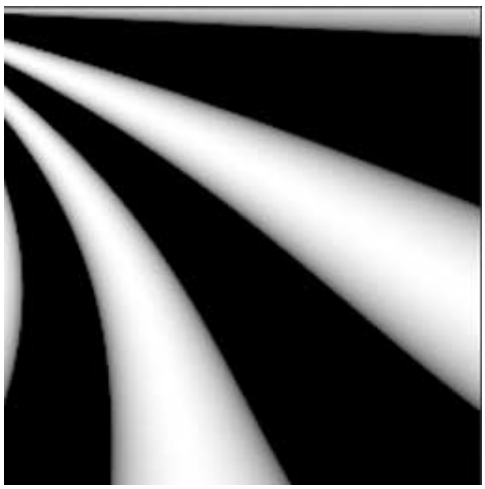

図 3.20: 風波マップの変化

色味などを再調整し、完成です！

図 3.21: 完成画像

3.4 実戦投入に向けて

端末で動作することは確認できましたが、Shader による計算量の数値は高めで、リッチな物になっていました。負荷的な部分は聞き取り中ですが、3D を使用しないシーンや、多量の設置をしなければ、実装自体は出来ると予見しています。また、負荷対策は入れていませんので、問題があれば、以下のような調整を入れていく予定です。

- カラーの着彩：グラデーションマップの代わりに Lerp を用いた簡易グラデーション
- 負荷の高い箇所を画像化：一定のルールで出来上がっている波などを画像化し、回転 + 変形で代用

3.5 考察

変形させる元の画像を加工することにはなりましたが、追加のテクスチャは「0」で仕上げることが出来ました。

読み込み素材が「0」という点での成果は出ましたが、動きをリッチにすると、それなりにShader は複雑になってくるので、負荷の面は気にしておかないとならないかなと思います。ですが、画像「0」で出来た分、負荷面で問題が出たとしても、画像化といった対策がとれるのは良いですね。

計算だけで組み立てるのは大変ですが、応用の利く部分もたくさんあるので、この分野の研究はとても役にたってくると思います。

3.6 さいごに

2D アーティストは 2D 空間で 3D に見せる手法を多く学んでいるので、その点は凄く活かしていけるかなと思いました。

デフォルメも、リアルを理解して、誇張しつつも、そう見えるように調整することで、情報量が少なくても納得感のある世界を表現をしています。

自分は Unity で 2D 演出を行っていることもあり、DCC ツールとゲームエンジン、相互に触れることになります。

そうすると、DCC ツールでは出来るがゲームエンジンでは出来ないということが多々あります。ですが、それを理由に品質を落としたり、クリエイティブで表現出来ないということにはしたくないのです。

ユーザーはそんなこと知らないのだから…

そういう思いもありますが、ユーザーが楽しめる作品が作れるように、日々 ShaderGraph での実験を通じて、自身の技術の引き出しを増やしています。ただ、それだけだと疲れるので、遊びながら作ったりして、制作自体を楽しんで行こうとも思います。

そしてこれからも…

「作品で嘘をついて行こう」

第 4 章

shader graph で水表現

Takeichi Ayato

4.1　動機

筆者は少し前からグラフィック関係の知識・技術を身に付けたいという思いがあり shader graph を触り始めました。今回はその勉強の際に作っているものの一つの水表現について執筆しました。

4.2　shader graph とは

shader graph とは、shader で行うことをコードを書かずにノードと呼ばれる描画処理単位のパーツを組み合わせることで実装する機能です。パーツの接続で構成されているため処理の流れが見たままの通りになっており追いやすいことや、タイポをすることが無いメリットがあります。目で見て把握できる情報が多いので初心者にもおすすめな印象です。

4.3　取り組み内容

準備

今回水を表現するにあたって用意した水面用のオブジェクトは、 Unity に初めから入っている Plane 1 枚です。スクリーンショットを取る際に動きが少しでもわかりやすくなるよう円柱や UV テクスチャを足していますが、以降紹介する shader は全てこの Plane に対する操作を行っています。

水面の揺れ (基礎)

まず初めに水面の揺れを入れました。頂点シェーダー側でメッシュの頂点座標に対して波の向きと高さ、時間経過による動きを計算し頂点位置を移動させています。Vector3 ノードで Y1 のみを使用しているのは頂点の移動方向を縦のみに限定するためです。水面と UV テクスチャの境目が水平になっていないことから、揺れを作ることが出来ていることがわかります。

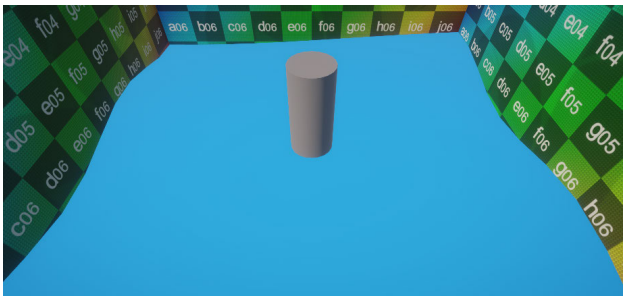

図 4.1: 水面の揺れ (基礎) の見た目

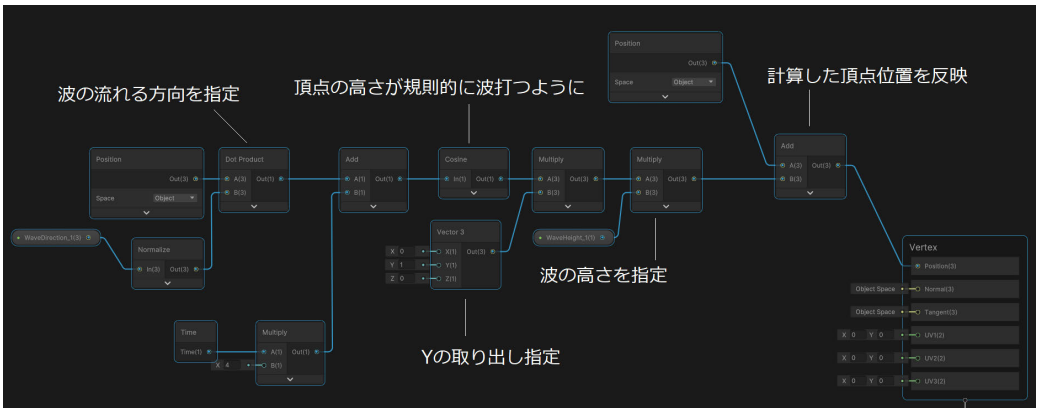

図 4.2: 水面の揺れ (基礎) のノード組み合わせ

水面の揺れ (法線)

次に水面の光の揺れを入れました。先ほど計算した規則的な波打ちの値を法線の x,z 情報として流し込んでいます。法線は光の当たり方を計算するための情報で、Normal(3) となっている繋ぎ先が法線情報として扱われます。これにより移動させた頂点位置と連動して水面に光が当たる絵が見えるようになっています。

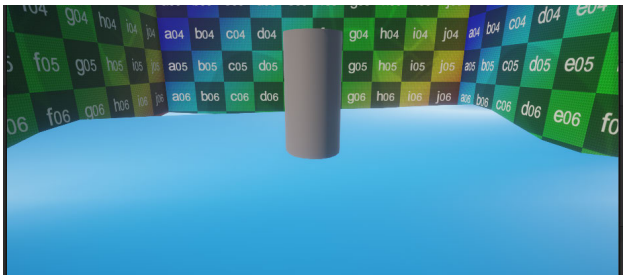

図 4.3: 水面の揺れ (法線) の見た目

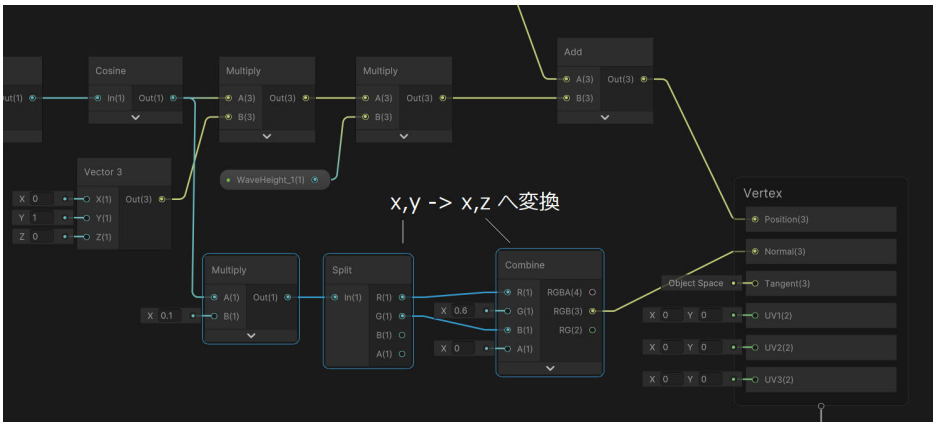

図 4.4: 水面の揺れ (法線) のノード組み合わせ

水面の揺れの合成

先ほどまでの揺れは波の方向を 1 方向に指定しているため単調な波になっています。なので揺れ方向をいくつか増やして複雑にすることを考えました。今回は 3 つに複製しています。複

製したそれぞれの波の向きの指定と高さの指定をバラバラになるように設定し、それぞれの最終出力を全て add ノードで合算することで合成しています。(複製元の先ほどまで紹介していたノードは画像外側の上のところにあります) 前ステップでは画面右手前から左奥へ波打つように見えている波が合成後は複雑になりました。

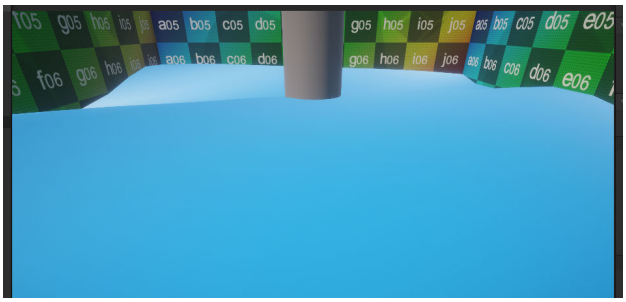

図 4.5: 水面の揺れの合成 の見た目

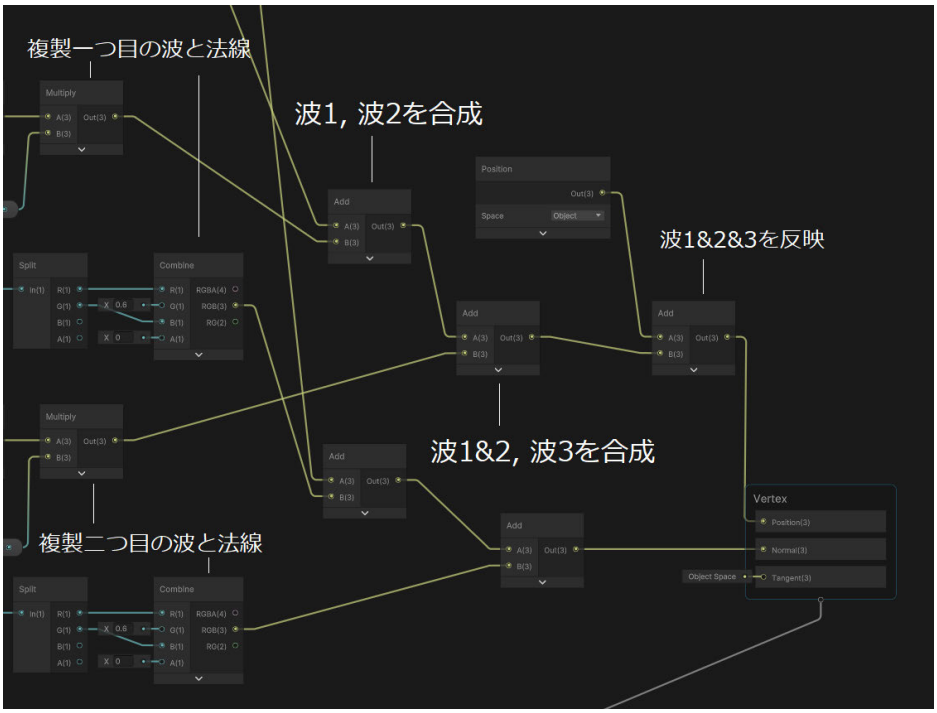

図 4.6: 水面の揺れの合成 のノード組み合わせ

水中の透過

ここまで頂点シェーダー側にて編集していましたが、ここからはフラグメントシェーダー側へ移ります。フラグメントシェーダーでは頂点シェーダーで決めた面に対してピクセル単位で色の計算を指定します。ここでは水中のオブジェクトを透過して水面に映す対応を入れました。Screen Position ノードと Scene Color ノードを接続することで画面に表示しているオブジェクトの向こう側の色が取得できます。先ほど法線情報を使ってライティングを入れているため、うっすら白く色がついていますがそれを無くすと、向こう側の色のみになります。

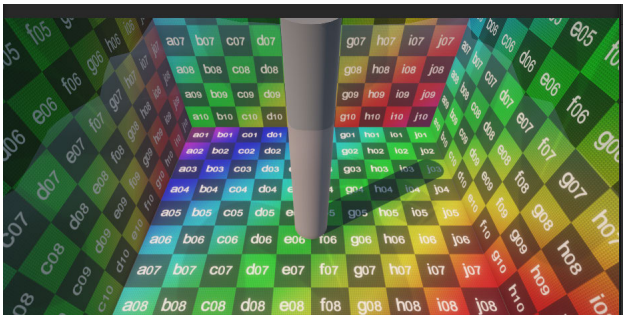

図 4.7: 水中の透過 の見た目

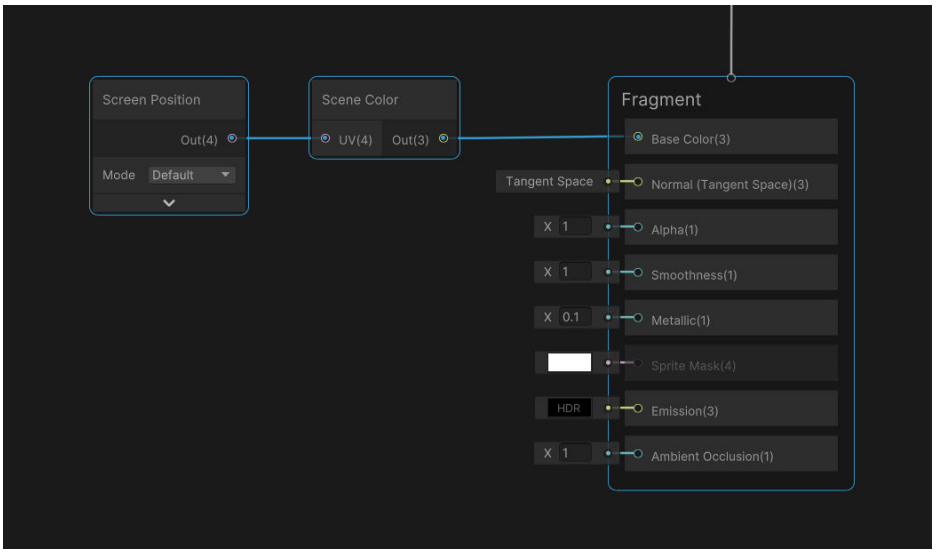

図 4.8: 水中の透過 のノード組み合わせ

水中描画対象の色収差

次に、水中のオブジェクトとして表示している対象に色収差を適用しました。光が水中に入る時に屈折の影響を受ける度合いが波長によって変わることを取り入れています。今回の色収差は専門的な計算は行わず、RGB それぞれの描画位置を水平方向にずらすことで表現しています。

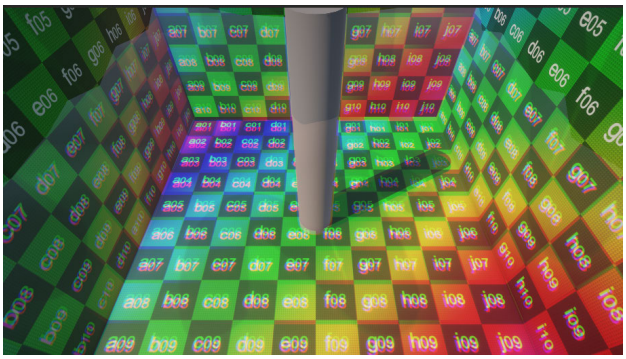

図 4.9: 水中描画対象の色収差 の見た目

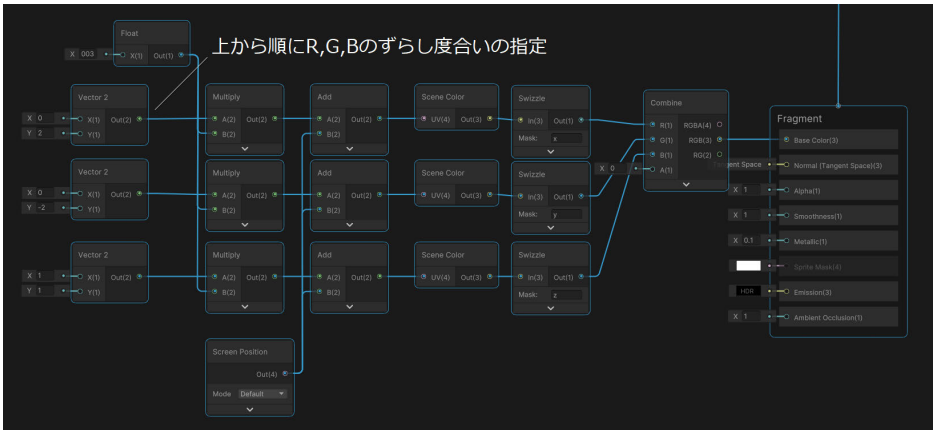

図 4.10: 水中描画対象の色収差 のノード組み合わせ

水の濁り (角度)

色収差の次は水の濁りを取り入れています。まずは角度が影響する濁りの表現から。水の深さと濁り具合にもよりますが、プールなどを思い浮かべると自分の視点から遠い位置は水が濃く見え、近い場所は足元まで見えていた経験があると思います。

先ほどの色収差 (水中の色) の出力先と水の色を指定した Color ノードを Lerp ノードに接続し、角度の影響を受けて水中の色と水の色を切り替えました。角度の影響度合いは Fresnel Effect ノードを使用しています。こちらはメッシュの面がカメラからの視線に対して垂直になるほど出力が高くなるノードです。

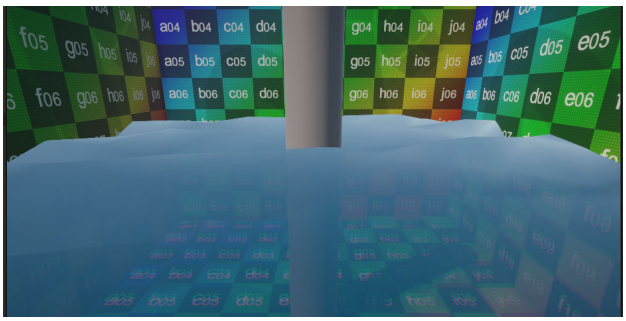

図 4.11: 水の濁り (角度) の見た目

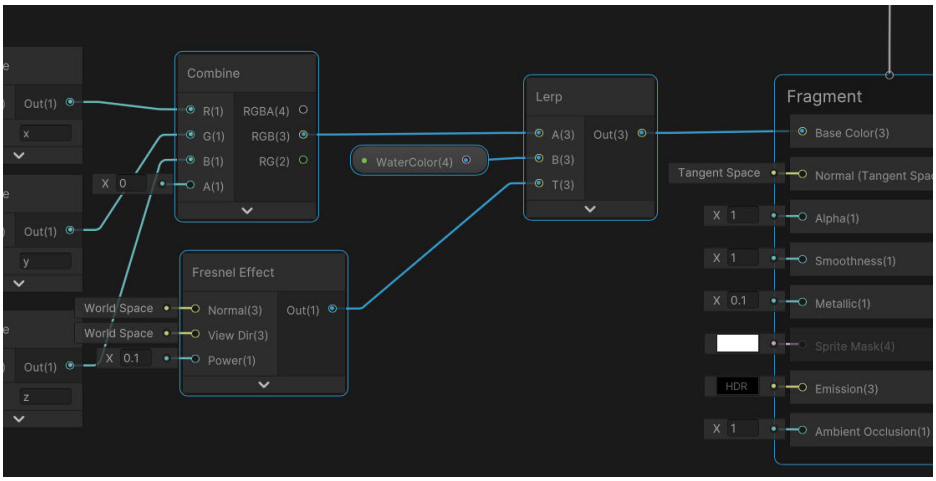

図 4.12: 水の濁り (角度) のノード組み合わせ

水の濁り (深さ)

先ほどの角度だけでは垂直に見下ろした際にどこまでも見えてしまうため、濁りの表現に水深も影響するように対応しました。Screen Depth というノードを使うことで深度バッファの情報を取得することが出来ます。これを元にカメラ目線から見た水面とその奥のオブジェクトの距離を計算し、浅いと判定するほど濁りが薄くなるようにしました。画面の左右端の壁に近い部分が見えていることで動作しているのがわかります。

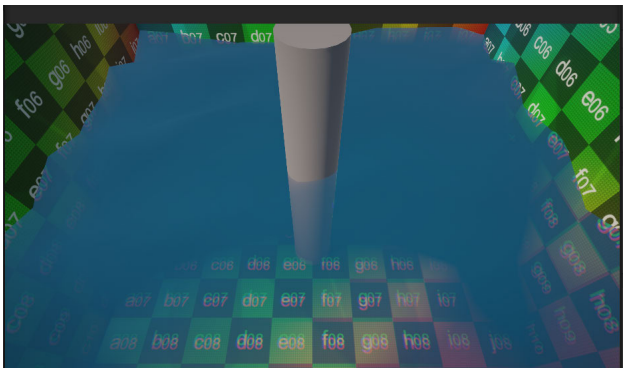

図 4.13: 水の濁り (深さ) の見た目

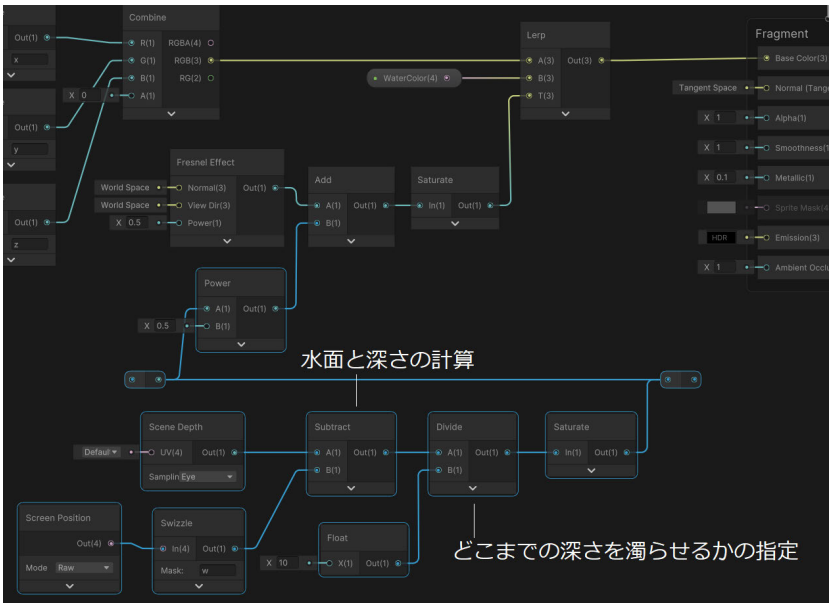

図 4.14: 水の濁り (深さ) のノード組み合わせ

最後の微調整

最後に微調整を行って完成です。撮影用に大きめに設定されていた色々なパラメタを自分が見たい水に近づくように設定し直しました。またどれだけ濁りがあっても最低限見えてほしい透き通り具合のイメージを持っていたため、最終出力の後に Lerp ノードを追加し、水中の色をわずかにブレンドすることで理想の出力に近づけました。自分の見たい出力を見ることが出来たため今回の水 shader graph の勉強は完了です。

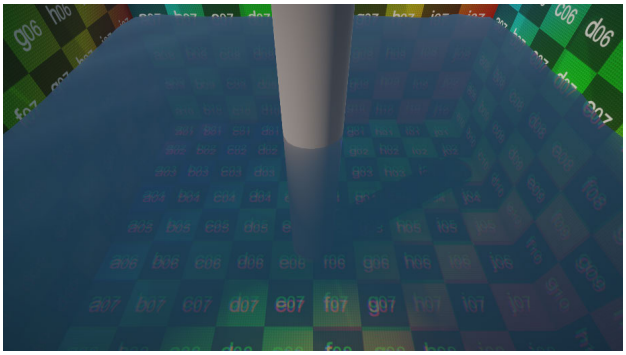

図 4.15: 最後の微調整 の見た目

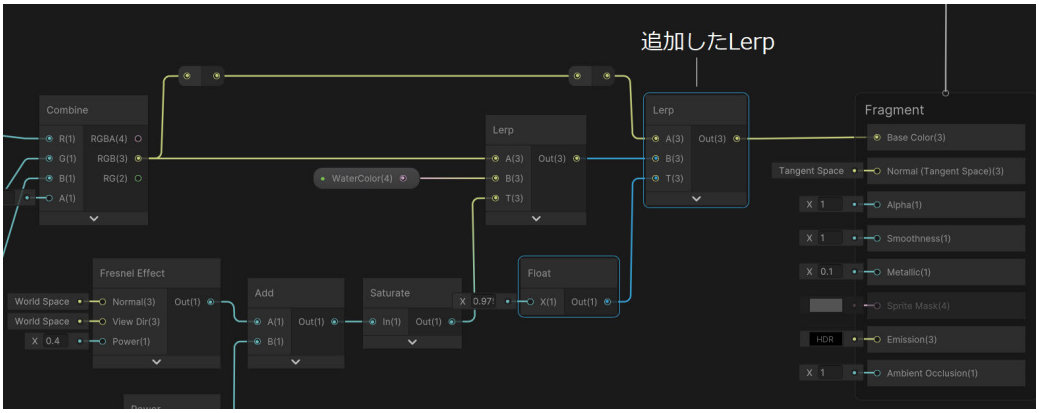

図 4.16: 最後の微調整 のノード組み合わせ

4.4 まとめ

今回 shader ではなく shader graph を選んだ理由は、 shader では覚えておくべき前提知識が多いと感じているからです。また、それらを把握していないと実現したいことのために編集する箇所も判断できないため、直観的に使いやすい shader graph を選びました。ただ、 処理の繰り返しや複数パスが必要になる制御は出来ないため、今後もこれを触りながら理解を深め、どこかの段階で shader にも手を出せたらなと考えています。

水の表現についても、今回は自分の思いついた水に対するイメージだけで実装しましたが、水面付近の泡立ちや水上から見た時と水中から見た時の見え方の違い、水中に投影する光の模様など出来ることはまだまだあります。機会があれば、その他の機能も試してみたいと思います。

第 5 章

XML プログラミング言語「XSLT」の可能性

Daisuke Makiuchi / @makki_d

5.1　ことのはじまり

それは、とある勉強会の懇親会でのことでした。出版業界の方とお話する中で、組版作業で XML と XSLT を利用することがあると聞きました。XSLT は XML を整形するためのテンプレート言語で、それ自体も XML で記述します。さらに驚くべきことに、XSLT はテンプレート言語でありながらチューリング完全だといいます。つまり、XSLT は **XML の XML による XML のためのプログラミング言語**なのです。

ところで筆者は以前 KLabTechBook Vol.1 にて、M4 というマクロ言語がチューリング完全であることを示しました[*1]。そうです、またなのです。確かめたくなってしまったのです。

今回も同様に、XSLT で Brainfuck のインタプリタを実装することで、XSLT がチューリング完全であることを確認したいと思います。

5.2　XSLT とは

XSLT（Extensible Stylesheet Language Transformations）は、主に XML を整形・変換するためのテンプレート言語です。XML 文書を受け取り、それを他の形式（HTML、テキスト、あるいは別の XML）に変換することができます。

[*1] テキストマクロプロセッサ「M4」のチューリング完全性について
https://makiuchi-d.github.io/2023/05/22/klabtechbook1-bfm4.ja.html

次の例は XSLT で XML 文書を表示用 HTML に変換するものです。リスト 5.1 は図書館の蔵書を表す XML 文書です。これにリスト 5.2 のように書かれた XSL テンプレートを適用することで、表示用に整形されたリスト 5.3 の HTML を出力しています。

リスト 5.1: 入力の XML 文書

```
<?xml version="1.0" encoding="utf-8"?>
<library>
  <book>
    <title>Learning XSLT</title>
    <author>John Doe</author>
  </book>
  <book>
    <title>XML in Action</title>
    <author>Jane Smith</author>
  </book>
</library>
```

リスト 5.2: XSL テンプレート

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html"/>
  <xsl:template match="/">
    <html>
      <body>
        <h1>Library Books</h1>
        <ul>
          <xsl:apply-templates select="library/book" />
        </ul>
      </body>
    </html>
  </xsl:template>
  <xsl:template match="book">
    <li>
      <strong>
        <xsl:value-of select="title"/>
      </strong> by <xsl:value-of select="author"/>
    </li>
  </xsl:template>
</xsl:stylesheet>
```

リスト 5.3: 出力 HTML

```
<html>
  <body>
    <h1>Library Books</h1>
    <ul>
      <li><strong>Learning XSLT</strong> by John Doe</li>
      <li><strong>XML in Action</strong> by Jane Smith</li>
    </ul>
  </body>
</html>
```

XSLT はテンプレートマッチングを駆使して文書の特定の部分に応じた処理を行います。この例では、XSL の<xsl:template match="/">のテンプレートが文書全体にマッチして呼び出され、その中の<xsl:apply-templates select="library/book"/>で、孫要素の<book>それぞれに対してマッチするテンプレート<xsl:template match="book">が呼び出される形となっています。

XSLT には xsl:if や xsl:for-each のような制御構文もありますが、このようなテンプレート言語が本当にプログラミング言語と言えるのでしょうか？

5.3 チューリング完全性について

チューリング完全とは

プログラミング言語であるためには、その言語がチューリング完全であることが必要だとよく言われます。

チューリング完全とは、チューリングマシンと同等の計算能力があることを意味します。チューリングマシンはアラン・チューリングによって考案された仮想の機械で、無限に長いテープ（記憶装置）と、そこに対してデータを読み書きするヘッドで構成されています。この機械に、ヘッドの移動や読み書きを指示するプログラムを与えることで動作する計算機です。

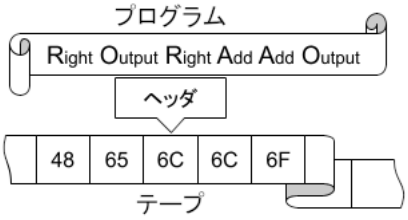

図 5.1: チューリングマシンの概念図

チューリングマシンは非常に単純な構造ですが、これまでに知られているどんな複雑な計算機械でも、理論上チューリングマシンで再現できることが知られています。そして計算可能なあらゆるアルゴリズムは、このチューリングマシンでも計算できるとされています[*2]。

プログラミング言語であるならば、あらゆる計算問題に対応できなくてはなりません。チューリング完全であれば、どんな計算問題でも計算できることが理論的に保証されるわけです。これが、プログラミング言語がチューリング完全であることを必要とする理由です。

Brainfuck とチューリング完全

ある言語がチューリング完全であることを示す方法のひとつが、すでにチューリング完全であることがわかっている言語の処理系をその言語で実装することです。チューリング完全な言語で書かれたプログラムをすべて実行できるのであれば、その言語を通せばあらゆる計算を実行できることになり、チューリング完全であると言えるわけです。

ところで、Brainfuck というプログラミング言語をご存知でしょうか？ この言語は記号のみで記述する、いわゆる難解プログラミング言語のひとつとして知られていますが、実は非常に有用な特徴を持っています。

リスト 5.4: Brainfuck で Hello, world!

```
+++++++++[>++++++++>++++++++++>+++++<<<-]>.>++.+++++++..+++.>-.
------------.<++++++++.--------.+++.------.--------.>+.
```

Brainfuck の処理系は、1 次元のメモリ配列とそのアドレスを示すポインタを 1 つ持ち、プログラムはポインタの移動とメモリ上の値の読み書きといった命令の列になっています。

[*2] 原義的には、計算可能性を定義するためにチューリングマシンが利用されています。

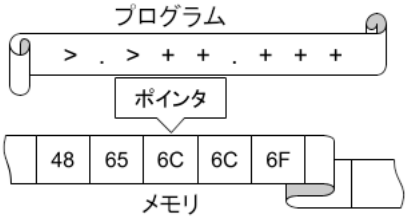

図 5.2: Brainfuck 処理系の概念図

ご覧のとおり、Brainfuck の処理系はチューリングマシンとほぼ同一の構成になっていて、チューリング完全であることが自明です。加えて、命令もわずか 8 種類のみであり、実装が容易です。このため、チューリング完全であることを示すために実装するにはうってつけの言語です。

表 5.1: Brainfuck の命令一覧

記号	処理
>	ポインタを右に移動
<	ポインタを左に移動
+	ポインタの指すメモリの値をインクリメント
-	ポインタの指すメモリの値をデクリメント
[	ポインタの指すメモリの値が 0 なら対応する] の次にジャンプ（ループ起点）
]	対応する [にジャンプ（ループ終点）
.	ポインタの指すメモリの値を ASCII 文字として出力
,	入力を受け取りポインタの指すメモリに書き込む

ということで、本題にもどりましょう。ここからは XSLT で Brainfuck の処理系を実装していきます。

5.4 XSLT による Brainfuck インタプリタの実装

これから紹介する XML による Brainfuck インタプリタの実装は GitHub にて公開していますので、適宜参照してみてください。

- `https://github.com/makiuchi-d/bfxslt`

インタプリタ本体は `bf.xsl` に実装しています。この他、`sample` ディレクトリにサンプルプログラムの XML ファイルをいくつか用意しました。

実行は XSLT プロセッサのコマンドやブラウザで行うことができます。リポジトリの README にも記載していますが、たとえば xsltproc を使用する場合は次のようにコマンドを実行してください。

リスト 5.5: xsltproc での実行

```
$ xsltproc bf.xsl sample/hello.xml
Hello, world!
```

XML による Brainfuck コードの表現

XSLT では、入力は XML でなくてはなりません。Brainfuck のコードは単純な命令の列なので、これを XML で表現します。また、Brainfuck は入力も受け付けるので、これも同じ XML ドキュメントにまとめてしまいましょう。まずルート要素として<bf>タグを配置し、その中に Brainfuck のコードの<code>タグ、入力データの<input>タグを入れ子にします。

たとえば、入力として「Aa」を読み込み、それぞれ 1 ずつインクリメントして「Bb」を出力するプログラムはリスト 5.6 のような XML になります。

リスト 5.6: XML による Brainfuck コードと入力の表現

```
<?xml version="1.0" encoding="utf-8"?>
<bf>
  <input>Aa</input>
  <code>
    <inc/><inc/>
    <loop>
      <dec/><right/><read/><inc/><print/><left/>
      <end/>
    </loop>
  </code>
</bf>
```

元の Brainfuck のコードは「++[->,+.<]」[*3]ですが、この各記号を表 5.2 のように定義した XML タグに置換して記述しています。

[*3] 単純に「,+.,+.」のようにも書けますが、ループの例を示すためにあえて複雑にしています

表 5.2: Brainfuck の命令とタグの対応

記号	xml タグ
>	<right/>
<	<left/>
+	<inc/>
-	<dec/>
[...]	<loop>...<end/></loop>
,	<read/>
.	<print/>

ここで、] は</loop>のようにタグを閉じるだけでなく、直前に必ず<end/>タグを挿入するようにしました。この理由は後ほど説明します。

実装の方針

XSLT では、特定の XML 要素に対応するテンプレートを定義して、それが呼び出されることで処理が進行します。インタプリタ実装の bf.xsl にはいくつかテンプレートが定義されていますが、最初に実行されるのはマッチングパターンが最も具体的な、リスト 5.7 のテンプレートです。このテンプレートは、プログラムの XML の全体を囲んでいる bf 要素にマッチして実行されます。

リスト 5.7: 最初に処理されるテンプレート

```
<xsl:template match="/bf">
  <xsl:variable name="input" select="input"/>
  <xsl:apply-templates select="code/*[1]">
    <xsl:with-param name="ptr" select="0"/>
    <xsl:with-param name="mem"><_0>0</_0></xsl:with-param>
    <xsl:with-param name="input" select="$input"/>
  </xsl:apply-templates>
</xsl:template>
```

ここではまず、入力にあたる<input>要素を xsl:variable を使って変数 input に格納しています。次に<code>の中の先頭の Brainfuck の命令にあたる要素に対して xsl:apply-template でテンプレートを適用しています。このときテンプレートにはパラメータとして、ポインタを表す ptr、メモリを表す mem、入力の input を<xsl:with-param>で渡します。ptr は数値で初期値は 0 です。mem はリスト 5.8 のような<_アドレス>要素の列としました。新たなアドレスにデータを書き込むたびに要素が増えていく形です。

リスト 5.8: メモリの表現

```
<_0>0</_0>
<_1>10</_1>
<_2>20</_2>
```

ところで、XSLT では変数は一度定義したら書き換えることができません。このため、グローバルな変数は使えず、変更を加えた値をパラメータとして次の命令の処理に引き回すことで、状態を更新するようにしています。

各命令のタグのテンプレートはリスト 5.9 の形をしています。

リスト 5.9: 各命令のテンプレートの基本形

```
<xsl:template match="命令のタグ名">
  <xsl:param name="ptr"/>
  <xsl:param name="mem"/>
  <xsl:param name="input"/>

  <!-- 各命令固有の処理 -->

  <xsl:apply-templates select="following-sibling::*[1]">
    <xsl:with-param name="ptr" select="$ptr"/>
    <xsl:with-param name="mem" select="$mem"/>
    <xsl:with-param name="input" select="$input"/>
  </xsl:apply-templates>
</xsl:template>
```

xsl:param で指定されているとおり、パラメータとして ptr、mem、input を受け取ります。そして各命令固有の処理をしたあと、次の命令に進むのですが、following-sibling::*[1] によって現在処理している要素の次の兄弟要素、つまり次の命令にあたる要素を指定して xsl:apply-templates で再帰的にテンプレートを適用します。こうすることで、次の命令の要素の名前とマッチするテンプレートが選ばれて処理され、これを繰り返すことでプログラムが順次処理されていくことになります。

各命令の実装

それでは各命令の処理をするテンプレートを見ていきましょう。

ポインタの移動： right, left（> <）

リスト 5.10 はポインタを右に移動する<right/>の処理をするテンプレートです。次の命令に進む xsl:apply-templates に渡すパラメータのうち、ptr の値を"$ptr + 1"とすることで、次の処理ではポインタが移動した状態となります。他のパラメータの mem と input は受け取ったものをそのまま次の命令の処理に渡しています。

リスト 5.10: right 命令のテンプレート

```
<xsl:template match="right">
  <xsl:param name="ptr"/>
  <xsl:param name="mem"/>
  <xsl:param name="input"/>

  <xsl:apply-templates select="following-sibling::*[1]">
    <xsl:with-param name="ptr" select="$ptr + 1"/>
    <xsl:with-param name="mem" select="$mem"/>
    <xsl:with-param name="input" select="$input"/>
  </xsl:apply-templates>
</xsl:template>
```

ポインタを左に移動する<left/>は、マッチングパターンを"left"とし、次に渡す ptr の値を"$ptr - 1"とするだけで実現できます。

メモリの値の加減算： inc, dec（+ -）

リスト 5.11 は、<inc/>の処理、つまり現在のポインタの指すメモリの値を 1 増加させるテンプレートです。まず値を取り出し、値を更新したメモリを生成し、次の命令のテンプレートを呼び出すという流になっています。

それぞれの処理がどう実現されているか見ていきましょう。

リスト 5.11: inc 命令のテンプレート

```
<xsl:template match="inc">
  <xsl:param name="ptr" />
  <xsl:param name="mem"/>
  <xsl:param name="input"/>
```

```
    <xsl:variable name="key" select="concat('_', $ptr)"/>
    <xsl:variable name="val" select="sum(exsl:node-set($mem)/*[name()=$key])"/>

    <xsl:variable name="mem">
      <xsl:call-template name="write-val">
        <xsl:with-param name="mem" select="$mem"/>
        <xsl:with-param name="key" select="$key"/>
        <xsl:with-param name="val" select="$val + 1"/>
      </xsl:call-template>
    </xsl:variable>

    <xsl:apply-templates select="following-sibling::*[1]">
      <xsl:with-param name="ptr" select="$ptr"/>
      <xsl:with-param name="mem" select="$mem"/>
      <xsl:with-param name="input" select="$input"/>
    </xsl:apply-templates>
  </xsl:template>
```

まず最初に値を取り出す処理として、リスト 5.12 の部分で変数 key と val を定義しています。

リスト 5.12: key と value の定義

```
    <xsl:variable name="key" select="concat('_', $ptr)"/>
    <xsl:variable name="val" select="sum(exsl:node-set($mem)/*[name()=$key])"/>
```

key は'_'とポインタの値を結合したもので、これがポインタの指す、mem の中の要素名になります。val の定義では、mem の中のノードの集合から、key と同じ名前の要素を取り出しています。要素が存在しなかったときに 0 となるように、sum 関数を利用しています。一番最初に mem の初期値として<_0>0</_0>を用意していたのは、mem の中身をノードの集合にしておくためです。

続いてこれらの値を使って、値を更新した状態のメモリを新たに作り、変数 mem を定義します[*4]。新たなメモリを作る処理は他の命令でも利用するので、リスト 5.13 のようにテンプレートで行っています。

[*4] XSLT では同じ名前の変数を同じスコープでは再定義できないのですが、この時点での mem はテンプレートのパラメータでありスコープが異なるので、同じ名前で定義できます

リスト 5.13: 値を書き込んだメモリを生成するテンプレート

```
<xsl:template name="write-val">
  <xsl:param name="mem"/>
  <xsl:param name="key"/>
  <xsl:param name="val"/>

  <xsl:for-each select="exsl:node-set($mem)/*[name()!=$key]">
    <xsl:copy-of select="."/>
  </xsl:for-each>
  <xsl:element name="{$key}">
    <xsl:value-of select="$val"/>
  </xsl:element>
</xsl:template>
```

このテンプレートでは、最初に xsl:for-each を使って、与えられた mem のうち要素名が key ではないものをすべて xsl:copy-of でコピーしています。その後に続ける形で、key の名前の要素を作り、内容を val に設定しています。この方法では要素の順序が書き換えるたびに入れ替わってしまうのですが、読み取るときには要素名でアクセスするので問題ありません。

<inc/>の処理では、このテンプレートに渡すパラメータは現在のメモリ mem と書き込む場所の key、そして val は"$val + 1"のように 1 加算した値となっていました。これによって、key の場所の値を 1 増やしたメモリが生成されるわけです。

最後にこの mem を次の命令のテンプレートに渡せば<inc/>の処理は完了です。<dec/>も同じ処理の流れで、加算していたところを"$val - 1"とするだけで実現できます。

ループ： loop, end ([ ])

Brainfuck の [は、ポインタの指すメモリの値が 0 なら対応する] の次の命令にジャンプするという命令で、] は対応する [に戻るという命令です。つまり、ポインタの指すメモリの値が 0 でない間 [と] の間にある命令が繰り返し実行されます。

リスト 5.14 が今回実装した [に相当する<loop>を処理するテンプレートです。メモリの値を読み取って val とする部分は inc の処理と同じです。その後、xsl:choose によって、val の値に応じて分岐します。val が 0 ではないとき、次に実行する要素として<loop>の内側の要素の先頭のものを"*[1]"で指定します。一方それ以外のときは、<loop>の外にある次の要素を"following-subling::*[1] で指定します。このように、[と] の対応関係を<loop>のネストで表現しているので、ジャンプ先を簡単に指定できます。

リスト 5.14: loop 命令のテンプレート

```
<xsl:template match="loop">
  <xsl:param name="ptr"/>
  <xsl:param name="mem"/>
  <xsl:param name="input"/>

  <xsl:variable name="key" select="concat('_', $ptr)"/>
  <xsl:variable name="val" select="sum(exsl:node-set($mem)/*[name()=$key])"/>

  <xsl:choose>
    <xsl:when test="$val != 0">
      <!-- 繰り返す: 子要素の先頭に進む -->
      <xsl:apply-templates select="*[1]">
        <xsl:with-param name="ptr" select="$ptr"/>
        <xsl:with-param name="mem" select="$mem"/>
        <xsl:with-param name="input" select="$input"/>
      </xsl:apply-templates>
    </xsl:when>
    <xsl:otherwise>
      <!-- 繰り返し終了: 次の要素に進む -->
      <xsl:apply-templates select="following-sibling::*[1]">
        <xsl:with-param name="ptr" select="$ptr"/>
        <xsl:with-param name="mem" select="$mem"/>
        <xsl:with-param name="input" select="$input"/>
      </xsl:apply-templates>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

ループの終わりは、<end/>を置いてから</loop>のように閉じています。この<end/>を処理するテンプレートをリスト 5.15 に示します。

リスト 5.15: end 命令のテンプレート

```
<xsl:template match="end">
  <xsl:param name="ptr"/>
  <xsl:param name="mem"/>
  <xsl:param name="input"/>

  <xsl:apply-templates select="parent::node()">
    <xsl:with-param name="ptr" select="$ptr"/>
    <xsl:with-param name="mem" select="$mem"/>
    <xsl:with-param name="input" select="$input"/>
  </xsl:apply-templates>
</xsl:template>
```

このテンプレートでは、次に処理する要素として親ノード、つまり自身を包んでいる<loop>を"parent::node()"で指定します。パラメータはここまで引き継がれてきた ptr、mem、input を渡します。こうすることで、次に実行される<loop>は、最新のメモリ状態を元にジャンプ先を分岐できるようになります。この<end/>が無い場合、親の<loop>は最初に実行された時点のポインタとメモリの状態しかわからないため、正しく処理を継続できません。また、この実装では end を書き忘れてしまうと、親の<loop>に処理を戻すことができず終了してしまいます。これが<end/>を挿入する理由です。

出力： print（.）

Brainfuck での . は、ポインタの指すメモリの値を ASCII コードとして出力します。リスト 5.16 は<print/>命令を実行するテンプレートです。メモリの値を取得する方法は他の命令と同様に、変数 val に値を保存します。その後<xsl:choose>と<xsl:when>で val の値ごとに分岐して ASCII コードに該当する文字を出力しています。

リスト 5.16: print 命令のテンプレート

```
<xsl:template match="print">
  <xsl:param name="ptr"/>
  <xsl:param name="mem"/>
  <xsl:param name="input"/>

  <xsl:variable name="key" select="concat('_', $ptr)"/>
  <xsl:variable name="val" select="sum(exsl:node-set($mem)/*[name()=$key])"/>

  <xsl:choose>
    <xsl:when test="$val=9"><xsl:text>&#9;</xsl:text></xsl:when>
    <xsl:when test="$val=10"><xsl:text>&#10;</xsl:text></xsl:when>
    <xsl:when test="$val=13"><xsl:text>&#13;</xsl:text></xsl:when>
    <xsl:when test="$val=32"><xsl:text
                        disable-output-escaping="yes"> </xsl:text></xsl:when>
    <xsl:when test="$val=33">!</xsl:when>
    (中略)
    <xsl:when test="$val=125">}</xsl:when>
    <xsl:when test="$val=126">~</xsl:when>
    <xsl:otherwise>&amp;#<xsl:value-of select="$val"/>;</xsl:otherwise>
  </xsl:choose>

  <xsl:apply-templates select="following-sibling::*[1]">
    <xsl:with-param name="ptr" select="$ptr"/>
    <xsl:with-param name="mem" select="$mem"/>
    <xsl:with-param name="input" select="$input"/>
  </xsl:apply-templates>
</xsl:template>
```

XSLT2.0 以降であれば ASCII コードと数値を相互変換する関数も提供されていますが、今回対象としているのは XSLT1.0 なので愚直に値ごとに分岐するようにしました。また、XSLT1.0 では扱える制御文字が HT(9)、LF(10)、CR(13) に限られているため、それ以外の制御文字や ASCII コード外の 127 以降の値は"&#数値;"という形式の文字列を出力するようにしました。

入力： read (,)

Brainfuck の , 命令は、入力から 1 文字受け取り、ポインタが指すメモリ位置にそれを ASCII コードとして書き込みます。リスト 5.17 はこの<read/>命令を処理するテンプレートです。

今回の XSLT の実装では、入力はコードと同じ XML に含まれていて、input パラメータとして引き回されてきました。この input から先頭 1 文字を取り出してメモリに書き込み、取り出した文字を削除した新しい input を次の命令に渡すのがこのテンプレートの処理の流れです。

まず最初に input の文字列長を string-length 関数を使って確認しています。もし空であれば入力終了を意味するので、ポインタが指す位置に EOF を表す 255 を書き込んだメモリを次のテンプレートに渡します。

input が空でない場合、まず substring($input, 1, 1) で先頭 1 文字を取り出し変数 c に保存します。そして次の命令に渡す input も substring($input, 2) のように 2 文字目以降を切り出しておきます。

取り出した先頭文字 c は、出力のときと同じように<xsl:choose>で文字種ごとに分岐して ASCII コードの数値に変換します。このとき、扱えない制御文字や ASCII コード外の文字は、すべて 255 とすることにします。この値を書き込んだメモリと、1 文字削った input を引数にして、次の命令のテンプレートを呼び出したら完了です。

リスト 5.17: read 命令のテンプレート

```
<xsl:template match="read">
  <xsl:param name="ptr"/>
  <xsl:param name="mem"/>
  <xsl:param name="input"/>

  <xsl:variable name="key" select="concat('_', $ptr)"/>

  <xsl:choose>
    <!-- inputが空のときは255(EOF)を書き込む -->
    <xsl:when test="string-length($input)=0">
      <xsl:apply-templates select="following-sibling::*[1]">
```

```
        <xsl:with-param name="ptr" select="$ptr"/>
        <xsl:with-param name="mem">
          <xsl:call-template name="write-val">
            <xsl:with-param name="mem" select="$mem"/>
            <xsl:with-param name="key" select="$key"/>
            <xsl:with-param name="val" select="255"/>
          </xsl:call-template>
        </xsl:with-param>
        <xsl:with-param name="input" select="$input"/>
      </xsl:apply-templates>
    </xsl:when>

    <!-- inputが空でないとき1文字取り出してメモリに書き込む -->
    <xsl:otherwise>
      <xsl:variable name="c" select="substring($input, 1, 1)"/>
      <xsl:variable name="input" select="substring($input, 2)"/>

      <xsl:variable name="val">
        <xsl:choose>
          <xsl:when test="$c='&#9;'">9</xsl:when>
          <xsl:when test="$c='&#10;'">10</xsl:when>
          <xsl:when test="$c='&#13;'">13</xsl:when>
          <xsl:when test="$c=' '">32</xsl:when>
          <xsl:when test="$c='!'">33</xsl:when>
          (中略)
          <xsl:when test="$c='}'">125</xsl:when>
          <xsl:when test="$c='~'">126</xsl:when>
          <xsl:otherwise>255</xsl:otherwise>
        </xsl:choose>
      </xsl:variable>

      <xsl:variable name="mem">
        <xsl:call-template name="write-val">
          <xsl:with-param name="mem" select="$mem"/>
          <xsl:with-param name="key" select="$key"/>
          <xsl:with-param name="val" select="$val"/>
        </xsl:call-template>
      </xsl:variable>

      <xsl:apply-templates select="following-sibling::*[1]">
        <xsl:with-param name="ptr" select="$ptr"/>
        <xsl:with-param name="mem" select="$mem"/>
        <xsl:with-param name="input" select="$input"/>
      </xsl:apply-templates>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

以上で、Brainfuck の 8 種類の命令をすべて実装することができました。実際に動くかどうかは、ぜひお手元で確認してみてください。

5.5 まとめと展望

この章では、XSLT による Brainfuck の処理系の実装例を紹介しました。実際に実装できたことから XSLT はチューリング完全であり、理論的には任意の計算が可能で、文書の変換や操作にとどまらない強力なツールになりえる可能性があります。

しかし、これを実用するには多くの課題があります。実際パフォーマンスも悪く、複雑なプログラム[*5]は途中で強制終了してしまうことも珍しくありません。また文法も、この章を読んでいただいた方なら感じられたと思いますが、人間が読み書きするのに向いておらず、デバッグやエラーハンドリングの面でも制約しかありません。くわえて、XSLT 2.0 や 3.0 が W3C より勧告[*6]されてすでに何年も経っているのですが、主要ブラウザでは未だサポートされておらず、今後の発展は期待しにくいでしょう。

一方で XSLT によるプログラミングでは、パターンマッチングや再帰呼び出しといった宣言型・関数型のパラダイムを強制されることになります。プログラミング用途としてはまったく実用には向きませんが、文法の厳しさも含めて、奇特な方向けのトレーニングにはうってつけの言語といえるでしょう。みなさんもぜひ XSLT でのプログラミングにチャレンジしてみてください。

[*5] たとえば、リポジトリの `sample/toupper.xml` はブラウザや `xsltproc` では実行できません。

[*6] `https://www.w3.org/TR/xslt20/`、`https://www.w3.org/TR/xslt-30/`

第 6 章

好きな C++ のコア機能発表ドラゴン

Hiro Onozawa

6.1 (投稿者コメント)

正論ロボ「業務で C++ は使わないのに...」

新卒で入社して 3 年目になったので初投稿です。
流行りにのって動画を作り、社内のフリーテーマ発表会で公開しました。その動画をベースに、好きな C++ のコア機能を発表します。

注意事項：

- 機能毎に解説を添えていますが、筆者の主観であり、事実と異なる可能性があります。
- サンプルコードは紙面に収める都合で一般的ではないフォーマットになっている場合があります。
- ♪ のマークが付いた行を括弧の中は読み飛ばしながら口ずさんで、あなたも発表ドラゴンになりましょう。

6.2 1 番

♪ C++ 好きな発表ドラゴンが
♪ 好きな (C++ の) コア機能を発表します

♪ const

リスト 6.1: 非メンバ関数を const を用いてオーバーロードする例

```
 1: #include <iostream>
 2:
 3: struct MyClass {
 4:     void Print() { std::cout << "void Print()" << std::endl; }
 5:     void Print() const { std::cout << "void Print() const" << std::endl; }
 6: };
 7:
 8: void Func1(void) {
 9:     MyClass myClass;
10:     const MyClass constMyClass;
11:     myClass.Print();
12:     constMyClass.Print();
13: }
```

const は不変性を表すことができます。C からある機能ですが、C++ ではより高度な使い方が可能です。

C++ の非 static メンバ関数は、そのインスタンスの const 修飾の有無に応じたオーバーロードが可能です (リスト 6.1 の 4-5 行目)。const な非 static メンバ関数では this の型も const 修飾されるため、非 static メンバ変数を書き換えることもできませんし[*1]、非 const な非 static メンバ関数を呼び出すこともできません。

他にも、コンテナの要素型を const にすることで要素が不変なコンテナ型を定義出来たり、参照と組み合わせて一時オブジェクトを効率よく変数に束縛出来たりします。不変性について強く制約できる、好き好き大好きな機能です。

♪ 定数式 (constexpr)

C++11 以降、constexpr というキーワードを付けて関数やコンストラクタ、メソッドを定義すると、定数式として扱われます。

定数式は、全ての引数がコンパイル時定数であればコンパイル時に計算を行い、そうでなければ実行時に計算を行います。定数式の適用可能な範囲は仕様の策定のたびに拡大されており、C++20 からは std::vector クラスのコンストラクタや std::sort テンプレート関数なども定数式となりました。

[*1] 非 static メンバ変数の宣言時に mutable キーワードで修飾した場合はその限りではありません。

リスト 6.2: constexpr を用いて文字列リテラルに含まれる小文字を数える例

```
 1: #include <iostream>
 2:
 3: using std::size_t;
 4: class CES {
 5:     const char* p; size_t sz;
 6: public:
 7:     template<size_t N>
 8:     constexpr CES(const char(&a)[N])
 9:     : p(a) , sz(N - 1)
10:     { }
11:
12:     constexpr char operator[](size_t n) const
13:     { return n < sz ? p[n] : throw "err"; }
14:
15:     constexpr size_t size() const
16:     { return sz; }
17: };
18:
19: constexpr size_t CntLower(const CES& s, size_t n = 0, size_t c = 0) {
20:     if (n == s.size()) { return c; }
21:     return ('a' <= s[n] && s[n] <= 'z')
22:         ? CntLower(s, n + 1, c + 1)
23:         : CntLower(s, n + 1, c);
24: }
25:
26: template<size_t n>
27: void PrintConstN()
28: { std::cout << n << std::endl; }
29:
30: void Func2() {
31:     PrintConstN<CntLower("Hello")>();
32:     PrintConstN<CntLower("world")>();
33: }
```

リスト 6.2 の PrintConstN 関数は非型テンプレート引数 n を受け取りますが、この n はコンパイル時定数である必要があります。CntLower 関数は定数式であり、s、n、c が共にコンパイル時定数であればコンパイル時に size_t 型のコンパイル時定数を返します。引数 s は CES クラスですが、このクラスのコンストラクタも定数式として定義されており、コンパイル時に文字列リテラルから CES クラスを定数式としてインスタンス化可能になっています。これにより、CntLower("Hello") もコンパイル時定数となり、PrintConstN 関数の非型テンプレート引数 n に渡すことが可能です。

コンパイル型言語である利点を最大限生かそうとする、好き好き大好きな機能です。

♪ (メンバ) 初期化子 (member initializer list)

リスト 6.3: メンバ変数の初期化子を用いた初期化の例

```
 1: #include <iostream>
 2:
 3: class C {
 4:     int m_i; std::string m_str;
 5: public:
 6: #if true // メンバ初期化子を用いてメンバ変数を初期化するパターン
 7:     C(int i, const char *pstr)
 8:     : m_i(i) , m_str(pstr)
 9:     { }
10: #else // メンバ変数への代入を行うパターン
11:     C(int i, const char *pstr)
12:     { m_i = i; m_str = pstr; }
13: #endif
14:     char f(void) const { return m_str[m_i]; }
15: };
16:
17: void Func3() {
18:     C c0(0, "Hello");
19:     std::cout << c0.f() << std::endl;
20: }
```

C++ のコンストラクタ定義において、メンバの初期化子を記述できます。

C++ では、初期化と代入は区別されます。リスト 6.3 の`#if` プリプロセッサディレクティブで有効になっている 7-9 行目の実装は、メンバ初期化子を用いているため、引数で渡した値を用いてメンバ変数が初期化され、代入は行われません。無効になっている 11-12 行目の実装は、メンバ初期化子を用いずコンストラクタでメンバ変数に引数を代入しているため、メンバ変数はまずデフォルト値で初期化され、その後引数で渡した値が代入されます[*2]。

細かな違いも規定する、C++ らしくて好き好き大好きな機能です。

♪ 添字に引数複数取るやつ

♪ 正式名称が　わからないコア機能も
♪ 好き　好き　大好き

[*2] リスト 6.3 のように単純な例では最適化により同一のバイナリになる可能性がありますが、メンバ変数の型がユーザー定義コンストラクタを持つクラスだったりすると明確な違いが現れます。

リスト 6.4: 複数の引数を取る添字演算子を定義した行列クラスの実装例

```
 1: #include <iostream>
 2: #include <memory>
 3:
 4: class Matrix {
 5:     std::unique_ptr<float[]> p;
 6:     size_t w, h;
 7: public:
 8:     Matrix(size_t w, size_t h)
 9:     : p(new float[w * h]{0})
10:     , w(w), h(h)
11:     { }
12:
13:     float& operator[](int x, int y)
14:     { return p[y * w + x]; }
15:     const float& operator[](int x, int y) const
16:     { return p[y * w + x]; }
17:
18:     void print() const {
19:         using std::cout, std::endl;
20:         for(auto i = 0z; i < w * h;) {
21:             cout << p[i] << " ";
22:             if (++i % w == 0) { cout << std; }
23:         }
24:     }
25: };
26:
27: void Func4() {
28:     Matrix A(3, 4);
29:     A[0, 2] = 1.0;
30:     A.print();
31: }
```

この機能の正式名称はわかりませんが、C++23 から添字演算子 (operator[]) が複数の引数を取れるようになりました[*3]。これで行列のような型のインデックスアクセスを自然に書けるようになります。

かつてはカンマ演算子をオーバーロードするという黒魔術により対処するケースもありましたが、そのような回りくどい対応が不要になり歓喜した方も多いと思います。

プログラマのニーズに応えてくれていると感じられる、好き好き大好きな機能です。

[*3] この機能の導入に先駆けて、C++20 で添字演算子内でのカンマ演算子の使用が非推奨化されています。

6.3 2 番

♪ C++ 好きな発表ドラゴンが
♪ 好きな (C++ の) コア機能を発表します

♪ auto

リスト 6.5: auto を用いて変数を定義する例

```
 1: void Func5() {
 2:     int x = 1;
 3:     int& r = x;
 4:
 5:     // どちらも int 型
 6:     int   a_cp_x = r;
 7:     auto  b_cp_x = r;
 8:
 9:     // どちらも int& 型
10:     int&  c_ref_x = r;
11:     auto& d_ref_x = r;
12: }
```

現代の C++ においてあらゆる変数宣言で用いられるであろうキーワード、auto です。

auto は C89 の時代から存在しており、初期の C++ にも同じ機能で引き継がれました。C の時代から auto が担っていたのは記憶域クラス指定子としての機能で、static や extern、register[*4]と同格の機能がありました。ただし auto を明示した変数宣言は、記憶域クラス指定子を何も付けない場合の変数宣言と全く同じ記憶域クラス指定となる仕様でした。それゆえ、おそらく C 言語を書いた事のある 99% の方は使ったことがないものと思います。

それくらい使う必要のない機能だったこともあり、C++11 からは型をコンパイラが推論するキーワードとして機能を変えることとなりました。なお、auto による型の推論では一部の文脈で std::initializer_list へ推論されるケースがある点を除き、テンプレートにおける型の推論と同じ規則が適用されます。1 つの規則を覚えれば 2 つの機能を理解できるようになっていて、嬉しいですね。

正確な型名を記述する大変さを取り除いてくれた、好き好き大好きな機能です。

[*4] register 修飾子は C++11 で非推奨化、C++17 でキーワードから削除 (ただし予約語のまま) されています。

♪ decltype

リスト 6.6: auto と decltype を用いた変数定義の比較例

```
 1: void Func6() {
 2:     int x = 1;
 3:     int& r = x;
 4:
 5:     // decltype(auto) var_name = exp; は
 6:     // decltype(exp) と等しい
 7:
 8:     auto           a_cp_x  = r; // int
 9:     decltype(r)    b_ref_x = r; // int&
10:     decltype(auto) c_ref_x = r; // int&
11:
12:     decltype(auto) d_cp_x  = x;   // int
13:     decltype(auto) e_ref_x = (x); // int&
14: }
```

auto のように型の記述を肩代わりするキーワードに decltype があります。decltype はオペランドで指定した式の型となります。

decltype は式の代わりに auto を与えることも可能です。この場合、変数宣言の初期化子や関数の戻り値などを decltype のオペランドに与えた場合と等価です。auto と decltype では型推論の結果が微妙に異なり、誤解を恐れずに書けば「参照が考慮されるか否か」が異なります[*5]。また decltype は厳密に式の型を推論する性質ゆえ、式に括弧を付けるだけで推論結果の型が変わる場合があるという興味深い側面をもちます。

細かなルールを押し付けてくるところも C++ らしくて、好き好き大好きな機能です。

♪ コンセプト (concept)

型の制約を記述できる、著者が個人的に最も待望していた C++20 の新機能です。これによりテンプレートの特殊化が、格段に、容易に記述できるようになります。

C++17 以前でも、後述する SFINAE と呼ばれるテンプレートに関する仕様を巧みに利用することで、テンプレート引数を制約することが可能でした。しかしそれは記述性、可読性、保守性、いずれの観点でも優れているとは言えないものでした。

[*5] 筆者は完全な差異を正確に記述できる程正しく理解していないため、参考文献を当たってください。筆者はテンプレート完全ガイドを 8 割読んで体感 2 割くらい理解しました。

リスト 6.7: concept を用いてテンプレート引数を制約する例

```
 1: template <class T>
 2: concept Speakable = requires (T& x) { x.speak(); };
 3: template <class T>
 4: concept UnSpeakable = !Speakable<T>;
 5:
 6: template <Speakable T>
 7: void Call(T& x) {
 8:   std::cout << "Call as Speakable : " << std::endl;
 9:   x.speak();
10: }
11: template <UnSpeakable T>
12: void Call(T& x) {
13:   std::cout << "Call as UnSpeakable." << std::endl;
14: }
15:
16: struct Alice {
17:   void speak() { std::cout << "I'm Alice." << std::endl; }
18: };
19: struct Bob {
20:   void speak() { std::cout << "I'm Bob." << std::endl; }
21: };
22: struct Eve {
23:   void listen() { }
24: };
25:
26: void Func7() {
27:   Alice a; Call(a);
28:   Bob   b; Call(b);
29:   Eve   e; Call(e);
30: }
```

リスト 6.7 では初めに「型 T が speak メソッドを持つ」という制約を Speakable として、「型 T が Speakable を満たさない」という制約を UnSpeakable として、それぞれ定義しています。続いて関数 Call のテンプレート引数を Speakable、UnSpeakable それぞれで制約して多重定義しています。この多重定義により、Speakable を満たす Alice クラスや Bob クラスのインスタンスを引数に渡した場合と、Speakable を満たさない Eve クラスのインスタンスを引数に渡した場合とで、異なるテンプレート定義が実体化され呼び出されます。

C++11 に盛り込むか否かで論争を巻き起こし、C++11 の承認を遅らせることになったという歴史もあるくらいの一大トピックで、好き好き大好きな機能です。

♪ 戻り値の型を後ろに書くやつ

♪ 正式名称が　わからないコア機能も
♪ 好き　好き　大好き

リスト 6.8: 戻り値型を後置した関数定義の例

```
 1: auto TRTFunc1(void) -> int { return 0; }
 2:
 3: struct Callable {
 4:     int operator()(void) { return 0; }
 5: };
 6:
 7: template<class T>
 8: auto TRTFunc2(T& f) -> decltype(f()) { return f(); }
 9:
10: // C++14以降 : return 文から推論される
11: template<class T>
12: auto TRTFunc3(T& f) { return f(); }
13:
14: void Func8() {
15:     Callable f;
16:     TRTFunc1();
17:     TRTFunc2(f);
18:     TRTFunc3(f);
19: }
```

あまり知られていないように思いますが、C++ では関数やラムダの戻り値型を後置することが可能です。

この機能は主にテンプレート関数で戻り値型の記述を簡略化する意図があり C++11 で導入された機能でした。しかし続く C++14 で、return 文から戻り値型を推論する機能も導入されたことにより、戻り値型を後置で明示する必要は薄くなってしまいました。

そんな日の目を見なくなってしまった機能も、好き好き大好きです。

6.4　3 番

♪ C++ 好きな発表ドラゴンが
♪ 好きな (C++ の) コア機能を発表します

♪ 参照 (reference)

参照、「C++ 何もわからん」となる機能の筆頭候補ではないでしょうか。

ただでさえ正しく理解するのが難しい機能ですが、さらに C++11 から右辺値参照とムーブセマンティクスが導入され、これとテンプレートと組み合わさり、理解する難易度が飛躍的に上がったように思います。筆者も人並みにはその特性を理解している心づもりでおりますが、少ない紙面にサンプルを添えて解説することは諦めました。それぐらい奥が深い機能です。

人知の及ぶ範囲を越えていそうな気がしつつ、それでも実行効率を追求した実装ができる余地を残そうとするところも、好き好き大好きです。

♪ 名前検索 (name lookup)

リスト 6.9: 名前検索を説明する例

```
 1: namespace NS1 {
 2:     struct Color {
 3:         int r, g, b;
 4:         Color(int r, int g, int b)
 5:         : r(r), g(g), b(b) { }
 6:     };
 7:
 8:     void Print(Color& c) {
 9:         std::cout << c.r << std::endl;
10:         std::cout << c.g << std::endl;
11:         std::cout << c.b << std::endl;
12:     }
13: }
14:
15: void Func9() {
16:     NS1::Color c(255, 0, 0);
17:     Print(c);
18: }
```

名前検索、これは言語機能というよりはコンパイラの機能という方が妥当かもしれません。あらゆる「名前」に対して、それが何を指しているのか、コンパイル時に検索が行われます。もし名前検索に失敗したら、コンパイルは失敗します。

リスト 6.9 は正しくコンパイルされます。一見何の問題もないコードに見えますが、17 行目の `Print(c)` という関数呼び出しに注目してみてください。ここは本来 `NS1::Print(c)` であるべきはずですが、なぜ `Print(c)` でも問題なくコンパイルでき、期待通り動作するので

しょうか? これは「引数依存検索 (ADL:Argument-Dependent Lookup)」という名前検索のルールによるものです。

括弧にくくられていない「修飾されていない名前[*6]」に引数リストが続くとき、ADL が働きます。ADL が働く場合、通常の検索範囲からではなく、その引数と関連する名前空間や関連するクラスを検索範囲として、名前検索が行われます。厳密な定義は規格や参考文献に任せて割愛しますが、大雑把にはその引数自身のクラスやそのクラスが定義された namespace と思ってください。リスト 6.9 の例では、`Print(c)` の引数は `NS1::Color` 型ですから、検索範囲は `NS1::Color` クラスと `namespace NS1` となり、`Print` の名前検索結果は `NS1::Print` となります。

この機能の必要性に疑問を抱いた方は、`a + b` という式が `operator+(a, b)` と等価であることを思い出してみて下さい。もしこの機能がなければ、せっかくユーザー定義クラス向けに演算子のオーバーロードをしたのに、その演算子を使った式でそれが呼ばれず、コンパイルエラーを生じるという悲惨な結果となってしまいます。

他にテンプレートに関する名前検索のルールとして「二段階検索 (Two-Phase Lookup)」と呼ばれるものもあります。解説は割愛しますが、こちらもテンプレートを自然に記述することを可能とするための興味深い性質を持っています。

コンパイラが直感的に振舞うよう、一生懸命ルールを定義してくれたえらい人も、好き好き大好きです。

♪ SFINAE (Substitution Is Not An Error)

SFINAE (Substitution Is Not An Error) はテンプレートの機能が拡充される過程で必要になり、C++11 で正式に導入された機能[*7]です。これを活用して高度なテンプレートメタプログラミングが可能になってしまった側面もありますが、それはあくまでも副産物です。SFINAE はテンプレートを使う限りいつの時代でも必要になる大切な機能です。

C++ では関数のオーバーロードが可能です。これはテンプレート関数にも言えます。そのため、テンプレート関数をインスタンス化する際には複数のテンプレート関数定義のインスタンス化を試みることになります。テンプレート関数をオーバーロードしているとき、通常はそれぞれのテンプレート関数定義は特定の型に対してのみ有効なものとして定義する事になります。裏を返せば、特定の型以外ではインスタンス化できないようなテンプレート関数定義となるわけです。

[*6] スコープ解決演算子 (`::`) やメンバアクセス演算子 (`.` または`->`) によって名前が属するスコープが明示されていない名前

[*7] 筆者は正確な歴史を把握できていませんが、C++03 では「仕様で明文化されておらずコンパイラの実装依存」だったものと理解しています。

リスト 6.10: SFINAE を用いてテンプレート関数の多重定義が問題なくインスタンス化される例

```
 1: template<class T>
 2: auto OverloadFunc(const T& v) -> decltype(v.valueOf()) {
 3:     return v.valueOf();
 4: }
 5: template<class T>
 6: auto OverloadFunc(const T& v) -> decltype(v.toString()){
 7:     return v.toString();
 8: }
 9:
10: struct Valuable {
11:     int valueOf(void) const { return 10; }
12: };
13: struct Stringable {
14:     const char* toString(void) const { return "string"; }
15: };
16:
17: void Func9() {
18:     std::cout << OverloadFunc(Valuable{}) << std::endl;
19:     std::cout << OverloadFunc(Stringable{}) << std::endl;
20: }
```

リスト 6.10 では `const T&`型の仮引数 `v` に対し、`valueOf` を呼び出すことが可能ならその戻り値を返す定義と、`toString` を呼び出すことが可能ならその戻り値を返す定義の 2 つの定義でオーバーロードされています。もし 1 つでもテンプレート関数のインスタンス化に失敗したらコンパイルエラーとなるという仕様であれば、`OverloadFunc(Valuable{})` をインスタンス化する際に 2 つ目の定義のインスタンス化に失敗し、コンパイルエラーとなってしまいます。実際には SFINAE によりテンプレートのインスタンス化に失敗しても直ちにコンパイルエラーとはしないため、1 つめの定義のみがインスタンス化され呼び出されるよう正しくコンパイルされます。

その具体的な手順は割愛しますが、SFINAE を活用することでテンプレート引数の制約を柔軟に行うことが可能で、これに歓喜した方も苦しんだ方もいることでしょう。幸いコンセプトが導入されたことで、C++20 以降テンプレート引数の制約の為に SFINAE を活用する必要はなくなりました。しかしこれからも、テンプレートの裏でコンパイラが SFINAE により良い感じにコンパイルしてくれることに違いはありません。皆さんも SFINAE に感謝してたくさんテンプレートをオーバーロードしていきましょう。

それはそれとして、うっかり本来の意図とは違う方向に世界を広げてしまう機能が紛れ込んでしまうところも、好き好き大好きです。

♪ 不等号で挟まれているやつ (三方比較演算子、three-way comparison operator)

♪ 一貫比較かな
♪ それって一貫比較だね
♪ 比較　大好き

リスト 6.11: 三方比較演算子を実装することで複数の比較演算子が導出されていることを確認する例

```
 1: class Comparelable1 {
 2:     int x;
 3: public:
 4:     Comparelable1(int x) : x(x) { }
 5:     auto operator<=>(const Comparelable1&) const = default;
 6: };
 7:
 8: class Comparelable2 {
 9:     int x, y;
10: public:
11:     Comparelable2(int x, int y) : x(x), y(y) { }
12:     auto operator<=>(const Comparelable2& rop) const
13:     { return (x <=> y) <= 0 ? x<=>rop.x : y<=>rop.y; }
14:     auto operator==(const Comparelable2& rop) const
15:     { return x == rop.x && y == rop.y; }
16: };
17:
18: void Func10() {
19:     Comparelable1 c1{1}, c2{2};
20:     Comparelable2 c3{1,2}, c4{3,4};
21:     auto conv = [](const auto& a, const auto& b)
22:     { return
23:         (a <  b) + (a <= b) + (a >  b) +
24:         (a >= b) + (a == b) + (a != b);
25:     };
26:     conv(c1, c2);
27:     conv(c3, c4);
28: }
```

C++20 から三方比較演算子 (`operator<=>`) が導入されました。この演算子は両辺の値を比較して、左辺が小さいか、大きいか、等しいかの 3 通りの値を返します。

この性質から、三方比較演算子の定義一つで 6 種類の比較演算子 (リスト 6.11 の 23-24 行目) が自動で導出でき、一定の条件を満たすとそれらが暗黙のうちに定義される、という特徴

があります。Comparelable1 のように、単純なクラスであれば三方比較演算子の定義のみで 6 種の比較演算子が暗黙のうちに定義されますが、多くの場合は Comparelable2 のように三方比較演算子と等価比較演算子 (operator==) の 2 つを定義することで、残り 5 種の比較演算子を暗黙に定義させることになるでしょう。

うまく使いこなすのはやや難しい機能ではありますが、暗黙に様々な定義を導出してくれるところも、好き好き大好きです。

6.5 (おわりに)

♪ 好きな (C++ の) コア機能が　また出てきたその時は
♪ 発表したい
♪ 発表したい

6.6 (親作品)

好きな惣菜発表ドラゴン / 重音テト

- ニコニコ動画：sm42515407
- YouTube：https://youtu.be/OnCFEo_pXaY

参考文献

- David Vandevoorde, Nicolai M. Josuttis (2010)「C++ テンプレート完全ガイド」津田 義史 訳, 翔泳社.
- Scott Meyers (2015) 「Effective Modern C++」 千住 治郎 訳, O’Reilly Japan.
- 「C++ リファレンス」 https://ja.cppreference.com/w/
- 「cpprefjp - C++ 日本語リファレンス」https://cpprefjp.github.io/index.html

執筆者・スタッフコメント

第 1 章 Daisuke Yamaguchi / @_gutio_

日課はベランダ菜園。この夏はミニトマトを子どもと楽しみました。

第 2 章 Shunsuke Ito / @fgshun

mypy の復習中。コーヒーの淹れ方も復習中。

第 3 章 Norimasa Shibata

高校生の頃の自分へ。三角関数って仕事で使えるぞ！

第 4 章 Takeichi Ayato

shader graph 楽しんでます

第 5 章 Daisuke Makiuchi / @makki_d

眼鏡っ娘が好きです

第 6 章 Hiro Onozawa

一番手に馴染むのは C++。一番使い込んだ自負があるのは C。業務で書くのは専ら bash と C#。

企画進行・イラスト・デザイン

UME

代表者として企画進行を担当しました。業務の繁忙期と重なったけどなんとかなった…。

たのりん

デザイン周り担当しました。素敵なイラストで思いっきりパロディできて楽しかったです！久しぶりにパッケージデザインをした気分…。

やもり

キャライラストを担当しました。自分の中で色々試し挑戦させて頂く機会となりました。課題が残る部分もあり…またリベンジしたいです！

いか

背景を担当しました。息づいているような素敵なキャラと細部まで凝ったパッケージ風デザインを、隅々まで楽しんでいただけたら嬉しいです。

既刊・電子版ダウンロード

https://www.klab.com/jp/blog/tech/2024/tbf17.html

KLab Tech Book Vol. 14

2024 年 11 月 2 日　技術書典 17 版（1.0）
著　者　KLab 技術書サークル
編　集　梅澤 寿史、牧内 大輔
発行所　KLab 技術書サークル
印刷所　日光企画

KLab Tech Book vol.14

1人 GitHub Actions OpenAI shader graph

KLab株式会社の有志のエンジニアが各自好きな内容で記事を執筆するKLabTechBook。バックナンバーも好評頒布中です。物理本には数に限りがありますが、PDF版はVol.1から最新号まで無料でダウンロード可能です。
KLablog(https://www.klab.com/jp/blog/)も随時更新中。「KLablog」で検索!
"Alphabet"de kaitearuto soreppoku mieruyouna kigashimasenka. Demo kono bunni imiwa arimasen. konna tokoromade oyomiitadaki arigatougozaimasu. zikaino KLabTeckBookmo otanoshimini!!

注意

特に注意することはありません。お好きなページからお楽しみください。購入報告や感想などをSNSで投稿していただけると執筆者・運営の励みになります。
<Xを使用されている方へ>
KLab技術広報Xアカウント（@klab_tech）をぜひフォローください。(2024年11月現在情報)

KLabTechBook Contents

KLab Inc.

2024 11021117 (8)