05. アルゴリズム詳解: パディング (Texture Dilation)

Matcap Maker のエクスポート機能には、**UV境界での色漏れ (Texture Bleeding)** を防ぐためのパディング処理（Dilation）が実装されています。このセクションでは、そのアルゴリズムの詳細を解説します。

問題の背景

3DモデルのUV展開図において、UVアイランドの境界ギリギリまでしか色が塗られていない場合、以下の状況でテクスチャの背景色（黒や透明）がモデル上に現れてしまい、継ぎ目（シーム）が見えてしまいます。

1. **Mipmapping**: 遠く離れたときに低解像度版のテクスチャが使われる際、隣接する透明ピクセルと色がブレンドされてしまう。
2. **Bilinear Filtering**: テクスチャ拡大時の補間処理で、境界外の色を拾ってしまう。

これを防ぐには、有効な色の領域をUV境界の外側へ数ピクセル「拡張（Dilate）」する必要があります。

実装アルゴリズム: Iterative Neighbor Filling

本ツールでは、NumPyを活用したCPUベースの反復アルゴリズムを採用しています。距離変換（Distance Transform）やJump Flooding Algorithm (JFA) のようなGPUベースの手法もありますが、Python環境での実装容易性と、エクスポート処理（非リアルタイム）である点を考慮し、以下の手法を選びました。

処理フロー

入力: `Image (H, W, 4)` RGBA画像

1. **透過部分の特定**: アルファチャンネルが0のピクセルを「ホール（穴）」と定義します。 `mask = (alpha > 0)`
2. **近傍シフト (Shift Operations)**: 画像を8方向（上下左右＋斜め）に1ピクセルずつずらしたコピーを作成します。 NumPyのスライス機能 `array[1:, :]` 等を使うことで、メモリコピーは発生しますが高速に処理できます。
3. **塗りつぶし候補の検出**: 「現在のピクセルがホール」であり、かつ「ずらした画像の同じ位置に有効な色がある」場合、その色は塗りつぶしの候補となります。
4. **色の合成**: 1つのホールに対して複数の方向から有効な色が提案される可能性があります（例：角のピクセル）。 これらの色の **平均値** を計算し、ホールの新しい色として採用します。 $$ C_{new} = \frac{\sum C_{neighbor}}{N_{valid}} $$
5. **反復**: この処理をユーザーが指定したパディングサイズ（px数）だけ繰り返します。1ループごとに有効領域が1ピクセルずつ外側に広がっていきます。

NumPyによるベクトル化実装

Pythonの `for` ループで全ピクセルを走査すると非常に低速ですが、NumPyのブロードキャスト機能を使うことで、C言語レベルの速度で処理を行っています。

```
# 概念的なベクトル化コード
def dilate_step(image):
    accum_color = zeros_like(image)
    count = zeros((H, W))

    for shift in [(0,1), (0,-1), (1,0)...]:
        shifted_img = roll(image, shift)
        valid_mask = shifted_img.alpha > 0

        # ホール部分にのみ、シフトした画像の色を加算
        accum_color[holes & valid_mask] += shifted_img[holes & valid_mask]
        count[holes & valid_mask] += 1

    # 平均化して適用
    new_colors = accum_color / count
    image[holes] = new_colors
```

この実装により、2K/4Kテクスチャであっても数秒以内で高品質なパディング処理が完了します。