2025-10-19

```
# STEP 3 — GRADIENT NOISE (Perlin idea) WITHOUT FADE
#   Random GRADIENTS at lattice corners; dot with offsets; linear interp.
# ------------------------------
# 8 normalized gradient directions
GRADS = np.array([
    [ 1, 0], [-1, 0], [ 0, 1], [ 0,-1],
    [ 1, 1], [-1, 1], [ 1,-1], [-1,-1]
], dtype=np.float64)
GRADS /= np.linalg.norm(GRADS, axis=1, keepdims=True)

# Random gradient field on the same coarse grid
grad_idx = rng.randint(0, len(GRADS), size=(Gh, Gw))
gx = GRADS[grad_idx, 0]
gy = GRADS[grad_idx, 1]

# Corner gradients
g00x = gx[yi,     xi    ]; g00y = gy[yi,     xi    ]
g10x = gx[yi,     xi + 1]; g10y = gy[yi,     xi + 1]
g01x = gx[yi + 1, xi    ]; g01y = gy[yi + 1, xi    ]
g11x = gx[yi + 1, xi + 1]; g11y = gy[yi + 1, xi + 1]

# Dot products (corner gradient · offset)
d00 = g00x*xf      + g00y*yf
d10 = g10x*(xf-1.) + g10y*yf
d01 = g01x*xf      + g01y*(yf-1.)
d11 = g11x*(xf-1.) + g11y*(yf-1.)

# Linear interpolation (NO fade)
lin_x1 = d00*(1-xf) + d10*xf
lin_x2 = d01*(1-xf) + d11*xf
grad_linear = lin_x1*(1-yf) + lin_x2*yf
```

```
# Normalize to [0,1] for display
def norm01(a):
    m, M = a.min(), a.max()
    return (a - m) / (M - m + 1e-12)

save_img("step3_gradient_linear.png", norm01(grad_linear),
         "Step 3: Gradient noise (linear, NO fade)")
```

$$GRADS = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ 1 & 1 \\ -1 & 1 \\ 1 & -1 \\ -1 & -1 \end{bmatrix}$$

Ok, grad_idx is like randomly selecting from the 8 direction vectors, the value is between (0,7) (I hate starting from 0!). gx, gy will be the two components of the direction selected at random for each BLOB. It will be 16*16, because we have 16 blobs. (gx,gy)

[ ]

Feeling very confused about the notations, I decided to ask GPT to use a concrete example, turning those annoying notations into actual numbers so that I can easily understand their meanings and impacts. So now I understand the fractional position as a measure of the distance from one individual pixel from the four corners. Because there are four corners, each corner's value will impact each location, hence we need to measure the distance from the four corners, that's what 2) is saying: four pairs of values measuring the distance from each corner.

Now, interesting part comes, what's the meaning of the dot product? Then we need to ask: what does dot product mean? Think about when you will get a zero, when you will get a max: when the two vectors are perpendicular you will get zero, when they are in the same direction you will get a max. So the dot product is measuring how two vectors are similar. Since the end results are called 'influence', basically it is a measure of how much the direction at the four corners should influence the individual location. It's not clear at this stage it's measured this way, so the further question we should ask is: what effects does such method do?

Now 4), the interpolation. The function for fade weights is about the relationship between what and what? Then what is lerp()? Asking GPT for explanation, sounds like input t is location of a pixel, the name suggests it's calculating the weights (of what) based on the location. So step 3) we already calculate the influence from the four directions at the 4 corners, now we calculate the weights? Weights of what? Isn't the influence number sort of weights? What are x1 and x2? The pixel values?

Ok now we need to gather our thoughts.

Let's do it. Instead of recalling what GPT tells us, now we have some brief ideas, let's think about what we would do. Say we have 8*8 pixels, and we decide to cut into 2*2 big blobs, hence each blob has 4*4 pixels. Note that in the white noise model, we need to generate 8*8 random numbers, controlling the brightness for each pixel, giving us the impression of 'noise', with no structures.

Now in the blob model, we treat the 2*2 blobs with random brightness, but within each block, we interpolate the pixel values from the four corners, hence, giving you the impression that there is sort of structures within. Say, we have 0.12, 0.52, 0.1, 0.2 respectively. Now comes to the interpolation details, what would I have done?

Take the bottom left blob for example, knowing the values at the four corners, for the 2^{nd} row, 3^{rd} column pixels, how would you calculate the pixel intensity? Interpolation means it's somehow averaging the four pixel values at the corners, but the question is, how to average, what is the weight? A natural way is not assigning equal weights, but a distance-dependent weight: the further away it is from the corner, the less the impact. So it is 25% away from the top left corner in the vertical direction, 50% away from it in the horizontal direction. I feel a bit confused with this components approach, the decomposition in the xy component. A more natural way is to think of distance: calculate how far it is from the four corners, d1, d2, d3, d4, then the weight is the distance over the total distance!

Let's see how GPT does it:

```
val_x1 = v00*(1-xf) + v10*xf
val_x2 = v01*(1-xf) + v11*xf
value_noise = val_x1*(1-yf) + val_x2*yf
```

Looks like it calculates the weights pairwise: treat the two corners above as a pair, and the two bottom ones a pair, assigning offsetting weight: if v00 contributes 20%, then v10 contributes 80%, make sense because as you gets further away from one corner, you moves closer to the other. So val_x1 is the average contribution from top 2 corner, val_x2 is from bottom 2 corners. But only the distance from x direction is accounted for. Now we attributes the weights from the vertical distance, making sure the top two's weight is compensated by the bottom two's: controlled by the yf and (1-yf) factors. It is a simple way to factor the location into the weight allocation. But, how about if we use the distance/sum of distance to assign the weight? Would the results be similar?

Ok, now on this gradient noise. It is still about mixing the four corner pixels to come up with the inner pixels, but the question is how to mix. The previous method only consider distance. Now we also consider directions. But going through the details: where are the random pixels? How to factor in the direction? Let's pull up the big picture for Perlin method.