2025-10-26

Stage 2: Interpolation (mixing by distance)

Blend those four influences exactly like Eq (1) but using fade-weighted coordinates

$u = f(x_f), v = f(y_f)$

with the smoothstep polynomial

$f(t) = 6t^5 - 15t^4 + 10t^3.$

$$n(x,y) = (1-u)(1-v)\,d_{00} + u(1-v)\,d_{10} + (1-u)v\,d_{01} + uv\,d_{11}.$$

So if we compare the formula to the value-noise one, the structure is the same. The only differences are:

1. D00 instead of v00, where d() are the direction-based influence, v00 are purely the corner random pixel value.
2. The weights are still distance dependent, but processed in a non-linear way.
   - We can park the question regarding, why, what's the effect of doing so, and are other alternatives. Actually this get me to think of a three step questions when learning some new method: 1) Why doing this? 2) Compare to not doing this, what's the effect? 3) Are there any other alternatives?

$$n(x,y) = (1-x_f)(1-y_f)\,v_{00} + x_f(1-y_f)\,v_{10} + (1-x_f)y_f\,v_{01} + x_f y_f\,v_{11}.$$

Seems that step 4 will answer the question about those fade-weighted coordinates. So first of all, GPT points out the problem with linear weights like $(1-x_f)(1-y_f)v00 + \cdots$: the problem is the slope jumps at cell borders. Well, is this a visible symptom, or merely math technicalities? I mean, can you see the jumpiness? Then GPT points out the math technicalities: the derivatives jump from 0 to 1 immediately. Excuse me, which one to differentiate? And differentiate what respect to what? I think once a concept of 'derivative' emerge, several other concepts must also emerge:

1) Which relationship between what and what?
2) Then whose change with what?

if I just look at the formula n(x,y)=$(1-x_f)(1-y_f)v00 + x_f(1-y_f)$ .... , there are so many symbols, so which one is the dependent, which one is the independent variable? If I view $w(x_f, y_f) = (1-x_f)(1-y_f)$, that means I am viewing a relationship between the weights and the individual distance to the leftmost corner? If we differentiate, we will have partial derivatives: $w_{xf} = -(1-y_f)$, $w_{xf} = -(1-y_f)$. Is this the derivative GPT is talking about??? So what's the problem with this one? We can ask GPT to clarify, or we can keep reading and make educated guess. Let's do that.

So instead of using x_f directly, we plug it into a 'fade function': $f(t) = 6t^5 - 15t^4 + 10t^3$. Remember x_f is between 0 and 1, marking horizontally percentage wise how far it is from the leftmost corner. Why pick this function, how to make sure it is legit? So first sanity check is, at the endpoints, they retain the value: f(0)=0, f(1)=1, the question is why? GPT does not explain, but it's a legit question from a student. Well that's because when it's right at the corner, so $x_f = 0$, the contribution is 100%. (well explain in more detail if we are doing a course). So these end-value check are the minimum. Then the derivative, $f'(t) = 30t^4 - 60t^3 + 30t^2$. At the boundary, $f'(0) = 0, f'(1) = 0$.

Ok, there are lots of gaps and holes in GPT's explanation. More contexts are needed. The specific questions are:

1) Whose derivatives jump from 0->1 immediately?
2) Now using the fade function, how is this problem solved?
3) GPT gives us pieces of information, we need to put them together into a coherent narrative.

Keep reading, seems GPT is doing more explanation.

$$n(x,y) = (1-x_f)(1-y_f)d_{00} + x_f(1-y_f)d_{10} + (1-x_f)y_f\, d_{01} + x_f y_f\, d_{11} \quad (1)$$

It asks us to look along a 1D slice at yf=0:

$$n(x) = (1-x_f)d00 + x_f d_{10}$$

Ok so letting yf=0, means we are looking at the edge from left to right. So the top two corners don't contribute anything. So the pixel does not depends on y, that's why it's removed. The formula of n(x) is a relationship between horizontal distances x_f. So the derivative is in terms of xf: n'(x)=-d00+d10. Where can you see that it jumps from d10-d00 to 'the next cell's derivative'? Now using the fade function, how is the resolved?

The new function reads: $n(x) = [1 - f(x_f)]d00 + f(x_f)d10$, now what does the derivative become? $n'(x) = -f'(x_f)d00 + f'(x_f)d10$.

Ok I think this is what GPT means:

1) The first simple linear case, $n'(x) = d10 - d00$, the difference between two bottom corner values. But the problem is, for the current blob, it will be one set of corner values (d10,d00), but for the next blob, it will be a different set of corner values. So the derivate at the corner from the left will be different from that from the right. The change is not smooth.
2) But the above is the math technicalities. I think we need to first SEE the problem from the graph visually. The rugginess, the jumpiness.
3) Then the visual symptom will have a math counterparts too, that's the jumpy derivative at the boundary!
4) Once the math symptom is identified, it can be tackled mathematically, by using a fade function to ensure smooth transition at the boundary. Once the math symptom is eliminated, the visual symptom is gone too, how amazing!

Ok, so basically we understand step4:

```
# -----------------------------
# STEP 4 — CLASSIC PERLIN (with fade curve)
#   Same as Step 3 but replace linear weights with smooth fade.
# -----------------------------
def fade(t):
    # 6t^5 - 15t^4 + 10t^3 : continuous up to 2nd derivative
    return t*t*t*(t*(t*6 - 15) + 10)

u = fade(xf)
v = fade(yf)

per_x1 = d00*(1-u) + d10*u
per_x2 = d01*(1-u) + d11*u
perlin_single = per_x1*(1-v) + per_x2*v

save_img("step4_perlin_single.png", norm01(perlin_single),
         "Step 4: Perlin single octave (with fade)")
```

not sure why it's called single octave? But let's move on.

Now let's do step 5:

```
# ----------------------------
# STEP 5 — fBM (stack multiple octaves of Perlin)
# ----------------------------
def perlin_from_grads(xf, yf, xi, yi, gx, gy):
    """Perlin single octave from precomputed grid/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]
    d00 = g00x*xf      + g00y*yf
    d10 = g10x*(xf-1.) + g10y*yf
    d01 = g01x*xf      + g01y*(yf-1.)
    d11 = g11x*(xf-1.) + g11y*(yf-1.)
    u = fade(xf); v = fade(yf)
    x1 = d00*(1-u) + d10*u
    x2 = d01*(1-u) + d11*u
    return x1*(1-v) + x2*v
```

```
def fbm_perlin(H, W, cell, octaves=6, lac=2.0, gain=0.5, rng=None):
    if rng is None:
        rng = np.random.RandomState(0)
    total = np.zeros((H, W), dtype=np.float64)
    amp = 1.0
    freq = 1.0
    amp_sum = 0.0
    # base grid size
    base_Gh, base_Gw = H // cell + 2, W // cell + 2
    for _ in range(octaves):
        Gh = int(base_Gh * freq)
        Gw = int(base_Gw * freq)
        # gradients for this octave
        grad_idx = rng.randint(0, len(GRADS), size=(Gh, Gw))
        gx = GRADS[grad_idx, 0]
        gy = GRADS[grad_idx, 1]
        # coordinates in this octave's grid space
        y = np.linspace(0, Gh-2, H)
        x = np.linspace(0, Gw-2, W)
        yy, xx = np.meshgrid(y, x, indexing='ij')
        xi = np.floor(xx).astype(int)
        yi = np.floor(yy).astype(int)
        xf = xx - xi
        yf = yy - yi
        total += amp * perlin_from_grads(xf, yf, xi, yi, gx, gy)
        amp_sum += amp
        amp *= gain
        freq *= lac
    return total / max(amp_sum, 1e-12)
```

```
fbm = fbm_perlin(H, W, cell=64, octaves=6, lac=2.0, gain=0.5, rng=rng)
save_img("step5_fbm.png", norm01(fbm), "Step 5: fBM (stacked Perlin)")
```

I did not go into the details, but read what GPT describes I think I got the main idea. In the above step, we chop the fine grid into many 32*32 blobs. Now what if we chop them into 16*16, 8*8, 4*4 blobs? And for each case, we use the same method to create the texture? Then for each case, the frequency of repetition is different? That's why they are referred to 'frequency'. The idea is to add up these multi-frequency picture together. Of course we can look at the end result and see the difference, but we need to understand why. It must be in line with how nature multiply itself!

Understand this high level, let's move on to step6. The code is just a few sentences:

```
# ----------------------------
# STEP 6 — RIDGED VARIANT (sharper mountains)
#   ridged = 1 - |fBM|
# ----------------------------
ridged = 1.0 - np.abs(fbm)
save_img("step6_ridged.png", norm01(ridged), "Step 6: Ridged (1 - |fBM|)")
```

Looks like it's inverting the pixel number with that '1-' subtraction? So pervious bright becomes dark and vice versa? Anyway, now step 7

```
# ----------------------------
# STEP 7 — DOMAIN WARP (organic twist)
#   Warp sample coords by a low-frequency field before evaluating fBM.
# ----------------------------
def domain_warp_base(H, W, cell_base=96, warp_strength=0.6, rng=None):
    if rng is None:
        rng = np.random.RandomState(1)
    # Low-frequency warp fields (u,v)
    fbm_u = fbm_perlin(H, W, cell=cell_base, octaves=4, lac=2.0, gain=0.5, rng=rng)
    fbm_v = fbm_perlin(H, W, cell=cell_base, octaves=4, lac=2.0, gain=0.5, rng=rng)
    u = (fbm_u - fbm_u.mean()) * warp_strength
    v = (fbm_v - fbm_v.mean()) * warp_strength
    return u, v

# base coordinate grid for a mid-frequency fBM
base = fbm_perlin(H, W, cell=48, octaves=6, lac=2.0, gain=0.5, rng=rng)

u, v = domain_warp_base(H, W, cell_base=96, warp_strength=0.7, rng=rng)

# Sample the base fBM at warped coords by re-running per octave with shifted coords
# (cheap approximation: shift the phase of the base image by u,v using grid sampling)
# We'll do a quick bilinear sampler for the 2D array "base".
# We'll do a quick bilinear sampler for the 2D array "base".
def bilinear_sample(img, x, y):
    H, W = img.shape
    x = np.clip(x, 0, W-1 - 1e-6)
    y = np.clip(y, 0, H-1 - 1e-6)
    xi = np.floor(x).astype(int)
    yi = np.floor(y).astype(int)
    xf = x - xi
    yf = y - yi
    v00 = img[yi,     xi    ]
    v10 = img[yi,     xi + 1]
    v01 = img[yi + 1, xi    ]
    v11 = img[yi + 1, xi + 1]
    x1 = v00*(1-xf) + v10*xf
    x2 = v01*(1-xf) + v11*xf
    return x1*(1-yf) + x2*yf

# Construct warped coordinates
yy0, xx0 = np.meshgrid(np.arange(H), np.arange(W), indexing='ij')
warped = bilinear_sample(base, xx0 + u*40, yy0 + v*40)  # scale displacement
save_img("step7_domain_warp.png", norm01(warped), "Step 7: Domain warp (organic)")

print("Saved:",
      *[os.path.abspath(p) for p in
        ["step1_white.png","step2_value_noise.png","step3_gradient_linear.png",
         "step4_perlin_single.png","step5_fbm.png","step6_ridged.png","step7_domain_warp.png"]],
      sep="\n  ")
```

PROJECT: simulate cloud emergence

Key math atoms:

1. Random number generator
2. Vectors:
   - The visual definition
   - Dot product
3. Concept of function is not that strong, but still something.
4. Derivative
   - When we diagnose the math symptoms of ‘jumpiness’.