**2) Downsample + Blur + Upsample**

Another **common trick** to achieve a large blur is:

1. **Downsample** the texture to a smaller resolution.
2. **Blur** at the smaller resolution with a *moderate* kernel size.
3. **Upsample** back to the original resolution.

This can drastically reduce the computational load, especially if your final usage only needs the “blurred look” rather than pixel-perfect fidelity.

**2.1 Downsample**

* For a **1024×1024** texture, you might downsample to **512×512** or even **256×256**.
* Each dimension halved → the area is ¼ of the original → significantly fewer pixels to process.

**Pseudo-code** (CPU approach):

cpp

Copy code

#include <algorithm>

void Downsample2x2\_RGBA8(

const uint8\* InPixels,

uint8\* OutPixels,

int InWidth, int InHeight, int BytesPerPixel

)

{

int OutWidth = InWidth / 2;

int OutHeight = InHeight / 2;

for (int y = 0; y < OutHeight; y++)

{

for (int x = 0; x < OutWidth; x++)

{

// We'll average a 2×2 block from the input

int inX = x \* 2;

int inY = y \* 2;

// Indices for the top-left pixel in the 2×2 block

int idx0 = (inY \* InWidth + inX) \* BytesPerPixel;

int idx1 = (inY \* InWidth + inX + 1) \* BytesPerPixel;

int idx2 = ((inY + 1) \* InWidth + inX) \* BytesPerPixel;

int idx3 = ((inY + 1) \* InWidth + inX + 1) \* BytesPerPixel;

// Accumulate RGBA

uint32 sumB = InPixels[idx0 + 0] + InPixels[idx1 + 0] + InPixels[idx2 + 0] + InPixels[idx3 + 0];

uint32 sumG = InPixels[idx0 + 1] + InPixels[idx1 + 1] + InPixels[idx2 + 1] + InPixels[idx3 + 1];

uint32 sumR = InPixels[idx0 + 2] + InPixels[idx1 + 2] + InPixels[idx2 + 2] + InPixels[idx3 + 2];

uint32 sumA = InPixels[idx0 + 3] + InPixels[idx1 + 3] + InPixels[idx2 + 3] + InPixels[idx3 + 3];

// Average

int outIdx = (y \* OutWidth + x) \* BytesPerPixel;

OutPixels[outIdx + 0] = (uint8)(sumB / 4);

OutPixels[outIdx + 1] = (uint8)(sumG / 4);

OutPixels[outIdx + 2] = (uint8)(sumR / 4);

OutPixels[outIdx + 3] = (uint8)(sumA / 4);

}

}

}

**More advanced**: You could use bicubic or Lanczos downsampling for higher quality, but 2×2 averaging is quick and often sufficient for a blur scenario.

**2.2 Blur the Downsampled Image**

* Now you have a **512×512** (or 256×256) image.
* You can do a smaller Gaussian kernel here (say 15×15 or even 9×9).
* Because there are fewer pixels, the same kernel is cheaper.

cpp

Copy code

// Now blur at the smaller size

GaussianBlurCPU(DownsampledPixels, OutWidth, OutHeight, KernelSize, Sigma);

**2.3 Upsample Back to Original Size**

* After blurring, **upsample** back to **1024×1024**.
* A simple approach is bilinear filtering:

cpp

Copy code

void UpsampleBilinear\_RGBA8(

const uint8\* InPixels,

uint8\* OutPixels,

int InWidth, int InHeight,

int OutWidth, int OutHeight,

int BytesPerPixel

)

{

for (int y = 0; y < OutHeight; y++)

{

// Map y to [0, InHeight - 1]

float v = (float)(y) \* (float)(InHeight - 1) / (float)(OutHeight - 1);

// "floor" and "fraction" for bilinear

int y0 = (int) v;

float fy = v - y0;

int y1 = std::min(y0 + 1, InHeight - 1);

for (int x = 0; x < OutWidth; x++)

{

float u = (float)(x) \* (float)(InWidth - 1) / (float)(OutWidth - 1);

int x0 = (int) u;

float fx = u - x0;

int x1 = std::min(x0 + 1, InWidth - 1);

// Indices for the four neighbors

int idx00 = (y0 \* InWidth + x0) \* BytesPerPixel;

int idx01 = (y0 \* InWidth + x1) \* BytesPerPixel;

int idx10 = (y1 \* InWidth + x0) \* BytesPerPixel;

int idx11 = (y1 \* InWidth + x1) \* BytesPerPixel;

// Bilinear interpolation

for (int c = 0; c < BytesPerPixel; c++)

{

float c00 = InPixels[idx00 + c];

float c01 = InPixels[idx01 + c];

float c10 = InPixels[idx10 + c];

float c11 = InPixels[idx11 + c];

float c0 = c00 + fx \* (c01 - c00);

float c1 = c10 + fx \* (c11 - c10);

float cFinal = c0 + fy \* (c1 - c0);

OutPixels[(y \* OutWidth + x) \* BytesPerPixel + c] =

(uint8)std::clamp<int>((int)(cFinal + 0.5f), 0, 255);

}

}

}

}

For real-time or near-real-time usage, you might do this on the **GPU** (e.g., via a simple fullscreen pass or compute shader).

**Putting It All Together**

cpp

Copy code

void DownsampleBlurUpsample(

uint8\* InOutPixels,

int Width, int Height,

int DownsampleFactor, // e.g., 2 or 4

int KernelSize, // e.g., 15

float Sigma // e.g., 5.0

)

{

// 1. Downsample

int DWidth = Width / DownsampleFactor;

int DHeight = Height / DownsampleFactor;

std::vector<uint8> Downsampled(DWidth \* DHeight \* 4);

Downsample2x2\_RGBA8(InOutPixels, Downsampled.data(), Width, Height, 4);

// For factor=2, you can do a repeated approach if factor=4, or a custom function.

// 2. Blur the downsampled data

GaussianBlurCPU(Downsampled.data(), DWidth, DHeight, KernelSize, Sigma);

// 3. Upsample back

std::vector<uint8> Upsampled(Width \* Height \* 4);

UpsampleBilinear\_RGBA8(

Downsampled.data(),

Upsampled.data(),

DWidth, DHeight,

Width, Height,

4

);

// 4. Copy back to InOutPixels

FMemory::Memcpy(InOutPixels, Upsampled.data(), Width \* Height \* 4);

}

* For **DownsampleFactor=2**, a 29×29 blur might be approximated by e.g. a 15×15 blur at half resolution. The effective blur radius is roughly doubled when scaled back up.
* Adjust the downsample factor, kernel size, and sigma to fine-tune visual fidelity vs. performance.

**Visual Accuracy Considerations**

1. **Iterative Smaller Gaussians**
   * Mathematically, convolving smaller Gaussians accumulates. If you pick the right standard deviations such that σ12+σ22+…≈σbig\sqrt{\sigma\_1^2 + \sigma\_2^2 + \dots} \approx \sigma\_{\text{big}}σ12​+σ22​+…​≈σbig​, you get close to the large-kernel result.
   * Typically, the result is quite close, especially for blur effects.
2. **Downsample + Blur + Upsample**
   * This can lose some high-frequency details. Because you’re discarding data in the downsample step, the blur might look slightly softer or have different edges compared to a direct 29×29 blur at full res.
   * Using bilinear or bicubic upsampling can help maintain smoothness.
   * If your pipeline requires a near-exact match to a 29×29 blur, this approximation might not be perfect, but visually it’s often “good enough” for many real-time scenarios.
3. **Combine Both**
   * You can downsample to half or quarter size, do 2 or 3 passes of a small Gaussian (like 5×5 or 7×7), then upsample. This can approximate a *very large* blur for a fraction of the cost.
4. **Profile and Adjust**
   * If the result isn’t quite strong enough, **increase** the kernel or sigma in the downsampled blur.
   * If it’s too blurry, **reduce** the kernel or sigma.
   * If you have enough GPU/CPU budget, consider a smaller downsample factor (like 2× instead of 4×) so you lose less detail.

**Summary of Approaches**

1. **Iterative Blurs**
   * Apply a 7×7 or 9×9 blur multiple times.
   * Tweak kernel size and number of passes to approximate a big 29×29 kernel.
2. **Downsample-Blur-Upsample**
   * Downsample (say 2×), blur with a moderate kernel (e.g., 15×15), then upsample.
   * Much cheaper than a full 29×29 blur at 1024×1024, but slightly less accurate.
   * For many real-time applications, the result is visually similar and significantly faster.

**Either or both** can help maintain near–real-time performance (like 10 times a second) while keeping a similar “big blur” effect. Ultimately, **profile** each approach on your target hardware and choose the best trade-off between quality and performance.