



Lightform FX

PLUGIN DEVELOPERS GUIDE

API v4 · Created by Michael S. Simone

Contents

1.	Introduction
2.	Anatomy of a Plugin
3.	Creating a Plugin Project
4.	Declaring Parameters
5.	Writing the Metal Kernel
6.	Packing Uniforms
7.	The Render Method
8.	Proxy-Aware Parameters
9.	Configurable Output Size
10.	On-Viewer Handles
11.	Validation
12.	Bundled Resources
13.	Multi-Pass Plugins
14.	Coordinate-Warp Kernels
15.	Custom Icons
16.	Unit Testing with MockRenderContext
17.	Building & Installing
18.	Writing Plugins in C++
19.	ABI Versioning & Debugging
20.	Full Sepia Source

Introduction

Lightform FX plugins are dynamic libraries that add nodes to the flowgraph. A plugin bundles a Metal compute kernel, a typed parameter schema, and optional extras (custom icons, viewer handles, validation hooks). The host compiles the Metal source once at load, builds the inspector UI from the schema, and dispatches the kernel whenever the node is evaluated.

The goal of the plugin API is that **a simple node is a short file**. The worked example in this guide — Sepia — is under 150 lines including the shader and illustrates every major capability the API offers. You only use the parts you need: a plain LUT-free color shift is just a `LightformPlugin` conformance and a kernel; a multi-pass blur or a node with on-viewer handles opts in to additional protocols.

This guide walks through each capability using Sepia as the running example. If you just want to see everything at once, skip to [Full Sepia Source](#).

What you need

- macOS 14 or later
- Xcode 15 (or the Swift 5.9+ toolchain)
- A Lightform FX install to run against
- Familiarity with Metal Shading Language (compute kernels, texture IO)

Anatomy of a Plugin

Every plugin conforms to `LightformPlugin`. That protocol fixes seven things:

- `identifier` — reverse-DNS string written into save files
- `displayName`, `category`, `version` — shown in the palette
- `inputSockets` — defaults to `[BG, FG, Matte]`
- `parameterSchema` — typed parameter list; host builds inspector UI from it
- `metalSource` + `metalEntryPoint` — kernel source and entry name
- `additionalEntryPoints` — extra kernels for multi-pass plugins (default: none)
- `packUniforms(_:)` — turns parameter values into the byte buffer the kernel reads

Everything else is opt-in via capability protocols:

- `IconProvidingPlugin` — custom SVG icon
- `ConfigurableOutputPlugin` — declare output dimensions (for generators)
- `ValidatablePlugin` — self-check at load time
- `ViewerHandleProvidingPlugin` — draggable handles on the viewer
- `ResourceLoadingPlugin` — read co-located files from the dylib's folder

Conformance to these is free — non-conformers simply don't get the feature. Old plugin binaries keep loading after new capabilities are added because nothing in `LightformPlugin` changes shape.

Creating a Plugin Project

A plugin is a Swift package that produces a dynamic library and depends on the

`LightformPluginAPI` package. Sepia's `Package.swift` :

```
// swift-tools-version:5.9
import PackageDescription

// Sample external plugin for Lightform FX. Builds a dynamic library that the
// host app loads at launch via dlopen. Copy the built .dylib into
// ~/Library/Application Support/Lightform/Plugins/ and restart Lightform.
//
// Build:
//   cd Plugins/Sepia && swift build -c release
//   cp .build/release/libSepia.dylib "$HOME/Library/Application Support/Lightform/Plu

let package = Package(
    name: "Sepia",
    platforms: [.macOS(.v14)],
    products: [
        .library(name: "Sepia", type: .dynamic, targets: ["Sepia"])
    ],
    dependencies: [
        // Depend on the standalone plugin-API package so this dylib links the
        // exact same LightformPluginAPI.dylib that the host loads at runtime.
        .package(path: "../../LightformPluginAPI")
    ],
    targets: [
        .target(
            name: "Sepia",
            dependencies: [
                .product(name: "LightformPluginAPI", package: "LightformPluginAPI")
            ],
            path: "Sources/Sepia"
        )
    ]
)
```

Key points:

- `.library(type: .dynamic, ...)` — produces a `.dylib` the host can `dlopen` .
- `.package(path: "../../LightformPluginAPI")` — link the same API package the host was built against. Version mismatches are caught at load time by the ABI sentinel.
- Minimum platform `.macOS(.v14)` matches the host.

Layout: one Swift file per plugin class is the common shape. Multiple plugins can live in the same dylib — the `lightform_plugin_create` C entry point returns a `LightformPluginList` so one package can ship a family of related nodes (keyers, tracker variants, etc.).

Declaring Parameters

`parameterSchema` is a list of `ParameterDescriptor` s. Each entry declares a key, a display name, a type (with range and default), and an optional caption that the inspector surfaces as a tooltip.

```
let parameterSchema: [ParameterDescriptor] = [
  ParameterDescriptor(
    key: "tint",
    displayName: "Tint",
    type: .float(min: 0, max: 1, defaultValue: 0.6),
    caption: "0 = original colors, 1 = full sepia at the falloff center."
  ),
  ParameterDescriptor(
    key: "falloff",
    displayName: "Falloff",
    type: .float(min: 0, max: 2, defaultValue: 0.8),
    caption: "How quickly the tint fades away from the center. 0 = uniform."
  ),
  ParameterDescriptor(
    key: "centerX",
    displayName: "Center X",
    type: .float(min: 0, max: 1, defaultValue: 0.5),
    caption: "Horizontal position of the tint center (drag the viewer handle)."
  ),
  ParameterDescriptor(
    key: "centerY",
    displayName: "Center Y",
    type: .float(min: 0, max: 1, defaultValue: 0.5),
    caption: "Vertical position of the tint center (drag the viewer handle)."
  ),
]
```

Parameter types

- `.float(min:max:defaultValue:)` — renders as a slider
- `.int(min:max:defaultValue:)` — stepper or slider
- `.bool(defaultValue:)` — checkbox
- `.color(defaultValue:)` — color well (RGBA scene-linear)
- `.enumeration(options:defaultIndex:)` — pop-up menu
- `.filePath(extensions:defaultValue:)` — file picker (LUTs, references)
- `.string(defaultValue:)` — text field

Keys are stable API.

Once a plugin ships, don't rename a key — project files reference parameters by string key. If a renamed key is loaded in an older save, the host silently falls back to the default.

Reading values

At render time the host passes a `ParameterValues` dictionary (`[String: ParameterValue]`).
Typed accessors make extraction painless:

```
let tint    = values.float("tint", default: 0.6)
let center  = values.color("center", default: .zero)
let useAlt  = values.bool("useAlt", default: false)
```


Writing the Metal Kernel

The kernel is plain Metal Shading Language. Every plugin that uses the default `render()` dispatches with the standard binding layout: **BG at texture(0), FG at texture(1), Matte at texture(2), Output at texture(3), uniforms at buffer(0).**

```
let metalSource = ""
#include <metal_stdlib>
using namespace metal;

// Members must match the order of `parameterSchema` – `UniformsLayout.pack`
// emits one float per scalar/bool/enum entry. Pad to 16 bytes at the end
// so `setBytes` is happy.
struct SepiaParams {
    float tint;
    float falloff;
    float centerX;
    float centerY;
};

kernel void sepia_kernel(
    texture2d<float, access::read> bg    [[texture(0)]],
    texture2d<float, access::read> fg    [[texture(1)]],
    texture2d<float, access::read> matte [[texture(2)]],
    texture2d<float, access::write> dst  [[texture(3)]],
    constant SepiaParams& p            [[buffer(0)]],
    uint2 gid                          [[thread_position_in_grid]])
{
    if (gid.x >= dst.get_width() || gid.y >= dst.get_height()) return;
    float4 c = bg.read(gid);

    // Normalized distance from the tint center. `falloff` scales it so
    // 0 keeps the tint uniform; larger values localize it tightly.
    float2 uv = float2(float(gid.x) / float(dst.get_width()),
                       float(gid.y) / float(dst.get_height()));
    float2 d = uv - float2(p.centerX, p.centerY);
    float r = length(d);
    float local = clamp(1.0 - r * p.falloff * 2.0, 0.0, 1.0);

    // BT.709 luminance + warm multiplier recipe.
    float gray = dot(c.rgb, float3(0.2126, 0.7152, 0.0722));
    float3 sepia = float3(gray) * float3(1.07, 0.82, 0.63);
    float mix_amt = clamp(p.tint, 0.0, 1.0) * local;
    dst.write(float4(mix(c.rgb, sepia, mix_amt), c.a), gid);
}
""
```

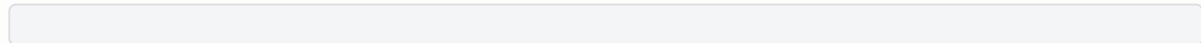
Notes on the kernel shape:

- Guard against out-of-bounds threads — `dispatchThreads` rounds up to the threadgroup size, so some threads land past the image edge.
- Input textures are `rgba32Float`, scene-linear, premultiplied by convention.
- Unconnected input sockets are bound to a 1×1 black texture, so sampling any slot is always safe.
- Output is written via `dst.write(..., gid)`. The host handles allocation — do not call `texture2d_array` sampling APIs that expect the output to be readable.

Packing Uniforms

The kernel reads parameters through a `constant Params&` pointer at `buffer(0)`. The plugin's `packUniforms(_:)` method produces that byte buffer. Metal's constant-buffer alignment rules are strict: `float3` and `float4` are both 16-byte aligned, `float2` is 8-byte aligned, and the struct itself is padded up to a multiple of 16 bytes. Getting this wrong silently corrupts uniforms.

The `UniformsLayout` helper does the arithmetic for you. The schema-driven form emits one scalar field per `parameterSchema` entry in declared order:



The kernel's `SepiaParams` struct just has to match the order of the schema's scalar entries:

```
struct SepiaParams {
    float tint;      // schema[0]
    float falloff;   // schema[1]
    float centerX;   // schema[2]
    float centerY;   // schema[3]
};
```

Manual packing

When you need vectors or arrays that don't map 1:1 to schema entries, use the explicit field form:

```
UniformsLayout.pack([
    .float(values.float("tint")),
    .float4(values.color("fillColor")),
    .int32(Int32(values.int("iterations"))),
])
```

Match the kernel struct field-for-field. The packer inserts alignment padding exactly where Metal expects it.

File-path and string parameters are skipped

by the schema-driven packer. Use them to resolve assets in `render()` (`ctx.loadLUT(path:)`, `file reads`) rather than trying to stuff them into the uniform buffer.

The Render Method

You don't have to implement `render()` at all — the default dispatches the kernel with the standard binding layout at the size of the BG input, or 1×1 if nothing is connected. Sepia uses the default.

If you need to do something custom (multi-pass, generator sizing, skip-dispatch when a switch is off), override it:

```
func render(_ ctx: RenderContext, values: ParameterValues) -> MTLTexture {
    let bg = ctx.input(at: 0)
    let out = ctx.makeOutputTexture(width: bg.width, height: bg.height)
    let u = packUniforms(values)
    u.withUnsafeBytes { raw in
        ctx.dispatchStandard(output: out,
                               uniforms: raw.baseAddress,
                               uniformLength: u.count)
    }
    return out
}
```

The `RenderContext` gives you everything the host has wired up for this render:

- `device` / `commandBuffer` — for manual Metal work
- `pipeline` — the compiled pipeline for the primary kernel
- `pipeline(named:)` — additional kernels (see Multi-Pass)
- `input(at:)` / `isConnected(at:)` — input textures and wiring state
- `makeOutputTexture(...)` / `makeIntermediateTexture(...)` — pooled textures
- `loadLUT(path:)` — parse + upload a `.cube` file, cached by path
- `fetchTemporal(socket:frame:)` — evaluate upstream at an arbitrary frame (for Retime, FrameBlend, Deflicker)
- `frameIndex` — the frame being rendered, 1-based
- `proxyScale` — current edit-time downscale factor (1.0 at full res, 0.5 at ½ proxy, etc.). See [Proxy-Aware Parameters](#).

Input textures are scene-linear.

The host applies any source-space decoding (sRGB EOTF, Cineon log, ARRI LogC, S-Log3, Log3G10, V-Log) at the Input node boundary, before `render()` is called. Plugins always operate on linear-light values regardless of how the user's source plates were encoded — do *not* apply your own log decode or gamma curve to `ctx.input(at:)` results. Output textures should also stay scene-linear; the viewer's display transform handles the encoding for monitor display.

Proxy-Aware Parameters

Lightform FX has an interactive proxy-resolution mode: users can drop the viewer to 1/2, 1/4, or 1/8 resolution for faster editing on heavy footage. Source plates are re-uploaded to the GPU at the reduced size, and every plugin evaluates against that smaller pixel count. The host automatically restores full resolution before any render or export.

Most plugins need to do nothing. If every parameter in your `parameterSchema` is already resolution-invariant — normalized (0..1), angles, percentages, enumerations, booleans, colors, mix factors — proxy mode is transparent to you. Your kernel runs on a smaller texture, produces correct output, done.

If your plugin has pixel-unit parameters — translate offsets expressed in pixels, blur radii, kernel sizes, displacement amplitudes, anything whose semantic meaning is "pixels in the final output" — you must multiply those values by `ctx.proxyScale` before packing them into your uniform buffer. Without it, a 100-pixel translate at 1/2 proxy lands 100 pixels on a half-sized canvas, which is double the intended visual offset.

Opt in via `PixelScaledParameters`

For plugins using the default `render()` (the common case, where you just implement `packUniforms`), conform to `PixelScaledParameters` and list your pixel-unit keys. The default render path pre-scales those values before `packUniforms` sees them, so your packing code stays unchanged.

```
final class DisplacePlugin: LightformPlugin, PixelScaledParameters {
    let parameterSchema: [ParameterDescriptor] = [
        ParameterDescriptor(key: "amount", displayName: "Amount (px)",
            type: .float(min: 0, max: 200, defaultValue: 20))
    ]
    let pixelScaledParameterKeys = ["amount"]

    // packUniforms, metalSource, etc. unchanged — "amount" arrives pre-scaled.
    func packUniforms(_ values: ParameterValues) -> Data {
        var p = SIMD4<Float>(values.float("amount"), 0, 0, 0)
        return Data(bytes: &p, count: MemoryLayout<SIMD4<Float>>.size)
    }
}
```

Only list keys that represent pixels in the final output. Listing a normalized or angle param here would shrink values that should stay constant — a bug that's invisible at Full but breaks at 1/2.

Manual scaling in overridden `render()`

Plugins that override `render()` (multi-pass, generators, coordinate-warp plugins using `dispatchIsolated`) should scale pixel params inline when they read them:

```
func render(_ ctx: RenderContext, values: ParameterValues) -> MTLTexture {
    // Translate offsets are pixel-unit – scale by the host's proxy factor.
    let tx = values.float("tx") * ctx.proxyScale
    let ty = values.float("ty") * ctx.proxyScale
    // ...pack, dispatch as usual
}
```

Or, if you'd rather keep the `PixelScaledParameters` -declarative style in an overridden `render()`, call the public helper `proxyScaled(_:plugin:proxyScale:)` on your values before `packUniforms`:

```
let scaled = proxyScaled(values, plugin: self, proxyScale: ctx.proxyScale)
let u = packUniforms(scaled)
```

What about generator sizes?

If your plugin conforms to `ConfigurableOutputPlugin` and returns `.fixed(width:height:)`, multiply those dimensions by `ctx.proxyScale` when proxy mode is active so the generator matches the scale of the rest of the graph. The default `ConfigurableOutputPlugin.render()` doesn't do this for you — because some generators (color charts, gradients) are intentionally resolution-independent — so it's a per-plugin call.

Configurable Output Size

Generators (Constant, Gradient, Noise) don't have a BG input to size against. Conform to `ConfigurableOutputPlugin` and declare the output size — the protocol's default `render()` still runs `dispatchStandard`, so you don't have to write the encoder boilerplate.

```
final class ConstantPlugin: LightformPlugin, ConfigurableOutputPlugin {
    let inputSockets: [InputSocket] = []    // pure generator

    func outputSize(values: ParameterValues) -> OutputSize {
        .fixed(width: values.int("width", default: 1920),
              height: values.int("height", default: 1080))
    }
    // ...parameterSchema, metalSource, packUniforms as usual
}
```

`OutputSize` cases:

- `.matchInput(socket: Int)` — clone the texture size at that socket (0 = BG, 1 = FG, 2 = Matte). Falls back to 1×1 when unconnected.
- `.fixed(width:height:)` — absolute size, regardless of inputs.

On-Viewer Handles

Some nodes are easier to dial in by dragging than by typing numbers. Sepia's "center" is a good example — the user wants to place the tint's focal point visually, not nudge a pair of X/Y sliders.

Conform to `ViewerHandleProvidingPlugin` and return a list of `ViewerHandle` s. The host renders a standard handle widget at each position and, on drag, writes the new normalized x/y back into the parameters you declared.

Fields on ViewerHandle

- `id` — stable identifier for this handle
- `normalizedPosition` — `SIMD2<Float>` in image space (0..1, origin top-left)
- `shape` — `.circle`, `.square`, or `.cross`
- `color` — fill color, RGBA scene-linear
- `xParamKey` / `yParamKey` — parameter keys the drag writes into. Either may be `nil` to lock that axis.

Parameters bound to handles

must be declared as `.float(min: 0, max: 1, ...)` — the host writes normalized 0..1 values directly, without range conversion. If the handle should represent pixel coordinates or world units, convert inside the kernel.

Validation

`ValidatablePlugin.validate()` runs once per plugin immediately after compile. Return an empty array for "OK" or a list of human-readable issues. The host logs each issue and marks the plugin with a compile-error status that the inspector can surface.

What's worth validating:

- Every parameter key your `packUniforms` / `render` reads actually exists in `parameterSchema` (renaming a key is the most common way to silently break a plugin).
- Your uniform struct byte count matches the kernel's `sizeof(Params)` (catch it early, not at the first render).
- Any asset the plugin expects to ship with the dylib exists when `resourceDirectory` is non-nil.

Bundled Resources

Sometimes a plugin needs data files beyond a `.cube` LUT — a noise texture, a pretrained weights blob, a wavetable. Conform to `ResourceLoadingPlugin` and the host will set `resourceDirectory` to the folder containing your dylib at load time.

```
final class NoisePlugin: LightformPlugin, ResourceLoadingPlugin {
    var resourceDirectory: URL? // host populates this

    func render(_ ctx: RenderContext, values: ParameterValues) -> MTLTexture {
        // Safe to call; returns nil in test harnesses where the path isn't set.
        if let data = loadResource(named: "blue_noise_64.exr") {
            // upload and use...
        }
        // ...
    }
}
```

Ship the resource alongside the dylib when distributing. For internal development, put it in the same folder you copy the build output into (typically `~/Library/Application Support/Lightform/Plugins/`).

Multi-Pass Plugins

Separable blurs, two-stage keyers, and any plugin that needs intermediate passes declare their extra kernels in `additionalEntryPoints` and dispatch them via `ctx.dispatch(named:...)`:

```
final class BlurPlugin: LightformPlugin {
    let metalEntryPoint = "blur_h"
    let additionalEntryPoints = ["blur_v"]

    let metalSource = """
#include <metal_stdlib>
using namespace metal;
kernel void blur_h(...) { ... }
kernel void blur_v(...) { ... }
"""

    func render(_ ctx: RenderContext, values: ParameterValues) -> MTLTexture {
        let bg = ctx.input(at: 0)
        let tmp = ctx.makeIntermediateTexture(width: bg.width, height: bg.height)
        let out = ctx.makeOutputTexture      (width: bg.width, height: bg.height)
        let u = packUniforms(values)
        u.withUnsafeBytes { raw in
            // Horizontal pass - bg → tmp - uses the primary pipeline via dispatchStar
            ctx.dispatchStandard(output: tmp, uniforms: raw.baseAddress,
                                uniformLength: u.count)

            // Vertical pass - tmp → out - uses the named extra pipeline.
            ctx.dispatch(named: "blur_v", textures: [tmp], output: out,
                        uniforms: raw.baseAddress, uniformLength: u.count)
        }
        return out
    }
}
```

Intermediate textures come from the host's pool (`ctx.makeIntermediateTexture`), so allocations are cheap and get recycled between frames automatically.

Coordinate-Warp Kernels

If your kernel samples `bg` at coordinates *derived from the output pixel* — lens distortion, custom displacement, optical-flow resample, radial warps, anything where a thread at `gid` reads `bg` at some other position — use `ctx.dispatchIsolated(...)` instead of `dispatchStandard`.

"Pixel-local" kernels (color grade, gamma, blend, keyer, blur) where each thread reads `bg` at its own `gid` should keep using `dispatchStandard`. Only coordinate-warp patterns need the isolated path.

```
func render(_ ctx: RenderContext, values: ParameterValues) -> MTLTexture {
    let bg = ctx.input(at: 0)
    let connected = ctx.isConnected(at: 0)
    let w = connected ? bg.width : 1
    let h = connected ? bg.height : 1

    let u = packUniforms(values)
    let result: MTLTexture? = u.withUnsafeBytes { raw in
        ctx.dispatchIsolated(width: w, height: h,
                             uniforms: raw.baseAddress,
                             uniformLength: u.count)
    }
    return result ?? ctx.makeOutputTexture(width: w, height: h)
}
```

What `dispatchIsolated` does for you

- Allocates the output as a **fresh** `.private` texture, bypassing the host's shared pool.
- Blit-copies `bg` (input slot 0) into a fresh scratch texture and binds the scratch at slot 0 in place of `bg`, materializing the input data with explicit GPU-side read-then-write ordering inside your command buffer before the kernel samples it.
- Encodes the compute dispatch against your primary `metalEntryPoint` with the three input slots and output slot bound exactly as `dispatchStandard` would.

When to reach for this

Pool-recycled textures plus coordinate-dependent sampling has produced a transient "random noise in the first ~second of playback" artifact on a small number of plugins. The fresh-texture + explicit blit pattern has been stable across every reproducer we've hit. The cost is one extra blit of the `bg` texture and two fresh allocations per render; negligible for per-frame work in a compositor but not free — the recommendation is to use it when your kernel matches the warp pattern, and use `dispatchStandard` otherwise.

Kernel signature.

Your kernel can still declare slot 0 as `texture2d<float, access::sample>` or `access::read` as you prefer. The helper binds a regular `rgba32Float` texture — both access modes work. For warp kernels, `access::sample` with a `constexpr sampler` is usually the better choice because it gives you free bilinear filtering at the warped coordinate.

Custom Icons

Plugins that don't provide an icon fall back to a generic puzzle-piece. Conform to `IconProvidingPlugin` and return an SVG string. Keep it self-contained — no external references, no scripts; the host rasterizes to a fixed size for the palette and flowgraph body.

The `LightformNodeIconTemplate` helper wraps any SVG symbol in the standard hardware-module chrome so your node reads as part of the same rack look as the built-ins:

```
let iconSVG: String = LightformNodeIconTemplate.svg(title: "SEPIA", symbol: """"
  <defs>
    <radialGradient id="sg" cx="50%" cy="45%" r="55%">
      <stop offset="0%" stop-color="#f4d8a8"/>
      <stop offset="55%" stop-color="#b88846"/>
      <stop offset="100%" stop-color="#5a3a1a"/>
    </radialGradient>
  </defs>
  <circle cx="48" cy="48" r="34" fill="url(#sg)" stroke="#3a2410" stroke-width="2"/>
  <path d="M30 58 Q48 40 66 58" fill="none" stroke="#3a2410" stroke-width="2.5" stroke-dasharray="5 5"/>
  <circle cx="40" cy="44" r="2.2" fill="#3a2410"/>
  <circle cx="56" cy="44" r="2.2" fill="#3a2410"/>
""")
```

The template expects your symbol to use a 0..96 coordinate system; it draws the symbol inside the "screen" area of a faceplate with a title label. Pass a short uppercase title and monochrome/limited-palette art — vivid colors read as LEDs against the cool-dark screen tint.

Unit Testing with MockRenderContext

The API package ships a `MockRenderContext` you can use from a plugin's test target. Give it a real Metal device and the input textures you want to exercise — `makeOutputTexture` allocates real `rgba32Float` textures you can read back and assert on.

```
import XCTest
import Metal
import LightformPluginAPI
@testable import Sepia

final class SepiaTests: XCTestCase {
    func testTintIsAppliedAtCenter() throws {
        let device = MTLCreateSystemDefaultDevice()!
        let queue = device.makeCommandQueue()!
        let cb = queue.makeCommandBuffer()!

        let plugin = SepiaPlugin()
        let lib = try device.makeLibrary(source: plugin.metalSource, options: nil)
        let fn = lib.makeFunction(name: plugin.metalEntryPoint)!
        let pso = try device.makeComputePipelineState(function: fn)

        let input = /* build a 16×16 rgba32Float test texture */
        let ctx = MockRenderContext(device: device,
                                    commandBuffer: cb,
                                    pipeline: pso,
                                    inputs: [input])

        let values: ParameterValues = [
            "tint": .float(1.0),
            "falloff": .float(0.0),
            "centerX": .float(0.5),
            "centerY": .float(0.5),
        ]
        let out = plugin.render(ctx, values: values)
        cb.commit(); cb.waitUntilCompleted()
        // Read out and assert on the center pixel's warm tint.
    }
}
```

For `packUniforms` -only tests (verifying schema/struct alignment), you don't even need a device — just call it and inspect the returned `Data`.

Building & Installing

From the plugin's package root:

```
swift build -c release
cp .build/release/libYourPlugin.dylib \
  "$HOME/Library/Application Support/Lightform/Plugins/"
```

Then restart Lightform FX. The plugin loader scans the directory on launch, ABI-checks each dylib, and registers every plugin it finds.

During development, keep the Lightform FX console open (via the Console.app "Lightform FX" process filter, or by launching from Xcode). Plugin load, compile errors, validate issues, and runtime errors all log there with the plugin's identifier as a prefix.

Iteration loop.

Lightform never calls `dlclose` on loaded plugins — Swift runtime type objects baked into your dylib stay referenced by already- created plugin instances. To reload, quit and relaunch the host. A short relaunch is cheap; attempting to hot-reload is not supported.

Writing Plugins in C++

If you'd rather work in C++, the SDK ships a parallel example `Examples/SepiaCpp/` that puts the kernel source, parameter table, uniform packing, and viewer-handle math in C++. A small Swift adapter (~30 lines of forwarding boilerplate) handles the `LightformPlugin` protocol conformance, since Swift protocols can't be implemented from C++ directly.

Why a Swift adapter at all? The host's plugin API is a Swift protocol. Conforming to it requires Swift code; there's no way to satisfy a Swift protocol from a pure C/C++ binary. The compromise: write everything that's actually per-plugin business logic in C++, and let the adapter forward.

Project layout

```
Examples/SepiaCpp/
├─ Package.swift           ← two targets: C++ core + Swift adapter
├─ Sources/
│   └─ SepiaCppCore/      ← pure C++ – author lives here
│       ├── include/
│       │   └─ SepiaCppCore.h ← C functions the adapter calls
│       └─ SepiaCppCore.cpp  ← kernel string, params, packing, handles
└─ SepiaCpp/              ← Swift adapter – copy unchanged
    └─ SepiaCppPlugin.swift ← forwards LightformPlugin → C functions
```

What goes where

- **C++ side** (you edit): plugin identifier & display name, Metal kernel source string, parameter table (key + range + default + caption), uniform-buffer packing, viewer-handle position math, validation. Anything that's actually plugin-specific.
- **Swift adapter** (you copy unchanged): conforms to `LightformPlugin` / `IconProvidingPlugin` / `ViewerHandleProvidingPlugin`, forwards each method to the C functions declared in the C header, exports the `lightform_plugin_create` / `lightform_plugin_abi_version` entry points.

Build & install

```
cd Examples/SepiaCpp
swift build -c release
cp .build/arm64-apple-macosx/release/libSepiaCpp.dylib \
  "$HOME/Library/Application Support/Lightform/Plugins/"
```

Relaunch Lightform FX; *Sepia* (C++) appears in the Color category of the node palette. The C++ target uses `cxxLanguageStandard: .cxx17`; bump in `Package.swift` if you need newer

features.

Starting your own C++ plugin

1. Copy the entire `Examples/SepiaCpp/` directory.
2. In the new directory, search-and-replace `Sepia / sepia` → your plugin name (case-sensitive — both Swift and C source).
3. Rewrite `Sources/<Yours>Core/<Yours>Core.cpp` : change the metadata strings, parameter table, uniform-packing logic, and Metal kernel source.
4. The Swift adapter under `Sources/<Yours>/` is generic — leave the body alone, just rename the type.

Crossing the Swift/C++ boundary

Keep the C interface narrow. The example uses only:

- `const char*` for strings (Swift converts via `String(cString:)`)
- `float` arrays for parameter values and handle coordinates
- `int` indices and counts
- A small POD struct for parameter descriptors

No SIMD types, no Swift enums, no `MTLTexture` — those stay on the Swift side. The Swift adapter packs `ParameterValues` into a flat float array in schema order before handing them to C++; the C++ side never has to know about Swift's parameter dictionary.

If you need to invoke `RenderContext` methods directly (custom `render()` override, multi-pass dispatch, viewer handles beyond a single XY pair), do that work in the Swift adapter and call into C++ for the per-pixel math. The adapter is a thin forwarder by default, but nothing stops you from putting a richer override in there.

Performance.

The C/Swift boundary is not a hot path. `packUniforms` runs once per frame, not per pixel. The actual kernel runs entirely on the GPU — Swift ↔ C++ has no involvement during pixel work. Expect identical performance to a pure-Swift plugin.

ABI Versioning & Debugging

Swift's protocols and enums don't have stable ABI. A plugin dylib built against an older `LightformPluginAPI` can crash inside the Swift standard library when the host loads it — witness-table slots and enum layouts shift between versions.

The host guards against this with a version sentinel. Every plugin exports:

```
@cdecl("lightform_plugin_abi_version")
public func lightform_plugin_abi_version() -> UInt32 {
    LIGHTFORM_PLUGIN_ABI_VERSION
}
```

If the value doesn't match the host's build, the plugin is refused at `dlopen` time with a readable log line — no crash, no partial registration. Bump happens when you rebuild against the current `LightformPluginAPI`; you don't touch the constant yourself.

Current ABI version: 4. Plugins built against ABI 3 or earlier need to be rebuilt against the current `LightformPluginAPI`. The bump to 4 added `RenderContext.proxyScale` for [proxy-resolution support](#) — most plugins need no code changes beyond a rebuild.

Common failures & what they mean

- **"missing lightform_plugin_abi_version"** — plugin built against an old API, rebuild.
- **"ABI mismatch (plugin=X, host=Y)"** — `LightformPluginAPI` package version differs, rebuild.
- **"dlopen failed"** — missing or broken dylib; check that it actually landed in the Plugins folder.
- **"compile failed for <id>"** — Metal shader doesn't compile; error message follows. Plugin is skipped; host still loads.
- **"entry point <name> not found"** — your `metalEntryPoint` or `additionalEntryPoints` string doesn't match any `kernel void` function in the source. Typo check.
- **"validate: ..."** — your own `validate()` returned issues; plugin still registers but is flagged.

Crashes inside Swift stdlib

when a plugin loads mean the ABI check was bypassed (the version sentinel is missing or the plugin was built with a mismatched compiler). Rebuild with the toolchain used for the host, and confirm the plugin exports `lightform_plugin_abi_version`.

Full Sepia Source

Reference: the complete `SepiaPlugin.swift`, shown for reading end-to-end. Every feature discussed in this guide appears here.

```

import Foundation
import simd
import LightformPluginAPI

/// Example external plugin: applies a sepia tone with a falloff centered on a
/// user-positioned viewer handle. Demonstrates the full plugin surface:
///
/// • Metal compute kernel with the standard BG/FG/Matte/Output binding layout.
/// • Typed parameter schema – the host auto-generates the inspector UI.
/// • `UniformsLayout.pack(values:schema:)` – schema-driven uniform packing,
///   no hand-padded Params struct needed.
/// • `ValidatablePlugin.validate()` – author-defined sanity checks, surfaced
///   at load time by the host.
/// • `ViewerHandleProvidingPlugin.viewerHandles(...)` – data-driven handle
///   that the host renders on the viewer and drags back into parameters.
/// • `IconProvidingPlugin.iconSVG` – custom palette/flowgraph icon.
final class SepiaPlugin: LightformPlugin,
                        IconProvidingPlugin,
                        ValidatablePlugin,
                        ViewerHandleProvidingPlugin {
    let identifier = "com.lightform.sample.sepia"
    let displayName = "Sepia"
    let category = "Color"
    let version = "2.0.0"

    let inputSockets: [InputSocket] = [.bg]

    let parameterSchema: [ParameterDescriptor] = [
        ParameterDescriptor(
            key: "tint",
            displayName: "Tint",
            type: .float(min: 0, max: 1, defaultValue: 0.6),
            caption: "0 = original colors, 1 = full sepia at the falloff center."
        ),
        ParameterDescriptor(
            key: "falloff",
            displayName: "Falloff",
            type: .float(min: 0, max: 2, defaultValue: 0.8),
            caption: "How quickly the tint fades away from the center. 0 = uniform."
        ),
        ParameterDescriptor(
            key: "centerX",
            displayName: "Center X",
            type: .float(min: 0, max: 1, defaultValue: 0.5),
            caption: "Horizontal position of the tint center (drag the viewer handle)."
        ),
        ParameterDescriptor(
            key: "centerY",
            displayName: "Center Y",
            type: .float(min: 0, max: 1, defaultValue: 0.5),
            caption: "Vertical position of the tint center (drag the viewer handle)."
        ),
    ],
}

```

```

let iconSVG: String = LightformNodeIconTemplate.svg(title: "SEPIA", symbol: ""
<defs>
  <radialGradient id="sg" cx="50%" cy="45%" r="55%">
    <stop offset="0%" stop-color="#f4d8a8"/>
    <stop offset="55%" stop-color="#b88846"/>
    <stop offset="100%" stop-color="#5a3a1a"/>
  </radialGradient>
</defs>
<circle cx="48" cy="48" r="34" fill="url(#sg)" stroke="#3a2410" stroke-width="2"/>
<path d="M30 58 Q48 40 66 58" fill="none" stroke="#3a2410" stroke-width="2.5" stro
<circle cx="40" cy="44" r="2.2" fill="#3a2410"/>
<circle cx="56" cy="44" r="2.2" fill="#3a2410"/>
""")

let metalEntryPoint = "sepia_kernel"
let metalSource = ""
#include <metal_stdlib>
using namespace metal;

// Members must match the order of `parameterSchema` - `UniformsLayout.pack`
// emits one float per scalar/bool/enum entry. Pad to 16 bytes at the end
// so `setBytes` is happy.
struct SepiaParams {
  float tint;
  float falloff;
  float centerX;
  float centerY;
};

kernel void sepia_kernel(
  texture2d<float, access::read> bg    [[texture(0)]],
  texture2d<float, access::read> fg    [[texture(1)]],
  texture2d<float, access::read> matte [[texture(2)]],
  texture2d<float, access::write> dst   [[texture(3)]],
  constant SepiaParams& p              [[buffer(0)]],
  uint2 gid                            [[thread_position_in_grid]])
{
  if (gid.x >= dst.get_width() || gid.y >= dst.get_height()) return;
  float4 c = bg.read(gid);

  // Normalized distance from the tint center. `falloff` scales it so
  // 0 keeps the tint uniform; larger values localize it tightly.
  float2 uv = float2(float(gid.x) / float(dst.get_width()),
                     float(gid.y) / float(dst.get_height()));
  float2 d = uv - float2(p.centerX, p.centerY);
  float r = length(d);
  float local = clamp(1.0 - r * p.falloff * 2.0, 0.0, 1.0);

  // BT.709 luminance + warm multiplier recipe.
  float gray = dot(c.rgb, float3(0.2126, 0.7152, 0.0722));
  float3 sepia = float3(gray) * float3(1.07, 0.82, 0.63);
  float mix_amt = clamp(p.tint, 0.0, 1.0) * local;

```

```

        dst.write(float4(mix(c.rgb, sepia, mix_amt), c.a), gid);
    }
    """"

func packUniforms(_ values: ParameterValues) -> Data {
    // Schema-driven packing. Each parameter in `parameterSchema` becomes
    // one uniform field in declared order – the kernel's `SepiaParams`
    // struct just has to follow the same layout.
    UniformsLayout.pack(values: values, schema: parameterSchema)
}

func validate() -> [String] {
    // Sanity check: every parameter key the kernel uses appears in the
    // schema. Catches typos renaming a param without updating the other
    // side. The host logs any returned strings at plugin-load time.
    let required = ["tint", "falloff", "centerX", "centerY"]
    let declared = Set(parameterSchema.map(\.key))
    return required.compactMap { declared.contains($0) ? nil : "missing parameter:"
}

func viewerHandles(values: ParameterValues) -> [ViewerHandle] {
    let x = values.float("centerX", default: 0.5)
    let y = values.float("centerY", default: 0.5)
    return [
        ViewerHandle(
            id: "center",
            normalizedPosition: SIMD2<Float>(x, y),
            shape: .circle,
            color: SIMD4<Float>(1.0, 0.8, 0.3, 1.0),
            xParamKey: "centerX",
            yParamKey: "centerY"
        )
    ]
}

}

/// Required C entry point. The host dlopen this dylib and calls
/// `lightform_plugin_create`; the returned opaque pointer is cast back to a
/// `LightformPluginList`. Returning a list (rather than a single plugin) lets
/// one dylib ship multiple related plugins.
@cdecl("lightform_plugin_create")
public func lightform_plugin_create() -> UnsafeMutableRawPointer {
    let list = LightformPluginList([SepiaPlugin()])
    return Unmanaged.passRetained(list).toOpaque()
}

/// Required ABI-version sentinel. Host refuses to load any plugin whose
/// reported version doesn't match `LIGHTFORM_PLUGIN_ABI_VERSION` at link time,
/// so stale dylibs fail fast with a readable log instead of crashing inside
/// the Swift stdlib.
@cdecl("lightform_plugin_abi_version")
public func lightform_plugin_abi_version() -> UInt32 {

```

LIGHTFORM_PLUGIN_ABI_VERSION

}