

Native Plug-ins for Hope Creative

A guide for developers of native Hope Creative plug-ins (`.hpx`): how the SDK works, what the host does for you, and how to build, test, sign and ship effects and modifiers.

SDK 1.0 · Plug-in API version 1 · Host version 1.0 (pre-release) · Windows x64 (macOS planned) · October 2026

Contents

1. What a native plug-in is
2. Quick facts
3. The SDK
4. Quick start
5. Anatomy of a bundle
6. Pictures, coordinates, colours and time
7. Parameters
8. GPU effects
9. CPU effects
10. Other moments and layer inputs
11. Modifiers
12. Instances, threads and determinism
13. Messages and crashes
14. Debugging
15. Where users find your plug-in
16. Packaging and signing (.hpx)
17. Developer Mode
18. macOS
19. Compatibility and versioning
20. Not supported yet
21. Testing checklist

1. What a native plug-in is

Hope Creative is a layer-based motion graphics and compositing application. Besides OpenFX plug-ins (see the *OFX Plug-in Guide*), it has its own native plug-in format: a compiled library (`.dll` on Windows, `.dylib` on macOS) built against the Hope Creative SDK and shipped as a signed `.hpx` package.

Native plug-ins are on the same footing as the app's built-in effects:

- **GPU effects** run on the app's own WebGPU device. You write WGSL shaders that run on Windows and macOS alike, and the host makes no copies of your pictures.
- **CPU effects** get the picture in memory and the host's thread pool. The trip to the CPU and back is optimised: about 7.5 ms for a 1080p frame, including the GPU readback.
- **Modifiers** change a property's value over time on top of its keyframes and expression, like the built-in Shake or Bounce.

Effects are applied to a **layer**. Your effect receives that layer's own picture (a video, a text, a shape, a solid...) before the layer is transformed into the composition. Effects on adjustment layers receive what lies below them. Native plug-ins run in Hope Creative and in Hope Creative Studio.

2. Quick facts

Item	Value
Interface	C (<code>hope_plugin.h</code>), every struct versioned by its size; optional C++ helpers

Item	Value
Entry point	one exported function: <code>const HopePluginV1* hope_plugin_v1(void)</code>
API version	<code>HOPE_API_VERSION</code> = 1
Entries per bundle	any number of effects (GPU or CPU) and modifiers
GPU	WebGPU (Dawn) on the host's device, WGSL shaders; all <code>wgpu*</code> functions from the host
Pixel format	RGBA, 32-bit float, premultiplied alpha; linear light, or display values on request
Coordinates	texture pixels, origin top left, y down
Resolution	multi-resolution: the texture follows the viewer's resolution (<code>scale</code>)
Tiles	no: you always get the whole texture
Temporal access	yes, announced in <code>framesNeeded</code>
Layer inputs	yes: other layers of the comp as input pictures
Viewer handles	points (crosshairs), automatically for every POINT parameter
Package	<code>.hpx</code> : encrypted, Ed25519-signed, one library per platform
Platform	Windows x64; macOS planned (build the dylib already, see section 18)

3. The SDK

The SDK comes with Hope Creative: **Developer ▶ Developer Resources ▶ Open SDK and Examples** opens it (folder *Plug-in SDK*), and the guides are in the same menu.

Path	What it is
<code>include/hope/hope_plugin.h</code>	the interface (C)
<code>include/hope/hope_plugin.hpp</code>	C++ helpers, header only, optional: parameters in one line, <code>ComputePass</code> , <code>workTexture</code> , <code>forRows</code>
<code>include/hope/hope_webgpu.h</code> <code>src/hope_webgpu.c</code>	WebGPU for your plug-in: compile the <code>.c</code> file into it; every <code>wgpu*</code> function then calls the host's
<code>include/webgpu/</code>	the WebGPU headers of the host's Dawn, with Dawn's header-only C++ wrapper <code>webgpu_cpp.h</code> (BSD licence included)
<code>cmake/HopePlugin.cmake</code>	<code>hope_add_plugin(<target> <sources>...)</code> : a correctly set up library target
<code>examples/starter/</code>	a bundle with a GPU effect (Duotone), a CPU effect (Posterize) and a modifier (Oscillate)
<code>templates/build-macos.yml</code>	a GitHub Actions workflow that builds the macOS library without a Mac
<code>tools/hopex-sign.exe</code>	certificates, packing, signing, license files

Link no WebGPU library of your own and don't define `WGPU_SHARED_LIBRARY`. Your plug-in imports nothing from Hope Creative. `hope_webgpu.c` looks every function up by name when the bundle loads, so the device and textures you're handed are always used with the host's own WebGPU.

4. Quick start

```
cd examples/starter
cmake -B build
cmake --build build --config Release
```

1. In Hope Creative, turn on **Developer ▶ Developer Mode**.
2. Copy `StarterPlugins.dll` into `Documents\Hope Creative\Plug-ins\` (a subfolder per bundle is fine).
3. Restart Hope Creative. **Effects ▶ Plugins ▶ Examples** shows Duotone and Posterize. Every number or point property offers Oscillate in its right-click menu under **Modify with**.

In Developer Mode the app loads your library from a copy, so you can rebuild while it runs. When a new build is there, the status line says so, and a restart loads it.

5. Anatomy of a bundle

```
#include "hope/hope_plugin.hpp" // C++ helpers (hope_plugin.h alone works too)

static HopeStatus describe(const HopeDescribeSuite* s, HopeDescriber* d) {
    if (hopeWebGpuLoad(s->host) != 0) return HOPE_FAILED; // only needed for GPU effects

    HopeEffectDesc e = {sizeof e};
    e.id = "glow"; // unique in the bundle, never renamed
    e.name = "Soft Glow"; // shown in the Effects panel
    e.category = "Your Company"; // your folder: Effects ▶ Plugins ▶ Your Company
    e.description = "A soft glow around bright parts: neon, highlights, dreamy looks.";
    e.flags = HOPE_EFFECT_GPU;
    e.createInstance = glowCreate; e.destroyInstance = glowDestroy;
    e.padding = glowPadding; e.renderGpu = glowRender;
    HopeEntry* entry = s->addEffect(d, &e);

    using namespace hope::param;
    hope::addParams(s, entry, {slider("radius", "Radius", 20, 0, 500),
                              percent("amount", "Amount", 60),
                              color("tint", "Tint", 1, 0.9, 0.7)});

    return HOPE_OK;
}

static const HopePluginV1 kPlugin = {sizeof kPlugin, HOPE_API_VERSION,
    "com.yourcompany.effects", "Your Effects", "Your Company", "1.2.0", describe};

extern "C" HOPE_PLUGIN_EXPORT const HopePluginV1* hope_plugin_v1(void) { return &kPlugin; }
```

- **Structs start with their size.** Set `structSize = sizeof(...)` on everything you hand in. Hosts read older and newer plug-ins correctly: each side reads only the fields both know.

- **Strings you hand in** (ids, names, menu entries) must live as long as the library is loaded: use string literals or static arrays. What the host hands to you lives until your call returns.
- **Ids are forever.** Projects store effects as `hpx.<bundle id>.<entry id>` and parameters as `.../<param id>`. Never rename an id that has shipped; add new ones instead. Ids use letters, digits, `_` and `-`; bundle ids also use dots (reverse domain: `com.yourcompany.effects`).
- `describe()` **is called once**, when the bundle loads. Keep it fast and free of side effects: no dialogs, no network. Do a license check lazily, at `createInstance` or the first render.
- **The description** is shown to users, and to Claude when someone works with Hope Creative through Claude. Write what the effect does and what it's good for, in a sentence or two.

6. Pictures, coordinates, colours and time

Every render gets a `HopeFrame` :

Field	Meaning
<code>width</code> , <code>height</code>	the texture: the layer's content at the preview resolution, plus <code>padding</code> on every side
<code>content</code>	where the layer's content lies in the texture (pixels, y down)
<code>padding</code>	texels of room on each side (what your <code>padding()</code> asked for, at most 2048, plus other effects' room)
<code>scale</code>	texels per layer pixel: 1 at full resolution, 0.5 at half... Multiply sizes given in layer pixels by it.
<code>time</code> , <code>fps</code>	the layer's own time in seconds and the comp's frame rate. It counts from the layer's start (where its source begins), as the built-in effects do: trimming the layer doesn't shift your effect. Retiming happens before you: you get the frame the layer shows.
<code>seed</code>	differs per effect instance, stays the same across frames: use it for randomness
<code>timeFromInPoint</code> , <code>length</code>	the same moment counted from the layer's in point (0 there, also after trimming and in each part of a split, as modifiers count), and the layer's length from in to out point. Use them for an effect that should start over with the layer, or offer the choice as a checkbox (for example "Start at In Point"). Newer than the first fields: check <code>HOPE_HAS(frame, HopeFrame, length)</code> before reading them.

- **Pixels** are RGBA, 32-bit float, premultiplied. Effects work in **linear light**, unless they set `HOPE_EFFECT_DISPLAY_VALUES` : then they get display-encoded values (the sRGB curve), as most image plug-ins of other hosts expect. This works on the GPU and the CPU: the host converts the source, the inputs and the result for you. Values above 1 are allowed (HDR) and kept.
- **Write every output pixel.** The output isn't cleared for you.
- **POINT values** come already mapped into the texture (padding and scale included). In `padding()` they come in layer pixels × scale (the texture isn't made yet), in `framesNeeded()` in layer pixels.
- **COLOR values** come in the space of your pixels: linear light, or display-encoded with `HOPE_EFFECT_DISPLAY_VALUES` . You never convert them yourself.

7. Parameters

Type	Value and use
HOPE_PARAM_FLOAT	a number with a slider (<code>getFloat</code>)
HOPE_PARAM_INT	a whole number (<code>getInt</code>)
HOPE_PARAM_ANGLE	degrees, clockwise
HOPE_PARAM_PERCENT	shown with %, handed over as shown (50 = 50 %)
HOPE_PARAM_CHECKBOX	0 / 1 (<code>getInt</code>)
HOPE_PARAM_MENU	the index of one of <code>options</code> (<code>getInt</code>)
HOPE_PARAM_COLOR	RGBA (<code>getColor</code>), see section 6
HOPE_PARAM_POINT	a point in the layer with a crosshair in the viewer (<code>getPoint</code>); default as a fraction of the layer
HOPE_PARAM_LAYER	another layer of the comp as an input picture (effects only)
HOPE_PARAM_VALUE	modifiers only: an amount in the property's units, one per dimension (<code>getValue</code>)

- `minValue` / `maxValue` : what the value may be (both 0: anything). `sliderMin` / `sliderMax` : the slider's usual range, when it's smaller.
- `group` puts a parameter into a collapsible nest; "Glow/Shape" nests a nest.
- `showIf` greys a parameter while others don't allow it: "glow" (on / not 0), "!glow", "mode=1|3", joined with `&`. Hope Creative greys rows instead of hiding them, so the panel doesn't jump.
- `flags` : `HOPE_PARAM_LOG_SCALE` (dragged by ratios: frequencies, sizes), `HOPE_PARAM_LAYER_SIZE` (a size whose default is a fraction of the layer), `HOPE_PARAM_NOT_ANIMATED` (no stopwatch).
- `hint` : a sentence about what the parameter does.

Every parameter gets keyframes, expressions, modifiers, undo and dragging like the app's own. You read the values of the moment you're rendering; you never interpolate keyframes yourself.

8. GPU effects

Set `HOPE_EFFECT_GPU` and `renderGpu`. You get the host's `device` and `queue`, an `encoder` to record into, the `source` texture and the `output` texture (both `RGBA32Float`, the size of the frame). The output can be bound as a storage texture.

```

const char* kShader = R"(
struct P { amount : f32, a : f32, b : f32, c : f32, };
@group(0) @binding(0) var src : texture_2d<f32>;
@group(0) @binding(1) var dst : texture_storage_2d<rgba32float, write>;
@group(0) @binding(2) var<uniform> p : P;
@compute @workgroup_size(16, 16)
fn main(@builtin(global_invocation_id) id : vec3<u32>) {
    let size = textureDimensions(src);
    if (id.x >= size.x || id.y >= size.y) { return; }
    let c = textureLoad(src, vec2<i32>(id.xy), 0);
    textureStore(dst, vec2<i32>(id.xy), vec4<f32>(mix(c.rgb, vec3<f32>(c.a) - c.rgb, p.amount), c.a));
}
)";

void* create(const HopeHostSuite*, WGPUDevice device) { return new hope::ComputePass(device,
kShader); }
void destroy(void* instance) { delete static_cast<hope::ComputePass*>(instance); }

HopeStatus render(void* instance, const HopeGpuRender* r) {
    const HopeFrame* f = r->frame;
    struct { float amount, pad[3]; } u{float(f->host->getFloat(f->params, "amount") / 100.0)};
    static_cast<const hope::ComputePass*>(instance)->run(r, &u, sizeof u);
    return HOPE_OK;
}

```

- **Compile pipelines in** `createInstance`, not per frame.
- **Several passes:** make your own textures (`hope::workTexture`, or `wgpuDeviceCreateTexture`), keep them in your instance while the size stays the same, and run one pass after the other into the encoder.
- You may do anything the device allows, including your own buffers and your own `wgpuQueueSubmit`.
- **Your encoder is submitted only if you return** `HOPE_OK` **or** `HOPE_INCOMPLETE`. If you return `HOPE_FAILED`, or if your plug-in crashes, nothing you recorded runs and the layer shows unchanged.
- **Room around the layer:** a glow or shadow that reaches past the layer returns the texels it needs from `padding()` (at the given `scale`). The texture then grows by that much on every side.

9. CPU effects

Set `HOPE_EFFECT_CPU` and `renderCpu`. `source` holds the whole texture, with its rows `rowFloats` apart. Always use `rowFloats`: it can be larger than `width × 4`. Write into `output` (packed, `width × 4` floats a row, top-down).

```

HopeStatus render(void*, const HopeCpuRender* r) {
    const HopeFrame* f = r->frame;
    const float gain = float(f->host->getFloat(f->params, "gain"));
    hope::forRows(f, [&](int y) { // all cores, a row each
        const float* in = r->source.pixels + size_t(y) * r->source.rowFloats;
        float* out = r->output + size_t(y) * size_t(f->width) * 4;
        for (int i = 0; i < f->width * 4; i += 4) {
            out[i] = in[i] * gain; out[i + 1] = in[i + 1] * gain; out[i + 2] = in[i + 2] * gain;
            out[i + 3] = in[i + 3];
        }
    });
    return HOPE_OK;
}

```

- **Use the host's threads:** `host->parallelFor(count, fn, user)` (Or `hope::forRows`). The pool is made once and the calling thread works along. Calls of `fn` run at the same time; don't call `parallelFor` from inside one.
- **Performance:** getting a 1080p picture to the CPU and back costs about 7.5 ms. A well-threaded sort of every row of a 1080p frame takes about 6 ms more and plays in real time.
- **Display values:** with `HOPE_EFFECT_DISPLAY_VALUES` you get and return display-encoded, premultiplied values (GPU effects too). This is the natural choice for ports of image plug-ins written for other hosts.

10. Other moments and layer inputs

Temporal effects

An effect that looks at other moments of its layer (echoes, trails, frame holds, motion analysis) announces them in `framesNeeded(host, params, time, fps, offsets, max)`, as seconds from `time`. It's a pure function of its arguments: the host asks while it works out the frame, before any render. In the render, `sourceAt(render, offset)` returns that moment's picture: the layer's content and the effects above yours, drawn where the layer is now.

If a moment isn't there, `sourceAt` returns null (GPU) or 0 (CPU). Draw a stand-in and return `HOPE_INCOMPLETE`: the frame isn't cached and is rendered again. With an accurate `framesNeeded`, this doesn't happen.

Layer inputs

A `HOPE_PARAM_LAYER` parameter lets the user pick another layer of the comp. `layerInput(render, id)` returns it as it sits in the comp (its transform, effects and mattes), drawn into your texture's space and size. With None picked, you get null (GPU) or 0 (CPU).

11. Modifiers

A modifier changes a property's value on top of its keyframes and expression. Users add it from a property's right-click menu, **Modify with**. Modifiers on the same property run top to bottom, and their settings get keyframes and undo like any property.

```
HopeStatus modify(const HopeModify* m) {
    double amplitude[3] = {};
    m->host->getValue(m->params, "amplitude", amplitude, m->dimensions); // per component
    const double wave = std::sin(2 * 3.14159265358979 * m->host->getFloat(m->params, "frequency") *
m->time);
    for (int i = 0; i < m->dimensions; ++i) m->value[i] += amplitude[i] * wave;
    return HOPE_OK;
}
```

- `flags`: `HOPE_MODIFIER_NUMBERS` (Rotation, Opacity, sliders) and / or `HOPE_MODIFIER_POINTS` (Position, Scale, Anchor Point).
- **Values come as the property stores them:** points in pixels with y down, Rotation in degrees, Scale and Opacity in %. `dimensions` is 1 or 2.
- **Settings** can be FLOAT, INT, ANGLE, PERCENT, CHECKBOX and VALUE. A VALUE is an amount per component in the property's units. Users type a point's amount as a share of the comp, and you get pixels.
- `keys` lists the property's keyframes (layer time). `keyed(modify, time, value, velocity)` gives the keyed curve alone, with its eases, at any time: its value and its change per second. That's what a bounce or an overshoot after a key needs.

- `time` counts from the layer's **in point** (0 there, also after trimming) and `length` runs from the in point to the out point, so a curve always fits the layer as it is now (unlike effects, which count from the layer's start). The key times in `keys` are on the same clock; `seed` differs per property, so two of your modifiers never move alike.

`modify()` is called very often: for every property it sits on, every frame, and the timeline's curves. It is called from any thread. Keep it fast, allocation-free and without state between calls.

12. Instances, threads and determinism

- **Instances:** `createInstance` is called when an effect instance first renders. `destroyInstance` is called when it hasn't been used for a while (about half a minute) or the effect is deleted. Keep pipelines, buffers and caches there, not in globals.
- **Threads:** calls for one instance never overlap. Different instances (another layer, the export, which renders with its own instances) may run at the same time on other threads.
- **Determinism:** the same values at the same time must give the same picture, whatever rendered before. Frames render in any order, ahead of time, more than once, and again for export, possibly with a fresh instance. Don't carry state from one frame to the next, and use `seed` instead of clocks or counters.

13. Messages and crashes

- `host->report(params, level, text) : HOPE_REPORT_LOG` goes to the session log. `HOPE_REPORT_WARNING` and `HOPE_REPORT_ERROR` also show in the status line, and in the effect's row in Effect Controls until its next render. Report a problem once, not every frame.
- **Crashes:** on Windows every call into your plug-in is protected. An access violation, a stack overflow or a C++ exception you let escape doesn't take Hope Creative along. Your bundle is turned off until the next start, its effects pass their picture through, its modifiers change nothing, and the user sees "... crashed (render, code ...) and is turned off until the next start". A crash in `describe()` makes the bundle "not usable".
- Users find the session logs under **Developer ▶ Show Log Files**; ask for them in bug reports. While you develop, the Plug-in Console shows the same live (section 14).

14. Debugging

Everything for developers is in the **Developer** menu: Developer Mode, the Plug-in Console, installing a `.hpx`, the folders for plug-ins, extensions and scripts, and the log files.

The Plug-in Console

Developer ▶ Plug-in Console opens a page beside Effects (it turns Developer Mode on if needed). It has two pages:

- **Messages:** what your plug-in says through `log()` and `report()`, its crashes, and every **WebGPU and WGSLError** of its calls. Each entry shows the time, your bundle, the effect and the layer it sits on. The same text again right after is counted (×12), not repeated. *Issues* shows only warnings and errors, *Copy* puts the list on the clipboard for a bug report.
- **Timing:** per effect instance, how many renders and how long they took on average and at most; per modifier, its calls. For GPU effects the time is that of recording your commands. Turn on *Measure GPU time* to see how

long the GPU runs them: the host then waits for the GPU around your effect, which is accurate but slows playback down.

```
f->host->log(f->params, "radius %.1f at scale %.2f, %d passes", radius, f->scale, passes);
```

`log()` is printf-style and goes to the console and the session log only, never to users' status line. Pass the render's `params` so the line shows which effect and layer it came from (or `nullptr`).

WebGPU and WGSL errors

The host catches the WebGPU errors of each of your calls where they happen. A typo in a shader comes back with its line and column, for example `Error while parsing WGSL: :3:27 error: unresolved value 'undefinedThing'`. A bind group that doesn't fit its pipeline comes back with the layout it expected.

- Errors in `createInstance` (where your shaders compile) are shown with every render of that effect, in the console and in the effect's row, until you fix them.
- A render whose commands are invalid is **never submitted**: the layer shows unchanged instead of a black or half-drawn picture, and the error says why.

A debugger

Build your plug-in with debug information (Debug, or `RelWithDebInfo`). Start Hope Creative, then in Visual Studio choose **Debug** ▶ **Attach to Process...** and pick `HopeCreative.exe`. Breakpoints in your code are then hit as usual. In Developer Mode the app loads your library from a copy, and the debugger finds its symbols through the path stored in the library, so keep the `.pdb` where you built it. Note that the crash protection catches crashes before a debugger sees them: to stop at the crashing line, turn on breaking on first-chance access violations (*Exception Settings* ▶ *Win32 Exceptions*).

Logs

Developer ▶ **Show Log Files** opens the folder of the session logs. Each session's log has the messages of all plug-ins, the crashes with the module and offset, and the app's own messages around them.

15. Where users find your plug-in

- **Effects panel**: category **Plugins**, in the folder you name in `category` (your company or product: "Hope" → Plugins ▶ Hope). Without one, your effects sit right in Plugins. Searching for the folder's name finds all of your effects.
- **Modifiers**: in the right-click menu of every property they fit, under **Modify with**.
- **Developer** ▶ **Plug-ins...** lists every bundle with its vendor, version, effects and modifiers, and who signed it. A user can turn a bundle off: its entries leave the lists at once, and at the next start it isn't loaded at all. Effects of it already in a project are kept with their settings ("Turned off in Tools > Plug-ins").
- A project with an effect whose plug-in isn't installed keeps the effect and its settings, and the layer passes through until the plug-in is back.

16. Packaging and signing (.hpx)

An `.hpx` is a single file holding your libraries for every platform. It is encrypted and signed with your developer certificate (Ed25519); Hope Creative never opens a package that was changed after signing.

1. **A folder** with `manifest.json` and the libraries:

```
{
  "type": "plugin",
  "id": "com.yourcompany.effects",
  "name": "Your Effects",
  "version": "1.2.0",
  "libraries": { "win64": "win64/YourEffects.dll", "macos": "macos/YourEffects.dylib" }
}
```

The `id` must be the one in `hope_plugin_v1()`; a package whose library says otherwise isn't loaded.

2. **A certificate**, once. Keep it and its password safe: every update must be signed with the same one.

```
hopex-sign cert create --name "Your Name" --organization "Your Company" --password ... --out
you.hopercert
```

3. **Pack and sign:**

```
hopex-sign pack <folder> --out YourEffects.hpx --cert you.hopercert --password ...
hopex-sign verify YourEffects.hpx
```

Installing: users double-click the `.hpx` or drop it on Hope Creative. A dialog shows the name, the version and who signed it. The package goes to `%APPDATA%\Hope Studios\Hope Creative\Plugin-ins\<id>\`, and the plug-in is there after a restart. At every start the package is checked again and its library unpacked next to it.

- **Unsigned packages** install and load only in Developer Mode.
- **Updates** must be signed with the certificate of the installed version; otherwise the install is refused.
- **Selling:** `hopex-sign license --cert you.hopercert --password ... --data '{"user": "..."}' --out license.json` signs license files your plug-in can check with your public key (`hopex-sign key you.hopercert`). How you deliver and check them is up to you.

17. Developer Mode

- Turn it on in **Developer ▶ Developer Mode** (it also governs script extensions). **Developer ▶ Open Plug-ins Folder** opens the folder below, **Developer ▶ Install Plug-in...** installs a `.hpx`.
- Libraries in `Documents\Hope Creative\Plugin-ins\` (directly or one folder deep) load without a package. They load from a copy, so your build can overwrite the original while Hope Creative runs. A new build is reported in the status line, and a restart loads it.
- DLLs your plug-in needs can sit next to it.
- Use the CMake template (`hope_add_plugin`). It links the C runtime statically, so your library needs no runtime DLLs on users' machines.

18. macOS

The macOS version of Hope Creative is planned. You can build the macOS library today with `templates/build-macos.yml` on GitHub Actions (no Mac needed). It produces one library for Apple silicon and Intel, which goes into your package's `macos/` folder. WGSL shaders and the C interface are the same on both platforms. Sign the library with your Developer ID and notarize it for your customers. On macOS the crash protection of section 13 isn't available; test well.

19. Compatibility and versioning

- `HOPE_API_VERSION` is 1. A host loads every plug-in built for its API version or an older one, and a plug-in built for a newer API is reported as such and left out.
- Struct sizes make additions safe in both directions: new fields are added at the end, and each side reads what both know.
- The WebGPU headers are those of the host's Dawn. `hope_webgpu.c` looks every function up by name, and `hopeWebGpuLoad` returns how many it didn't find (0: all of them). Rebuild with a new SDK when you want newer WebGPU features.
- Projects refer to your effects and parameters by id. Keep ids stable across versions, add parameters freely, and a project saved with an older version opens with the new parameters at their defaults.

20. Not supported yet

- Sound effects and 3D effects (deformers, geometry) as plug-in entries.
- Your own viewer drawings beyond point handles, and your own UI elements in Effect Controls.
- Menus and colours in modifier settings.
- Reloading a plug-in without a restart.
- Layer Styles (effects in comp space after the transform).

21. Testing checklist

- Full, half and quarter resolution give the same picture at their scale (sizes multiplied by `scale`).
- The result reaching past the layer: `padding()` asks for enough room, and nothing is cut at the edges.
- Text, shape, solid, video, image and adjustment layers all work. A layer's content needn't start at 0, 0, so use `content`.
- Scrubbing backwards and jumping around give the same frames as playing (determinism).
- An export matches the viewer.
- Two instances of your effect on different layers don't influence each other.
- Temporal effects: no stand-in frames after the first render (`framesNeeded` is accurate).
- Modifiers: fast with many properties; correct on Position (pixels) and on single numbers.
- Turning your bundle off and on again in **Developer ▶ Plug-ins...** works, and a project keeps your effects while it's off.
- The signed `.hpx` installs on a machine without Developer Mode.