slughorn User Guide

GPU-native Vector Rendering

Contents

1 slughorn User Guide **6**

2 What Is Slug **7**
2.1 Terathon's Slug Library . . . 7
2.2 Why Not Tessellation . . . 8
2.3 Why Not MSDF . . . 8

3 What Is slughorn **10**
3.1 slughorn + osgSlug . . . 10
3.2 The Name . . . 10

4 Who Is This For **12**
4.1 Not a Good Fit . . . 13

5 Core Concepts **14**
5.1 The Five Types at a Glance . . . 14
5.2 `slug_t`, `_cv`, and `cv()` — No Silent Double Narrowing . . . 14
5.3 Key . . . 15
5.4 Shape . . . 15
5.4.1 The `expand` parameter . . . 16
5.5 Layer . . . 16
5.5.1 Layer::Color . . . 16
5.5.2 Layer::Transform . . . 17
5.5.3 Layer::drawMode . . . 17
5.5.4 Layer::blendMode . . . 17
5.6 CompositeShape . . . 17
5.7 Atlas . . . 18
5.7.1 Atlas::TextureData . . . 18
5.8 The build() Lifecycle . . . 18
5.9 PackingStats . . . 19

6 Getting Data In: Backends **20**
6.1 The Single-Translation-Unit Pattern . . . 20
6.2 Choosing a Backend . . . 20
6.3 FreeType (`slughorn/freetype.hpp`) . . . 21
6.3.1 Font metrics . . . 21
6.3.2 High-level Helpers . . . 21
6.3.3 Low-level Helpers . . . 21
6.3.4 COLRv1 emoji . . . 22
6.3.5 LoadConfig . . . 22

6.4 NanoSVG (`slughorn/nanosvg.hpp`) . . . 23
6.4.1 High-level: file and string loading . . . 23
6.4.2 Mid-level: NSVGimage . . . 24
6.4.3 Low-level: single shape . . . 24
6.4.4 What NanoSVG Supports . . . 24
6.5 Cairo (`slughorn/cairo.hpp`) . . . 24
6.5.1 decomposePath and loadShape . . . 25
6.5.2 What Cairo Supports . . . 25
6.6 Skia (`slughorn/skia.hpp`) . . . 25
6.6.1 decomposePath and loadShape . . . 25
6.6.2 Stroke Expansion . . . 25
6.6.3 Conic segment handling . . . 26
6.7 The Local-Origin Convention . . . 26

7 The Canvas API 27
7.1 Path . . . 27
7.1.1 Path Commands . . . 27
7.1.2 Convenience Shape Helpers . . . 28
7.1.3 Transform Stack . . . 28
7.1.4 Stroke Expansion . . . 28
7.1.5 Arc-length Sampling . . . 28
7.1.6 Composing Paths . . . 29
7.2 Canvas . . . 29
7.2.1 Implicit vs Explicit Path . . . 29
7.2.2 Commit Verbs . . . 30
7.2.3 CompositeShape Management . . . 31
7.3 The Core Patterns . . . 31
7.3.1 Pattern 1 — Single-layer Fill (auto-key) . . . 31
7.3.2 Pattern 2 — Named Shape . . . 31
7.3.3 Pattern 3 — Multi-layer Composite . . . 32
7.3.4 Pattern 4 — Geometry-only Shape . . . 32
7.3.5 Pattern 5 — Stroke as Commit Verb . . . 32
7.3.6 Pattern 6 — strokePath + defineShape (Geometry-only Stroke) . . . 32
7.3.7 Pattern 7 — Transform stack (Baked Tick Marks) . . . 33
7.3.8 Pattern 8 — Explicit Pivot Origin . . . 33
7.3.9 Pattern 9 — Standalone Path + Arc-length Sampling . . . 34
7.4 The Scale Parameter . . . 34
7.5 Text Placement . . . 34
7.5.1 Vertical Anchoring — `TextAnchorY` . . . 35
7.5.2 Horizontal Alignment — `TextAlignX` . . . 35
7.5.3 Coordinate Space . . . 35
7.5.4 Dependency Design . . . 36
7.6 Gradient Quick Reference . . . 36

8 Mixing Backends **37**
8.1 Single Atlas, One Font . . . 37
8.2 Multi-Font Scenes . . . 37
8.3 Multiple Atlases in a Scene (osgSlug) . . . 38

9 Gradients **39**
9.1 The Gradient Texture . . . 39
9.2 Registering a Gradient . . . 39
9.2.1 Canvas API . . . 39
9.2.2 Direct — GradientInfo . . . 40
9.3 Stops . . . 40
9.4 The Three Types . . . 40
9.4.1 Linear . . . 40
9.4.2 Radial . . . 41
9.4.3 Sweep (Conic) . . . 41
9.5 The Type Discriminator . . . 41
9.6 Shader Considerations . . . 41
9.7 What Backends Handle For You . . . 42

10 Band Placement & Atlas Tuning **43**
10.1 How the Indirection Table Works . . . 43
10.2 Uniform Placement . . . 43
10.3 The SplitStrategy API . . . 44
10.4 `computeAdaptiveSplits` — A Cautionary Tale . . . 45
10.5 Manual Placement . . . 45
10.5.1 A Concrete Result . . . 46
10.6 The Offline Editor . . . 49
10.7 What's Next: A Cost-Based Strategy . . . 49

11 Serialization **51**
11.1 Writing . . . 51
11.2 Reading . . . 51
11.3 The Command-Line Tool . . . 52
11.4 Binary Container Layout . . . 52

12 Python **53**
12.1 Submodules . . . 53
12.1.1 slughorn.render . . . 53
12.1.2 slughorn.canvas . . . 53
12.1.3 slughorn.emoji . . . 53
12.1.4 slughorn.freetype . . . 54
12.1.5 slughorn.nanosvg . . . 54
12.2 The `bin/slughorn` CLI . . . 55
12.3 What's Coming . . . 56

13 Connecting to Your Graphics Backend **57**
13.1 The SSBO Path (Default) . . . 57
13.1.1 Atlas Shape Buffer — Binding 0 . . . 58
13.1.2 Layer Buffer — Binding 1 . . . 58
13.1.3 Vertex Attributes . . . 58
13.1.4 The Geometry Loop . . . 59
13.2 The GL3 Path . . . 59
13.3 Texture Upload . . . 60
13.4 Blend Mode . . . 62
13.5 Reference Implementations . . . 62

14 Interactivity **63**
14.1 Runtime Mutation via the Layer Buffer . . . 63
14.2 The effectId System . . . 63
14.3 Layer Masking . . . 64
14.4 SubdividedDrawable and Mesh-Level Animation . . . 64
14.5 Path-Following Animation . . . 64
14.6 Putting It Together: A Clock . . . 65

15 CPU Rendering **67**
15.1 What It Is . . . 67
15.2 What It's For . . . 67
15.3 Python Surface . . . 68
15.4 `Atlas::Shape::curves` . . . 68

16 Debugging & Diagnostics **70**
16.1 slughorn-side tools . . . 70
16.1.1 Python band visualizer . . . 70
16.1.2 CLI subcommands . . . 70
16.2 osgSlug-side Tools . . . 71
16.2.1 Debug modes — `osgSlug_debugMode` . . . 71
16.2.2 Layer masking — `osgSlug_layerMask` . . . 73
16.2.3 Text Rendering Flags . . . 73
16.2.4 Text Pixel Alignment — `OSGSLUG_TEXT_PIXEL_ALIGN` . . . 74
16.3 Workflow Notes . . . 74

17 How This All Started **76**
17.1 Where It's All Going . . . 77

18 AI Disclosure & Sponsorship **78**

19 Changelog **79**
19.1 v0.1.0 — First Release . . . 79

1 slughorn User Guide

slughorn and osgSlug are developed by AlphaPixel.

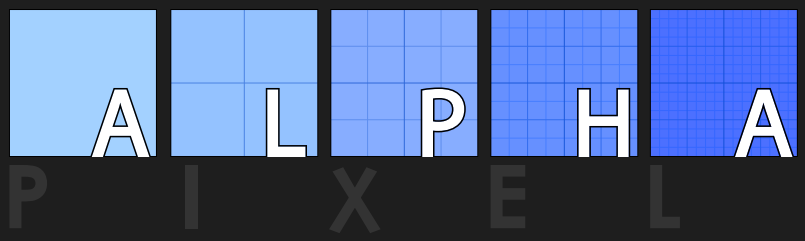


Figure 1: AlphaPixel

2 What Is Slug

"Slug" is a typesetting term: the full cast-metal line of type produced by a Linotype machine, delivered as one solid piece to the press. Terathon Software adopted it because the library's primary job is exactly that — laying out individual lines of text. The name says what it does.

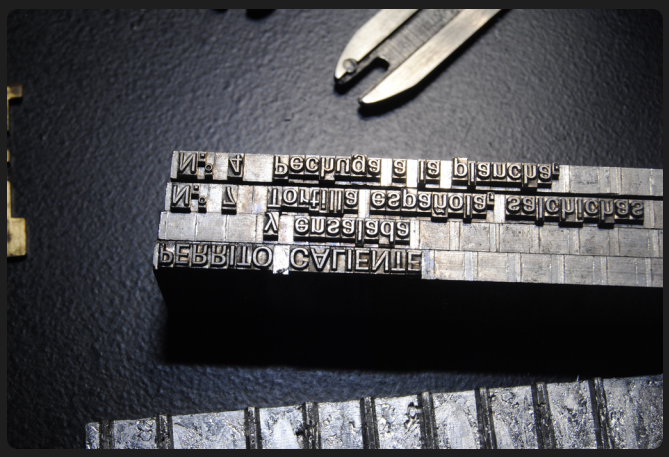


Figure 2: Linotype (by Antonio Molina)

The algorithm renders vector outlines entirely on the GPU at draw time. There are no precomputed glyph images and no signed distance fields. Every outline is composed of quadratic Bézier curves, and at render time the fragment shader evaluates the exact curve equations to determine whether each fragment is inside or outside the outline. The result is mathematically sharp at any scale or viewing angle — no rasterization artifacts, no SDF blurring, no re-tessellating for a new size.

Making per-fragment curve evaluation practical requires keeping the per-fragment work small. Slug achieves this through two GPU textures that the Atlas produces on `build()`:

- **Curve texture** (RGBA32F) — stores the quadratic Bézier control points for every shape packed into the atlas.
- **Band texture** (RGBA16UI) — divides each glyph's bounding box into narrow horizontal strips. Each strip records the small subset of curves that can possibly influence fragments within it. The fragment shader looks up the current fragment's band and evaluates only those few curves. A glyph with hundreds of control points may require only four or five evaluations per fragment. See Band Placement & Atlas Tuning for how band count and atlas width interact with packing efficiency.

2.1 Terathon's Slug Library

The Slug algorithm is the work of **Eric Lengyel** of Terathon Software LLC, and his decision to make the GLSL shader code publicly available is the direct reason slughorn can exist. None of this

— not the Atlas, not the backends, not a single line of the fragment math — would have been possible without that act of generosity. The technique was first described in Lengyel's 2017 Journal of Computer Graphics Techniques paper, *"GPU-Centered Font Rendering Directly from Glyph Outlines."* Terathon's Slug library ships a complete commercial product: a full text layout runtime covering kerning, ligatures, combining marks, 18 types of alternate substitution, bidirectional text, and paragraph layout; a font conversion tool that compiles `.ttf`/`.otf` fonts into `.slug` binaries; a vector graphics conversion tool for `.svg`/`.ovex` albums; and the reference GLSL shaders exposed via `GetFragmentShaderSourceCode()` / `GetVertexShaderSourceCode()`. If your project needs a complete, production-grade, commercially supported text layout engine, that is the product to evaluate.

slughorn is not Slug. slughorn is an independent open-source C++20 library that adopts the same two-texture GPU representation and uses shader mathematics derived from Lengyel's openly published work. It provides its own backends (FreeType, NanoSVG, Cairo, Skia, Canvas), its own `Atlas` abstraction and `.slugb` file format, and its own Canvas authoring API. For now, advanced text layout is Terathon's domain; if you need production-grade BiDi, ligature substitution, or paragraph layout today, that is the product to reach for. Comprehensive text support is on slughorn's horizon — but the work comes after what makes this library distinct, and the community's interest will shape how quickly that gap closes. Everything built in this guide rests on the foundation that Lengyel made accessible.

2.2 Why Not Tessellation

The straightforward alternative to curve-based rendering is tessellating glyph outlines — triangulating the filled regions and handing triangles to the GPU. This is how virtually all early vector-on-GPU approaches worked, and it has two problems that are fundamental rather than incidental.

The first is scale dependence. Triangles approximate curves; the quality of that approximation is fixed at the moment the triangles are generated. What looks smooth at 12 pt shows visible faceting at 200 pt. Avoiding that means re-tessellating every time the rendering size changes, or over-tessellating pessimistically for the largest size you might ever need — expensive, and still wrong the moment a size you didn't predict appears. The second is triangle count: achieving display-quality smoothness requires many triangles per glyph, and that cost multiplies with every shape on screen. Complex emoji or detailed SVG icons compound the problem quickly.

The band-based approach sidesteps both. Curve data is written to the atlas once at load time and is inherently resolution-independent — the same data produces correct results at any scale or under any perspective transform. The band texture keeps per-fragment shader cost bounded regardless of glyph complexity: only the curves that overlap the current horizontal strip need evaluating, not the entire outline.

2.3 Why Not MSDF

Multi-channel Signed Distance Fields are a genuine improvement over single-channel SDF, and the honest answer is: if you are rendering Latin text at a predictable size range in a flat 2D scene,

MSDF is excellent — simpler to integrate, cheaper per-fragment, and supported by a mature tooling ecosystem (msdfgen, msdf-atlas-gen, and many engine plugins). We are not going to pretend otherwise.

The question is whether it is the right default for what slughorn is actually for.

It is still a precomputed approximation. MSDF stores distance information out to a fixed *spread* radius baked into a texture at a chosen resolution. The multi-channel encoding dramatically reduces corner reconstruction errors compared to single-channel SDF, but it cannot recover detail finer than the spread captures, and it cannot recover precision lost to the texture's resolution. When a glyph is displayed far larger than it was baked — say, 10× its intended size — the distance field interpolation begins to soften edges noticeably. Avoiding this means rebaking at higher resolution, managing multiple atlas resolutions, or accepting the artifact. The Slug approach does not make this tradeoff: the same stored curves produce exact coverage at any scale without rebaking.

The spread/detail tradeoff has no single right answer. Thin strokes need a small spread for accurate reconstruction; thick shapes need a large spread for reliable coverage. A font with both fine-detail serifs and bold strokes requires a compromise that is wrong for at least one end of the range. Glyphs with complex overlapping topology — CJK, connected scripts, many SVG icons, HUD elements with intersecting subpaths — can confuse multi-channel reconstruction even at a spread value that works perfectly for simpler Latin glyphs. The Slug approach evaluates the same curve math regardless of topological complexity.

3D and perspective are where MSDF falls apart. The distance field is computed in flat 2D texture space: the shader samples it under the assumption that texture space and screen space have a consistent relationship. Under perspective projection — a near-plane widget vs. a far-plane label in VR, or text on a tilted panel in a 3D scene — that relationship breaks down. The sampling rate across the glyph is no longer uniform, and the field values that were computed for flat rendering arrive at the fragment slightly wrong. The resulting artifact is subtle, but in VR it contributes directly to eye strain. This is not a fixable artifact in the MSDF model; it is a consequence of the texture-space design. Resolution-independent curve evaluation sidesteps it entirely — the fragment shader computes correct in/out coverage from the actual curve geometry at every fragment, regardless of the viewing angle or distance.

It requires a separate preprocessing pipeline. msdfgen (or equivalent) must run over every glyph or shape, with resolution and spread chosen per asset, before rendering can begin. slughorn's Atlas handles all of this automatically as part of `build()`.

slughorn does include `msdfgen` as an optional backend (`SLUGHORN_MSDF`) for cases where pregenerated SDF data is useful as supplementary input. The two approaches are complementary rather than competing: MSDF is a practical choice when its constraints match the use case; the Slug-based path is the right default when they do not.

3 What Is slughorn

At its core, slughorn is a library for creating buffers of data (inside a `slughorn::Atlas`) based on standard vector graphics techniques and libraries whose purpose is to be consumed by a renderer using OpenGL, Vulkan, DirectX, etc. Whenever possible, slughorn is designed with the assumption that any parts of the vector graphics pipeline that *can* be offloaded to the GPU **will be**. Where dynamic content makes frequent GPU updates unavoidable, compute shaders are the recommended way to keep CPU overhead minimal (and we have examples of doing exactly that).

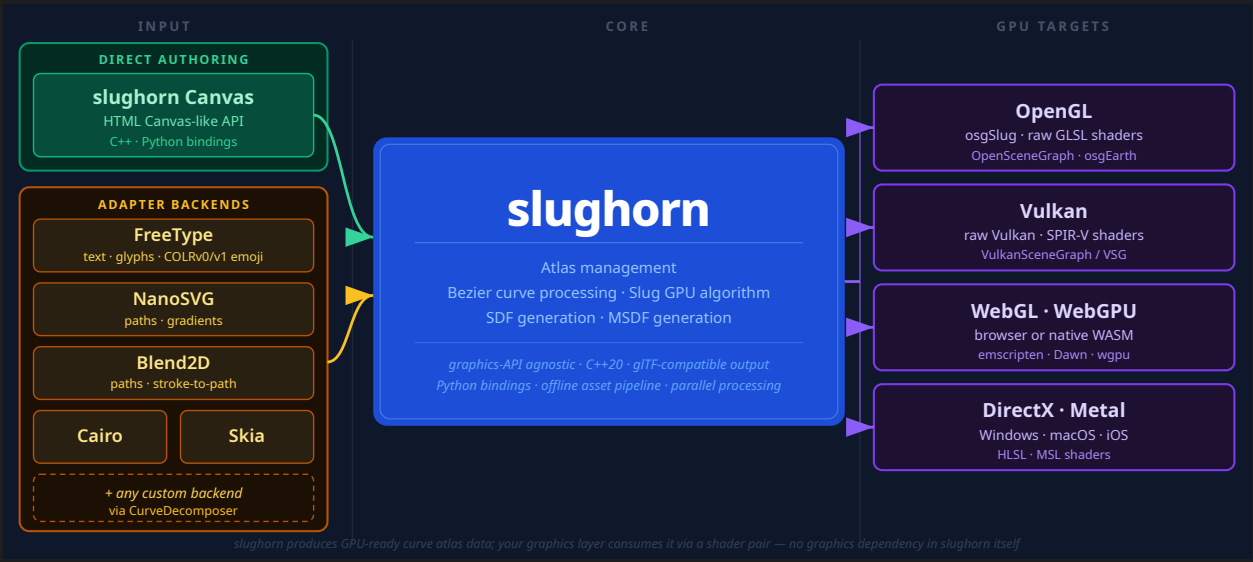


Figure 3: Architecture

3.1 slughorn + osgSlug

Throughout this guide you will see references to osgSlug, which is an OpenSceneGraph adapter for slughorn that has been developed in lock-step with the library from the very first day. It is the primary reference implementation and testbed — every GPU feature in slughorn was exercised first through osgSlug. When this guide shows GPU integration code, attribute layouts, or shader patterns, the examples come from osgSlug. The patterns apply equally to any custom renderer; osgSlug is just the one we can point to concretely.

In its current version, the data slughorn exports is tested in osgSlug using two distinct rendering approaches: a GL3-compatible path where all the necessary state is tightly packed into vertex attributes, and the preferred GL4-compatible path where "shader storage buffer objects" (SSBO) are used to represent both static and dynamic state.

3.2 The Name

The name carries two readings, and you are welcome to enjoy both.

The practical one: **slug** (the rendering algorithm) + **horn** (as in shoe-horn); essentially, "shoe-horning the Slug technique into different environments."

The other reading is for Harry Potter readers. There is a certain Hogwarts Potions professor whose surname also contains the word "slug," and whose fondness for collecting talented acquaintances — and crystallized pineapple — felt, to the author, like a reasonable metaphor for a library that collects rendering backends. The overlap was not accidental. No copyright lawyers were consulted in the naming of this project.

4 Who Is This For

slughorn is aimed at developers who need resolution-independent vector rendering and are willing to own their rendering pipeline. If you have a custom OpenGL, Vulkan, or Metal renderer and want GPU-sharp text and UI/HUD elements, this library is sized for that gap. The output is a few GPU textures and an SSBO/vertex attribute contract — everything else is yours.

Text and glyphs at any scale or perspective. If your application renders text in 3D space, under arbitrary transforms, or at sizes that vary widely at runtime, the Slug algorithm gives you mathematically exact coverage at every scale without re-baking. Note that slughorn is still catching up to Terathon's commercial Slug library in advanced text layout — full BiDi, ligature substitution, and paragraph layout are on the roadmap, and community interest will shape how quickly that gap closes.

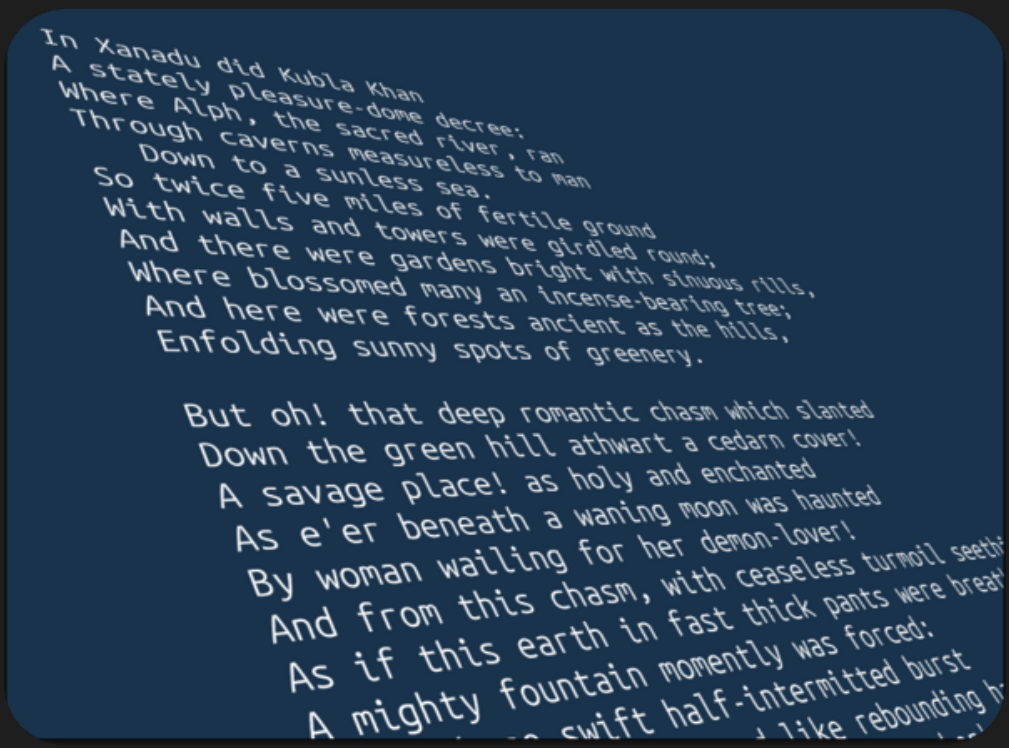


Figure 4: osgslug-text

HUDs, dashboards, and instrumentation. Gauges, arc charts, map overlays, connectors, animated widgets — anything a traditional HUD or vector canvas would need. This is the library's primary design target: resolution-independent UI geometry that is authored in code, baked at startup, and rendered frame-to-frame with no re-tessellation.

2D vector content from existing sources. If you already have geometry in FreeType, NanoSVG, Cairo, or Skia, the respective backends let you route it into slughorn's GPU representation without re-authoring. You do not have to abandon your current workflow to get resolution-independent

output. The Canvas API covers the case where you want to author shapes directly in C++ with an interface that will feel familiar from HTML Canvas.

VR and immersive displays. In VR, fuzzy UI and text cause eye strain and user dissatisfaction faster than almost any other visual artifact. Fragment-aware rendering stays sharp at any view distance or stereo perspective without re-baking — a natural fit for world-space widgets, HMD overlays, and spatial UI.

Offline asset pipelines. The `.slugb` serialization format lets you pre-bake a font atlas or icon set at build time and ship it alongside your application. The runtime never needs FreeType or any other font library present — just the atlas file and the shaders. This matters in constrained deployment environments where pulling in a font stack is not an option.

Python. slughorn's Python bindings are extensive: the full Atlas and Canvas API, gradient construction, serialization, and a pure-Python GLSL emulator for offline rendering and debugging. If you want to script your asset pipeline, inspect atlas packing, or drive the full rendering pipeline without touching C++, that path is well-supported. Alongside slughorn's Python bindings, an OpenSceneGraph Python integration (osgSlug.py) is in active development — a complete pipeline where atlas build, scene graph construction, and multipass rendering are all driven from Python.

4.1 Not a Good Fit

slughorn is probably the wrong choice if:

- You need **complete, production-grade BiDi, ligature substitution, or paragraph layout now.** That is Terathon Slug's domain, and it is the product to reach for if your text requirements are complex.
- You are building a **document layout engine** or rich-text editor. slughorn has no text flow, line breaking, or inline object model.
- You need **stroked shapes with round caps and joins at production quality.** Stroke support currently handles butt caps and miter/bevel joins. Round caps and joins are on the roadmap but not yet shipped.
- You want a **drop-in UI toolkit.** slughorn provides a rendering primitive, not a widget system. Layout, hit testing, and input handling are entirely the caller's responsibility.

> NOTE: As mentioned previously, slughorn will inevitably improve support for all of these areas as time goes on, depending largely on community interest/needs.

5 Core Concepts

slughorn organizes everything around five types. Understanding how they relate is the prerequisite for everything else in the library.

5.1 The Five Types at a Glance

Type	Role
`Key`	Stable name for a shape (codepoint or string)
`Shape`	One baked vector outline — band texture coords + metrics
`Layer`	One placed instance of a `Shape` — transform, color, gradient
`CompositeShape`	Ordered group of `Layers` with a layout advance
`Atlas`	Owns all shapes and their GPU textures

The data flow is one-way: you describe shapes to the `Atlas`, call `build()`, and the Atlas hands back raw GPU texture buffers. From that point on, rendering is entirely driven by `Layer` values that tell the GPU where and how to draw each shape.

5.2 `slug_t`, `_cv`, and `cv()` — No Silent Double Narrowing

Every numeric value in the slughorn API — coordinates, scale factors, color components, tolerance thresholds — has type `slug_t`. It is a `typedef` for `float`.

The Slug algorithm and its GPU shaders operate entirely in single precision. Matching that on the CPU side is not a style choice — it is the only way to guarantee that the values you specify are the values that reach the GPU. Early in slughorn's development, implicit `double → float` conversions produced subtle but real precision bugs: values that looked correct in a C++ debugger but arrived at the shader slightly wrong, causing rendering artifacts that were genuinely hard to trace back to a narrowing conversion.

The reason this gets its own section is that C++ will silently accept a `double` literal wherever a `float` is expected, truncate its precision, issue zero warnings, and move on. Write a bare `0.5` where you meant a slughorn value and you have a `double` quietly narrowed to `float`. **You do not want to deal with `double → float` size issues.** `slug_t`, `_cv`, and `cv()` exist so that you never have to. The name `cv` is short for *curve value* — or just *curve*, if that is what your brain prefers.

```
using namespace slughorn::literals;

// _cv: attach to any literal to produce a slug_t directly
```

```
slug_t x = 0.5_cv; // float — NOT a double narrowed to float
slug_t full = 1_cv; // prefer 1_cv over 1.0_cv

// cv(): static_cast<slug_t> — the right way to convert computed values
int i = 4;
slug_t t = cv(i) / 16_cv; // not float(i), not static_cast<float>(i)
```

When you see `float(i)` in slughorn context that is `cv(i)` spelled wrong. When you see `1.0f` that is `1_cv` spelled wrong. When you see a bare `1.0` assigned to a `slug_t` parameter — that is a `double` narrowing silently, which is exactly what this is here to prevent.

The `using namespace slughorn::literals;` declaration is assumed throughout this guide.

5.3 Key

A `Key` is a discriminated union that names a shape inside an `Atlas`. Two factory constructors cover the two use cases:

```
auto k1 = slughorn::Key(0x1F600); // glyph / emoji / your own number system
auto k2 = slughorn::Key("logo-body"); // arbitrary named shape
```

Keys are stable identifiers: they survive serialization round-trips (`.slug` / `.slugb`), tie Layers to Shapes at render time, and let you look up a Shape after `build()`. Every shape you add to the Atlas receives a Key at the moment it is added.

> The `Key` constructor is overloaded to support implicit conversion; any function that requires a `Key` can also accept a `string`/`uint32_t` in its place.

`KeyIterator` is a convenience factory that auto-increments so you rarely need to manage strings manually:

```
slughorn::KeyIterator ki("widget"); // produces "widget_0", "widget_1", …

auto ka = ki.next(); // "widget_0"
auto kb = ki.next(); // "widget_1"
```

5.4 Shape

A `Shape` is the baked output of one closed vector outline. It contains:

- Band texture coordinates — where in the Atlas GPU texture this shape lives
- Metrics: `bearingX`, `bearingY`, `width`, `height` (all in em-space)
- `computeQuad()` — converts a Layer's transform into a world-space bounding quad

You usually *don't* construct a `Shape` directly (although, it's entirely possible). Usually, backends (FreeType, NanoSVG, Cairo, the Canvas API) feed curve data into the Atlas, and the Atlas produces Shapes. After `build()` you retrieve a Shape by Key:

```
if(const auto shape = atlas.getShape(key); shape) {
    auto quad = shape->computeQuad(transform, scale, expand);
}
```

Shapes are resolution-independent. The same `Shape` can be rendered at any scale, with any transform, without re-baking.

5.4.1 The expand parameter

`computeQuad()` accepts an optional `expand` value (in em-space units) that enlarges the bounding quad by that amount on every side. It exists because the Slug fragment shader needs to sample slightly *outside* the shape's exact geometric bounds: anti-aliasing computes coverage by evaluating the distance field at the quad's corners and edges, and if the quad is clipped flush to the outline, fringe pixels at the boundary are discarded before the shader can soften them. A small margin — the default is `0.01` em-units — gives the algorithm enough room to produce a smooth result.

Set `expand` to `0` only when using `setAutoMetrics(false)` for GPU tiling, where em-coords must land exactly in `[0, 1]` and any margin would push them out of range.

5.5 Layer

A `Layer` is a placed instance of a `Shape`. It binds a `Key` to the rendering parameters/state needed to draw that shape once:

```
struct Layer {
    Key key; // which Shape to draw
    Color color; // RGBA flat color
    Transform transform; // x/y = world position; z reserved for depth
    slug_t scale; // em→world scaling (font size for text; 1.0 for SVG/Canvas)
    uint32_t effectId; // 0 = default; frontend hint
    slug_t effectParam; // 0 = default; frontend hint (per-layer float)
    uint32_t gradientId; // 0 = flat color; 1-based ID from addGradient()
    slug_t expand; // 0.01_cv = default
    DrawMode drawMode; // Visible = default
    BlendMode blendMode; // SrcOver = default
};
```

> `effectId` and `effectParam` are general-purpose frontend hints. slughorn does not interpret either value; they are passed wholesale to the renderer. `effectId` selects a named effect (e.g. rotation, glow); `effectParam` carries a per-layer scalar the effect needs (e.g. rotation speed, blur radius). They are the only fields in any slughorn POD struct that imply a backend-to-frontend relationship that slughorn itself does not define.

5.5.1 Layer::Color

When `gradientId` is non-zero, `color.rgb` is overridden by the gradient; only `color.a` (as a multiplier) remains active.

5.5.2 Layer::Transform

`x` and `y` place the shape in world space: `computeQuad()` reads them to offset the bounding quad. The frontend should pass them as the position component of any gradient lookup. `z` is a depth offset for 3D scene placement and defaults to `0`.

5.5.3 Layer::drawMode

Controls whether and how the layer participates in rendering:

- `Visible` — normal draw call (default)
- `Hidden` — shape is in the atlas but skipped at draw time; use for temporary toggles
- `Geometry` — curves are in the atlas but no quad is ever emitted; use when you need the curve data as a path source for sampling or animation without rendering the shape

5.5.4 Layer::blendMode

Controls how this layer composites over what is already in the framebuffer. `SrcOver` (premultiplied Porter-Duff over) is the default and the only mode guaranteed on all hardware. Photoshop-style modes (`Multiply`, `Screen`, etc.) require `GL_KHR_blend_equation_advanced`.

5.6 CompositeShape

A `CompositeShape` groups an ordered list of `Layers` with a single `advance` value (the horizontal advance in em-space used in text layout). `CompositeShape` is the highest-level type in slughorn, and is the mechanism by which simple `Shape` instances are grouped together with `Layers` and finally composited into something more dynamic.

> Emojis are the simplest way to visualize how `CompositeShape` works. A single rendered emoji maps to one `CompositeShape` containing several Layers stacked on top of each other (e.g., a COLRv1 emoji with a base outline, a color fill, and a gradient wash).

Beyond having a backend like FreeType/NanoSVG return a `CompositeShape` loaded from some on-disk file, the preferred way to build a `CompositeShape` is through the Canvas API (see `canvas.finalize()`). Once you have one, use the named accessor to configure specific layers rather than reaching into `.layers` directly:

```
auto clock = canvas.finalize();

clock.layer("hourHand").effectId   = 3;
clock.layer("minuteHand").effectId = 2;
clock.layer("secondHand").effectId = 1;
```

`layer(Key)` throws `std::out_of_range` if the key is not present. `operator[](size_t)`, `size()`, and `empty()` are also available for index-based access and container-style checks.

To compute the world-space bounding box of all participating layers, pass the built `Atlas` — making the query atlas-agnostic and safe to reuse with any compatible atlas:

```
if(auto bb = clock.boundingBox(*atlas)) {
    // bb->x0, bb->y0 (bottom-left)
    // bb->x1, bb->y1 (top-right)
}
```

When laying out a string, the caller iterates over codepoints, building up a `CompositeShape` and accumulating `advance` to compute each glyph's starting position.

5.7 Atlas

`Atlas` is the top-level owner. It accumulates shape descriptors from backends or direct calls, and on `build()` packs everything into two GPU textures:

- **Curve texture** — RGBA32F; stores the quadratic Bézier control points
- **Band texture** — RGBA16UI; stores the per-band curve index lists and indirection tables

```
slughorn::Atlas atlas; // default 512px texture width
slughorn::Atlas atlas(1024); // or specify a power-of-two width
```

5.7.1 Atlas::TextureData

After all shapes and gradients have been registered, call `build()`:

```
Atlas::TextureData td = atlas.build();
```

`TextureData` is plain byte buffers — no OpenGL state is touched. You upload the buffers to your GPU however your rendering layer prefers. `osgSlug` is the reference implementation; see Connecting to Your Graphics Backend for the full GL3/GL4 approaches.

After `build()`, the Atlas is immutable. To add new shapes you start a new Atlas. This is a deliberate design decision: static textures are cache-friendly and make GPU streaming predictable. For large glyph sets (CJK fonts), use font subsetting to keep the Atlas to a manageable size; see Band Placement & Atlas Tuning.

5.8 The build() Lifecycle

A complete slughorn session has six steps:

```
// 1. Create the Atlas
slughorn::Atlas atlas;

// 2. Add shapes via whichever backend(s) suit your content.
// Each backend returns Key values you use at render time.
// (See: Getting Data In: Backends)

// 3. Register gradients — must happen before build()
slughorn::GradientInfo gi;
```

```
gi.type = slughorn::GradientInfo::Type::Linear;
gi.stops = { {0_cv, red}, {1_cv, blue} };
gi.transform = slughorn::buildLinearGradientMatrix(x0, y0, x1, y1);

uint32_t gid = atlas.addGradient(gi); // returns a 1-based ID

// 4. Bake — packs curves and bands into GPU texture buffers
slughorn::Atlas::TextureData td = atlas.build();

// 5. Upload textures to your GPU (backend-specific)
// td.curveTexture — RGBA32F
// td.bandTexture — RGBA16UI

// 6. Render: for each Layer in each CompositeShape:
if(const auto shape = atlas.getShape(layer.key); shape) {
    auto q = shape->computeQuad(layer.transform, layer.scale, expand);
    // q.x0, q.y0 — bottom-left corner in world space
    // q.x1, q.y1 — top-right corner in world space
    // Emit quad vertices and set band/gradient vertex attributes from
    // shape and layer. See: Connecting to Your Graphics Backend.
}
```

5.9 PackingStats

`PackingStats` (accessible via `atlas.getPackingStats()` after `build()`) gives curve and band-texture utilization numbers useful for tuning and debugging. The `slughorn` Python CLI utility can also be used to query this information.

6 Getting Data In: Backends

Every backend does the same job: read curve data from an external source — a font file, an SVG document, a Cairo path, a Skia path — and register one or more shapes in the atlas. Each backend lives in its own header with no dependency beyond what that backend itself requires.

6.1 The Single-Translation-Unit Pattern

All backend headers are header-only with a gated implementation block. In exactly one `.cpp` file, define the implementation macro before including the header:

```
#define SLUGHORN_FREETYPE_IMPLEMENTATION
#include <slughorn/freetype.hpp>
```

All other translation units include the header without the define. No separate `.cpp` compilation unit is needed.

> The slughorn CMake configuration includes the ability to compile the backends directly into the resulting library; this simplifies "in-tree" usage of slughorn significantly (for example, when slughorn is included as a Git submodule).

6.2 Choosing a Backend

Backend	Header	When to use
FreeType	`slughorn/freetype.hpp`	TrueType/OpenType text and COLRv0/v1 emoji
NanoSVG	`slughorn/nanosvg.hpp`	SVG files/strings; no system dependency
Cairo	`slughorn/cairo.hpp`	Shapes authored via a `cairo_t*`
Skia	`slughorn/skia.hpp`	`SkPath` geometry; the only backend with stroke-to-fill expansion
Canvas	`slughorn/canvas.hpp`	Shapes authored directly in C++

The Canvas API is covered separately because it is slughorn's native authoring layer rather than a bridge to an external library. For new projects without existing font or SVG assets, start there.

6.3 FreeType (`slughorn/freetype.hpp`)

The FreeType backend decomposes TrueType/OpenType glyph outlines and COLRv0/v1 emoji into Atlas shapes. Requires FreeType 2 on the include and link path; no other graphics dependency.

6.3.1 Font metrics

`FontMetrics` carries dimensionless em-space ratios that scale to any font size. While the struct itself is defined in the `slughorn` top-level namespace (in order to support future backends "exporting" font metrics), the FreeType backend is currently the *only* one that does so.

```
// From an already-open face:
slughorn::FontMetrics m = slughorn::freetype::readFontMetrics(face);

// Or without managing an FT_Face yourself:
auto m = slughorn::freetype::loadFontMetrics("/usr/share/fonts/truetype/my.ttf");

slug_t capHeight = fontSize * m.capHeightRatio; // ~0.72 for Latin
slug_t lineHeight = fontSize * (1_cv + m.lineGapRatio);
```

All ratio fields are in `[0, 1]` as fractions of the em-square. `unitsPerEM` is the raw integer from the font's head table.

6.3.2 High-level Helpers

For the common case where you just want to load a font and move on:

```
slughorn::Atlas atlas;

// Printable ASCII (codepoints 32–126) — simplest path
slughorn::freetype::loadAsciiFont("/path/to/font.ttf", atlas);

// Explicit list of codepoints
slughorn::freetype::loadFontGlyphs(
    "/path/to/font.ttf",
    { 'A', 'B', 'C', 0x2603 }, // snowman included
    atlas
);

// Every glyph in the charmap (use with care on large fonts)
slughorn::freetype::loadAllFontGlyphs("/path/to/font.ttf", atlas);
```

6.3.3 Low-level Helpers

```
FT_Library ftLib; FT_Init_FreeType(&ftLib);
FT_Face face; FT_New_Face(ftLib, "/path/to/font.ttf", 0, &face);

slughorn::freetype::loadGlyph(face, 'A', atlas); // single codepoint
```

```
slughorn::freetype::loadGlyphRange(face, 0x0020, 0x007E, atlas); // contiguous range
slughorn::freetype::loadGlyphs(face, { 'A', 'Z', '0', '9' }, atlas); // explicit list
slughorn::freetype::loadAllGlyphs(face, atlas); // full charmap

FT_Done_Face(face);
FT_Done_FreeType(ftLib);
```

All low-level calls skip codepoints already present in the Atlas, so incremental loading is safe.

6.3.4 COLRv1 emoji

Color emojis expand into multi-layer `CompositeShapes`. The backend auto-detects COLRv1 (FreeType ≥ 2.11) and falls back to COLRv0:

```
// Obtain the palette (nullptr → resolveColor falls back to opaque white)
FT_Color* palette = nullptr;
FT_Palette_Data pd = {};

if(!FT_Palette_Data_Get(face, &pd) && pd.num_palettes > 0)
    FT_Palette_Select(face, 0, &palette);

// Single emoji
slughorn::CompositeShape cs;

slughorn::freetype::loadColorGlyph(face, 0x1F600, palette, atlas, cs);

// List of emojis
std::map<uint32_t, slughorn::CompositeShape> colorGlyphs;

slughorn::freetype::loadColorGlyphs(
    face, { 0x1F600, 0x1F4A9, 0x2764 }, palette, atlas, colorGlyphs
);

// High-level — no FT_Face to manage
slughorn::freetype::loadEmojiFont(
    "/path/to/NotoColorEmoji.ttf",
    { 0x1F600, 0x1F4A9 },
    atlas,
    colorGlyphs
);
```

`colorGlyphs` is keyed by codepoint. At render time, iterate `CompositeShape::layers` for each emoji just as you would for any other `CompositeShape`.

6.3.5 LoadConfig

All loading functions accept a `LoadConfig` struct as their final (optional) argument. Default construction is zero-cost and correct for the common case:

```
struct LoadConfig {
    Atlas::SplitStrategy strategy = {}; // optional band-split callable
    LogCallback log = {}; // optional diagnostic callback
    bool uniform = false; // shared em-space bbox for all glyphs
};
```

Logging — pass a `log` callback to diagnose failures without global state:

```
slughorn::freetype::loadAsciiFont(fontPath, atlas, {
    .log = [](int level, const std::string& msg) {
        if(level >= slughorn::freetype::LOG_WARN) std::cerr << msg << "\n";
    }
});
```

Log levels: `LOG_INFO = 0`, `LOG_NOTICE = 1`, `LOG_WARN = 2`.

Uniform bounding box — set `uniform = true` when all glyphs in a batch must share the same em-space bounding box. This is required for `setLayerShapeIndex` glyph-swap cycling (e.g. an animated digit counter). The backend auto-detects tabular advances and falls back to union-bbox centering when they differ:

```
slughorn::freetype::loadFontGlyphs(
    fontPath,
    { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' },
    atlas,
    { .uniform = true }
);
```

6.4 NanoSVG (`slughorn/nanosvg.hpp`)

The NanoSVG backend parses SVG files or strings and produces a `CompositeShape` with one `Layer` per filled SVG shape, back-to-front order preserved. NanoSVG is a single-header C library bundled with slughorn — no system dependency.

6.4.1 High-level: file and string loading

```
slughorn::Atlas atlas;
slughorn::KeyIterator(0x1234); // Or, something like "MySVG"...

auto cs0 = slughorn::nanosvg::loadFile("icon.svg", atlas, keyBase);
auto cs1 = slughorn::nanosvg::loadString(svgString, atlas, keyBase);
```

SVG coordinates are always normalized to `[0, 1]` em-space using `scale = 1 / image->width`. To recover authoring-space positions use `pixel_x = layer.transform.x * cfg.width`. World sizing is the caller's responsibility at render time via `Layer::scale` or a world transform.

`keyBase` is incremented for each shape registered. If you load multiple SVGs into the same Atlas, pass the same `keyBase` variable to each call and key allocations will not collide.

When a `KeyIterator` with a named prefix is used, any SVG shape that carries its own `id` attribute (`<path id="arrow-left" …>`) is registered under that id directly — the iterator is only consulted for anonymous shapes. To suppress this and force the iterator's own naming scheme regardless of what the SVG says, pass `true` as the second constructor argument:

```
// Uses SVG element ids where present; falls back to "icon_0", "icon_1", …
slughorn::KeyIterator keys("icon");

// Always uses "brand_0", "brand_1", … — SVG ids are ignored
slughorn::KeyIterator forced("brand", true);
```

6.4.2 Mid-level: NSVGimage

If you already have an `NSVGimage*`:

```
auto* image = nsvgParseFromFile("icon.svg", "px", 96_cv);
auto cs = slughorn::nanosvg::loadImage(image, atlas, keyBase);

nsvgDelete(image);
```

6.4.3 Low-level: single shape

`decomposePath` and `loadShape` operate on a single `NSVGshape*`:

```
for(const auto* shape = image->shapes; shape; shape = shape->next) {
    auto transform = slughorn::nanosvg::loadShape(shape, atlas, keyIterator);

    if(transform) layer.transform = *transform;
}
```

6.4.4 What NanoSVG Supports

Supported: filled shapes with solid color or linear/radial gradients; lines and cubics (split to quadratics at load time).

Not yet: stroked shapes, sweep/conic gradients, gradient `spreadMethod` reflect/repeat, clip paths, masks, group opacity/transforms, text elements.

6.5 Cairo (`slughorn/cairo.hpp`)

The Cairo backend decomposes the active path on a `cairo_t*` into slughorn curves. Use it when you are already generating geometry via Cairo and want to route it into the Atlas without re-authoring.

6.5.1 decomposePath and loadShape

```
cairo_t* cr = /* your cairo context */;

// Low-level: get curves and the local-origin offset matrix
auto [info, transform] = slughorn::cairo::decomposePath(cr, scale);

// Mid-level: register directly in the Atlas
Transform transform = slughorn::cairo::loadShape(cr, atlas, "my_shape", scale);

// Copy transform.x / y into layer.transform.x / y.
```

`scale` converts authoring-space coordinates into em-space. Any value works — `build()` computes `bandScaleX = numBands / em_width` and absorbs whatever scale was chosen. Pass `1_cv` to store coordinates as-is; pass `1_cv / 100_cv` if your Cairo canvas is drawn in a 100-unit space and you want em-space in `[0, 1]`.

6.5.2 What Cairo Supports

Cairo's native curve primitive is the cubic Bézier (`CAIRO_PATH_CURVE_TO`). The backend splits each cubic at its midpoint into two quadratics. `cairo_stroke_to_path()` is not part of Cairo's stable public API; for stroke outlines, use the Skia backend or author stroked shapes as explicit filled paths.

6.6 Skia (`slughorn/skia.hpp`)

The Skia backend decomposes `SkPath` objects and is the only backend with stroke-to-fill expansion via `skpathutils::FillPathWithPaint`.

6.6.1 decomposePath and loadShape

```
SkPath path;

path.moveTo(10, 10);
path.lineTo(90, 90);

// Low-level
auto [info, transform] = slughorn::skia::decomposePath(path, scale);

// Mid-level
auto transform = slughorn::skia::loadShape(path, atlas, "my_shape", scale);
```

6.6.2 Stroke Expansion

Until the Canvas API improves, Skia is currently the only backend in slughorn that can produce strokes with completely accurate joins/caps (although `slughorn::canvas` **will** eventually catch up).

```
// Expand a stroked outline into a filled path, then load it:
Transform transform = slughorn::skia::loadStrokedShape(
    path,
    atlas,
    "my_stroked_shape",
    8_cv, // stroke width
    scale,
    SkPaint::kRound_Join, // join style (default)
    SkPaint::kRound_Cap // cap style (default)
);

// Or as two steps — useful when you need to inspect the filled path first:
auto filled = slughorn::skia::strokeToFill(path, 8_cv);
auto transform = slughorn::skia::loadShape(filled, atlas, key, scale);
```

6.6.3 Conic segment handling

Skia paths may contain rational quadratic (conic) segments — circular arcs use weight `w = cos(angle/2)`. When `w == 1` the conic is an ordinary quadratic and passes through unchanged. When `w != 1`, the backend splits it at `t = 0.5` into two ordinary quadratics. This is transparent to callers.

6.7 The Local-Origin Convention

By default (`autoMetrics=true`) all backends shift curves so the shape's bounding box sits at `(0, 0)` inside the Atlas texture. The canvas-space offset that was subtracted is returned as a `Transform` (x/y set). Copy it into `layer.transform`; at render time, `Shape::computeQuad` adds it back to reconstruct the correct world position.

Pass `autoMetrics=false` to suppress the shift entirely — curves are stored exactly as authored and the returned `Transform` is zero. This is required for GPU tiling via `fract()`, where em-coords must remain in `[0, 1]`.

Passing `Origin::Centered` to any backend stores the bounding-box center instead of the corner, turning `layer.transform.x` / `y` into a GPU-side rotation pivot.

7 The Canvas API

The Canvas API is slughorn's native path-authoring layer. Its interface will feel familiar to anyone who has used HTML Canvas or SVG: you describe shapes with `moveTo` / `lineTo` / `arc` / `bezierTo` verbs, then *commit* them into the Atlas with `fill`, `stroke`, or `defineShape`. What you get back is a fully baked Atlas entry — resolution-independent, GPU-ready — rather than a retained display list.

Every builder method on both `Path` and `Canvas` returns `*this`, so calls can be chained, while commit verbs (`fill`, `stroke`, `defineShape`, `fillGradient`, `strokeGradient`) act as natural chain terminators, often returning a `Key` into the atlas. `finalize()` returns a `CompositeShape`, acting as a kind of "reset" during authoring.

Two cooperating types do all the work:

Type	Role
`Path`	Geometry state and authoring verbs; equivalent to HTML Canvas `Path2D`
`Canvas`	Stateful `CompositeShape` builder; holds an implicit `Path` and exposes commit verbs

7.1 Path

`Path` owns curve geometry and the current transform matrix (CTM). It can be used standalone (built and sampled independently) or through the `Canvas` forwarding API.

7.1.1 Path Commands

```
path.moveTo(x, y);
path.lineTo(x, y);
path.quadTo(cx, cy, x, y); // quadratic Bézier
path.bezierTo(c1x, c1y, c2x, c2y, x, y); // cubic Bézier
path.closePath(); // close sub-path and move curves to pending
```

All return `Path&`, so they chain naturally:

```
auto path = Path{}
   .moveTo(0.1_cv, 0.1_cv)
   .lineTo(0.9_cv, 0.1_cv)
   .lineTo(0.5_cv, 0.9_cv)
   .closePath()
;
```

7.1.2 Convenience Shape Helpers

Each helper calls `clear()` internally and replaces the current path — except `arc()`, which *appends* to the current path, enabling composition with `lineTo()` for pie slices, stadium shapes, and other arc-terminated outlines.

```
path.rect(x, y, w, h);
path.roundedRect(x, y, w, h, r);
path.circle(cx, cy, r);
path.ellipse(cx, cy, rx, ry);
path.arc(cx, cy, r, startAngle, endAngle, ccw=false); // appends; does NOT call clear()
path.arcTo(x1, y1, x2, y2, r); // rounded corner connecting two lines
```

7.1.3 Transform Stack

The CTM is applied to every path command before coordinates reach the curve accumulator. Geometry is baked in its transformed state; the CTM does not retroactively affect curves already accumulated.

```
path.save(); // push CTM onto stack
path.restore(); // pop CTM from stack
path.translate(tx, ty);
path.rotate(angle); // radians
path.scale(sx, sy);
path.setTransform(m); // replace CTM entirely
path.resetTransform(); // back to identity
```

`clear()` (i.e. `canvas.beginPath()`) does **not** reset the CTM — matching HTML Canvas behavior where `beginPath()` only clears geometry, never the transform.

7.1.4 Stroke Expansion

`strokePath(width)` expands the current path in-place from a centerline to a constant-width filled outline. The resulting curves replace the current geometry:

```
path.moveTo(0.1_cv, 0.5_cv).quadTo(0.25_cv, 0.05_cv, 0.5_cv, 0.5_cv);
path.strokePath(0.04_cv); // path is now a filled outline, not a centerline
```

The optional `cw` argument reverses winding so the outline subtracts coverage — useful for punch-out effects when combined with a filled region sharing the same commit call.

7.1.5 Arc-length Sampling

`Path::sample(t)` returns position and tangent direction at normalized arc-length $t \in [0, 1]$. The lookup table is built lazily and cached until the path geometry changes.

```
slughorn::canvas::Path p;

p.moveTo(0_cv, 0_cv).quadTo(0.5_cv, 1_cv, 1_cv, 0_cv);

auto mid = p.sample(0.5);
// mid.x, mid.y — world position at 50% arc length
// mid.angle — tangent angle in radians

// total arc length in authoring space
slug_t len = p.arcLength();
```

This is the mechanism used to place tick labels around a circular gauge or position an icon at an exact point along a curved path.

7.1.6 Composing Paths

`addPath(other)` appends all curves from `other` into this path's pending accumulator without consuming or modifying `other`. Use it to build compound shapes from reusable sub-paths or to merge paths produced independently:

```
slughorn::canvas::Path body, detail;

body.circle(0.5_cv, 0.5_cv, 0.4_cv);
detail.rect(0.3_cv, 0.3_cv, 0.4_cv, 0.4_cv);

// one commit covers both sub-paths
body.addPath(detail);

canvas.fill(body, RED);
```

7.2 Canvas

`Canvas` wraps an `Atlas` reference and an implicit `Path`. It forwards the full `Path` API so that for simple cases you never need to touch a `Path` directly. Commit verbs drain the current path into the Atlas and push a `Layer` onto the in-progress `CompositeShape`.

```
slughorn::Atlas atlas;

slughorn::canvas::Canvas canvas(atlas); // default auto-key prefix
slughorn::canvas::Canvas canvas(atlas, slughorn::KeyIterator("icon")); // custom prefix
```

7.2.1 Implicit vs Explicit Path

All authoring commands operate on the Canvas's internal path:

```
canvas.beginPath() // clear internal path; CTM unchanged
    .moveTo(0.1_cv, 0.5_cv)
```

```
    .quadTo(0.5_cv, 0.1_cv, 0.9_cv, 0.5_cv)
    .fill(RED) // commit internal path
;
```

Any commit verb also accepts an explicit `Path` argument. The path is read but never consumed — the same `Path` can be committed multiple times or to multiple canvases:

```
slughorn::canvas::Path star = buildStar();

canvas.fill(star, RED); // commit once in red — star is unchanged
canvas.fill(star, BLUE, 1_cv, "blue_star"); // commit again in blue
```

`canvas.path()` returns a copy of the internal path for sampling or continued building without disturbing the accumulator.

> **Note:** After `fill()` or `stroke()`, the accumulated curves remain in the internal path until the next `beginPath()` call. Call `beginPath()` explicitly when starting a new, unrelated shape to avoid inadvertently merging geometry.

7.2.2 Commit Verbs

7.2.2.1 fill — Flat Color

```
Layer fill(Color color, slug_t scale=1_cv, Atlas::ShapeInfo::Origin origin={});
Layer fill(Color color, slug_t scale, Key key, Atlas::ShapeInfo::Origin origin={});
```

Commits the current path as a flat-colored Layer and returns it. The auto-key overload assigns the next name from the `KeyIterator` (`"icon_0"`, `"icon_1"`, ...). The explicit-key overload makes the shape addressable by name after `build()`. Use `layer.key` when you need to pass the key to another API.

7.2.2.2 stroke — Stroke as Commit Verb

```
Layer stroke(slug_t width, Color color, slug_t scale=1_cv, Atlas::ShapeInfo::Origin origin={});
Layer stroke(slug_t width, Color color, slug_t scale, Key key, ...);
```

Calls `strokePath(width)` internally then commits the outline — equivalent to `strokePath` + `fill` but cleaner for the common case. Note that an explicit `Key` registers the shape *and* appends it to the composite accumulator. If you want the named shape but not the composite membership, call `beginComposite()` immediately after.

7.2.2.3 defineShape — Geometry Only

```
bool defineShape(Key key, slug_t scale=1_cv, Atlas::ShapeInfo::Origin origin={});
```

Registers geometry in the Atlas but does **not** push a Layer onto the composite accumulator. Use when you want the outline addressable independently — for example, to re-use it later with different colors or gradients managed by the caller.

7.2.2.4 fillGradient / strokeGradient

```
Layer fillGradient(const GradientHandle& handle, slug_t scale=1_cv, ...);
Layer strokeGradient(slug_t width, const GradientHandle& handle, slug_t scale=1_cv, ...);
```

Like `fill` / `stroke`, but the resulting Layer carries a `gradientId` instead of a flat color. Gradient construction is covered in Gradients; a quick-start reference is at the end of this section.

7.2.3 CompositeShape Management

Every `fill`, `stroke`, `fillGradient`, and `strokeGradient` call appends a Layer to an in-progress `CompositeShape`. `finalize()` seals it:

```
// Implicit reset — happens automatically at construction and after each finalize()
canvas.beginComposite(); // explicitly reset accumulator + clear path
canvas.setAdvance(0.6_cv); // em-space horizontal advance (for text layout)

// returns composite; does NOT register in Atlas
CompositeShape cs = canvas.finalize();

// registers in Atlas + resets accumulator
canvas.finalize("my_icon");
```

`beginComposite()` resets both the composite accumulator and the internal path. Call it when you want to discard an in-progress composite without producing a `CompositeShape`.

7.3 The Core Patterns

7.3.1 Pattern 1 — Single-layer Fill (auto-key)

The simplest case. The shape gets an auto-key; the composite gets a named key.

```
canvas.beginPath()
    .moveTo(0.5_cv, 0.9_cv)
    .lineTo(0.9_cv, 0.1_cv)
    .lineTo(0.1_cv, 0.1_cv)
    .closePath()
    .fill(RED)
;

canvas.finalize("my_triangle");
```

7.3.2 Pattern 2 — Named Shape

Use when you need the shape directly addressable after `build()`, e.g. for `atlas.getShape("circle_shape")` or the CLI `render` subcommand.

```
canvas.circle(0.5_cv, 0.5_cv, 0.4_cv).fill(BLUE, 1_cv, "circle_shape");
canvas.finalize("circle_composite");
```

7.3.3 Pattern 3 — Multi-layer Composite

Each `fill` appends one Layer; `finalize` seals all into a single `CompositeShape`.

```
canvas.rect(0.05_cv, 0.05_cv, 0.9_cv, 0.9_cv).fill(RED);
canvas.circle(0.5_cv, 0.5_cv, 0.35_cv).fill(BLUE);
canvas.roundedRect(0.25_cv, 0.25_cv, 0.5_cv, 0.5_cv, 0.08_cv).fill(GREEN);
canvas.finalize("layered_icon");
```

7.3.4 Pattern 4 — Geometry-only Shape

```
canvas
    .roundedRect(0.1_cv, 0.1_cv, 0.8_cv, 0.8_cv, 0.15_cv)
    // No Layer; the shape lives in the Atlas with no color attached.
    .defineShape("rrect_geom")
;
```

7.3.5 Pattern 5 — Stroke as Commit Verb

```
canvas
    .beginPath()
    .moveTo(0.1_cv, 0.5_cv)
    .quadTo(0.25_cv, 0.05_cv, 0.5_cv, 0.5_cv)
    .quadTo(0.75_cv, 0.95_cv, 0.9_cv, 0.5_cv)
    .stroke(0.06_cv, WHITE)
;

canvas.finalize("scurve_stroke");
```

7.3.6 Pattern 6 — strokePath + defineShape (Geometry-only Stroke)

The escape hatch: `strokePath()` expands the centerline in-place, then `defineShape()` commits the resulting outline as raw geometry with no color. Use when you need the outline for something the commit verbs cannot express directly (e.g. a clip region, or combined with additional sub-paths before a single `fill`).

`strokePath()` returns `bool` (success/empty), so the chain ends before it:

```
canvas
    .beginPath()
    .moveTo(0.2_cv, 0.85_cv)
    .quadTo(0.1_cv, 0.5_cv, 0.5_cv, 0.5_cv)
;

canvas.strokePath(0.08_cv);
canvas.defineShape("scurve_geom");
```

7.3.7 Pattern 7 — Transform stack (Baked Tick Marks)

All 12 ticks accumulate into one `_pendingCurves` buffer and are committed as a **single Atlas shape**. `save()`/`restore()` isolates each rotation so that subsequent iterations always start from the base CTM.

```
canvas.beginPath();

for(int i = 0; i < 12; i++) {
    canvas
        .save()
        .translate(CX, CY)
        .rotate(cv(i) * 2_cv * M_PI / 12_cv)
        .moveTo(0_cv, TICK_INNER)
        .lineTo(0_cv, TICK_OUTER)
    ;

    canvas.strokePath(TICK_WIDTH); // returns bool — chain ends here
    canvas.restore();
}

canvas.fill(DARK, 1_cv, "clock_ticks", Origin(Origin::Type::Centered));
canvas.finalize("clock_ticks_composite");
```

Because all sub-paths share one `fill` call, the entire tick ring is one atlas entry — no per-tick draw call overhead at render time.

7.3.8 Pattern 8 — Explicit Pivot Origin

`Origin(cx, cy)` stores the pivot into `Layer::transform.x/y`. The GPU consumer can then rotate the shape around that exact point without any CPU-side re-layout. This is how the clock hand demo works: the hand is authored with its base at `(CX, CY)`, and the GPU rotates it there each frame.

```
canvas.moveTo(CX, CY).lineTo(CX, CY + HAND_LENGTH);

canvas.stroke(
    HAND_WIDTH, HAND_COLOR, 1_cv,
    "clock_hand",
    Origin(CX, CY) // pivot = stroke base, stored in Layer::transform.x/y
);

canvas.finalize("clock_hand_composite");
```

7.3.9 Pattern 9 — Standalone Path + Arc-length Sampling

```
slughorn::canvas::Path p;

p
    .moveTo(0_cv, 0_cv)
    .lineTo(0.3_cv, 0_cv)
    .quadTo(0.5_cv, 1_cv, 0.7_cv, 0_cv)
    .lineTo(1_cv, 0_cv)
;

for(size_t i = 0; i <= 10; i++) {
    auto s = p.sample(cv(i) / 10_cv);

    std::cout << s.x << ", " << s.y << ", angle=" << s.angle << "\n";
}

// Commit via explicit-path overload — p is unchanged.
canvas.stroke(p, 0.06_cv, WHITE);
canvas.finalize("sample_path");
```

7.4 The Scale Parameter

Every commit verb accepts an optional `scale` parameter (default `1_cv`). It converts authoring-space coordinates into em-space before storing curves. Any value is valid — `build()` absorbs it via `bandScaleX = numBands / em_width`, so the GPU result is identical regardless of which scale you choose. `[0, 1]` em-space is a convention, not a requirement (the one exception is GPU tiling with `fract()`, which genuinely requires `[0, 1]` — see expand and GPU Tiling).

`scale` is never stored in `Layer::scale`, which is reserved for FreeType2 font sizes. For Canvas geometry, `1_cv` is almost always the right choice.

7.5 Text Placement

`Canvas::text()` places pre-loaded glyphs into the in-progress composite, handling em-space conversion, vertical anchoring, and optional horizontal centering automatically. The result is indistinguishable from a sequence of `fill()` calls — one `Layer` per glyph, all accumulating into the same composite as any other canvas geometry.

The glyphs must already exist in the Atlas (loaded via a font backend before `atlas.build()`). `Canvas` itself has no font dependency.

```
// font->metrics() returns a slughorn::FontMetrics — plain slug_t fields,
// defined in slughorn.hpp, no FreeType types anywhere in the struct.
slughorn::FontMetrics m = font->metrics();

// Coordinates are in canvas space — the same space as moveTo/lineTo.
// If a Y-flip CTM is active (SVG-style authoring), pass SVG-space coordinates.
canvas.text(
```

```
    "String",
    70_cv, // fontSize in canvas units
    180_cv, 305_cv, // x (anchor), y (anchor) — SVG canvas coords
    {1_cv, 1_cv, 1_cv, 1_cv},
    m,
    slughorn::canvas::TextAnchorY::Baseline,
    slughorn::canvas::TextAlignX::Center
);

canvas.finalize("card_text");
```

7.5.1 Vertical Anchoring — `TextAnchorY`

Value	y refers to	Metric used
`Baseline`	text baseline (default)	—
`CapCenter`	vertical centre of capital letters	`capHeightRatio`
`CapTop`	top edge of capital letters	`capHeightRatio`
`XCenter`	vertical centre of lowercase letters	`xHeightRatio`

All ratios are fractions of the em-square stored in `slughorn::FontMetrics`. Multiply by `fontSize` to get world-space distances.

7.5.2 Horizontal Alignment — `TextAlignX`

Value	x refers to	Pass count
`Left`	left edge of first glyph (default)	single
`Center`	horizontal centre of run	two (measures total advance first)
`Right`	right edge of last glyph	two

`Left` is single-pass with no overhead. `Center` and `Right` iterate the string twice — once to measure total advance, once to place. The advance values come from the Atlas shapes loaded by the font backend; if a glyph is not present in the Atlas a 0.6 em fallback is used.

7.5.3 Coordinate Space

`canvas.text()` applies the current CTM to `(x, y)` before converting to em-space, so it obeys the same coordinate space as every other canvas verb. If you set up a Y-flip for SVG-style authoring:

```
canvas.translate(0_cv, H).scale(1_cv, -1_cv);
```

then pass SVG-space coordinates to `text()` just as you would to `moveTo`. The CTM handles the conversion to world space internally. Passing raw world-space coordinates to `text()` while a flip CTM is active will place the text at the mirror position.

7.5.4 Dependency Design

`FontMetrics` lives in `slughorn/slughorn.hpp` as a plain struct of `slug_t` fields. Neither `canvas.hpp` nor `freetype.hpp` includes the other — both see only `slughorn.hpp`:

`freetype.hpp` ⟶ `slughorn.hpp` ⟵ `canvas.hpp`

`Canvas::text()` has no opinion on where the metrics came from. Pass metrics from FreeType, HarfBuzz, a future backend, or a manually filled struct — the API is identical. The application bridges the two at the call site.

7.6 Gradient Quick Reference

The Canvas provides three factory methods. All return a `GradientHandle` that you pass to `fillGradient` or `strokeGradient`:

```
// Linear — two endpoints in authoring space
auto g = canvas.createLinearGradient(x0, y0, x1, y1, stops);

// Radial — center (cx, cy), inner radius r0, outer radius r1 (r0=0 → point centre)
auto g = canvas.createRadialGradient(cx, cy, r0, r1, stops);

// Sweep (conic) — center, start and end angle in radians
// Use -π to +π with matching first/last stops for a seam-free full circle.
auto g = canvas.createSweepGradient(cx, cy, startAngle, endAngle, stops);
```

Stops are `{t, Color}` pairs with $t \in [0, 1]$. See Gradients for the full treatment of gradient transforms, aspect-ratio correction, and COLRv1 integration.

8 Mixing Backends

The dependency design pays off as soon as you need text alongside canvas geometry. Load glyphs with a font backend, draw shapes with `Canvas`, and commit everything into the same `Atlas` — the two cooperate through `FontMetrics` without either knowing about the other.

8.1 Single Atlas, One Font

The common case: one `Atlas` holds both canvas shapes and font glyphs. Load the font before calling `canvas.text()` (so advance widths are available for the centering pass), then call `atlas.build()`:

```
slughorn::canvas::Canvas canvas(*atlas);

canvas.translate(0_cv, 520_cv).scale(1_cv, -1_cv); // SVG-style Y-down authoring

// ... path drawing, canvas.fill(), canvas.stroke() ...
canvas.finalize("card");

auto font = osgx::make_ref<osgSlug::Font>("Sans.ttf", atlas);

font->load();
atlas->build();
atlas->packTextures();

// Coordinates are in SVG canvas space — the active CTM is applied internally.
canvas.text(
    "String",
    70_cv,
    180_cv, 305_cv,
    {1_cv, 1_cv, 1_cv, 1_cv},
    font->metrics(),
    slughorn::canvas::TextAnchorY::Baseline,
    slughorn::canvas::TextAlignX::Center
);

canvas.finalize("title");
```

One `Atlas`, one `build()` call, one `ShapeDrawable` — canvas shapes and text are indistinguishable at the GPU level.

8.2 Multi-Font Scenes

Loading two fonts into one `Atlas` has a hard constraint: `Font::load()` registers each glyph under its Unicode codepoint as the Atlas key. Two fonts covering the same codepoints — a regular and an italic variant of the same family, for instance — cannot coexist in one Atlas. The second `load()` silently skips every codepoint already registered.

The current solution is **one Atlas per font family**, with all canvas geometry for that family in the same Atlas.

> **Future: per-font key offset**
> The planned fix is a key offset on the `Font` object:
>
> ```
> auto fontRegular = osgx::make_ref<osgSlug::Font>("Sans.ttf", atlas);
> auto fontItalic = osgx::make_ref<osgSlug::Font>("Sans-Italic.ttf", atlas, /*keyOffset=*/0
> ```
>
> Glyphs from `fontItalic` would be stored at keys `0x10000 + codepoint`, leaving the regular range untouched. `canvas.text()` would accept a `Font*` (or a `FontMetrics` carrying the offset) to look up the correct range at placement time. Unicode codepoints top out at `0x10FFFF`, leaving ample room for many independent namespaces in a single Atlas. See `WhatsNext` for current status.

8.3 Multiple Atlases in a Scene (osgSlug)

When each font family lives in its own `Atlas`, each `ShapeDrawable` also needs its own Atlas state. The key constraint is that `SSBOShapeDrawable::compile()` writes the layer SSBO and the GLSL program into the drawable's own `StateSet` via `getOrCreateStateSet()`. The Atlas-specific state — shape SSBO, textures, uniforms — comes from `createDefaultStateSet()`. These must be *merged* into the drawable's existing `StateSet`, not used to replace it:

```
sd->compile();
sd->getOrCreateStateSet()->merge(*atlas->createDefaultStateSet());

sdItalic->compile();
sdItalic->getOrCreateStateSet()->merge(*atlasItalic->createDefaultStateSet());

auto geode = osgx::make_ref<osg::Geode>();

geode->addDrawable(sd);
geode->addDrawable(sdItalic); // drawn in addDrawable() order
// no StateSet on the Geode — each drawable is self-contained
```

Calling `sd->setStateSet(atlas->createDefaultStateSet())` instead would replace the drawable's `StateSet` entirely, discarding the compiled layer SSBO and making the drawable invisible. Use `getOrCreateStateSet()->merge()`; both drawables can share a single `Geode`.

9 Gradients

Gradients in slughorn are not computed per-fragment from raw stop data. Instead, `build()` bakes all registered gradients into a dedicated RGBA8 texture: 256 texels wide, one row per gradient. The fragment shader does one texture fetch at a computed `t` value to get the interpolated color — hardware bilinear filtering handles all blending between stops. No stop-count limit, no per-fragment iteration over a stop list.

9.1 The Gradient Texture

The gradient texture is a third GPU texture alongside the curve and band textures:

Texture	Format	Size	Accessor
Curve	RGBA32F	atlas_width × N	`getCurveTextureData()`
Band	RGBA16UI	atlas_width × N	`getBandTextureData()`
Gradient	RGBA8	256 × gradient_count	`getGradientTextureData()`

Upload it after `build()` bound to a separate sampler unit:

```
Atlas::TextureData td = atlas.build();
Atlas::TextureData gd = atlas.getGradientTextureData();

// gd.width = 256
// gd.height = number of registered gradients (0 if none)
// gd.format = RGBA8; upload with GL_LINEAR filter
```

If no gradients are registered, `gd.height` is 0 and you can skip the upload.

9.2 Registering a Gradient

9.2.1 Canvas API

For Canvas-authored content, three factory methods handle registration:

```
auto g = canvas.createLinearGradient(x0, y0, x1, y1, stops);
auto g = canvas.createRadialGradient(cx, cy, r0, r1, stops);
auto g = canvas.createSweepGradient(cx, cy, startAngle, endAngle, stops);

// commit current path
canvas.fillGradient(g);

// with explicit key
canvas.fillGradient(g, 1_cv, "named_gradient");
```

Gradient coordinates are in the same authoring space as your path commands — no coordinate conversion needed.

9.2.2 Direct — GradientInfo

For custom backends or shapes registered outside the Canvas API, use `GradientInfo` directly:

```
slughorn::GradientInfo gi;

gi.type = GradientInfo::Type::Linear;
gi.stops = { {0_cv, RED}, {0.5_cv, YELLOW}, {1_cv, BLUE} };
gi.transform = buildLinearGradientMatrix(x0, y0, x1, y1);

// 1-based ID; must be called before build()
uint32_t gid = atlas.addGradient(gi);

// Store gid in Layer::gradientId.
```

`Layer::gradientId = 0` means flat color (default). When non-zero, `Layer::color.rgb` is ignored and the gradient color is used instead; `Layer::color.a` remains active as a per-layer opacity multiplier.

9.3 Stops

Stops are `{t, Color}` pairs with $t \in [0, 1]$. They are sorted by `t` during `build()`, so order does not matter. Colors beyond the first and last stop are clamped (pad spread). 256 texels is enough resolution for any ramp visible at screen sizes.

```
std::vector<GradientStop> stops = {
    {0_cv, {1, 0, 0, 1}}, // red
    {0.5_cv, {1, 1, 0, 1}}, // yellow
    {1_cv, {0, 0, 1, 1}}, // blue
};
```

9.4 The Three Types

9.4.1 Linear

`buildLinearGradientMatrix(x0, y0, x1, y1)` encodes the axis as a direction vector. The shader projects each fragment's em-coordinate onto the axis: `t = dot(emCoord, dir) + offset`.

```
gi.type = GradientInfo::Type::Linear;
gi.transform = buildLinearGradientMatrix(0.1_cv, 0.5_cv, 0.9_cv, 0.5_cv);
gi.stops = stops;
```

9.4.2 Radial

`buildRadialGradientMatrix(cx, cy, r1)` encodes the center and outer radius. `GradientInfo::innerRadius` sets the inner radius for an annular gradient (0 = solid center, the common case). The shader computes `t = (length(emCoord - center) - innerRadius) / (outerRadius - innerRadius)`.

```
gi.type = GradientInfo::Type::Radial;
gi.transform = buildRadialGradientMatrix(0.5_cv, 0.5_cv, 0.4_cv);
gi.innerRadius = 0_cv; // point center; set > 0 for a ring
gi.stops = stops;
```

The NanoSVG and FreeType COLRv1 backends use `Type::AffineRadial` internally for elliptical radials — this is handled automatically; callers never construct one directly.

9.4.3 Sweep (Conic)

`buildSweepGradientMatrix(cx, cy, startAngle, arcSpan)` encodes the center and angular range. Angles are in radians from the +X axis; `arcSpan = endAngle - startAngle`. The shader computes `t = (atan2(dy, dx) - startAngle) / arcSpan`.

```
gi.type = GradientInfo::Type::Sweep;
gi.transform = buildSweepGradientMatrix(0.5_cv, 0.5_cv, -M_PI, 2.0 * M_PI);
gi.stops = stops;
```

For a seam-free full-circle sweep, use `startAngle = -π` and `arcSpan = 2π`. `atan2` returns values in `[-π, π]`, which maps exactly onto $t \in [0, 1]$ under this parameterization — no visible seam at the wrap point.

9.5 The Type Discriminator

Rather than a separate per-type attribute, the shader identifies the gradient type from the `w` component of the gradient transform attribute:

`a_gradientXform.w`	Type
`== 0.0`	Linear
`> 0.0`	Radial / AffineRadial
`< 0.0`	Sweep

This is guaranteed by the builder functions and requires no additional caller action.

9.6 Shader Considerations

Consuming gradients in your graphics backend requires additional information be fed to the shader pipeline:

Attributes SSBO (GL4) or appended to the vertex layout (GL3):

Attribute	Location	Content
`a_gradientMeta`	6	`(gradientId, centerX, centerY, r0_norm)`
`a_gradientXform`	7	Type-encoded gradient transform

- **Linear**: `a_gradientXform = (dirX, dirY, offset, 0.0)`.
- **Radial / AffineRadial**: `a_gradientXform` = column-major 2×2 B matrix; `a_gradientMeta.yz` = center; `a_gradientMeta.w` = normalized inner radius.
- **Sweep**: `a_gradientXform = (cx, cy, startAngle, -invArcSpan)`.

`a_gradientMeta.x` carries the gradient ID (an integer packed into a float). The shader decodes it to address the correct row of the gradient texture:

```
float gv = (float(v_gradientId) - 0.5) / float(slug_gradientCount);
vec4 gc = texture(slug_gradientTexture, vec2(t, gv));
```

Uniforms:

Uniform	Type	Value
`osgSlug_gradientTexture`	`sampler2D`	Gradient texture unit
`osgSlug_gradientCount`	`int`	`atlas.getGradientTextureData().hei`

The full SSBO/vertex attribute layout, including gradient packing, is covered in Connecting to Your Graphics Backend.

9.7 What Backends Handle For You

For content from external sources, gradient registration is automatic:

- **FreeType COLRv1** — `loadColorGlyph` / `loadEmojiFont` parse the font's paint graph, register all gradient stops via `addGradient`, and store the resulting ID in each `Layer::gradientId`. No `GradientInfo` construction needed.
- **NanoSVG** — `loadFile` / `loadImage` detect linear and radial gradient fills, convert their stop and transform data, and register gradients automatically. NanoSVG sweep gradients are not a standard SVG feature and are not yet supported.

Manual `GradientInfo` construction is only needed when authoring through the Canvas API or writing a custom backend.

10 Band Placement & Atlas Tuning

The Core Concepts section described how the band texture divides each shape's bounding box into horizontal strips, keeping per-fragment shader work small by limiting which curves each fragment must evaluate. This section covers what you can actually control: how many bands, where they go, and why those choices matter more than they might appear to.

> This area of slughorn is currently *experimental*, although we believe we **have** seen performance improvements under controlled conditions. As slughorn continues to evolve — and especially with the eventual visual editor — this component of the library will likely see a lot of change.

10.1 How the Indirection Table Works

slughorn attempts to extend the original Slug banding model in one important way: both axes are banded independently, and the fragment shader locates the right band on each axis via a small O(1) lookup table rather than a linear search.

Each shape's block in the band texture begins with two indirection arrays — one for Y, one for X — each `INDIRECTION_SIZE` entries wide (currently 32). To find the band covering a given em-space coordinate, the shader quantizes that coordinate to the nearest slot index and reads the band index directly from the table. The combined (X-band, Y-band) pair then determines which curves the shader evaluates for that fragment.

The 32-slot granularity means that meaningful split positions align with multiples of `1/32 = 0.03125`. Splits placed between quantization boundaries are snapped to the nearest slot; the planned editor (see The Offline Editor) will enforce that snapping visually.

10.2 Uniform Placement

By default, slughorn assigns band counts automatically and places splits at even intervals across the shape's bounding box — equivalent to what the original Slug library does, and a correct, practical baseline:

```
slughorn::Atlas::ShapeInfo info;

info.curves = curves;
info.numBandsX = 16; // 16 columns
info.numBandsY = 16; // 16 rows → 256 band cells total
```

For shapes where coverage is distributed roughly evenly — most glyphs, simple icons — uniform placement is hard to beat. Every band is the same width, so no single band accumulates a disproportionate share of curves.

`computeUniformSplits()` makes the baseline explicit and returns the same even fractions in the normalized `[0,1]` format used by `splitsX`/`splitsY`:

```
auto [sx, sy] = Atlas::computeUniformSplits(curves, 16, 16);
// sx = {0.0625, 0.125, 0.1875, ..., 0.9375} (15 interior splits → 16 bands)
// sy = same
```

It is most useful as a control baseline when benchmarking other strategies.

10.3 The SplitStrategy API

Every shape goes through the same band-building machinery regardless of how its splits were chosen. Split placement is a **policy layer**, fully separate from the atlas builder and the shader:

`caller chooses splits → ShapeInfo::splitsX/splitsY → band builder → shader`

Splits are **always** in normalized `[0,1]` space — `0.0` is the shape's left/bottom edge, `1.0` is its right/top edge, independent of authoring-space scale. This keeps split values stable regardless of how large or where the shape happens to be.

`Atlas::SplitStrategy` is the callable type that encapsulates a placement policy:

```
using SplitStrategy = std::function<
    std::pair<std::vector<slug_t>, std::vector<slug_t>>(const Atlas::Curves&)
>;
```

It takes the shape's curves and returns `{splitsX, splitsY}` as normalized fractions. The FreeType backend accepts a `LoadConfig` with an optional `strategy` field:

```
slughorn::freetype::loadAsciiFont(
    fontPath,
    atlas,
    {
        .strategy = [](const Atlas::Curves& curves,
            slug_t minX, slug_t rangeX, slug_t minY, slug_t rangeY
        ) {
            int n = std::min(16, std::max(1, (int)curves.size() / 2));

            return Atlas::computeCostSplits(curves, n, n, minX, rangeX, minY, rangeY);
        }
    }
);
```

The Canvas API exposes the same capability as a stateful setter — analogous to `fillStyle` in HTML Canvas — that applies to subsequent commit verbs until cleared:

```
canvas.setSplitStrategy([](const Atlas::Curves& curves) {
    return Atlas::computeUniformSplits(curves, 8, 8);
});
```

```
canvas.fill(color); // strategy applied here
canvas.fill(color2); // and here

canvas.clearSplits(); // back to default auto-band behavior
```

`setSplits()` sets both axes directly and clears any active strategy:

```
canvas.setSplits(splitsX, splitsY);
```

10.4 `computeAdaptiveSplits` — A Cautionary Tale

`Atlas::computeAdaptiveSplits()` was the first attempt at smarter band placement. It runs a sweep-line analysis over each curve's bounding-box footprint on each axis, builds a density profile, and places split boundaries at the lowest-density valleys — positions where the fewest curve bounding boxes cross.

The intuition sounds right: put band boundaries where few curves cross them, so each band inherits a small, clean curve list. In practice it reliably underperforms uniform placement on every shape tested so far, sometimes by 2–3×.

The diagnosis is in what the heuristic actually optimizes. The shader does not pay for boundary crossings; it pays for the total number of curves it must evaluate per fragment, which is proportional to the number of curves in the fragment's chosen X-band *and* the number in its chosen Y-band. Minimizing boundary crossings is not the same thing as minimizing that sum.

The specific failure modes:

- A tiny density valley consumes a precious split, while an adjacent large dense region remains under-partitioned — one expensive band where there could have been several cheaper ones.
- The X and Y axes are analyzed independently, so "locally good X" and "locally good Y" can combine into an expensive 2D result for the shader.
- The greedy valley selection does not account for band width, so an apparent optimum can be erased when positions snap to indirection-table boundaries.

`computeAdaptiveSplits()` is kept as a named function because the policy API it validates is sound and the function is useful as a comparison baseline. Treat it as a proof-of-concept rather than a production tuning tool.

10.5 Manual Placement

The most direct path to a performance improvement is manual placement: telling slughorn exactly where the band boundaries should go.

`ShapeInfo::splitsX` and `ShapeInfo::splitsY` accept normalized `[0,1]` fractions directly, overriding any `numBandsX`/`numBandsY` value when non-empty. The resulting band count is `splits.size() + 1`:

```
slughorn::Atlas::ShapeInfo info;

info.curves = curves;
info.splitsX = {0.25_cv, 0.5_cv, 0.75_cv}; // 4 X-bands
info.splitsY = {0.5_cv}; // 2 Y-bands
```

10.5.1 A Concrete Result

To make this concrete, consider a stroked rounded rectangle — one of the most common shapes in any UI toolkit:

```
canvas.roundedRect(0.1_cv, 0.1_cv, 0.9_cv, 0.9_cv, 0.05_cv)
    .stroke(0.05_cv, {0.85_cv, 0.66_cv, 0.26_cv, 1_cv});
```

Corner radius `0.05`, stroke width `0.05`. This shape has a very specific curve distribution: four arc pairs at the corners (an inner arc at `r ≈ 0.025` and an outer arc at `r ≈ 0.075`), straight stroke sides along the four edges, and a completely empty interior.

With uniform band placement, the indirection grid divides the shape evenly — placing band boundaries straight through the center regardless of whether any curves live there:

That empty center is a missed opportunity. Every fragment landing there still performs a band lookup and curve-list traversal, even though the result will always be zero coverage.

Geometry-aware placement changes the question from *"how many bands?"* to *"where are the curves, and how do I keep each band's curve count as low as possible?"* For this shape the answer is straightforward: place all splits tightly around the four corner zones and leave the interior as a single coarse band.

The `INDIRECTION_SIZE` constant is 32, so the finest addressable split position is `1/32 = 0.03125`. After normalizing the shape's bounding box to `[0, 1]`, the corner arcs occupy roughly the outer 9% of em-space on each side. Using all four valid split positions within that zone — `SPLIT_01` through `SPLIT_04` at the low end, and `SPLIT_28` through `SPLIT_31` at the high end — isolates each arc into its own small cell:

```
std::vector<slug_t> splitsX = {
    Atlas::SPLIT_01, Atlas::SPLIT_02, Atlas::SPLIT_03, Atlas::SPLIT_04,
    Atlas::SPLIT_28, Atlas::SPLIT_29, Atlas::SPLIT_30, Atlas::SPLIT_31,
};
std::vector<slug_t> splitsY = {
    Atlas::SPLIT_01, Atlas::SPLIT_02, Atlas::SPLIT_03, Atlas::SPLIT_04,
    Atlas::SPLIT_28, Atlas::SPLIT_29, Atlas::SPLIT_30, Atlas::SPLIT_31,
};

canvas.setSplits(splitsX, splitsY);
```

The resulting 9×9 grid looks like this:

The inner and outer arcs at each corner sit in their own dedicated cells. The straight stroke sides

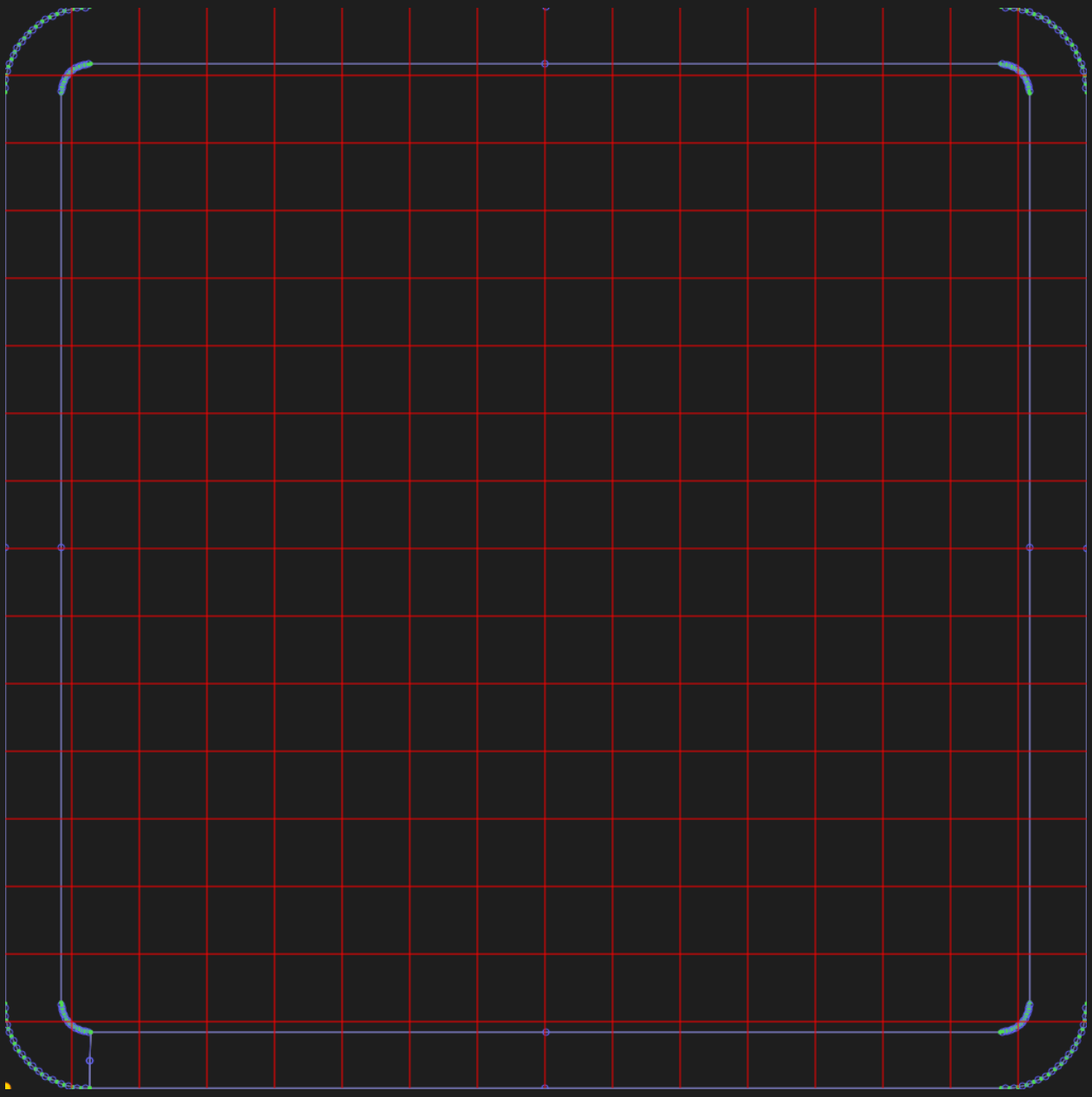


Figure 5: Uniform band placement on a stroked rounded rectangle

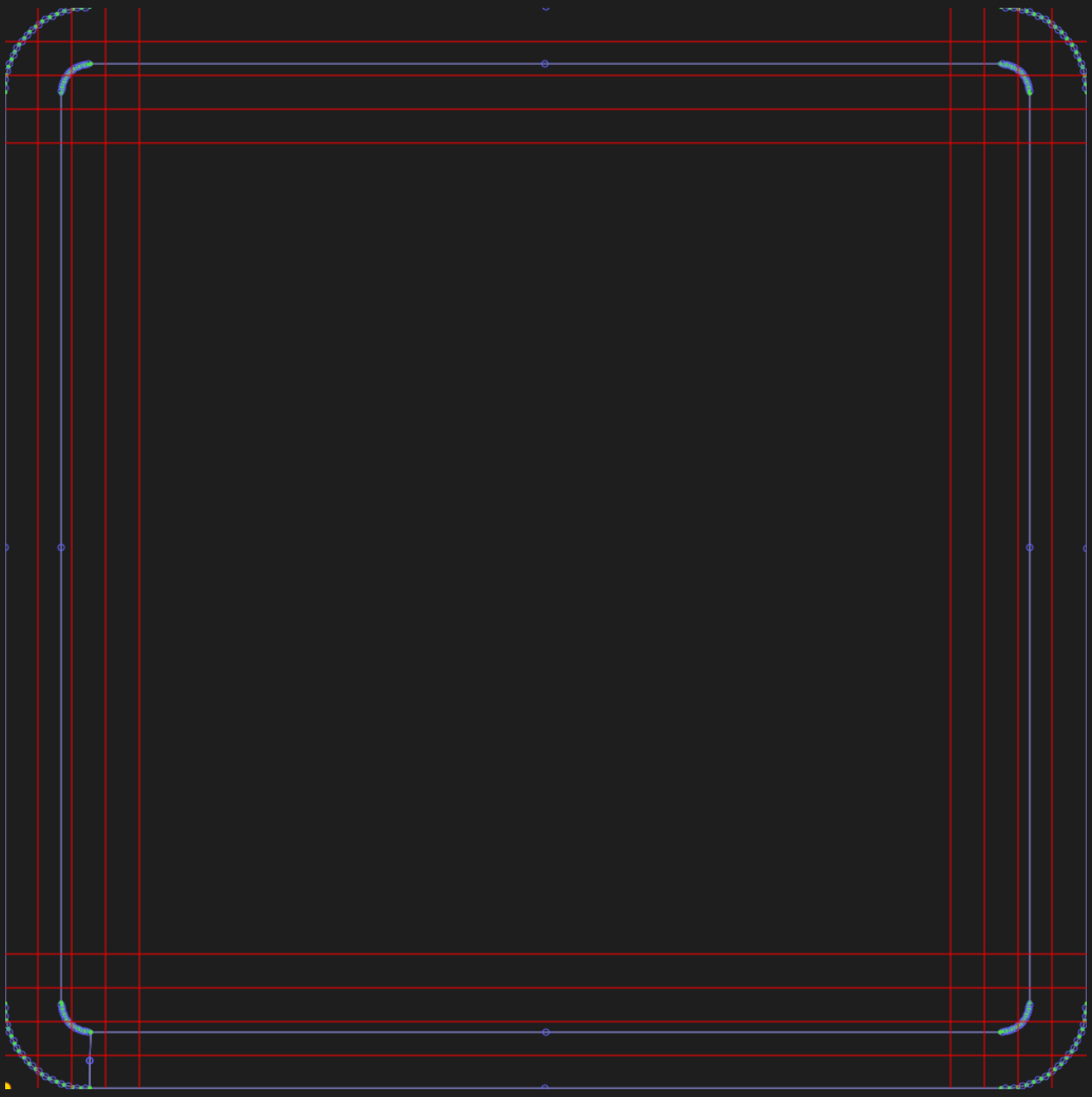


Figure 6: Geometry-aware band placement on a stroked rounded rectangle

are confined to the thin edge bands. The large empty center — covering the majority of every frame's fragments — is a single zero-curve band that the shader exits immediately.

Measured on the same hardware at the same window size:

Placement	GPU time	Stability
Uniform	110–120 µs	stable
Geometry-aware (8 splits/axis)	**~73 µs**	**stable**

That is a noticable reduction in GPU time, with little frame-to-frame variance, and at *lower texture bandwidth* — center fragments perform zero reads into the curve texture rather than fetching curve data that evaluates to no coverage.

The improvement is not specific to this shape. Any shape where the curve geometry is concentrated in a fraction of the bounding box — icons, glyphs, logos, UI chrome — is a candidate. The principle is always the same: **bracket the curves tightly, leave the empty space coarse.**

10.6 The Offline Editor

Identifying the right splits for a complex shape by hand is not always obvious. The planned `slughorn-editor` — a PySide6 application driven by slughorn's Python bindings — will make this visual and interactive. Phase 1 targets a single core workflow:

1. Load a `.slug` or `.slugb` file.
2. Display the shape with its current band grid overlaid as draggable guide lines.
3. Let the user drag, add, and delete band boundaries on each axis.
4. Write the adjusted `splitsX`/`splitsY` back via the normalized `[0,1]` API.

One subtlety the editor will handle explicitly: because the indirection table has 32 slots, band boundaries that do not fall on a multiple of `1/32 = 0.03125` are silently quantized. The editor will snap dragged lines to valid indirection positions so that what you see is exactly what the shader gets. A prototype already exists at `test/slughorn_editor.py`.

The benchmark result above was produced entirely with hand-written splits. The editor is the natural path to making that process accessible rather than arithmetic.

10.7 What's Next: A Cost-Based Strategy

The right objective for a future `compute*Splits` is not boundary density but **band cost**: some measure of the total shader work produced by a given partition. Useful targets include area-weighted average curves per band, worst-band curve count, and a combined X/Y cost that reflects how both axes interact in the shader rather than scoring them independently.

It will be added as a distinct function alongside the existing two, keeping benchmarking clean and allowing callers to fall back to a known baseline without restructuring their code:

```
Atlas::computeAdaptiveSplits(curves, numBandsX, numBandsY); // proof-of-concept
Atlas::computeUniformSplits(curves, numBandsX, numBandsY); // baseline / control
Atlas::compute*Splits(curves, numBandsX, numBandsY); // cost-based (planned)
```

A longer-term direction worth investigating once the editor is mature: **per-shape variable indirection resolution**. Today `INDIRECTION_SIZE = 32` is a global constant shared by every shape — the finest split you can express is `1/32 ≈ 3.1%` of em-space regardless of how large or complex the shape is. A future design could allow each shape to declare its own indirection size (and potentially independent X/Y sizes for non-square shapes), so a large detailed shape could use 64 slots for `~1.5%` granularity while a small simple shape uses 8 and pays almost nothing for its band table. The shader cost of the change is minimal — one runtime scalar read replaces a compile-time constant — and the editor is the natural place to expose the knob, since you can immediately see whether the finer grid actually isolates any additional curves.

11 Serialization

slughorn's serialization layer lives in `slughorn/serial.hpp` and supports two formats. Both represent the same data — a built Atlas with its GPU textures, shape metrics, composite definitions, and gradients — and the format is auto-detected on load.

Format	Extension	Analogy	When to use
JSON + base64	`.slug`	glTF `.gltf`	Debugging, inspection, source control
Binary container	`.slugb`	glTF `.glb`	Shipping, fast loading, minimal size

Like the backends, `serial.hpp` is a single-header implementation file. Define the implementation macro in exactly one `.cpp` before including:

```
#define SLUGHORN_SERIAL_IMPLEMENTATION
#include <slughorn/serial.hpp>
```

11.1 Writing

`build()` must be called before writing. Extension determines the format automatically:

```
Atlas::TextureData td = atlas.build();

slughorn::serial::write(atlas, "font.slug"); // JSON + base64
slughorn::serial::write(atlas, "font.slugb"); // binary container
```

For stream-based output, call `writeJSON` or `writeBinary` directly:

```
slughorn::serial::writeJSON(atlas, std::cout);
slughorn::serial::writeBinary(atlas, outputStream);
```

11.2 Reading

The file format is auto-detected from the first byte ({ → JSON, S → binary). The returned Atlas is fully pre-built — `is_built()` is `true` and `build()` does not need to be called again. No font library or backend is required at load time.

```
slughorn::Atlas atlas = slughorn::serial::read("font.slugb");

// Immediately usable — upload textures and render
Atlas::TextureData td = atlas.getCurveTextureData(); // already populated
```

11.3 The Command-Line Tool

The `slughorn` binary (built when `SLUGHORN_SERIAL` is enabled) provides six subcommands for working with atlas files from the command line:

```
slughorn info   <atlas> [key]   — atlas summary, or details for one shape/composite
slughorn list   <atlas>         — list all keys (--shapes / --composites to filter)
slughorn render <atlas> <key>   — render a shape to grayscale PNG
slughorn svg    <atlas> <key>   — export decoded curves + band boundaries as SVG
slughorn shell  <atlas> [key]   — interactive Python shell with atlas pre-loaded
slughorn font   <font> <output> — pack font glyphs into a new atlas
```

`render` and `svg` are particularly useful during development. `render` runs the full band-based coverage path (use `--reference` for the brute-force fallback) and writes a PNG you can inspect immediately. `svg` overlays the decoded curve control points and band boundary lines — the primary tool for diagnosing band placement or curve decomposition issues. `shell` drops you into an IPython or stdlib REPL with the atlas and the decoded shape already in scope.

The `font` subcommand is a self-contained pipeline for the common case:

```
# Printable ASCII from a system font
slughorn font /usr/share/fonts/truetype/dejavu/DejaVuSans.ttf ui.slugb --ascii

# A specific Unicode range
slughorn font myfont.ttf symbols.slugb --range U+2600-U+26FF

# Only the characters that appear in a string
slughorn font myfont.ttf label.slugb --chars "Health: 100%"
```

11.4 Binary Container Layout

The `.slugb` container is structurally identical to the glTF `.glb` binary format — not just inspired by it. The chunk magic values (`0x4E4F534A` for JSON, `0x004E4942` for BIN), chunk header layout, padding bytes, and `bufferViews` array all match the glTF spec exactly. The `asset` block and `generator` field follow glTF conventions as well.

The three differences from a formal glTF extension are minor:

1. **Magic / version** — `"SLUG"` / `1` instead of `"glTF"` / `2`. Trivial to swap if embedding in a real glTF file.
2. **No `buffers` array** — glTF formally requires a `buffers` entry that `bufferViews` reference. slughorn puts data directly in `bufferViews`, skipping one indirection level.
3. **Top-level namespace** — a proper glTF extension would place slughorn fields under `extensions.SLUGHORN_atlas`. The heavy lifting is done; formal compliance is a wrapper.

The result is that any tool capable of parsing `.glb` chunk structure can read the binary section of a `.slugb` file without modification.

12 Python

slughorn's Python bindings are unusually complete. Every public C++ type — `Atlas`, `Shape`, `CompositeShape`, `Layer`, `Key`, `KeyIterator`, `Color`, `Matrix`, `GradientInfo`, `GradientStop`, `SplitStrategy`, `FontMetrics` — is exposed directly with the same interface you'd use from C++. The bindings are built with pybind11 and live in `ext/slughorn-python.cpp`.

12.1 Submodules

The top-level `slughorn` module has five submodules.

12.1.1 slughorn.render

The software decode and rendering layer. This is not the GPU path — it is the validation path: it runs the Slug coverage algorithm in Python-accessible C++, operating directly on decoded curve and band data. It is how `bin/slughorn render` produces PNG output without a graphics context.

```
import slughorn

atlas = slughorn.read("ui.slugb")
sampler = slughorn.render.decode(atlas, slughorn.Key(ord("A")))

# Full grayscale coverage grid as a float32 array
grid = sampler.render_grid(size=256, margin=0.05)

# Or sample one point at a time
sample = sampler.render_sample_banded(x=0.5, y=0.5, ppe_x=0.01, ppe_y=0.01)

print(sample.fill)
```

`render_grid` returns a NumPy-compatible `float32` memoryview. `render_sample` takes all decoded curves (the reference path); `render_sample_banded` mirrors the actual GPU shader path, and is what `render_grid` uses by default.

12.1.2 slughorn.canvas

The HTML Canvas-style drawing context, identical in capability to the C++ `Canvas` API. See The Canvas API for the full reference; the Python binding exposes every method: `moveTo`, `lineTo`, `quadTo`, `cubicTo`, `beginShape`, `endShape`, `defineShape`, and all gradient factory helpers.

12.1.3 slughorn.emoji

A Unicode 15.1 RGI emoji lookup table with 973 single-codepoint entries. Names are CLDR short names, normalised to lowercase with spaces replaced by underscores.

```python
import slughorn

cp = slughorn.emoji.name_to_codepoint("dragon") # 0x1F409
name = slughorn.emoji.codepoint_to_name(0x1F409) # "dragon"

# Slack-style :colon: names are accepted too
cp = slughorn.emoji.slack_name_to_codepoint(":dragon:")
```

12.1.4 slughorn.freetype

The FreeType backend (available when built with `SLUGHORN_HAS_FREETYPE`). Three loading modes cover the common cases: ASCII, an explicit codepoint list, or the full font charmap. A fourth function, `load_emoji_font`, handles COLR layered emoji and returns a `dict[int, CompositeShape]` mapping codepoint to pre-built composite.

```python
import slughorn

atlas = slughorn.Atlas()
slughorn.freetype.load_font_glyphs(
    "/usr/share/fonts/truetype/dejavu/DejaVuSans.ttf",
    [ord(c) for c in "Hello, World!"],
    atlas,
)
atlas.build()
slughorn.write(atlas, "hello.slugb")
```

All loaders accept three optional keyword arguments mirroring `LoadConfig`:

- `strategy`: optional callable `(curves) -> (splits_x, splits_y)` for custom band placement
- `uniform`: if `True`, all glyphs share one em-space bounding box (required for `setLayerShapeIndex` cycling)
- `log`: optional callable `(level: int, msg: str)` for load-time diagnostics

```python
slughorn.freetype.load_font_glyphs(
    "/path/to/font.ttf",
    [ord(c) for c in "0123456789"],
    atlas,
    uniform=True,
    log=lambda level, msg: print(f"[ft:{level}] {msg}"),
)
```

12.1.5 slughorn.nanosvg

The NanoSVG backend (available when built with `SLUGHORN_HAS_NANOSVG`). Parses SVG files or strings and packs every filled shape into an atlas as a `CompositeShape` with one `Layer` per shape, back-to-front order preserved. Both solid-color and linear/radial gradient fills are handled; stroked shapes, clip paths, and text elements are not yet supported by this backend.

```
import slughorn

atlas = slughorn.Atlas()
composite = slughorn.nanosvg.load_file("icon.svg", atlas)

atlas.build()

slughorn.write(atlas, "icon.slugb")
```

Similar to the Canvas backend, the NanoSVG backend also makes use of the `KeyIterator` object, allowing the user more fine-grained control over the resultant `Shape` keys in the final atlas.

```
// We pass true as the "force" parameter, which ensures the internal element
// names are ignored; the default is to only use the KeyIterator when no "id"
// is found.
ki = slughorn.KeyIterator("svg", true)

composite_a = slughorn.nanosvg.load_file("a.svg", atlas, ki) # svg_0
composite_b = slughorn.nanosvg.load_file("b.svg", atlas, ki) # svg_1

atlas.build()
```

> The Cairo and Skia backends (`slughorn/cairo.hpp`, `slughorn/skia.hpp`) exist in C++ and will be bound to Python once a clean bridging path from a Python-side path type is identified. For now they are C++-only.

12.2 The `bin/slughorn` CLI

`bin/slughorn` is a standalone Python script that wraps the compiled extension into a six-subcommand CLI. No install step is required; it runs directly from the repo and auto-discovers the built extension module regardless of where `PYTHONPATH` points.

Subcommand	What it does
`info`	Print atlas summary; add a key to drill into one shape's metadata
`list`	List all keys; `--shapes` / `--composites` filters to one kind
`render`	Decode and rasterise a shape to a grayscale PNG
`svg`	Export decoded curves and band boundaries as an SVG
`shell`	Drop into an interactive REPL with the atlas pre-loaded (IPython if available)
`font`	Full pipeline: font file → packed `.slug` / `.slugb` atlas

```
# Inspect an atlas
slughorn info ui.slugb
slughorn info ui.slugb A # details for one shape

# Render and export
slughorn render ui.slugb A --size 512 --margin 0.05
slughorn svg ui.slugb A --size 1024

# Interactive exploration; 'atlas', 'shape', and 'decoded' are available
slughorn shell ui.slugb --key A

# Font packing (requires a --freetype build)
slughorn font /path/to/font.ttf output.slugb --ascii
slughorn font myfont.ttf output.slugb --chars "Health: 100%"
slughorn font myfont.ttf output.slugb --range U+2600-U+26FF
```

All subcommands that read an atlas accept both `.slug` and `.slugb`; the format is detected automatically from the file.

The `shell` subcommand deserves a special mention. It drops you into a full Python REPL (IPython, if installed; the stdlib `code` module otherwise) with `slughorn`, `atlas`, `make_key`, and optionally a pre-decoded `shape` and `sampler` already in scope. It is the fastest way to explore an unfamiliar atlas or prototype a new loading pipeline.

12.3 What's Coming

As of June 2026, a separate **AlphaPixel** project, `OSG.py`, is nearly ready for public use. Combined with the `osgSlug.py` bindings, the entire `slughorn → osgSlug → OSG` pipeline will be fully scriptable from Python — load shapes, configure the scene graph, and drive rendering without writing a line of C++.

13 Connecting to Your Graphics Backend

slughorn's output is fundamentally a fragment-shader problem. The Slug algorithm runs entirely in the fragment shader: it looks up bands, evaluates curve coverage, and decides per-pixel opacity. Every piece of per-vertex data exists because the fragment shader cannot derive it from what it already has. That is the right frame for understanding the data layout.

The division of responsibility is clean:

- **slughorn owns** — CPU-side curve baking, band packing, the data layout contract.
- **You own** — batching strategy, render pass, scene graph/ECS integration.

Two rendering paths are supported in `osgSlug`:

Path	Vertex attributes	GL requirement	`ShapeDrawable` type
SSBO (default)	2	GL 4.3 / GLSL 430	`SSBOShapeDrawable`
GL3 (opt-in)	8	GL 3.3 / GLSL 330	`GL3ShapeDrawable`

13.1 The SSBO Path (Default)

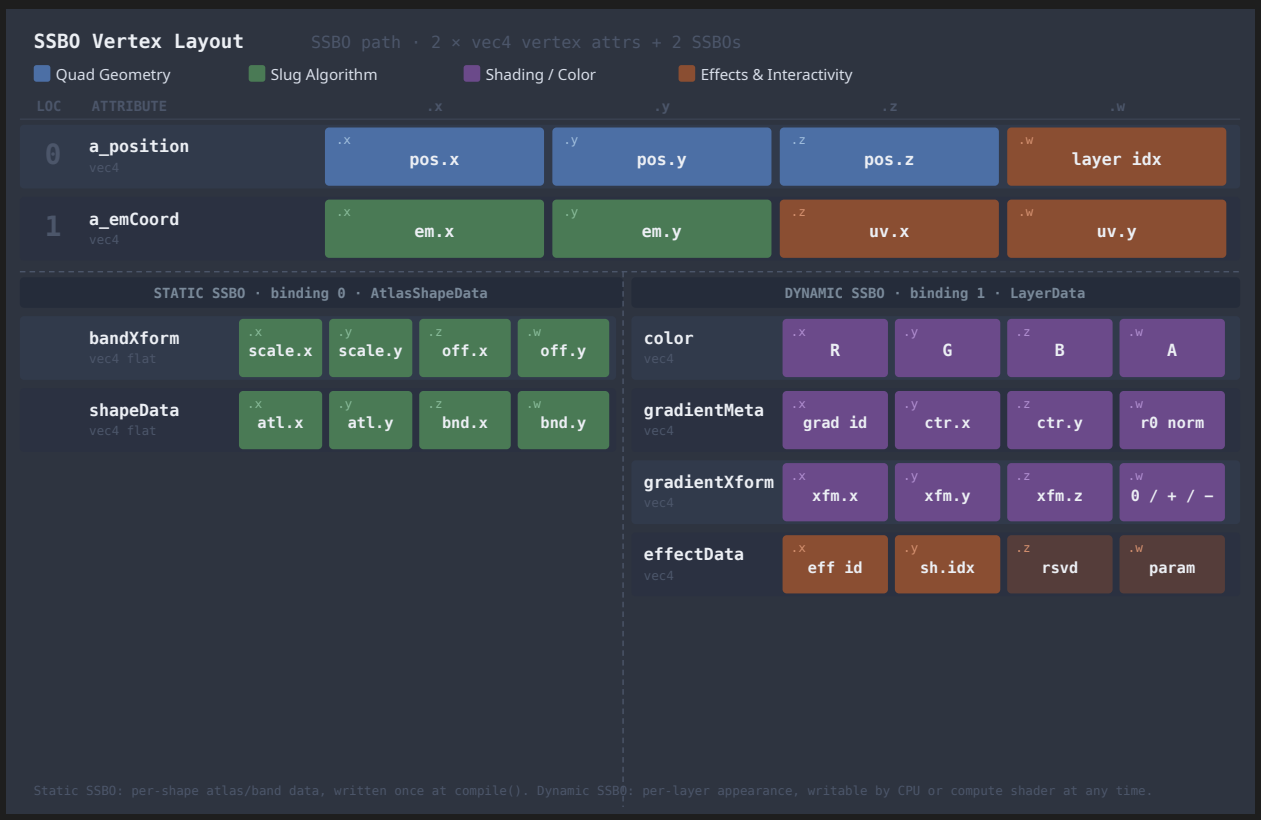


Figure 7: SSBO layout

The SSBO path splits per-shape data across two shader storage buffers bound at different points. The key insight is that some data is **atlas-static** (fixed once after `build()`) and some is **layer-dynamic** (can change per frame). Separating them makes runtime mutation nearly free.

13.1.1 Atlas Shape Buffer — Binding 0

Built once in `Atlas::packTextures()` and never written again. One entry per unique shape in the atlas; 3 vec4s (48 bytes) per entry.

Slot	Field	Content
0	`bandXform`	`bandScale{X,Y}, bandOffset{X,Y}`
1	`shapeData`	`bandTex{X,Y}, bandMax{X,Y}`
2	`originData`	`origin{X,Y}, 0, 0`

Bound once on the scene-graph node's state set via `createDefaultStateSet()`. Every drawable under that node shares it automatically.

13.1.2 Layer Buffer — Binding 1

Built at `compile()` time; one entry per layer in the drawable; 4 vec4s (64 bytes) per entry. **Runtime-mutable** — see Interactivity.

Slot	Field	Content
0	`color`	Flat RGBA
1	`gradientMeta`	`(gradientId, cx, cy, r0_norm)`
2	`gradientXform`	Type-encoded gradient transform
3	`effectData`	`(effectId, shapeIndex, 0, userParam)`

`effectData.y` carries the 0-based index of this layer's shape in the atlas shape buffer — the link between binding 1 and binding 0. `effectData.w` carries the world-space quad width used by `SubdividedDrawable` effects such as 9-slice scaling; it is 0 for flat quads.

13.1.3 Vertex Attributes

Only two vertex attributes are emitted:

Loc	Name	Type	Content
0	`a_position`	`vec4`	`xyz` = world-space corner; `w` = 1-based layer index into binding 1
1	`a_emCoord`	`vec4`	`xy` = em-space coordinate; `zw` = UV [0,1] per corner

13.1.4 The Geometry Loop

Each `Layer` in a `CompositeShape` produces one quad: four vertices, two triangles.

```
static constexpr slug_t EXPAND = 0.01_cv;
size_t layerIndex = 0;

for(const slughorn::Layer& layer : compositeShape.layers) {
    const auto shape = atlas.getShape(layer.key);

    if(!shape) { layerIndex++; continue; }

    // World-space quad corners
    Quad q = shape->computeQuad(layer.transform, layer.scale, cv(EXPAND));

    // Em-space corners — expand must match computeQuad exactly
    slug_t emX0 = shape->bearingX - EXPAND;
    slug_t emY0 = (shape->bearingY - shape->height) - EXPAND;
    slug_t emX1 = (shape->bearingX + shape->width) + EXPAND;
    slug_t emY1 = shape->bearingY + EXPAND;

    slug_t lidx = cv(layerIndex + 1); // 1-based, packed into a_position.w

    // Emit 4 vertices: a_position (xyz corner, lidx) + a_emCoord (em xy, uv)
    // Pack 4 vec4s into the layer buffer at layerIndex * 4

    layerIndex++;
}
```

> **The expand invariant:** the value passed to `computeQuad` and the value subtracted/added to the em-coord corners must be identical. If they drift, the coverage mapping breaks silently — the shape renders but with incorrect anti-aliasing.

13.2 The GL3 Path

The GL3 path replicates all per-shape data as per-vertex attributes instead of using SSBO buffers. It is the correct choice for GL 3.x hardware or when SSBO support is unavailable. Use `GL3ShapeDrawable` (or `GL3SubdividedDrawable`) to opt in; the atlas build, texture upload, and blend mode are identical between the two paths.

The vertex attribute layout diagram below documents the GL3 path.

Loc	Name	Type	What it carries	Why the shader needs it
0	`a_position`	`vec3`	World-space quad corner	Places the quad

Loc	Name	Type	What it carries	Why the shader needs it
1	`a_color`	`vec4`	Flat RGBA	Shading output; `rgb` ignored when `gradientId != 0`, `a` stays active
2	`a_emCoord`	`vec2`	Em-space coordinate for this corner	Primary input to the coverage test
3	`a_bandXform`	`vec4`	`(bandScaleX, bandScaleY, bandOffsetX, bandOffsetY)`	Maps em-coords to indirection-table slot indices; `flat` across the quad
4	`a_shapeData`	`vec4`	`(bandTexX, bandTexY, bandMaxX, bandMaxY)`	Band texture origin and max band counts; `flat` across the quad
5	`a_effectId`	`float`	Per-layer effect policy ID	Selects per-layer vertex shader behaviour; `0` = standard fill
6	`a_gradientMeta`	`vec4`	`(gradientId, cx, cy, r0_norm)`	Selects gradient texture row; carries radial center and inner radius
7	`a_gradientXform`	`vec4`	Type-encoded gradient transform	Maps em-space → gradient `t` value

Note that `a_bandXform` and `a_shapeData` carry identical values for all four vertices of a quad. The `flat` qualifier prevents the driver from interpolating them, which would corrupt their integer-aligned fields.

13.3 Texture Upload

Three textures must be uploaded after `build()`:

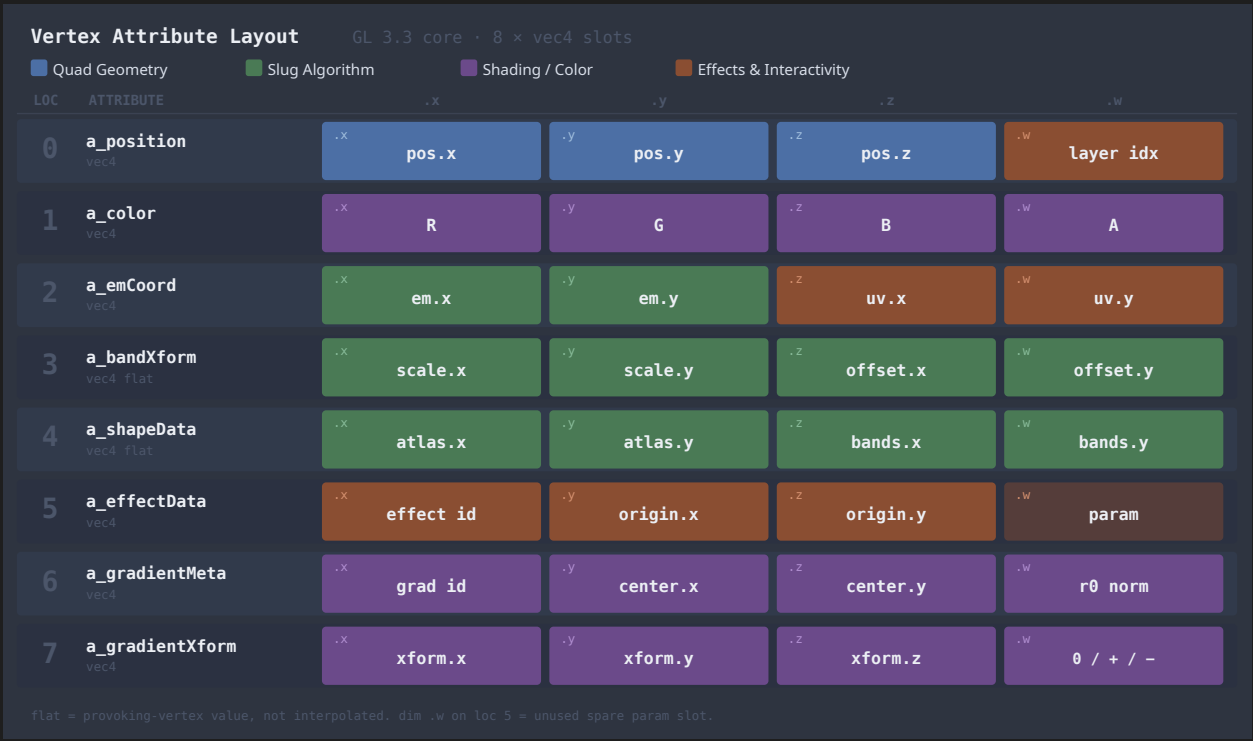


Figure 8: Vertex attribute layout

Texture	Format	Filter	Sampler type	Why
Curve	RGBA32F	`GL_NEAREST`	`sampler2D`	Fetched with `texelFetch` — exact texel addressing
Band	RGBA16UI	`GL_NEAREST`	`usampler2D`	Same; integer format, must not be filtered
Gradient	RGBA8	`GL_LINEAR`	`sampler2D`	Sampled with `texture()` — hardware bilinear gives free stop interpolation

```
// Curve — RGBA32F
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F,
    td.width, td.height, 0, GL_RGBA, GL_FLOAT, td.bytes.data());

// Band — RGBA16UI  (note GL_RGBA_INTEGER internal format)
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16UI,
```

```
    td.width, td.height, 0, GL_RGBA_INTEGER, GL_UNSIGNED_SHORT, td.bytes.data());

// Gradient — RGBA8 (skip entirely if getGradientTextureData().height == 0)
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8,
    gd.width, gd.height, 0, GL_RGBA, GL_UNSIGNED_BYTE, gd.bytes.data());
```

13.4 Blend Mode

The correct blend equation for slughorn content is premultiplied alpha:

```
glBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
```

When gradients are active the shader combines coverage and gradient alpha into a premultiplied result; straight-alpha blending (`GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA`) will produce incorrect compositing for semi-transparent gradient shapes.

13.5 Reference Implementations

osgSlug is the primary reference implementation and has been developed in sync with slughorn from the beginning. `SSBOShapeDrawable` is the default drawable type; `GL3ShapeDrawable` is available via explicit instantiation for GL3-only targets. Its `StateSet` manages the full texture and uniform binding chain including gradient support, all debug modes, and the stem-darkening / gamma text path. For OSG-based applications it is the path to follow. For other scene graphs, `SSBOShapeDrawable::compile()` is the authoritative reading of the geometry loop.

The example `osgslug-ssbo` demonstrates the runtime mutation pattern: layer colors and effect IDs updated via `getLayerBuffer()` + `dirty()` with no VBO rebuild.

slughorn-example-glfw (`example/slughorn-example-glfw.cpp`) is a minimal single-file raw OpenGL example with no framework dependencies. It covers the core geometry loop and texture upload and is useful as the simplest possible end-to-end reading of the pipeline in isolation.

14 Interactivity

slughorn's GPU architecture makes runtime mutation unusually cheap. The two-SSBO split in the SSBO path (see Connecting to Your Graphics Backend) means that changing a shape's color, swapping its effect, or redirecting its gradient is a direct memory write followed by a single `dirty()` call — no vertex data is touched, no `compile()` is required.

14.1 Runtime Mutation via the Layer Buffer

After `compile()`, every `SSBOShapeDrawable` and `SSBOSubdividedDrawable` holds a reference to its layer buffer via `getLayerBuffer()`. Each layer occupies 4 consecutive vec4 entries at offset `layerIndex * 4`:

Offset	Field	What to write
`+ 0`	`color`	New RGBA (premultiplied)
`+ 1`	`gradientMeta`	x = new gradientId (0 = flat color)
`+ 2`	`gradientXform`	Gradient transform (normally fixed at compile time)
`+ 3`	`effectData`	x = effectId; y = shapeIndex; z = 0 (reserved); w = userParam

A typical color-swap in an `UpdateCallback` for `osgSlug` would look like:

```
// Adjust ONLY the 0th Layer, triggering a "SubData" sync
sd->setLayerColor(0, {1_cv, 0.5_cv, 0.5_cv, _alpha});
sd->dirtyLayers(0);
```

That is the entirety of the hot path for a runtime color change. No geometry is rebuilt; no texture is re-uploaded; no state set is recreated.

> **What `dirty()` does:** it sets a flag on the buffer object causing OSG (and the driver) to upload the changed range to the GPU via `glBufferSubData` at the next render traversal. The upload is lazy — multiple writes between frames cost one transfer, not one per write.

14.2 The effectId System

`effectId` is a per-layer integer packed into `effectData.x`. The vertex shader in osgSlug reads it and branches to apply a per-layer geometric transformation before passing the vertex to the fragment shader.

Changing an effect at runtime is the same buffer write described above — overwrite `effectData.x` and call `dirty()`.

14.3 Layer Masking

The `osgSlug_layerMask` uniform controls per-layer visibility without touching any geometry. It is a bitmask where bit `N + 1` (1-based) controls layer `N`:

```
// Hide layer 1 (second layer), show everything else:
stateSet->addUniform(new osg::Uniform("osgSlug_layerMask", 0b00000100));
```

A mask value of `0` (the default) shows all layers. Because masking happens in the fragment shader at coverage-test time it has no CPU cost and no draw-call overhead.

Practical uses: toggling HUD sub-panels on/off, A/B testing visual variants in a composite, progressive reveal of multi-layer text decorations.

14.4 SubdividedDrawable and Mesh-Level Animation

`SSBOSubdividedDrawable` replaces the flat four-vertex quad with a dense mesh driven by a `PositionCallback`:

```
auto* sd = new osgSlug::SSBOSubdividedDrawable();

sd->setStepsU(32);
sd->setStepsV(8);

sd->setPositionCallback([](slug_t u, slug_t v) -> osgSlug::Vec3 {
    // u, v in [0, 1]; return world-space position
    return {u * width, v * height + sin(u * PI_CV) * amplitude, 0_cv};
});
```

The em-coord mapping is computed from `(u, v)` independently of the position callback, so the position callback can freely distort world-space geometry — curved monitors, spheres, banners wrapped around cylinders — without affecting the Slug coverage calculation.

14.5 Path-Following Animation

`Canvas::Path` provides arc-length-parameterised sampling of any path authored via the Canvas API. Combined with `osg::MatrixTransform` and an `UpdateCallback` it places and orients a shape along a curve each frame at zero CPU cost:

```
slughorn::Canvas canvas;

// ... build a path with moveTo/lineTo/quadTo/bezierTo/arcTo ...
auto path = canvas.path(); // snapshots the current path; non-destructive

struct PathFollowCallback: public osg::NodeCallback {
    slughorn::Canvas::Path path;

    float speed;
```

```
    void operator()(osg::Node* node, osg::NodeVisitor* nv) override {
        auto* mt = static_cast<osg::MatrixTransform*>(node);
        auto t = fmod(nv->getFrameStamp()->getSimulationTime() * speed, 1.0f);
        auto s = path.sample(cv(t)); // { x, y, angle }

        osg::Matrix m;

        m.makeRotate(s.angle, osg::Vec3f(0, 0, 1));
        m.setTrans(s.x, s.y, 0);

        mt->setMatrix(m);

        traverse(node, nv);
    }
};

auto* mt = new osg::MatrixTransform();

mt->addChild(shapeDrawable);
mt->setUpdateCallback(new PathFollowCallback{path, 0.2f});
```

`Canvas::Path::sample(t)` returns a `Sample { x, y, angle }` where $t \in [0, 1]$ is normalized arc-length and `angle` is the tangent direction in radians. Multiple shapes can sample the same `Path` at different `t` offsets to produce a convoy or flight-formation effect.

14.6 Putting It Together: A Clock

The clock demo (`osgslug-clock.cpp`) ties all the above patterns together: three shapes in one `CompositeShape`, each with `Origin::Centered` so their quads all anchor at the same world point, and one shape (the hand) driven by `effectId 8` (rigid rotation about its origin).

The key lesson from the clock: `Origin::Centered` (or a custom `Origin` at the shared pivot) makes all layers in the composite share the same `transform.x/y`, so `computeQuad` places all quads around the same world point without any manual coordinate arithmetic.

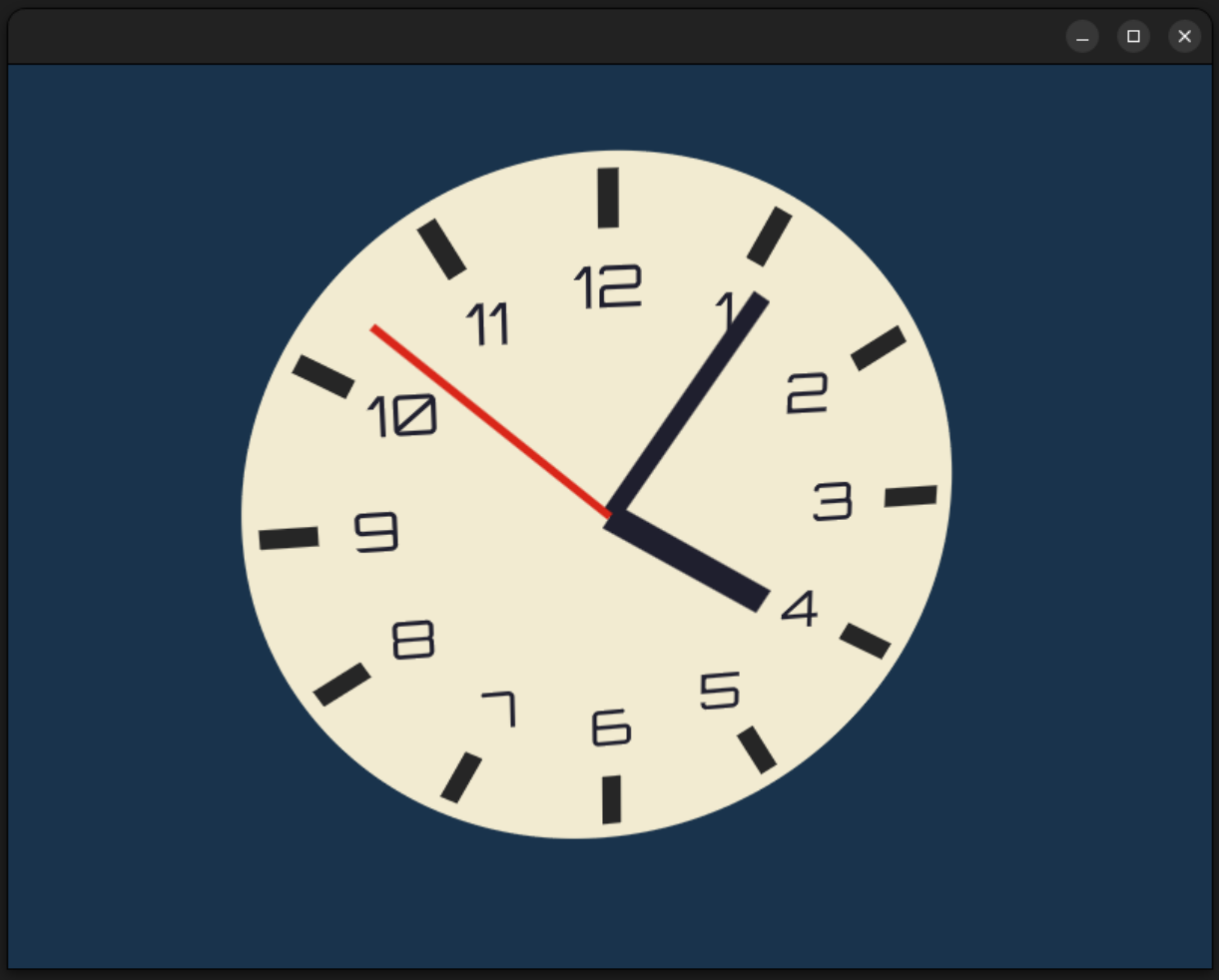


Figure 9: osgslug-clock

15 CPU Rendering

`slughorn/render.hpp` is a pure C++ port of the GPU coverage formula — the same analytic Slug algorithm that runs in the fragment shader, re-implemented for the CPU. It has no GPU dependency and no Python dependency; it is a standalone header.

15.1 What It Is

`render.hpp` exposes three types in the `slughorn::render` namespace:

Type	Purpose
`Sample`	Result of evaluating coverage at a single (x, y) point
`Sampler`	Per-shape decoding context; owns the reconstructed curve list
`Grid`	Row-major `float` grid returned by `renderGrid()`

A `Sampler` is created by calling one of two `decode()` overloads:

```
#include "slughorn/render.hpp"

// Primary overload — requires a fully-built atlas
slughorn::render::Sampler s = slughorn::render::decode(atlas, key);

// Serial overload — reconstructs from raw texture data; no full atlas needed
slughorn::render::Sampler s = slughorn::render::decode(shape, curveTex, bandTex);
```

Once you have a `Sampler`, you can render:

```
// Single point — returns a Sample with fill, xcov, ycov, xwgt, ywgt, iters
slughorn::render::Sample p = s.renderSample(x, y);

// Single point using band-accelerated path (faster on glyphs with many bands)
slughorn::render::Sample p = s.renderSampleBanded(x, y);

// Full grid — returns a Grid (row-major float vector, width×height)
slughorn::render::Grid g = s.renderGrid(width, height);

// Access a pixel: row = y, col = x
float coverage = g.at(row, col);
```

15.2 What It's For

Primary use cases:

- **Debugging band placement.** Render a shape CPU-side and overlay the band boundaries to see exactly what the GPU sees. The `bin/slughorn svg` subcommand uses this path.
- **Testing.** `test/slughorn-test-render` uses `renderGrid()` to verify that shapes produce non-zero coverage and produce recognizable ASCII art — a fast sanity check with no GPU.
- **Advanced Processing.** The curve data reconstructed by `decode()` is the same data stored directly on `Atlas::Shape::curves` (populated at `build()` time and at `serial::read()` time). Both C++ and Python callers can access raw glyph geometry without re-running a font backend.

15.3 Python Surface

The `slughorn.render` submodule wraps `render.hpp` directly. The API is unchanged from before the C++ extraction — the speedup (0.8–1.0 s → 0.118 s per shape) is a side-effect of moving the hot coverage loop out of pybind11's shadow, where the compiler can vectorize it.

```
import slughorn
import slughorn.render as sr

atlas = slughorn.Atlas()
# ... load shapes, build ...

sampler = atlas.decode(key) # → slughorn.render.Sampler
grid = sampler.render_grid(64) # → float32 numpy array, shape (64, 64)
sample = sampler.render_sample(0.5, 0.5) # → slughorn.render.Sample
print(sample.fill, sample.xcov, sample.ycov)
```

`atlas.decode(key)` is the Python spelling of `slughorn::render::decode(atlas, key)`.

15.4 `Atlas::Shape::curves`

Raw em-space curves are stored permanently on every `Atlas::Shape` after `build()` or `serial::read()`:

```
// Works at any build lifecycle stage — pre-build returns font metrics + em-space curves;
// post-build also includes GPU band fields. Returns nullopt if key not registered.
const auto shape = atlas.getShape(key);
if(shape) {
    // shape->curves is always valid — no need to call decode() just to get geometry
    // shape->advance, shape->bearingX/Y, shape->width, shape->height all available
}
```

For contour-aware curve access (stroking, canvas baking), use the separate accessor:

```
// Pre-split into closed contours. Contour breaks are detected where
// p3 of curve[i] != p1 of curve[i+1]. Always returns a vector (empty if not found).
Atlas::Contours contours = atlas.getShapeContours(key);
```

Use `shape->curves` (via `getShape`) when passing data to a pixel-level rasterizer (contour order doesn't matter). Use `getShapeContours` when building a `Path` for stroking or filling — each contour must be handled independently to avoid phantom connector strokes across subpath breaks.

`Canvas::strokeText()` uses `getShapeContours()` internally for exactly this reason.

16 Debugging & Diagnostics

Most of the diagnostic tooling lives in **osgSlug** — it is inherently GPU-side work. slughorn itself contributes a few CPU-side and Python-side tools. This section covers both layers and is clear about which belongs where.

16.1 slughorn-side tools

16.1.1 Python band visualizer

The `slughorn.render` submodule can export a shape's curve and band structure as an image or SVG with band boundary overlays drawn on top. This is the primary tool for diagnosing band placement problems — too few bands (bad AA on dense glyphs), bands in the wrong place (dark artifact lines), or an indirection table that isn't snapping boundaries correctly.

```
import slughorn.render as sr

# PNG with band boundary overlay
sr.save_curves("my_shape.slug", "key", "debug.png", show_bands=True)

# SVG version (scalable, easier to inspect at high zoom)
sr.save_curves_svg("my_shape.slug", "key", "debug.svg", show_bands=True)
```

Band boundaries are derived from the indirection table transitions — the same O(1) lookup the shader uses — so what you see in the image is exactly what the GPU sees.

16.1.2 CLI subcommands

The `bin/slughorn` CLI has three subcommands useful for debugging:

```
# Print a full atlas summary: shape count, band counts, texture dimensions, gradient list.
slughorn info my_atlas.slug

# Render one shape or composite to a PNG — no GPU required.
slughorn render my_atlas.slug my_key out.png

# Export a shape's curves and band boundaries as a scalable SVG.
slughorn svg my_atlas.slug my_key out.svg
```

`render` uses the same GLSL emulator as `slughorn.render`, so it faithfully mirrors what osgSlug will produce (minus effects and gradients, which are GPU-only).

`slughorn svg` draws every curve in the shape as a quadratic Bézier with control-point indicators and overlays the band grid in red. It is the fastest way to inspect raw path geometry — especially useful for diagnosing stroke problems, where the curve count alone tells you whether tessellation is behaving correctly. To serialize the atlas from C++ for use with the CLI, pipe it through

`slughorn::serial::writeJSON`:

```
// Serialize to JSON for inspection or CLI use:
slughorn::serial::writeJSON(*atlas, std::cerr);

// Or write a .slug file the CLI can open directly:
slughorn::serial::write(*atlas, "debug.slug");
```

16.1.2.1 Diagnosing stroke geometry with `slughorn svg` The most effective use of `slughorn svg` is a **before/after comparison**. When a stroke renders incorrectly, export the broken shape and a known-good reference and compare raw file sizes:

```
# Broken version (large coordinates, stroke looks dashed):
slughorn svg broken.slug stroke_key broken.svg

# Known-good version (normalized coordinates, stroke looks correct):
slughorn svg okay.slug stroke_key okay.svg
```

A 10× or greater difference in line count is an immediate red flag: the broken shape has far more curves than the good one. Open both in a browser and zoom into the corners — if curves are "bunched up" with dozens of points at the same integer coordinate, tessellation has exploded. The fix is an adaptive flattenCurve tolerance, not band tuning.

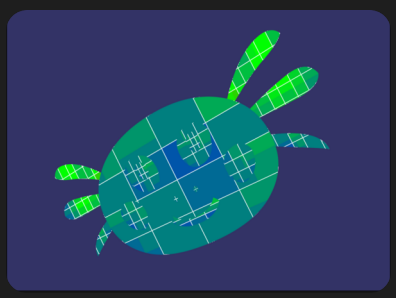
This exact workflow has revealed many bugs during development.

16.2 osgSlug-side Tools

16.2.1 Debug modes — `osgSlug_debugMode`

Set this integer uniform on the geode's state set to switch all visible shapes into a diagnostic view. Mode 0 is normal rendering; all other modes replace or augment the output color. Use the environment variable `OSGSLUG_DEBUG=#` to enable these modes.

Mode	Name	What it shows	Example
1	Checkerboard	Alternating dark/light fill per band cell — confirms band count and grid alignment at a glance	

Mode	Name	What it shows	Example
2	Band edge lines	1px anti-aliased lines drawn at every band boundary, inverted against the shape color	
3	Quad border	Pixel-perfect border around each shape's bounding quad; border color is a deterministic hash of the shape's band transform, so each shape gets a unique color	
4	Iteration heatmap	Fragment color encodes Slug iteration count: blue = cheap, green = moderate, red = expensive	
5	Heatmap + grid	Mode 4 with white 1px band grid lines overlaid — see cost and band layout simultaneously	
6	Show quads	Fully uncovered fragments render as half-white instead of being discarded — reveals the extent of every shape's bounding quad, useful for spotting oversized `expand` values	

Mode 1 (checkerboard) is the fastest way to confirm that your `SplitStrategy` is doing what you expect — each band cell gets a distinct color, so you can count bands and check that boundaries are landing where the strategy placed them.

Mode 4/5 (heatmap) is the right tool for performance work. High iteration counts (red) on a dense

glyph mean the band placement is forcing the shader to evaluate too many curves per fragment. Adding more bands — or moving existing ones to where the curves actually cluster — will shift those fragments toward blue.

Mode 3 (quad border) uses the true `[0,1]` UV coordinates baked into `a_emCoord.zw`, so the border is exactly one pixel wide regardless of shape size or camera distance. The hashed border color lets you visually distinguish overlapping quads in a dense composite.

16.2.2 Layer masking — `osgSlug_layerMask`

Set this integer uniform to isolate or hide individual layers in a `CompositeShape` without rebuilding geometry. The value is a bitmask: bit N+1 controls layer N (1-based, matching the layer index packed into `a_position.w`). The default value of 0 means all layers visible.

```
// Show only layer 0 (bit 1 set — all other bits clear):
sdg->getOrCreateStateSet()->addUniform(
    new osg::Uniform("osgSlug_layerMask", 0b10)
);

// Hide layer 1, show everything else:
sdg->getOrCreateStateSet()->addUniform(
    new osg::Uniform("osgSlug_layerMask", ~0b100)
);

// Back to all-visible:
sdg->getOrCreateStateSet()->addUniform(
    new osg::Uniform("osgSlug_layerMask", 0)
);
```

This is particularly useful when inspecting COLRv1 emoji (which may have 20+ gradient layers) or a complex multi-layer HUD composite where one layer is visually obscuring another.

> **Capacity:** the bitmask is a 32-bit int, so up to 30 layers can be individually controlled (bits 1–30). Layers beyond 30 are always visible when the mask is non-zero, at least for the time being.

16.2.3 Text Rendering Flags

Three uniforms control rendering quality for `osgSlug::Text` layers. They have no effect on non-text shapes.

Uniform	Type	Default	Effect
`osgSlug_textMode`	`bool`	`false`	Switches from `slug_Render` to `slug_RenderText` (MSAA + sub-pixel AA); enables stem darkening and gamma
`osgSlug_stemDarken`	`bool`	`false`	Applies stem darkening to partially-covered edge fragments — thickens thin stems at small sizes

Uniform	Type	Default	Effect
`osgSlug_gamma`	`float`	`1.0`	Gamma exponent applied to edge coverage after stem darkening; values < 1 brighten edges

`osgSlug_textMode` is the master switch. `osgSlug_stemDarken` and `osgSlug_gamma` are no-ops when `osgSlug_textMode` is false.

16.2.4 Text Pixel Alignment — `OSGSLUG_TEXT_PIXEL_ALIGN`

Set this environment variable to `1` (or `true`) to snap each glyph's advance position to the nearest integer pixel boundary before placing the next glyph. This prevents sub-pixel cursor drift from accumulating across a long text run at small sizes.

```
OSGSLUG_TEXT_PIXEL_ALIGN=1 ./my-osgslug-app
```

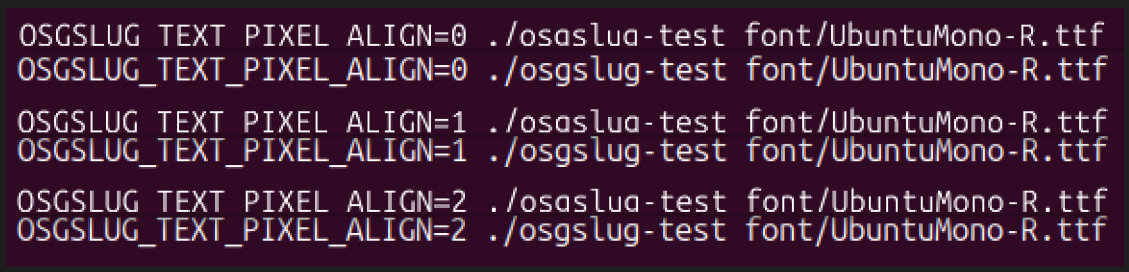


Figure 10: Text pixel alignment comparison

Each panel shows osgSlug text (top) positioned directly above the equivalent gnome-terminal output (bottom) for direct comparison. Mode 0: no snapping — sub-pixel drift accumulates across the run. Mode 1: `round()` — clear improvement, consistent inter-glyph spacing. Mode 2: `round() + 0.5` — negligible difference from mode 1 at the sizes tested.

The snapping currently uses `std::round(cursorX)`. A half-pixel variant (`std::floor(cursorX) + 0.5`) — matching Cairo's convention of placing glyph origins on texel centers rather than texel edges — was tested and produced no perceptible difference at the sizes evaluated, so both variants remain available for further experimentation.

> **Future direction — matching FreeType raster quality:** For applications that need small-size text to match a CPU rasterizer (terminal emulators, constrained ortho-2D UIs), the planned approach is a **hybrid renderer**: slughorn/osgSlug handles large text, animations, and HUD shapes; a pre-rendered FreeType bitmap atlas takes over below a configurable size threshold. The crossover is invisible when both sides use grayscale AA and a matched gamma. Subpixel LCD rendering in the fragment shader (a deferred milestone) is the alternative path that would narrow the gap without a fallback. See `WhatsNext` for detail.

16.3 Workflow Notes

A typical debug session for a new shape looks like this:

0. **`slughorn svg`** first if you're working with a Canvas stroke — confirm the curve count is reasonable and corners aren't exploding. A 10× size difference between a working and broken export means it's a geometry problem, not a GPU problem. Fix it there before touching band settings.
1. **Mode 6** — confirm the quad is the right size and position. Oversized quads waste fill rate; undersized quads clip the shape.
2. **Mode 1** — count the bands. Compare against your `SplitStrategy` expectation.
3. **Mode 2** — check boundary positions. A boundary in the wrong cell produces a visible artifact line in normal rendering.
4. **Mode 4/5** — check iteration cost. If you see red on a shape that doesn't need it, add bands in the expensive region.
5. Back to **mode 0** — confirm the shape looks correct in normal rendering.

For multi-layer composites, combine `osgSlug_layerMask` with the debug modes: isolate one layer at a time in mode 1 or mode 4 before going back to the full composite.

17 How This All Started

In early April of 2026, we noticed that Pelican Mapping had added "Slug text support" to osgEarth. Having long been obsessed with all things *vector graphics*, I found myself **shocked** at not having any idea what Slug actually was! Once we had extracted the Slug-related code out into its own standalone example, the first question we began asking ourselves was: *what else can we DO with this!?*

After getting approval to do some research on the subject, the first experiment was straightforward: I would "inject" the necessary curve data directly into the existing, heavily font-centric code, replacing the letter `F` with a simple 3-curve triangle. To my surprise, it worked without a hitch (and you can **still** see this remnant in the `osgslug-simple.cpp` example, which continues to perform that exact same step).

Figure 11: Original Triangle Test

After that, the floodgates opened. I quickly started partitioning the code into reusable pieces of generic C++20, and that became the basis for slughorn itself. osgSlug was begun *alongside* slughorn, serving as the de facto "testbed" for our development and experimentation. I rapidly

added support for ingesting Cairo, Skia, NanoSVG and FreeType2 data via a common interface: `slughorn::CurveDecomposer`. Once that all fell into place, the natural next step was a slughorn-native authoring API, from which `slughorn/canvas.hpp` was born.

Other features came quickly (such as the Python bindings, seeing as how we'd been working on modern OpenSceneGraph bindings, and already had a great deal of experience coercing `pybind11` into behaving how we needed), glTF-compatible serialization, the `slughorn.render` submodule (a full GLSL Slug simulator in C++), as well as experimentation into how we might be able to not only *use* Slug but actually **contribute back** to it in the form of the `slughorn::SplitStrategy` approach (which continues to be *experimental*, but promising).

17.1 Where It's All Going

At this point, we're confident saying that slughorn has an impressive set of features, and this guide is almost *certain* to leave many of them out! The best way to see what the future holds — which will be guided **entirely** by community interest and support — is to check out our evolving What's Next. If any of these features would be valuable to you *sooner rather than later*, feel free to reach out and let us know!

18 AI Disclosure & Sponsorship

The entirety of this work was funded by AlphaPixel Development, a respected leader in graphics software development and contracting/consulting, specializing in performance and GPU challenges.

We developed this library to be the definitive method for bringing Eric Lengyel's incredible Slug algorithm to the developer community, and open sourced it with a permissive license so there is no reason to ever re-invent it. We want this to be the best and only Slug implementation so everyone can share and benefit from all improvements.

> Both slughorn and osgSlug were developed in a "pair-programming" style using both `claude` and `codex`. **Nothing was "vibe-coded"**. Instead, we used AI for research, test generation, automating small changes, and exploring how *other projects* had solved similar problems.

That being said, we also have bills to pay and groceries to buy.

If you're interested in implementing Slughorn in your software project, consider contracting us to do the work. We are the world's leading experts in this library, and we can probably implement it into almost any software for less total investment than anyone else, including doing it in-house on your own. If you'd like to see some aspect of Slughorn extended or improved (especially the things you see in the What's Next document), we are the obvious candidate to implement those improvements. We've already researched and dabbled into many of them, but haven't been able to invest the resources to complete them. With your patronage, we can finish the work we've already begun.

And finally, if you have difficult problems you need solved that DON'T involve Slug and Slughorn, consider talking to us. For literal decades we have been experts in OpenGL, Vulkan, OpenSceneGraph, VulkanSceneGraph, Qt, typography rendering, GIS/mapping, 3D, VR/AR/xR and all kinds of fun technologies. You may have even used code we wrote in the past without knowing it, as we develop and contribute to many open source projects.

What difficult problem of yours can we solve today?

19 Changelog

19.1 v0.1.0 — First Release

Initial public release.