nAIVE

The AI-Native Game Engine

Product Requirements Document & Technical Specification

Version 1.0

February 2026

ASU Endless Games and Learning Lab

CONFIDENTIAL

Property	Value
Document Type	Product Requirements Document (PRD) & Technical Specification
Project	nAIVE Engine + Relic Demo Game
Primary Target	Claude Code implementation agent
Language	Rust (engine), SLANG (shaders), Lua (scripting), YAML (data)
Platform	macOS (Apple Silicon primary), Linux, Windows, WebGPU
Rendering	wgpu + SLANG shader compiler

Status	Greenfield — build from scratch

Table of Contents

Executive Summary

nAIVE (AI-Native Interactive Visual Engine) is a new game engine designed from the ground up for AI-driven game development. Unlike Unity and Unreal, which invest 80% of their engineering in graphical editors, nAIVE has no visual editor. The AI agent IS the editor.

The core thesis: with modern AI coding agents (Claude Code), AI asset generators (Meshy, ElevenLabs), and AI vision models, the entire game development workflow can happen through text files, CLI commands, and pipes. The graphical editor is dead weight.

This thesis was validated at a Supercell AI Hackathon where a complete game was built using Claude Code as the sole development interface, with Unity reduced to nothing more than a build runner and play-mode viewer.

Design Philosophy

Everything is a file. Scenes, entities, materials, render pipelines, input bindings, event schemas — all human-readable YAML files that an AI agent can read, understand, and modify.

Unix pipes connect everything. AI services (Meshy for 3D, ElevenLabs for audio, Flux for textures) are accessed via standardized CLI pipes that read from stdin and write to stdout or files.

SLANG shaders from day one. Nvidia's SLANG shading language compiles to Metal (macOS), SPIR-V (Vulkan), WGSL (WebGPU), and CUDA. It supports automatic differentiation for neural rendering, modules for composable shader code, and a reflection API that auto-generates GPU resource bindings.

Gaussian Splatting as first-class primitive. 3D Gaussian Splats are not a plugin. They are a native renderable type alongside traditional meshes, with hybrid depth compositing built into the render pipeline.

AI observability built in. The engine can stream frames for AI vision analysis, expose game state via a command socket, log all events for AI consumption, and accept simulated input for automated playtesting.

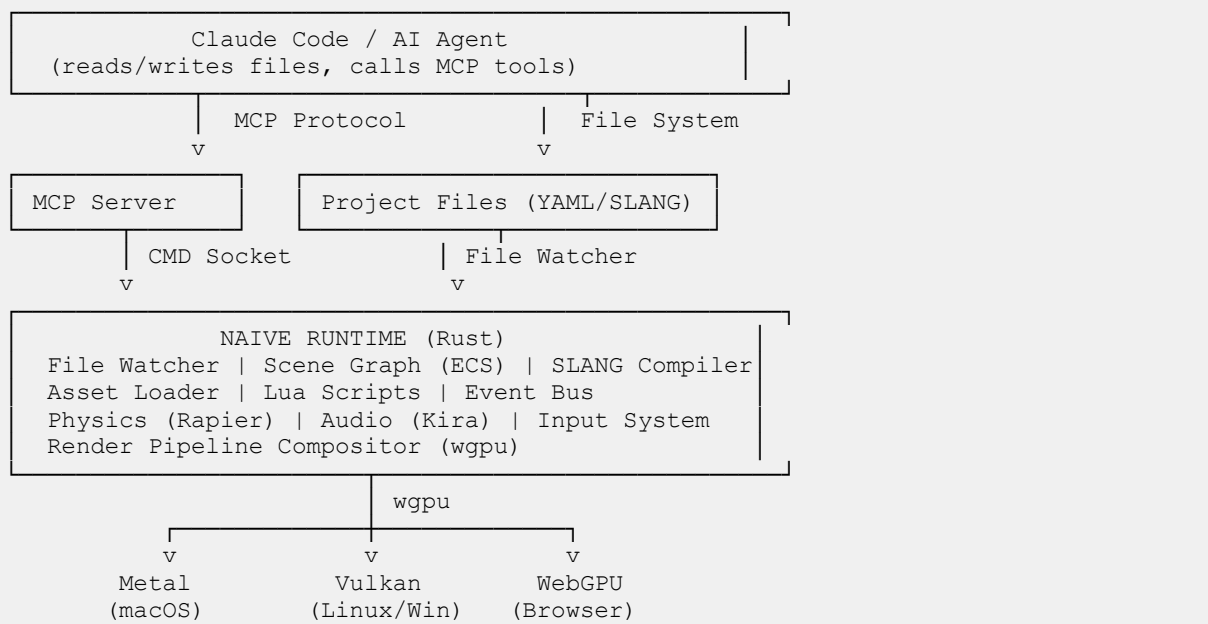
Target Deliverables

- **nAIVE Runtime:** A Rust-based game engine that loads YAML scenes, compiles SLANG shaders, renders via wgpu, runs Lua scripts, and supports hot-reload of all assets.
- **nAIVE CLI:** Command-line tools for project creation, building, testing, asset generation, and runtime control.
- **nAIVE MCP Server:** A Model Context Protocol server that gives Claude Code native tool access to the running engine.
- **Relic Demo Game:** A single-room dungeon crawler that demonstrates every engine feature: Gaussian splat environment, mesh-based interactive objects, dynamic lighting, combat, item collection, and AI playtesting.

Architecture Overview

The nAIVE engine follows a layered architecture where each layer communicates through well-defined file formats and protocols.

System Architecture Diagram



Core Technology Decisions

Decision	Choice	Rationale
Language	Rust	Memory safety without GC, excellent GPU bindings, growing gamedev ecosystem
GPU Abstraction	wgpu	Cross-platform (Metal/Vulkan/WebGPU), pure Rust, web export for free
Shader Language	SLANG	Compiles to MSL/SPIR-V/WGSL, auto-diff, modules, reflection API, Khronos-backed
ECS	hecs	Lightweight, no framework lock-in, fast iteration
Scene Format	YAML	Human/AI readable, diffable, standard tooling
Mesh Format	glTF 2.0	Industry standard, every AI tool exports it
Scripting	Lua 5.4 (mlua)	Small runtime, fast, AI generates near-perfect Lua
Physics	Rapier 3D	Pure Rust, deterministic, fast
Audio	Kira	Game-focused audio, spatial sound, pure Rust
File Watching	notify	Cross-platform filesystem event detection
IPC	Unix domain socket + JSON	Simple, fast, works with any language

Project File Structure

Every nAIVE project follows this directory layout. All files are human-readable and AI-editable. No binary proprietary formats.

Directory Layout

```
project/
├── project.yaml              # Project manifest (name, version, settings)
├── scenes/
│   └── relic.yaml              # Scene definitions (entity graph)
├── entities/
│   ├── player.yaml            # Reusable entity templates
│   ├── guardian.yaml
│   ├── door.yaml
│   ├── torch.yaml
│   └── relic_chalice.yaml
├── assets/
│   ├── meshes/                # .gltf/.glb files
│   ├── splats/                # .ply Gaussian splat files
│   ├── textures/              # .png, .ktx2 textures
│   ├── materials/             # .yaml material definitions
│   └── audio/                 # .wav, .ogg audio files
├── shaders/
│   ├── modules/               # Shared SLANG modules
│   │   ├── camera.slang
│   │   ├── pbr.slang
│   │   ├── lighting.slang
│   │   └── math_utils.slang
│   ├── passes/                # Render pass shaders
│   │   ├── gbuffer.slang
│   │   ├── gaussian_splat.slang
│   │   ├── deferred_light.slang
│   │   ├── composite.slang
│   │   └── tonemap.slang
│   └── effects/               # Post-processing effects
│       ├── torch_flicker.slang
│       └── glow.slang
├── logic/
│   ├── player_fps.lua         # Gameplay scripts
│   ├── guardian.lua
│   ├── door_swing.lua
│   ├── torch_flicker.lua
│   └── relic_pickup.lua
├── input/
│   └── bindings.yaml          # Input action mappings
├── pipelines/
│   └── render.yaml            # Render pipeline definition
├── events/
│   └── schema.yaml            # Event type definitions
├── tests/
│   └── test_relic.lua          # Automated playtest scripts
└── .naive/
    └── ci.yaml                # CI/CD pipeline definition
```

File Format Summary

File Type	Format	Purpose	Hot Reload

.yaml (scene)	YAML	Scene graph: entity hierarchy, component data	Yes — diffs and patches ECS
.yaml (entity)	YAML	Reusable entity templates with component definitions	Yes
.yaml (material)	YAML	Material properties: shader ref, textures, uniforms	Yes — updates GPU uniforms
.yaml (pipeline)	YAML	Render pass graph: inputs, outputs, shader refs	Yes — recompiles pipeline
.yaml (bindings)	YAML	Input action mappings: keyboard, mouse, gamepad	Yes
.yaml (events)	YAML	Event type schemas with payload definitions	Yes
.slang	SLANG	GPU shaders: vertex, fragment, compute	Yes — recompiles via SLANG
.lua	Lua 5.4	Gameplay scripts with lifecycle hooks	Yes — preserves state
.gltf/.glb	glTF 2.0	3D mesh assets with materials and animations	Yes — rebuilds GPU buffers
.ply	PLY	Gaussian splat point cloud data	Yes — rebuilds splat buffer
.png/.ktx2	Image	Texture assets	Yes — re-uploads to GPU
.wav/.ogg	Audio	Sound effects and music	Yes

Scene Format Specification

Scenes are the top-level container for game content. A scene file defines entities and their components in a YAML format that serves as the single source of truth for the game world.

Scene YAML Schema

```
# scenes/relic.yaml
name: "Relic"                    # Human-readable scene name

settings:                        # Scene-wide settings
  ambient_light: [0.05, 0.04, 0.03]  # RGB ambient light color
  fog:
    enabled: true
    color: [0.02, 0.02, 0.03]
    density: 0.15
  music: assets/audio/dungeon_ambient.ogg
  gravity: [0, -9.81, 0]        # Physics gravity vector

entities:                        # Array of entity definitions
  - id: entity_unique_id        # Unique string identifier
    tags: [tag1, tag2]           # Searchable tags
    extends: other_entity_id     # Optional: inherit from another entity
    components:                  # Component data (see Component Catalog)
      transform:
        position: [x, y, z]
        rotation: [pitch, yaw, roll]  # Euler angles in degrees
        scale: [x, y, z]
      mesh_renderer:
        mesh: path/to/mesh.gltf
        material: path/to/material.yaml
      # ... additional components
```

Component Catalog

Every component type supported by the engine. Claude Code must know these to create and modify entities.

transform

Required on all entities. Defines position, rotation, and scale in world space.

```
transform:
  position: [0, 1, 0]           # Vec3: world position
  rotation: [0, 90, 0]          # Vec3: Euler angles (degrees)
  scale: [1, 1, 1]              # Vec3: scale factors
```

mesh_renderer

Renders a 3D mesh with a material.

```
mesh_renderer:
  mesh: assets/meshes/model.gltf    # Path to glTF file
  material: assets/materials/mat.yaml  # Path to material definition
  cast_shadows: true                # Default true
  receive_shadows: true             # Default true
```

gaussian_splat

Renders a Gaussian splat point cloud. Can coexist with mesh_renderer on the same entity.

```
gaussian_splat:
  source: assets/splats/scene.ply   # Path to PLY file
  opacity_threshold: 0.02           # Min opacity to render (performance)
  lod_bias: 0.0                     # Level-of-detail adjustment
```

camera

Defines a camera for rendering.

```
camera:
  fov: 75                           # Field of view in degrees
  near: 0.1                        # Near clip plane
  far: 100                         # Far clip plane
  role: main                       # "main" = primary render camera
```

rigid_body

Physics simulation via Rapier 3D.

```
rigid_body:
  type: dynamic                     # dynamic | static | kinematic
  mass: 70.0                       # Mass in kg (dynamic only)
  collider: capsule                 # box | sphere | capsule | mesh
  height: 1.8                      # Capsule-specific
  radius: 0.3                      # Capsule/sphere-specific
  lock_rotation: [true, false, true]  # Lock X, Y, Z rotation
  gravity_scale: 1.0               # Per-entity gravity multiplier
```

character_controller

Alternative to rigid_body for player-like entities with grounded movement.

```
character_controller:
  height: 1.8
  radius: 0.3
  step_height: 0.3                 # Max step-up height
  slope_limit: 45                  # Max walkable slope in degrees
```

collider

Physics collision shape without full rigid body simulation.

```
collider:
  type: box                         # box | sphere | capsule | compound
  size: [2, 3, 0.3]                # Box dimensions
  trigger: false                    # If true, detects overlap but doesn't block
  # Compound collider example:
  type: compound
  shapes:
    - type: box
      size: [0.3, 3, 20]
```

```
      offset: [-6, 0, 0]
    - type: box
      size: [12, 3, 0.3]
      offset: [0, 0, 10]
```

point_light

Dynamic point light source.

```
point_light:
  color: [1.0, 0.7, 0.3]         # RGB color
  intensity: 8.0                  # Light intensity
  range: 6.0                     # Maximum light range
```

audio_source

Spatial or 2D audio emitter.

```
audio_source:
  clip: assets/audio/sound.wav    # Audio file path
  spatial: true                   # 3D positional audio
  loop: false                     # Loop playback
  volume: 0.7                     # Volume 0.0-1.0
  play_on_start: false            # Auto-play when spawned
```

interactable

Marks entity as player-interactable.

```
interactable:
  prompt: "Press E to open"       # UI prompt text
  action: interact                # Input action that triggers it
  radius: 2.0                    # Interaction range from player
```

health

Damageable entity with health tracking.

```
health:
  max: 100                        # Maximum health
  current: 100                    # Starting health
```

trigger_volume

Invisible detection zone.

```
trigger_volume:
  shape: sphere                   # sphere | box
  radius: 8.0                    # Sphere radius
  # Or for box:
  size: [4, 3, 4]                # Box dimensions
```

animator

Skeletal animation controller.

```
animator:
  default_animation: idle         # Animation to play on spawn
  blend_time: 0.3                # Default crossfade duration
```

script

Attaches a Lua gameplay script with configuration.

```
script:
  source: logic/enemy_patrol.lua  # Path to Lua file
  config:                         # Becomes global 'config' table in Lua
    patrol_speed: 2.0
    attack_damage: 15
    patrol_points:
      - [5, 0, 0]
      - [5, 0, 10]
```

Entity Inheritance (extends)

Entities can inherit from other entities defined in the same scene. The child entity inherits all components and overrides only the specified fields. This is critical for DRY scene authoring — define a torch once, place it four times.

```
# Base entity
- id: torch_01
  components:
    transform:
      position: [-3, 2.5, -6]
    mesh_renderer:
      mesh: assets/meshes/wall_torch.gltf
    point_light:
      color: [1.0, 0.7, 0.3]
      intensity: 8.0
    script:
      source: logic/torch_flicker.lua

# Inherited entity - only overrides position
- id: torch_02
  extends: torch_01
  components:
    transform:
      position: [3, 2.5, -6]
```

Scene Reconciliation on Hot Reload

When a scene YAML file changes, the runtime diffs against the current ECS state and patches it. This is inspired by React's virtual DOM diffing.

- New entities in the YAML are spawned into the ECS.
- Removed entities are destroyed.
- Modified components are patched in-place (no entity recreation).
- Scripts receive an on_reload() callback with state preserved.
- The scene is never fully reloaded — the game continues from its current state.

Material Format Specification

Materials define how surfaces look by binding a SLANG shader to texture and uniform inputs. Materials are YAML files that Claude Code can read and modify to change visual appearance.

Material YAML Schema

```yaml
# assets/materials/player_mat.yaml
shader: shaders/passes/gbuffer.slang   # SLANG shader file
properties:
  albedo_map: textures/player/albedo.png
  normal_map: textures/player/normal.png
  roughness: 0.6                       # Float uniform
  metallic: 0.1                        # Float uniform
  emission: [0, 0, 0]                  # Vec3 uniform
blend_mode: opaque                     # opaque | alpha_blend | additive
double_sided: false
cull_mode: back                        # back | front | none
```

The engine reads this file, loads the referenced SLANG shader, uses the SLANG reflection API to determine the required bind group layout, and automatically maps the named properties to shader uniforms. Claude Code can change any property value and the material hot-reloads.

Render Pipeline Specification

The render pipeline is declared as a YAML file that defines a directed acyclic graph (DAG) of render passes. The engine compiles this into wgpu render and compute pipelines at startup and on hot-reload.

Pipeline YAML Schema

```yaml
# pipelines/render.yaml
version: 1
settings:
  resolution: [1920, 1080]
  vsync: true
  max_fps: 60
  hdr: true

resources:
  - name: gbuffer_albedo
    type: texture_2d
    format: rgba8
    size: viewport
  - name: gbuffer_normal
    type: texture_2d
    format: rgb16f
    size: viewport
  - name: gbuffer_depth
    type: texture_2d
    format: depth32f
    size: viewport
  - name: splat_color
    type: texture_2d
    format: rgba16f
    size: viewport
  - name: splat_depth
    type: texture_2d
    format: depth32f
    size: viewport
  - name: hdr_buffer
    type: texture_2d
    format: rgba16f
    size: viewport

passes:
  - name: geometry_pass
    type: rasterize
    shader: shaders/passes/gbuffer.slang
    inputs:
      scene_meshes: auto
      scene_materials: auto
    outputs:
      color: gbuffer_albedo
      normal: gbuffer_normal
      depth: gbuffer_depth
    sort: front_to_back
    cull: frustum

  - name: splat_pass
    type: compute
    shader: shaders/passes/gaussian_splat.slang
    inputs:
```

```
      scene_splats: auto
      camera: auto
    outputs:
      color: splat_color
      depth: splat_depth
    dispatch: per_splat

  - name: lighting_pass
    type: compute
    shader: shaders/passes/deferred_light.slang
    inputs:
      gbuffer_albedo: gbuffer_albedo
      gbuffer_normal: gbuffer_normal
      gbuffer_depth: gbuffer_depth
      splat_color: splat_color
      splat_depth: splat_depth
      scene_lights: auto
    outputs:
      color: hdr_buffer

  - name: tonemap_pass
    type: fullscreen
    shader: shaders/passes/tonemap.slang
    inputs:
      hdr: hdr_buffer
    outputs:
      color: swapchain
```

Pipeline Compilation Process

At startup and on hot-reload, the runtime performs the following steps:

1. Parse the pipeline YAML and validate all resource references.
2. Build a dependency DAG from pass input/output connections.
3. Topologically sort passes to determine execution order.
4. Compile each referenced SLANG shader via the shader-slang Rust crate.
5. Use SLANG reflection API to extract bind group layouts for each shader.
6. Auto-generate wgpu BindGroupLayouts and RenderPipeline/ComputePipeline objects.
7. Allocate GPU resources (textures, buffers) declared in the resources section.
8. Create wgpu BindGroups wiring resources to shader bindings.

CRITICAL IMPLEMENTATION NOTE: The SLANG reflection API is the bridge between the declarative pipeline YAML and the wgpu GPU resources. The shader declares what it needs (uniforms, textures, samplers), the reflection API reports those needs, and the engine auto-generates the binding code. No manual bind group layout code should ever be written.

Adding/Removing Render Passes

Claude Code can modify the render pipeline by editing the YAML file. Common operations include adding post-processing effects, changing the lighting model, or inserting debug visualization passes. The engine recompiles the pipeline DAG on hot-reload.

SLANG Shader System

SLANG is the sole shader language for nAIVE. All GPU code is written in SLANG and compiled at runtime to the appropriate target: WGSL for wgpu on macOS (Metal) and web (WebGPU), or SPIR-V for Vulkan on Linux/Windows.

SLANG Compilation Architecture

```
SLANG Source (.slang files)
       |
       v (shader-slang Rust crate)
  SLANG Compiler
       |
  +----+----+
  |         |
  v         v
WGSL     Reflection Data (JSON-like)
  |         |
  |         v
  |    Auto-generate wgpu
  |    BindGroupLayouts
  |         |
  v         v
wgpu creates GPU pipelines
       |
  +----+--------+
  |    |        |
  v    v        v
Metal Vulkan  WebGPU
```

Compilation Target Selection

```
// Rust pseudocode for target selection
fn select_slang_target() -> slang::CompileTarget {
    // For wgpu, WGSL is the safest universal target
    // wgpu's naga translates WGSL -> MSL (Mac) or keeps as WGSL (WebGPU)
    slang::CompileTarget::Wgsl

    // Alternative: compile to SPIR-V, enable wgpu's "spirv" feature
    // Richer feature set but requires feature flag
    // slang::CompileTarget::Spirv
}
```

Decision: Start with SLANG → WGSL as the primary path. This gives maximum cross-platform compatibility through wgpu. If specific SLANG features require SPIR-V (advanced compute, certain auto-diff patterns), add SPIR-V as a secondary target with wgpu's spirv feature flag.

Shader Module System

SLANG supports a module system with import statements. Shared code (camera data, PBR functions, math utilities) lives in the modules/ directory and is imported by pass shaders.

Required Shader Modules

Module	File	Purpose
Camera	shaders/modules/camera.slang	Camera uniforms (view, projection, position), world-to-screen helpers
PBR	shaders/modules/pbr.slang	Cook-Torrance BRDF evaluation, Fresnel, GGX distribution
Lighting	shaders/modules/lighting.slang	Point light evaluation, attenuation, shadow helpers
Math Utils	shaders/modules/math_utils.slang	Quaternion math, color space conversion, noise functions

Camera Module Reference

```
// shaders/modules/camera.slang
struct CameraData {
    float4x4 view;
    float4x4 projection;
    float4x4 view_projection;
    float3 position;
    float near_plane;
    float far_plane;
    float2 viewport_size;
};

ParameterBlock<CameraData> camera;

float3 world_to_screen(float3 world_pos) {
    float4 clip = mul(camera.view_projection, float4(world_pos, 1.0));
    return clip.xyz / clip.w;
}

float linear_depth(float raw_depth) {
    return camera.near_plane * camera.far_plane /
        (camera.far_plane - raw_depth * (camera.far_plane - camera.near_plane));
}
```

Gaussian Splat Shader

This is the core differentiating shader. It renders 3D Gaussian splats as a compute pass.

```
// shaders/passes/gaussian_splat.slang
import camera;

struct GaussianSplat {
    float3 position;
    float3 scale;
    float4 rotation;        // quaternion
    float opacity;
    float3 sh_coeffs[16];   // Spherical harmonics
};

StructuredBuffer<GaussianSplat> splats;
RWTexture2D<float4> output_color;
RWTexture2D<float> output_depth;
uniform uint splat_count;
```

```
[shader("compute")]
[numthreads(256, 1, 1)]
void cs_main(uint3 tid : SV_DispatchThreadID) {
    if (tid.x >= splat_count) return;
    GaussianSplat splat = splats[tid.x];

    // Project to screen, compute 2D covariance, evaluate SH color
    // Alpha-blend onto output textures
    // (Full implementation in engine code)
}

// Auto-diff enabled for potential in-engine splat optimization
[Differentiable]
float3 evaluate_sh(float3 coeffs[16], float3 dir) {
    float3 result = coeffs[0] * 0.28209479;
    // Spherical harmonic band evaluation
    return result;
}
```

Shader Hot-Reload Process

When any .slang file changes on disk:

9. File watcher detects the change.
10. SLANG compiler recompiles the modified module and all dependents.
11. SLANG reflection API extracts updated bind group layout.
12. If layout changed, wgpu pipeline is recreated; if only code changed, only the shader module is swapped.
13. Next frame renders with the updated shader.

Target latency: Under 300ms from file save to visual update.

Lua Scripting API

Lua is the gameplay scripting language. Every entity with a script component gets its own Lua environment. Scripts define behavior through lifecycle hooks and event listeners.

Script Lifecycle Hooks

Hook	Signature	When Called
init	init(entity)	Once when entity spawns or script first loads
update	update(entity, dt)	Every frame, dt = delta time in seconds
fixed_update	fixed_update(entity, fixed_dt)	Fixed timestep (50Hz default), for physics
on_destroy	on_destroy(entity)	When entity is about to be destroyed
on_reload	on_reload(entity)	When script file is hot-reloaded (state preserved)
on_interact	on_interact(entity, interactor)	When player interacts (if interactable component)

Core API Namespaces

Entity Access

```
entity.id                          -- string: unique entity ID
entity:get("component_name")       -- returns mutable component table
entity:set("component_name", data) -- bulk-set component fields
entity:has("component_name")       -- bool check
entity:add("component_name", data) -- add new component at runtime
entity:remove("component_name")    -- remove component
entity:destroy()                   -- mark for destruction
entity:clone()                     -- spawn a copy
```

Scene Queries

```
scene.find(entity_id)              -- find by ID, returns handle or nil
scene.find_with_tag(tag)           -- first entity with tag
scene.find_all_with_tag(tag)       -- array of handles
scene.find_in_radius(pos, radius)  -- spatial query
scene.find_with_component(name)    -- all entities with component
scene.has_tag(entity, tag)         -- check if entity has tag
scene.spawn(yaml_path, overrides)  -- spawn from template
scene.spawn_at(yaml_path, pos, rot)
scene.load(scene_path)             -- load new scene (async)
scene.load_additive(scene_path)    -- add to existing
```

Input

```
input.pressed(action)              -- true while held
input.just_pressed(action)         -- true only on press frame
input.just_released(action)        -- true only on release frame
```

```
input.get_axis_2d(action)          -- returns {x, y} normalized
```

Physics

```
physics.raycast(origin, direction, max_dist) -- returns hit or nil
  hit.entity, hit.point, hit.normal, hit.distance
physics.raycast_all(origin, dir, max_dist)   -- penetrating
physics.overlap_sphere(center, radius)       -- array of entities
physics.overlap_box(center, half_extents, rotation)
```

Audio

```
audio.play(path)                    -- 2D fire-and-forget
audio.play_at(path, position, opts) -- 3D spatial
  opts: { volume, pitch, min_distance, max_distance, loop }
audio.play_music(path, { fade_in })
audio.stop_music({ fade_out })
audio.set_bus_volume(bus_name, volume)
```

Events

```
emit(event_name, payload_table)     -- emit a game event
listen(event_name, handler_fn)     -- register event listener
  -- handler receives event table with payload fields
```

Tweening

```
tween(target_table, field, end_value, duration, opts)
  opts: { easing, delay, loop, ping_pong, on_complete }
  easing values: "linear", "ease_in_quad", "ease_out_quad",
    "ease_in_out_quad", "ease_out_cubic", "ease_in_out_cubic"
```

Coroutines

```
coroutine.start(function() ... end) -- start async sequence
coroutine.wait(seconds)             -- pause for duration
coroutine.yield_until(event_name)  -- pause until event
coroutine.yield_frames(n)          -- pause for N frames
```

Timers

```
timer.after(seconds, callback)      -- one-shot delayed call
timer.every(seconds, callback)     -- repeating timer
```

Debug

```
log.info(message), log.warn(message), log.error(message)
debug.draw_line(from, to, color, duration)
debug.draw_sphere(center, radius, color)
debug.draw_box(center, size, rotation, color)
debug.draw_text_3d(position, text, color)
```

Math Types Available in Lua

```
vec2(x, y), vec3(x, y, z), vec4(x, y, z, w)
quat(x, y, z, w), quat.from_euler(pitch, yaw, roll)
-- Vector ops: +, -, *, dot, cross, length, normalized,
--   lerp, distance, angle, reflect, project, clone
-- Transform helpers: forward(), right(), up(), look_at(target)
```

Hot-Reload Behavior

When a .lua file changes on disk, the runtime creates a new Lua environment, copies the state table from the old environment, calls on_reload(entity), and drops the old environment. The init() function is NOT called again. The config table is re-read from the entity YAML.

Event System

The event bus is the nervous system of nAIVE. Every meaningful occurrence flows through it as a named, typed event with a payload. Events are data, not function calls.

Event Schema Format

```
# events/schema.yaml
# Input Events (emitted by input system)
input.action_pressed:
  payload:
    action: { type: string }
    player_index: { type: int, default: 0 }

input.action_released:
  payload:
    action: { type: string }
    held_duration: { type: float }

# Physics Events (emitted by physics engine)
physics.collision_enter:
  payload:
    entity_a: { type: entity_id }
    entity_b: { type: entity_id }
    contact_point: { type: vec3 }
    impulse: { type: float }

physics.trigger_enter:
  payload:
    trigger: { type: entity_id }
    visitor: { type: entity_id }

# Lifecycle Events (emitted by runtime)
lifecycle.entity_spawned:
  payload:
    entity: { type: entity_id }
    source_file: { type: string }

lifecycle.component_changed:
  payload:
    entity: { type: entity_id }
    component: { type: string }
    source: { type: enum, values: [script, hot_reload, command, physics] }

# Game Events (defined by game developer / Claude Code)
game.damage_dealt:
  payload:
    target: { type: entity_id }
    source: { type: entity_id }
    amount: { type: float }
    damage_type: { type: string }

game.entity_died:
  payload:
    entity: { type: entity_id }
    killer: { type: entity_id, optional: true }
    cause: { type: string }

game.item_collected:
```

```
  payload:
    item: { type: string }
    collector: { type: entity_id }

game.level_complete:
  payload:
    level: { type: string }
    time: { type: float }
    health_remaining: { type: float }
```

Event Processing Order

The frame loop processes events in a deterministic order:

14. Input system reads OS events, emits input.* events.
15. First event bus flush (scripts react to previous frame's physics/input).
16. Script update() runs for all entities, may emit game.* events.
17. Physics step runs, emits physics.* events.
18. Second event bus flush (late-update systems see everything from this frame).
19. Animation update.
20. Render.

Event Logging

All events are logged to a ring buffer and optionally to a file. An AI agent can query recent events via the command socket or MCP server for game state understanding.

```
# CLI event monitoring
$ naive events --follow --filter "game.*"
[00:12.341] game.damage_dealt target=enemy_03 amount=25 type=bullet
[00:12.342] game.entity_died entity=enemy_03 killer=player
[00:14.887] game.item_collected item=health_pack collector=player
```

Input System

Input is handled in two layers: raw OS events are mapped to semantic actions via a YAML bindings file, and scripts consume only semantic actions. This decouples input devices from game logic.

Input Bindings YAML

```yaml
# input/bindings.yaml
action_maps:
  - name: player_movement
    actions:
      - name: move
        type: axis_2d
        bindings:
          - device: keyboard
            composite: wasd
            keys: { up: W, down: S, left: A, right: D }
          - device: gamepad
            source: left_stick
            deadzone: 0.15

      - name: look
        type: axis_2d
        bindings:
          - device: mouse
            source: delta
            sensitivity: 2.5
          - device: gamepad
            source: right_stick
            deadzone: 0.1

      - name: jump
        type: button
        bindings:
          - device: keyboard
            key: Space
          - device: gamepad
            button: south

      - name: interact
        type: button
        bindings:
          - device: keyboard
            key: E
          - device: gamepad
            button: west

      - name: fire
        type: button
        bindings:
          - device: mouse
            button: left
          - device: gamepad
            trigger: right
            threshold: 0.5
```

Input Injection for AI Playtesting

The runtime accepts synthetic input via the command socket, allowing AI agents to simulate player actions:

```
# Simulate button press
{"cmd": "inject_input", "action": "jump", "type": "press"}

# Simulate analog movement
{"cmd": "inject_input", "action": "move", "type": "axis", "value": [0.5, 1.0]}

# Auto-release after N frames
{"cmd": "inject_input", "action": "fire", "type": "press", "hold_frames": 10}
```

MCP Server Specification

The nAIVE MCP (Model Context Protocol) server gives Claude Code native tool access to the running engine. Instead of shell commands and file edits (which work but are indirect), the MCP server provides high-level tools that map directly to engine operations.

MCP Tools Catalog

The following tools must be implemented. Each tool communicates with the running nAIVE runtime via the command socket.

Scene and Entity Tools

Tool	Purpose	Key Parameters
naive_spawn_entity	Spawn entity from YAML template	template (path), overrides (object), entity_id (optional)
naive_query_entity	Get current state of entity/component	entity_id, component (optional)
naive_modify_entity	Modify component data on live entity	entity_id, component, data
naive_destroy_entity	Remove entity from scene	entity_id
naive_list_entities	List all entities with tags/components	filter_tag, filter_component (optional)

Event Tools

Tool	Purpose	Key Parameters
naive_emit_event	Emit game event into running scene	event_name, payload
naive_query_events	Query event log for recent events	filter (glob), last_seconds, last_n, involving_entity
naive_watch_events	Subscribe to real-time event stream	filter (glob), watch_id

Input Simulation Tools

Tool	Purpose	Key Parameters
naive_inject_input	Simulate player input action	action, type (press/release/axis), value, hold_frames
naive_play_sequence	Execute timed input sequence	sequence (array of timed actions)

Observation Tools

Tool	Purpose	Key Parameters

naive_screenshot	Capture current frame	camera, resolution, output_path
naive_observe	Comprehensive state snapshot	include_screenshot, include_entities, include_events, include_perf

Runtime Control

Tool	Purpose	Key Parameters
naive_runtime_control	Pause/resume/step/time scale	command (pause/resume/step/set_time_scale/restart_scene), value
naive_hot_reload	Force hot-reload of file(s)	file (optional, all if omitted)
naive_generate_asset	Generate asset via AI service	type, prompt, service, output_path, options

Command Socket Protocol

The runtime listens on a Unix domain socket (or TCP for remote) at /tmp/naive-runtime.sock. Commands are JSON objects sent as single lines, responses are JSON objects returned as single lines.

```
// Request format
{"cmd": "query_entity", "entity_id": "player", "component": "transform"}

// Response format
{"status": "ok", "data": {"position": [2.3, 1.0, -4.1], "rotation": [0, 45, 0],
"scale": [1,1,1]}}

// Error format
{"status": "error", "message": "Entity not found: player2"}
```

Relic Demo Game Specification

Relic is the first game built on nAIVE. It is a single-room first-person dungeon crawler designed to exercise every engine system. The entire game must be buildable by Claude Code without any graphical editor.

Game Design

Setting: A torch-lit stone dungeon chamber containing a guarded treasure.

Objective: Enter the room, defeat the goblin guardian, and claim the golden chalice relic.

Duration: Approximately 2–5 minutes of gameplay.

Perspective: First-person.

Controls: WASD movement, mouse look, E to interact, left-click to attack, Space to jump.

Entity Manifest

The following entities comprise the Relic scene. Each must be fully defined in YAML with all component data.

Entity ID	Type	Key Components	Description
dungeon_room	Environment	gaussian_splat	The room itself, rendered as a Gaussian splat for photorealistic volumetric look
player	Player	transform, character_controller, camera, health, script	First-person player with movement, look, interact, and attack
gate	Interactive	transform, mesh_renderer, collider, interactable, audio_source, script	Heavy door that swings open when interacted with
torch_01–04	Ambient	transform, mesh_renderer, point_light, audio_source, script	Wall-mounted torches with flickering light animation
guardian	Enemy	transform, mesh_renderer, rigid_body, health, trigger_volume, audio_source, script	Goblin that guards the relic, detects player, chases and attacks
relic	Objective	transform, mesh_renderer, point_light, interactable, audio_source, script	Golden chalice that hovers and rotates, picked up to win
floor	Collision	transform, collider	Invisible floor collision (splat has no physics)
walls	Collision	transform, collider (compound)	Invisible wall collision matching the splat room geometry

Relic Scene YAML

This is the complete scene definition that Claude Code must produce:

```
# scenes/relic.yaml
name: "Relic"
settings:
  ambient_light: [0.05, 0.04, 0.03]
  fog:
    enabled: true
    color: [0.02, 0.02, 0.03]
    density: 0.15
  music: assets/audio/dungeon_ambient.ogg
  gravity: [0, -9.81, 0]

entities:
  # Environment (Gaussian Splat)
  - id: dungeon_room
    components:
      transform:
        position: [0, 0, 0]
        scale: [1, 1, 1]
      gaussian_splat:
        source: assets/splats/dungeon_room.ply
        opacity_threshold: 0.02

  # Player
  - id: player
    tags: [player]
    components:
      transform:
        position: [0, 1, -8]
      mesh_renderer:
        mesh: assets/meshes/player_hands.gltf
        material: assets/materials/skin.yaml
      character_controller:
        height: 1.8
        radius: 0.3
        step_height: 0.3
      camera:
        fov: 75
        near: 0.1
        far: 100
        role: main
      health:
        max: 100
        current: 100
      script:
        source: logic/player_fps.lua
        config:
          move_speed: 4.0
          look_sensitivity: 2.0
          interact_range: 2.5
          attack_damage: 25
          attack_range: 2.0
          attack_cooldown: 0.5

  # Door
  - id: gate
    tags: [interactable]
    components:
      transform:
        position: [0, 1.2, -3]
      mesh_renderer:
```

```
      mesh: assets/meshes/heavy_door.gltf
      material: assets/materials/iron_wood.yaml
    collider:
      type: box
      size: [2.4, 2.8, 0.3]
    interactable:
      prompt: "Open door"
      action: interact
      radius: 2.0
    audio_source:
      clip: assets/audio/door_scrape.wav
      spatial: true
    script:
      source: logic/door_swing.lua
      config:
        swing_angle: -110
        swing_speed: 1.5

# Torches (4x with inheritance)
- id: torch_01
  components:
    transform:
      position: [-3, 2.5, -6]
    mesh_renderer:
      mesh: assets/meshes/wall_torch.gltf
      material: assets/materials/torch_mat.yaml
    point_light:
      color: [1.0, 0.7, 0.3]
      intensity: 8.0
      range: 6.0
    audio_source:
      clip: assets/audio/torch_crackle.wav
      spatial: true
      loop: true
      volume: 0.3
    script:
      source: logic/torch_flicker.lua
      config:
        flicker_speed: 8.0
        intensity_range: [5.0, 10.0]
- id: torch_02
  extends: torch_01
  components:
    transform:
      position: [3, 2.5, -6]
- id: torch_03
  extends: torch_01
  components:
    transform:
      position: [-3, 2.5, 2]
- id: torch_04
  extends: torch_01
  components:
    transform:
      position: [3, 2.5, 2]

# Guardian Enemy
- id: guardian
  tags: [enemy, npc]
  components:
    transform:
      position: [0, 0, 3]
      rotation: [0, 180, 0]
```

```
    mesh_renderer:
      mesh: assets/meshes/goblin.gltf
      material: assets/materials/goblin_mat.yaml
    rigid_body:
      type: dynamic
      mass: 50
      collider: capsule
      height: 1.2
      radius: 0.4
    health:
      max: 60
      current: 60
    trigger_volume:
      shape: sphere
      radius: 6.0
    audio_source:
      clip: assets/audio/goblin_growl.wav
      spatial: true
    script:
      source: logic/guardian.lua
      config:
        attack_damage: 20
        attack_range: 1.8
        attack_cooldown: 1.5
        chase_speed: 3.0

# The Relic
- id: relic
  tags: [interactable, objective]
  components:
    transform:
      position: [0, 1.2, 7]
    mesh_renderer:
      mesh: assets/meshes/golden_chalice.gltf
      material: assets/materials/gold_emissive.yaml
    point_light:
      color: [1.0, 0.9, 0.5]
      intensity: 3.0
      range: 2.0
    interactable:
      prompt: "Take the relic"
      action: interact
      radius: 1.5
    audio_source:
      clip: assets/audio/holy_pickup.wav
      spatial: true
    script:
      source: logic/relic_pickup.lua
      config:
        hover_amplitude: 0.1
        hover_speed: 2.0
        rotation_speed: 30

# Invisible collision geometry
- id: floor
  components:
    transform:
      position: [0, 0, 0]
    collider:
      type: box
      size: [12, 0.1, 20]
- id: walls
  components:
```

```
    transform:
      position: [0, 1.5, 0]
    collider:
      type: compound
      shapes:
        - type: box
          size: [0.3, 3, 20]
          offset: [-6, 0, 0]
        - type: box
          size: [0.3, 3, 20]
          offset: [6, 0, 0]
        - type: box
          size: [12, 3, 0.3]
          offset: [0, 0, -10]
        - type: box
          size: [12, 3, 0.3]
          offset: [0, 0, 10]
```

Lua Script Specifications

Each script must implement the following behavior. Claude Code should write these scripts implementing the full Lua API specified in Section 7.

player_fps.lua

First-person player controller with WASD movement, mouse look, interaction, and melee attack. Must emit game.damage_dealt on attack hits via physics.raycast. Must handle game.damage_dealt events where target is self (flash red, update health). Must emit game.entity_died on health reaching zero.

door_swing.lua

Door that toggles open/closed on interaction. Uses tween() to animate rotation. Plays door sound via audio_source. Removes collider when open, re-adds when closed.

torch_flicker.lua

Ambient flickering using layered sine waves at different frequencies on the point_light intensity. Each torch gets a random phase offset for organic feel. Slight color temperature shift on the green channel.

guardian.lua

State machine enemy with states: IDLE, ALERT, CHASE, ATTACK, DEAD. Listens for physics.trigger_enter to detect player. Chases player when alerted. Attacks when in range, emitting game.damage_dealt. On death: plays death animation, emits game.entity_died, ragdolls, fades out and destroys after 10 seconds.

relic_pickup.lua

Hovering/rotating chalice using sine wave on Y position and steady rotation on Y axis. On interaction: coroutine sequence of float up, bright flash, shrink, emit game.item_collected and game.level_complete, destroy self.

AI-Generated Assets Required

All visual and audio assets should be generated via AI pipes. No hand-crafted art required.

Asset	Type	AI Service	Prompt / Description
dungeon_room.ply	Gaussian Splat	Photogrammetry/3DGS	Medieval stone dungeon room, torch-lit, single chamber, arched ceiling
heavy_door.gltf	3D Mesh	Meshy	Heavy wooden door with iron hinges and medieval studs, game-ready
wall_torch.gltf	3D Mesh	Meshy	Wall-mounted torch with flame, medieval style
goblin.gltf	3D Mesh	Meshy	Goblin warrior with wooden club, low-poly game character
golden_chalice.gltf	3D Mesh	Meshy	Ornate golden chalice with gemstones, fantasy relic
player_hands.gltf	3D Mesh	Meshy	First-person view hands and arms, medieval gauntlets
door_scrape.wav	SFX	ElevenLabs	Heavy stone door scraping open with metallic latch
torch_crackle.wav	SFX	ElevenLabs	Torch fire crackling, ambient loop
goblin_growl.wav	SFX	ElevenLabs	Goblin growl, aggressive, fantasy creature
holy_pickup.wav	SFX	ElevenLabs	Holy relic pickup sound, magical shimmer and choir
dungeon_ambient.ogg	Music	ElevenLabs/Suno	Dark ambient dungeon, mysterious, low strings, looping

Implementation Phases

The nAIVE engine should be built incrementally with each phase producing a runnable milestone.

Phase 1: Foundation (Weeks 1–2)

Milestone: A window opens, renders a colored triangle from a SLANG shader, hot-reloads on file change.

- Rust project setup with Cargo.toml and all dependencies.
- wgpu initialization: window (winit), surface, device, queue.
- SLANG compiler integration via shader-slang crate.
- SLANG → WGSL compilation path working.
- Simple vertex + fragment shader rendering a triangle.
- File watcher (notify) on the shaders/ directory.
- Shader hot-reload: detect .slang change, recompile, recreate pipeline.

Phase 2: Scene Loading (Weeks 3–4)

Milestone: Load a YAML scene file, display multiple glTF meshes with materials and a camera.

- YAML scene parser (serde_yaml) with entity/component deserialization.
- ECS world (hecs) populated from parsed scene data.
- glTF mesh loader with vertex buffer creation.
- Material system: YAML material files mapped to shader uniforms via reflection.
- Camera system: view/projection matrices from camera component.
- Transform hierarchy and world matrix computation.
- Multiple entities rendering in a single frame.
- Scene file hot-reload with ECS reconciliation (diff and patch).

Phase 3: Render Pipeline Compositor (Weeks 5–6)

Milestone: Pipeline YAML drives a multi-pass deferred renderer with dynamic lighting.

- Pipeline YAML parser with resource and pass definitions.
- DAG builder: dependency graph from pass inputs/outputs.
- Topological sort for execution order.
- GPU resource allocation from pipeline resource declarations.
- SLANG reflection-based auto-generation of wgpu bind group layouts.
- G-buffer pass: albedo + normal + depth render targets.

- Deferred lighting pass: point light evaluation.
- Tone mapping pass: HDR to SDR output.
- Pipeline YAML hot-reload: recompile entire pipeline DAG.

Phase 4: Gaussian Splatting (Weeks 7–8)

Milestone: Load and render a .ply Gaussian splat file alongside traditional meshes with correct depth compositing.

- PLY file parser for Gaussian splat data (positions, covariances, SH coefficients, opacity).
- GPU buffer for splat data.
- Compute shader for splat sorting (radix sort by camera depth).
- Compute shader for splat rendering (2D Gaussian projection, alpha blending).
- Depth-based compositing of splat and mesh render targets in the lighting pass.
- gaussian_splat component type in the ECS.

Phase 5: Physics and Input (Weeks 9–10)

Milestone: Player can walk around the scene with WASD/mouse, collide with walls, and interact with objects.

- Rapier 3D integration: world creation, body/collider management.
- Sync ECS transforms to/from Rapier bodies each frame.
- Character controller implementation (grounded movement, step-up, slope).
- Input system: winit events mapped to semantic actions via bindings.yaml.
- Input state: pressed/just_pressed/just_released/axis_2d.
- First-person camera: mouse delta applied to player rotation.
- Trigger volumes and collision events.

Phase 6: Lua Scripting (Weeks 11–12)

Milestone: Lua scripts attached to entities execute lifecycle hooks and respond to events.

- mlua integration: create Lua VM, register API functions.
- Script component: load .lua file, create per-entity Lua environment.
- Lifecycle hooks: init, update, fixed_update, on_destroy, on_reload.
- Entity API exposed to Lua: get/set components, find entities, spawn.
- Input API in Lua: pressed, just_pressed, axis_2d.
- Physics API in Lua: raycast, overlap queries.
- Script hot-reload: new environment with state preservation.

Phase 7: Event System and Audio (Weeks 13–14)

Milestone: Events flow between systems, audio plays spatially, torch lights flicker.

- Event bus: emit, listen, flush, ring buffer log.
- Event schema validation from events/schema.yaml.
- Event file logging and CLI event stream.
- Kira audio integration: spatial audio, music, sound effects.
- Audio API exposed to Lua.
- Tween system: property animation with easing functions.
- Coroutine support in Lua for sequential game logic.

Phase 8: Command Socket and MCP Server (Weeks 15–16)

Milestone: Claude Code can query and control the running game via MCP tools.

- Unix domain socket listener in the runtime (tokio async).
- JSON command protocol: parse, route, execute, respond.
- All MCP tools implemented: spawn, query, modify, events, input injection, screenshot, observe.
- MCP server binary that bridges MCP protocol to command socket.
- Input injection system: synthetic input indistinguishable from real input.
- Screenshot capture: render to offscreen buffer, save as PNG.

Phase 9: Relic Game Assembly (Weeks 17–18)

Milestone: The complete Relic demo game is playable: walk, open door, fight goblin, pick up relic, win.

- Generate all assets via AI pipes (or use placeholder art).
- Write all Lua scripts for player, door, torches, guardian, relic.
- Assemble the complete scene YAML.
- Tune gameplay: combat damage, enemy speed, detection range.
- Add UI overlays: health bar, interaction prompt, victory screen.
- Polish: particle effects, screen shake on damage, music transitions.

Phase 10: AI Playtesting (Weeks 19–20)

Milestone: Automated tests verify the game is playable end-to-end.

- Test framework: run Lua test files that inject input and assert events.
- test_full_playthrough: walk to door, open, fight, collect relic, verify game.level_complete.
- test_player_can_die: set health to 1, verify death event fires.
- test_no_escape_routes: try all directions, verify player stays in bounds.
- Vision-augmented playtest script using Claude's vision API.
- CI pipeline: run tests on every commit.

Cargo.toml Reference

The complete Rust dependency configuration for the nAIVE runtime:

```
[package]
name = "naive-runtime"
version = "0.1.0"
edition = "2021"

[dependencies]
# GPU rendering
wgpu = { version = "24", features = ["spirv"] }
winit = "0.30"

# SLANG shader compilation
shader-slang = "0.1"

# ECS
hecs = "0.10"

# Scene format
serde = { version = "1", features = ["derive"] }
serde_yaml = "0.9"
serde_json = "1"

# Asset loading
gltf = "1.4"
image = "0.25"

# Scripting
mlua = { version = "0.10", features = ["lua54", "vendored"] }

# File watching
notify = "7"

# Math
glam = "0.29"

# Physics
rapier3d = "0.22"

# Audio
kira = "3"

# IPC / Async
tokio = { version = "1", features = ["net", "io-util", "rt-multi-thread", "sync"] }

# Logging
tracing = "0.1"
tracing-subscriber = "0.3"

# CLI
clap = { version = "4", features = ["derive"] }

[profile.dev]
opt-level = 1          # Slightly optimize dev builds for GPU work

[profile.release]
```

```
opt-level = 3
lto = true
```

Build and Run Commands

```
# Build
$ cargo build

# Run with a scene
$ cargo run -- run --scene scenes/relic.yaml --pipeline pipelines/render.yaml

# Run tests
$ cargo run -- test tests/test_relic.lua

# Start MCP server
$ cargo run -- mcp-serve
```

Runtime Output Modes

The runtime supports multiple output targets configured via CLI flags.

Mode	Flag	Use Case
Window	--output window	Development: renders to a desktop window with hot-reload
Headless	--output headless	CI/CD: no window, runs logic and tests only
Frames	--output frames --dir ./renders/	Capture individual frames as PNG files
Stream	--output stream --port 8080	WebRTC stream for AI vision observation
Screenshot	naive screenshot --output shot.png	On-demand frame capture via CLI

CLI Reference

The nAIVE CLI provides project management and runtime control commands.

```
# Project management
$ naive init <project-name>        # Create new project from template
$ naive run --scene <scene.yaml>   # Start runtime with scene
$ naive build --target <platform>  # Build for distribution
$ naive test <test-file.lua>       # Run automated tests

# Asset generation (pipes)
$ naive generate-asset --type mesh_3d --prompt "..." --output path
$ naive generate-asset --type sfx --prompt "..." --output path
$ naive generate-asset --type texture --prompt "..." --output path

# Runtime control (while running)
$ naive cmd pause
$ naive cmd resume
$ naive cmd step                   # Step one frame
$ naive cmd set-time-scale 0.5

# Monitoring
$ naive events --follow --filter "game.*"
$ naive screenshot --output shot.png
$ naive perf                       # Performance metrics

# MCP
$ naive mcp-serve                  # Start MCP server
```

Critical Implementation Notes for Claude Code

This section contains specific guidance for the AI agent implementing nAIVE. These are hard-won lessons from the design process.

SLANG Integration

- **Always compile SLANG to WGSL first.** This is the universal target that works on macOS (Metal), Linux/Win (Vulkan via wgpu), and browsers (WebGPU). Only use SPIR-V if a specific feature requires it.
- **Use the reflection API for everything.** Never manually write wgpu bind group layouts. Compile the SLANG shader, extract bindings from reflection data, and auto-generate the wgpu structures.
- **SLANG prebuilt binaries exist for aarch64 macOS.** The shader-slang crate handles downloading them. Do not build SLANG from source.

wgpu Patterns

- **wgpu is async for device creation.** Use pollster::block_on() or tokio for the initial setup, then the render loop is synchronous.
- **Pipeline recreation is fast.** Don't fear recreating wgpu pipelines on shader hot-reload. It takes ~10ms.
- **Use wgpu's naga for WGSL validation.** If SLANG generates invalid WGSL, naga will catch it with clear error messages.

ECS Architecture

- **Components are plain Rust structs.** No inheritance, no vtables. Use hecs::World::query() for system iteration.
- **Systems are plain functions.** fn physics_system(world: &mut World, dt: f32) { ... }. No system registration framework.
- **Entity IDs in YAML map to hecs Entity handles via a HashMap<String, Entity>.** This registry enables entity lookup by name for scripts and commands.

Lua Integration

- **Each script gets its own Lua VM (mlua::Lua instance).** This provides isolation but means inter-script communication must go through the event bus.
- **Expose Rust structs to Lua via mlua's UserData trait.** Vec3, Transform, Entity handles are all UserData.

- **The config table is populated from YAML before init() runs.** Use serde to convert YAML config to Lua table.
- **Hot-reload preserves the state table.** Serialize state to a Lua string, create new VM, deserialize state back.

Performance Targets

Metric	Target	Measurement
Frame rate (Relic scene)	60 fps	On MacBook Pro M3/M4
Shader hot-reload	< 300ms	File save to visual update
Scene hot-reload	< 100ms	YAML save to ECS patch
Lua hot-reload	< 10ms	Script save to new behavior
Command socket latency	< 5ms	JSON command to response
Gaussian splats (1M)	30-60 fps	Depends on screen coverage

Testing Strategy

Automated Gameplay Tests

Tests are Lua scripts that inject input and assert game events occurred:

```
-- tests/test_relic.lua

function test_full_playthrough()
    scene.load("scenes/relic.yaml")
    wait_for_event("lifecycle.scene_loaded")
    local player = scene.find("player")

    -- Walk to door
    input.inject("move", "axis", {0, 1})
    wait_until(function()
        return player:get("transform").position.z > -4
    end, 10)
    input.inject("move", "axis", {0, 0})

    -- Open door
    input.inject("interact", "press")
    wait_seconds(2)

    -- Walk to guardian
    input.inject("move", "axis", {0, 1})
    wait_for_event("game.enemy_alerted", 10)

    -- Fight
    for i = 1, 20 do
        input.inject("fire", "press")
        wait_frames(5)
        input.inject("fire", "release")
        wait_frames(10)
    end

    assert(event_occurred("game.entity_died", {entity = "guardian"}))

    -- Walk to relic and pick up
    input.inject("move", "axis", {0, 1})
    wait_seconds(3)
    input.inject("interact", "press")
    wait_for_event("game.level_complete", 5)

    log.info("Full playthrough test passed!")
end

function test_player_can_die()
    scene.load("scenes/relic.yaml")
    wait_for_event("lifecycle.scene_loaded")
    scene.find("player"):get("health").current = 1
    input.inject("move", "axis", {0, 1})
    wait_seconds(8)
    assert(event_occurred("game.entity_died", {entity = "player"}))
end
```

Test Runner CLI

```
$ naive test tests/test_relic.lua
Running 2 tests...
  OK test_full_playthrough (12.3s game time)
  OK test_player_can_die (3.2s game time)
All tests passed.
```

Glossary

Term	Definition
nAIVE	AI-Native Interactive Visual Engine. The game engine described in this document.
SLANG	Nvidia’s shader language (Khronos-hosted). Compiles to MSL, WGSL, SPIR-V, HLSL, CUDA.
wgpu	Rust cross-platform GPU abstraction. Maps to Metal, Vulkan, DX12, WebGPU.
ECS	Entity Component System. Data-oriented architecture: entities are IDs, components are data, systems are functions.
Gaussian Splat	3D scene representation using millions of colored 3D Gaussians instead of triangle meshes.
MCP	Model Context Protocol. Standard for AI agents to interact with tools and services.
Hot-Reload	Updating a running application by detecting file changes and patching state without restart.
Relic	The first demo game built on nAIVE. A single-room dungeon crawler.
PLY	Polygon File Format. Used for Gaussian splat point cloud data.
glTF	GL Transmission Format. Standard 3D mesh interchange format.
DAG	Directed Acyclic Graph. Used for render pipeline pass ordering.
WGSL	WebGPU Shading Language. wgpu’s native shader format, SLANG’s compilation target.