PSD3 Kernel Separation Architecture

Overview

This document outlines the architecture for separating PSD3's force simulation into pluggable kernels, enabling different performance/platform tradeoffs while maintaining a unified PureScript coordination layer.

Motivation

The successful implementation of the WASM force kernel demonstrated:

1. **3.4x speedup** over D3.js at 10,000 nodes using Rust/WASM
2. **Zero overhead** from PureScript adapter layer (validated via three-way benchmark)
3. **Visual parity** between implementations (same algorithm, same results)

This validates a broader architectural pattern: PureScript as a **coordination layer** that orchestrates high-performance kernels written in platform-optimal languages.

Architecture

Current State

```
psd3-simulation
├── ForceEngine/
│   ├── Core.purs         ← D3 FFI bindings
│   ├── Simulation.purs   ← Simulation wrapper
│   ├── WASM.purs         ← WASM FFI bindings (new)
│   └── WASMEngine.purs   ← WASM adapter (new)
└── Scene/
    ├── Engine.purs       ← EngineAdapter interface
    └── Handle.purs       ← High-level API
```

Target State

```
psd3-simulation-core          ← Types, interfaces, scene orchestration
├── psd3-d3-kernel            ← D3.js implementation (JavaScript target)
├── psd3-wasm-kernel          ← Rust/WASM implementation (all JS targets)
├── psd3-native-kernel        ← C++ implementation (future: desktop apps)
└── psd3-distributed-kernel   ← Erlang implementation (future: large-
scale)
```

Core Interface

All kernels implement the `EngineAdapter` interface:

```
-- psd3-simulation-core/src/PSD3/Simulation/Engine.purs

type EngineAdapter node =
  { getNodes            :: Effect (Array node)
  , capturePositions    :: Array node -> PositionMap
  , interpolatePositions :: PositionMap -> PositionMap -> Number -> Effect
Unit
  , updatePositions     :: PositionMap -> Effect Unit
  , applyRulesInPlace   :: Array (NodeRule node) -> Effect Unit
  , reinitializeForces  :: Effect Unit
  , reheat              :: Effect Unit
  }

-- Core simulation types
type SimulationNode r =
  { id :: Int
  , x :: Number
  , y :: Number
  , vx :: Number
  , vy :: Number
  , fx :: Number  -- NaN if not fixed
  , fy :: Number  -- NaN if not fixed
  | r
  }

type PositionMap = Object { x :: Number, y :: Number }

type NodeRule node =
  { name :: String
  , select :: node -> Boolean
  , apply :: node -> node
  }
```

Force Configuration (Kernel-Agnostic)

```
-- Common force configuration types
type ManyBodyConfig =
  { strength :: Number      -- Positive = attract, negative = repel
  , theta :: Number         -- Barnes-Hut approximation (0.9 typical)
  , distanceMin :: Number
  , distanceMax :: Number
  }

type LinkConfig =
  { distance :: Number
  , strength :: Number
  , iterations :: Int
  }

type CenterConfig =
```

```
  { x :: Number
  , y :: Number
  , strength :: Number
  }
```

Kernel Implementations

psd3-d3-kernel

Target: JavaScript (browser, Node.js) **Dependencies**: D3.js (d3-force) **Characteristics**:

- Mature, well-tested
- Good performance for <5,000 nodes
- Extensive force types (collision, radial, position)
- Familiar to D3 users

```
-- psd3-d3-kernel/src/PSD3/Kernel/D3.purs
module PSD3.Kernel.D3 (create, mkAdapter) where

create :: forall r. SimulationConfig -> Effect (D3Simulation r)
mkAdapter :: forall r. D3Simulation r -> EngineAdapter (SimulationNode r)
```

psd3-wasm-kernel

Target: JavaScript with WASM support (browsers, Node.js, Deno) **Dependencies**: Rust toolchain, wasm-pack **Characteristics**:

- 3-4x faster than D3 for large graphs
- Barnes-Hut O(n log n) many-body force
- Fixed memory layout (cache-friendly)
- ~65KB WASM binary + ~15KB JS glue

```
-- psd3-wasm-kernel/src/PSD3/Kernel/WASM.purs
module PSD3.Kernel.WASM (initWasm, create, mkAdapter) where

-- Must be called once before creating simulations
initWasm :: String -> Aff Unit

create :: forall r. Array (SimulationNode r) -> Array Link -> WASMConfig -
> Effect (WASMSim r)
mkAdapter :: forall r. WASMSim r -> EngineAdapter (SimulationNode r)
```

Build artifacts:

```
psd3-wasm-kernel/
├── src/                    ← PureScript source
```

```
├── rust/                    ← Rust source (force-kernel)
│   ├── src/
│   │   ├── lib.rs          ← Main simulation
│   │   └── quadtree.rs     ← Barnes-Hut implementation
│   └── Cargo.toml
├── pkg/                     ← wasm-pack output (generated)
│   ├── force_kernel.js
│   ├── force_kernel_bg.wasm
│   └── force_kernel.d.ts
└── spago.yaml
```

psd3-native-kernel (Future)

Target: C++ backend, desktop applications **Use case**: Electron apps, native visualizations, offline processing **Characteristics**:

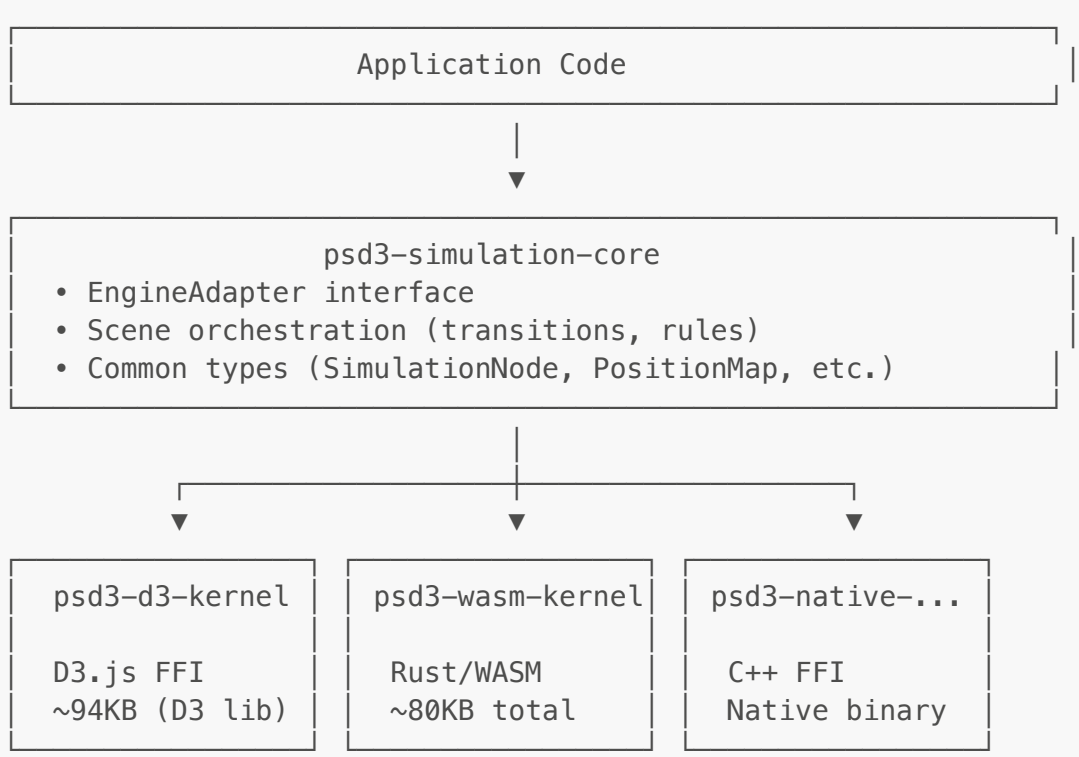
- Direct CPU access, SIMD optimization potential
- No WASM overhead
- Could use existing C++ force libraries

psd3-distributed-kernel (Future)

Target: Erlang/OTP via purerl **Use case**: Large-scale simulations across multiple nodes **Characteristics**:

- Partition large graphs across Erlang processes
- Fault-tolerant simulation
- Real-time updates via OTP messaging

Package Dependencies



Build System Considerations

Current Complexity

Building a full PSD3 application may require:

Tool	Purpose	Language
spago	PureScript compilation	PureScript → JS
wasm-pack	Rust to WASM	Rust → WASM
cargo	Rust dependencies	Rust
esbuild	JS bundling	JS
purepy	Python backend	PureScript → Python
purerl + rebar3	Erlang backend	PureScript → Erlang

Proposed Build Orchestration

A declarative build configuration that understands cross-language dependencies:

```
# psd3-build.yaml (hypothetical)
project: my-visualization

targets:
  browser:
    purescript:
      entry: src/Main.purs
      backend: javascript
    kernels:
      - psd3-wasm-kernel  # triggers wasm-pack build
    bundle:
      tool: esbuild
      output: dist/app.js

  server:
    purescript:
      entry: src/Server.purs
      backend: erlang

  analysis:
    purescript:
      entry: src/Analysis.purs
      backend: python
    dependencies:
      - numpy
      - pandas
```

Near-Term: Makefile/Justfile

Until dedicated tooling exists, a `Makefile` or `justfile` coordinates builds:

```
# Makefile for psd3-wasm-kernel

.PHONY: all clean wasm purescript bundle

all: wasm purescript bundle

wasm:
    cd rust && wasm-pack build --target web --out-dir ../pkg

purescript:
    spago build

bundle:
    esbuild src/Entry.js --bundle --outfile=dist/bundle.js

clean:
    rm -rf pkg/ output/ dist/
```

Long-Term: Dedicated Orchestrator

A PureScript-native build tool that:

1. Parses dependency graph across language boundaries
2. Determines build order automatically
3. Caches artifacts intelligently
4. Provides unified error reporting
5. Supports watch mode across all languages

Migration Path

Phase 1: Extract Core (Current)

1. ✅ Implement WASM kernel alongside D3
2. ✅ Define EngineAdapter interface
3. ✅ Validate performance and correctness
4. Document kernel requirements

Phase 2: Separate Packages

1. Create `psd3-simulation-core` with shared types
2. Extract D3 code to `psd3-d3-kernel`
3. Move WASM code to `psd3-wasm-kernel`
4. Update existing applications

Phase 3: Expand Kernels

1. Implement collision, radial forces in WASM kernel
2. Explore native kernel for desktop
3. Prototype distributed kernel for large-scale

Phase 4: Build Tooling

1. Document manual build process thoroughly
2. Create helper scripts for common workflows
3. Evaluate/build dedicated orchestration tooling

Performance Comparison

Benchmark: Clustered graph, 5 clusters, ~2x links per node

Nodes	D3.js (ms/tick)	WASM (ms/tick)	Speedup
100	0.65	0.28	2.3x
1,000	1.51	1.28	1.2x
5,000	8.2	3.1	2.6x
10,000	41.8	12.2	3.4x

Note: Speedup increases with node count due to Barnes-Hut O(n log n) vs D3's O(n²)

Kernel Selection Guidelines

Scenario	Recommended Kernel
< 2,000 nodes, simple forces	D3 kernel
> 2,000 nodes, performance critical	WASM kernel
Need collision/radial forces	D3 kernel (until WASM implements)
Desktop app, native performance	Native kernel (future)
Distributed/large-scale	Distributed kernel (future)

Open Questions

1. **Registry support**: How do we publish packages with non-PureScript artifacts (WASM binaries)?
2. **Initialization ergonomics**: WASM requires async init. How do we make this seamless?
3. **Feature parity**: Should all kernels implement all forces, or specialize?
4. **Testing**: How do we ensure kernels produce identical results? Property-based testing across implementations?
5. **Versioning**: How do we version kernels independently while maintaining compatibility?

Appendix: File Sizes

Current implementation sizes (unminified, uncompressed):

Artifact	Size	Notes
D3.js	94 KB	Full d3.v7 library
force_kernel_bg.wasm	65 KB	Rust WASM binary
force_kernel.js	15 KB	WASM JS bindings
ps-bundle.js	44 KB	PureScript + FFI

Total WASM solution: ~124 KB (smaller than D3 alone, 3x faster)

References

- WASM Force Demo - Three-way benchmark
- PSD3 Simulation - Current implementation
- Barnes-Hut Algorithm - O(n log n) force calculation
- wasm-bindgen - Rust/JS interop