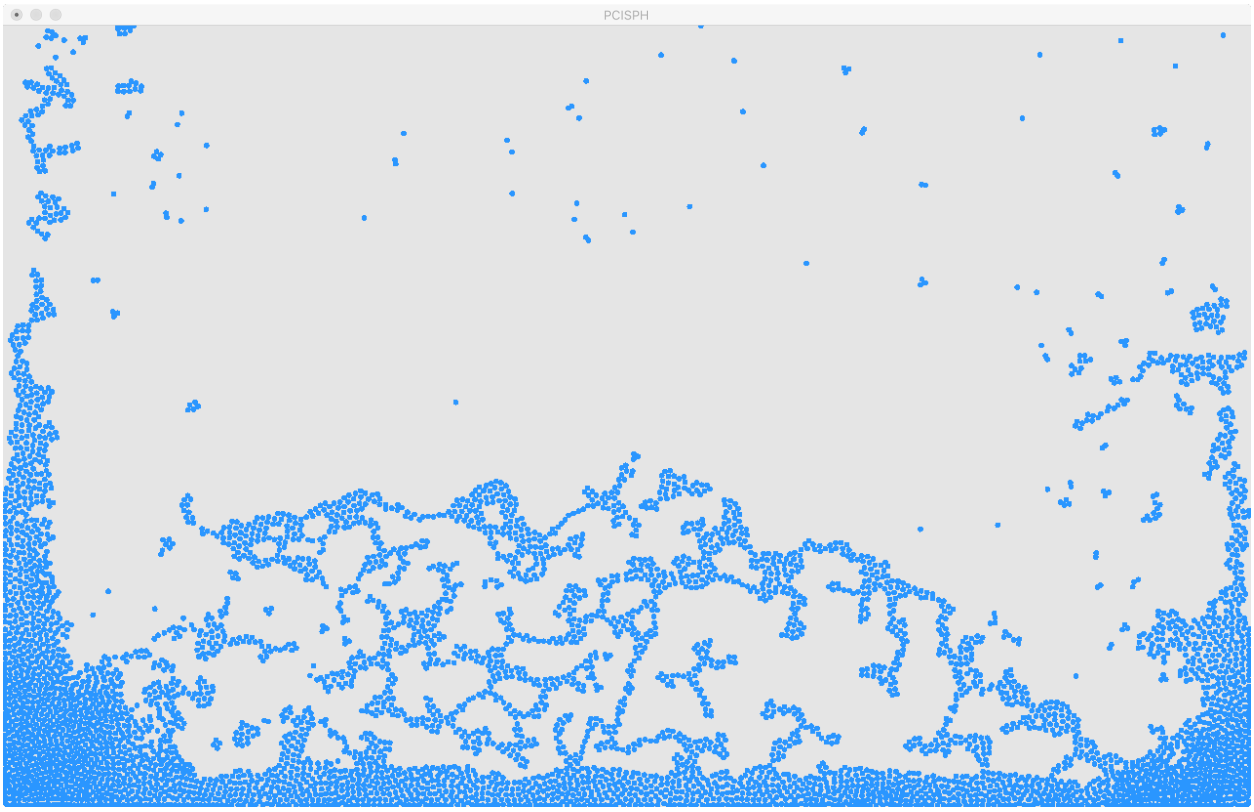

`shux` Language Proposal

Lucas Schuermann (lvs2124), John Hui (jzh2106), Mert Ussakli (mu2228), Andy Xu (lx2180)

February 7, 2017

1 Goals and Philosophy

We would like to create a language optimized for expressing, simulating, and rendering particle-based (Lagrangian) physics problems. Currently, though many implementations of solvers for such problems exist, they are frequently overly verbose, poorly organized and poorly optimized, and cluttered with helper code for rendering, spatial gridding, and multiprocessing. By introducing a revised syntax, better-suited semantics and adequate abstraction, we wish to build a general-purpose language better catered towards the needs of particle-based physics simulations.

Our language aims to be fundamentally based upon coroutines comprised of local variables and pure functions, which can be processed asynchronously over the set of all simulated particles. Other features, such as variable state lookback, inferred strong typing, and built in functional mathematical expression representation and optimization features help to facilitate a maximally concise and minimally error-prone user experience when solving domain-specific problems, largely in physical phenomena simulation, as modeled by particle-discretized partial differential equations.

2 Problem Description

Though physics simulation is a massive field with many areas of active research, much emphasis is given to so-called Lagrangian problems, or particle-based discretization schemes, commonly encountered when modeling phenomena such as granular materials, fluid dynamics, cloth, and many others. Past work in particle-based simulation includes discrete element methods (DEM) for granular materials (i.e. sand), smoothed particle hydrodynamics-type methods (SPH) for free surface fluid flow (i.e. water), extended predictive-corrected SPH and other methods (PCISPH) for more complex fluid flows (i.e. viscous fluids, such as honey, viscoelastic materials), ball-spring models for cloth simulation, astrophysical n-body simulations, and, in more recent literature, unified particle-based physics solvers for use in real-time simulation by means of explicit boundary condition modeling.

To briefly expound upon the structure of a canonical particle-based simulation, there are a number of common parts which we will aim to simplify. Consider the case of fluid simulation with smoothed particle hydrodynamics. In general, a number of particles are defined, each storing quantities such as position, velocity, pressure, density, and so on. The simulation as a whole must store the state of a number of particles (usually hundreds or thousands), define a number of methods to update these states, and then loop over time, calling these methods before re-rendering to the screen. Common methods include integration, boundary conditions, and pressure/density calculations, applied without order to all particles, but with only the latter depending on neighboring particles in the simulation. In the case of methods which depend on neighboring particles (imagine spring forces or gravitational forces), the algorithmic efficiency can be improved from $O(n^2)$ to $O(n)$ by means of a spatial gridding system, into which all particles are sorted at each time step (loop). Other considerations include ease of expression of different functions applying to particles, for example, wanting to

switch out different integration functions from Euler (1st order) to Runge-Kutta (4th order) and the ease of parallelization since applications of rules such as integration to the set of particles can be executed at the same time, proportional to the number of computing cores on the machine.

3 Language Features

1. C-like syntax
2. Type inference, strong/strict typing
3. Immutable variables by default
4. Coroutines (locally stateful) and kernels (pure functions) for easy parallelization
5. Easy (built-in) control flow for iteration over element sets
6. Variable state lookback feature: Historical variable access for use in integrators and other past state dependent methods
7. Anonymous functions
8. Anonymous variables for writing simple math (temporary intermediate objects)
9. 1st class and higher order functions, assignable to objects, support maps etc.
10. Types: int (8 bytes), float, string, boolean, vector, matrix, grid
    (a) Grid includes easy filters and selectors for iteration
11. Extensible structs for polymorphic data type definitions

`shux` will introduce a "lookback" feature for immutable variables, which exposes historical values of variables in an iterated function to the programmer. This is an often necessary feature in mathematical and physical models like Euler's approximations, fluid dynamics calculations, or the Runge-Kutta method.

Our language will also have type inferencing, but still enforce strong typing, which is most convenient for mathematical programming.

`shux` will implement **coroutines**, locally stateful functions, to represent iterative steps. Coroutines will serve as a convenient way to represent iterative computations, and can be thought of in an equivalent sense as **generators**. We will be handling multiple coroutines asynchronously and with trivial extension to parallel processing.

Along with coroutines, we will support **kernels**, which are pure functions with map-like features to produce new set of variables from and old set of variables and update coroutine iterations. These two features are intended to be best-suited for physics simulations, where

iterative mathematical computation is always necessary.

We don't want to encounter synchronization issues, therefore we will implement variables that will be **immutable** by default. All variable updates are map-like, in that they can only produce a new set of variables. This is also in line with our functional philosophy.

Lastly, rendering will be handled by built-in language features to simplify the process of creating a window, and displaying particles from vertex buffers, intended to be generated from an array of particle positions at the end of each coroutine loop.

4 Example Program: Euler's Method

```
// declare type-inferred global constants
let dt = 0.01;
let g = <0, -9.8>;

// define abstraction struct for particle
struct particle {
    float x, v;
}


// named definition of kernel function
// takes in a parameter of type particle, and returns another particle
fn euler_kernel(particle p) -> particle {

    // implicitly returns last expression in a block
    // constructs an anonymous temporary particle value to be returned
    particle : {
        // since we're trying to implicitly lookback on an argument
        // this kernel cannot be used on history-less variables
        .v = p.v..1 + g * dt;
        .x = p.x..1 + p.v..1 * dt;
    }
}


// define coroutine that takes in a list of particles and returns
// another list of particles
co euler(particle[] init) -> particle[] {

    // create named value to store its history
    // use `@` operator to map through all_particles with a lambda
    val particle[] all_particles = all_particles @ p -> {
```

```
     particle : {
            .v = p.v..1 + g * dt;
            .x = p.x..1 + p.v..1 * dt;
     }
     // returned particle uses the previous value of the particle
    } : init
    // default value of init if lookback of 1 does not yet exist

    // since the above is the only (and thus last) expression of this coroutine
    // its value is yielded every iteration
}

// code entry point
fn main() {

    // initialize particle array of size 10, all fields 0'd out
    var particle[10] p_init = {0};

    // declare another couple of arrays to which we will assign
    var particle[] p_40, p_40_select;

    // a coroutine generates an array (partcle[][]), meaning we can map over it
    for 1000 euler(p_init) @ state -> {

        // display every state (particle[]) by sending it to OpenGL
        // using library function
        displayf(state);

        // since last line ends with ';', nothing is returned
    }

    // can also just let euler() run for 40 iterations without doing anything
    // and retrieve the resulting state
    // arrays copied over using '='
    p_40 = do 40 euler(p_init);

    // use '::' operator to filter through p_40 and return subset
    p_40_select = p_40 :: p -> {

        // only return particles to the southeast of <4, 4>
        bool : p.x<0> < 4 && p.x<1> < 4
    };

    displayf(p_40_select);
}
```