A contrived example of using FLECS for Simulation

A guide to the code

Michael Sparks, October 2025

Contents

What it simulates

It simulates a collection of asteroids floating around affected by "simplified" gravity. They have all the usual features you'd expect.

It's a contrived example which is a cross between Conway's game of life and a gravity simulation of asteroids in a toroidal space.

This means:

- The behaviour of an entity depends on its current state
- That state gets updated based on nearby asteroids. (in particular acceleration is updated)
- The "nearby" asteroids is determined based on a grid, and asteroids in the current grid cell and neighbouring grid cells is taken into account.
- Furthermore only the closest K nearest neighbours are used to determine this.

Building, Running, Notes

Building the code - either use the container (preferred) or the following if the headers are installed.

```
g++ -std=c++17 asteroids_knn.cpp -o asteroids_knn -lflecs
```

Running this code results in the simulation running and being visualised using ascii art:

```
./asteroids_knn (num)
```

If you pass in "num" this is used for "K nearest neighbours", otherwise a default of 10 is used,

Note: The tutorial code was *bootstrapped* using AI under **very** tight instruction. This file walks through what the code does and why. None of the text in this document however is AI generated. (I wanted it to be correct) I've simplfied the code by doing this manually as well.

This should be sufficiently different from what you're seeking to do so it's project agnostic. But should hopefully show a pattern you can use.

ECS

This tutorial assumes the reader has explored the use of flecs with some starter exercises. Later versions of this tutorial won't

World

The world in this system is called "world"

```
flecs::world world(argc, argv);
```

This is slightly different from the default from the demos and starter project:

```
flecs::world ecs;
```

Components

The components this project uses:

```
struct Position { double x, y; };
struct Velocity { double dx, dy; };
struct Accel    { double ddx, ddy; };
struct Mass     { double m; };
struct AsteroidTag {};
```

These get registered as follows:

```
world.component<Position>();
world.component<Velocity>();
world.component<Accel>();
world.component<Mass>();
world.component<AsteroidTag>();
```

All of these are likely obvious what they're for. The `AsteroidTag` component here is actually superfluous, but shows how to use this to differentiate between things.

Systems

The following systems are created in the that world in the following order:

- A system to clear the accelerations
- A system to update the accelerations
- A system to apply the accelerations

Then, essentially the following code runs:

```
for (int i = 0; i < STEPS; ++i) {
    // Important: bins correspond to *current* positions,
    // so rebuild before systems run
    rebuild_bins();
    world.progress();
    render_ascii(vp, asteroids);
```

sleep_briefly();

```
}
```

Note that world.progress runs the systems in the order described above

Constants

```
G        = 1.0;  // Gravity factor
DT       = 0.02; // *Logical* time period inside simulation (not actual time delta)
SOFTEN2  = 1e-4; // Make the numbers not go stupid when asteroids distance is close to zero

// ---- Toroidal domain ----  (screen where edges wrap around)
```

```
W = 40.0;           // half-width  (domain x in [-W, W])
H = 20.0;           // half-height (domain y in [-H, H])
BOX_W = 2.0 * W;    // full width
BOX_H = 2.0 * H;    // full height

// ---- Uniform grid (conceptual 5x5) ----
static constexpr int GX = 5;
static constexpr int GY = 5;
```

Entity Creation - Random Numbers

```
    // Random number source
    std::mt19937 rng( std::random_device{}()  );

    // 4 Random number generators within certain bounds
    std::uniform_real_distribution<double> UposX(-W, W);
    std::uniform_real_distribution<double> UposY(-H, H);
    std::uniform_real_distribution<double> Uvel(-0.4, 0.4);
    std::uniform_real_distribution<double> Umass(0.5, 2.0);
```

Entity Creation - Memory Allocation

```
    const int N = 150;  // Number of asteroids

    std::vector<flecs::entity> asteroids;  // place to store them
    asteroids.reserve(N);  // Create the space
```

Entity Creation - Looping

Note this just loops 150 times and create 150 random asteroids.

```
    for (int i = 0; i < N; ++i) {
        asteroids.push_back(
            world.entity()
                .add<AsteroidTag>()
                .set<Position>({UposX(rng), UposY(rng)})
                .set<Velocity>({Uvel(rng), Uvel(rng)})
                .set<Accel>({0.0, 0.0})
                .set<Mass>({Umass(rng)}));
    }
```

Bins 1/3

As noted, the space is divided up into a grid - in this case hardcoded to a 5 x 5 grid. Each cell is referred to as a bin, so when this code runs:

```
for (int i = 0; i < STEPS; ++i) {
    // Important: bins correspond to *current* positions,
```

```
    // so rebuild before systems run
    rebuild_bins();
```

It's clear that the position of the asteroids changes each tick so this needs updating. (There are nicer ways of doing this, but this is clear & works)

Bins - 2/3

So `bins` is actually a grid of of integer buckets.

```
std::vector<std::vector<int>> bins(GX * GY);
```

- The grid is represented by a vector that is (grid-width x grid-height) sized.
- The things inside that vector is vectors of integers
- The integers are indices into `asteroids`

The function for building the buckets then has this logic:

```
auto rebuild_bins = [&](){
  // Clear the bins
  // Loop through the asteroids using `i` as an index
  // If for any reason asteroid `i` doesn't have a position, skip it
  // Identify the bucket for asteroid `i`, and add `i` to that
  // bucket
```

Bins - Revisited 3/3

The code looks like this:

```
    auto rebuild_bins = [&](){
        for (auto &b : bins) { b.clear(); }
        for (int i = 0; i < (int)asteroids.size(); ++i) {
            if (!asteroids[i].has<Position>())  continue;

            Position p = asteroids[i].get<Position>();
            auto [cx, cy] = pos_to_cell(p);
            bins[cy*GX + cx].push_back(i);
        }
    };
```

Perhaps the most interesting part here is `auto [cx, cy] = pos_to_cell(p);` This is a a ***structured binding***, like in python & javascript.

Systems Overview - Ordering

The ordering of the systems is actually done using the default pipeline:

- https://www.flecs.dev/flecs/md_docs_2Systems.html#builtin-pipeline

This allows you to say "this system is this kind of system" and the default pipeline runs the different kinds of systems in this specific order:

```
flecs::OnStart
flecs::OnLoad
flecs::PostLoad
flecs::PreUpdate
flecs::OnUpdate
flecs::OnValidate
flecs::PostUpdate
flecs::PreStore
flecs::OnStore
```

Systems Overview - Ordering

The ordering of the systems is actually done using the default pipeline:

- https://www.flecs.dev/flecs/md_docs_2Systems.html#builtin-pipeline

This allows you to say "this system is this kind of system" and the default pipeline runs the different kinds of systems in this specific order:

```
flecs::PreUpdate
flecs::OnUpdate

flecs::PostUpdate
```

The ones we do not use here have been removed from the above.

Systems Overview - Clearing Acceleration

This is the first system - which zeros the acceleration for all the asteroids

```
// 0) Clear accelerations
world.system<Accel>()
    .with<AsteroidTag>()
    .kind(flecs::PreUpdate)
    .each([](Accel& a){ a.ddx = 0.0; a.ddy = 0.0; });
```

Note the use of `flecs::PreUpdate` here.

Systems Overview - Setting Acceleration

So this is the high level view of this function:

```
// 1) Update the accelerations - KNN gravity using only current cell + 8 neighbours
world.system<const Position, Accel, const Mass>()
    .with<AsteroidTag>()
    .kind(flecs::OnUpdate)
    .each([&](flecs::entity self, const Position& pi, Accel& ai, const Mass& /*mi*/){
            auto [cx, cy] = pos_to_cell(pi); // Find my cell
```

- *Gather candidate indices from surrounding neighbourhood (with cell wrap)*
- *Build distances to candidates - taking into account wrap around/toroidal space*
- *Take the K-closest candidates*
- *Calculate the acceleration caused by those candidate's gravity*
- *Store that as this entity's current acceleration*

```
});
```

NOTE the use of `flecs::OnUpdate` here.

Systems Overview - Applying Acceleration, Velocity etc

Lastly the state gets updated based upon acceleration, velocity etc:

```
// 2) Apply the accelerations
world.system<Position, Velocity, const Accel>()
    .with<AsteroidTag>()
    .kind(flecs::PostUpdate)
    .each([](Position& p, Velocity& v, const Accel& a){
        v.dx += a.ddx * DT;
        v.dy += a.ddy * DT;
        p.x  += v.dx * DT;
        p.y  += v.dy * DT;
        p.x = wrap_coord(p.x, W);
        p.y = wrap_coord(p.y, H);
    });
```

ASCII Art? `render_ascii` 1

The code uses a simple viewport definition

```
struct Viewport { int w,h; double scale; };
...
    Viewport vp{100, 30, 0.6};
```

… and some terminal shenanigans to render the display as ascii art.

The code then operates as follows:

- The screen is cleared and cursor hidden, using escape codes.
- The code creates a "blank display" structure - which is a vector of strings representing screen rows
- The code loops through the asteroids, mapping their positions into the specific string in that vector
- The code then moves the cursor to the screen top left (again, escape code) and renders that.

ASCII Art? `render_ascii` 2

- The screen is cleared and cursor hidden, using escape codes.

```
std::cout << "\\x1b[2J\\x1b[?25l"; // Clear the screen and hide the cursor
```

- The code creates a “blank display” structure - which is a vector of strings representing screen rows

```
std::vector<std::string> buf(vp.h, std::string(vp.w, ' '));
```

- The code loops through the asteroids, mapping their positions into a specific string

```
for (auto e : asteroids) {
    if (e.has<Position>()) {
        Position p = e.get<Position>();
        plot(p.x, p.y, '.');
    }
}
```

ASCII Art? `render_ascii` 3

Actually rendering is then trivial - just print the strings.

- The code then moves the cursor to the screen top left and renders that.

```
std::cout << "\\x1b[H";   // Move the cursor to the home of the terminal.
for (auto& row : buf) std::cout << row << "\\n";    // Print out the rows
std::cout.flush();
```

The pattern used here.

So the pattern this uses is:

- Clear “new state info” (clear acceleration **PreUpdate**)
- Calculate the “new state information” needed per entity across all entities or across all nearby entities. (define per entity acceleration **OnUpdate**)
- Apply that new state info / state change (apply acceleration **PostUpdate**)

Full Source

```
#include <flecs.h>
#include <cmath>
#include <iostream>
#include <vector>
#include <random>
#include <thread>
#include <chrono>
#include <algorithm>
#include <utility>

struct Position { double x, y; };
struct Velocity { double dx, dy; };
struct Accel    { double ddx, ddy; };
struct Mass     { double m; };
struct AsteroidTag {};

static constexpr double G        = 1.0;
static constexpr double DT       = 0.02;
static constexpr double SOFTEN2  = 1e-4;

// ---- Toroidal domain ----
static constexpr double W = 40.0;           // half-width  (domain x in [-W, W])
static constexpr double H = 20.0;           // half-height (domain y in [-H, H])
static constexpr double BOX_W = 2.0 * W;    // full width
static constexpr double BOX_H = 2.0 * H;    // full height

// ---- Uniform grid (conceptual 5x5) ----
static constexpr int GX = 5;
static constexpr int GY = 5;

// Minimum-image displacement for one axis
static inline double min_image(double d, double half_len, double box_len) {
    if (d >  half_len) d -= box_len;
    if (d < -half_len) d += box_len;
    return d;
}

// Wrap a coordinate into [-L, L]
static inline double wrap_coord(double v, double half_len) {
    if (v < -half_len) v += 2.0 * half_len;
    if (v >  half_len) v -= 2.0 * half_len;
    return v;
}

// Map position -> grid cell index (0..GX-1, 0..GY-1), with toroidal wrap
static inline std::pair<int,int> pos_to_cell(const Position& p) {

    // Shift to [0, BOX_*), normalise, then scale to bins
```

```
    double nx = (p.x + W) / BOX_W; // [0,1)
    double ny = (p.y + H) / BOX_H; // [0,1)
    int cx = int(std::floor(nx * GX));
    int cy = int(std::floor(ny * GY));

    // Robust wrap for edge cases where p==+W or +H after rounding
    if (cx < 0) cx += GX; if (cx >= GX) cx -= GX;
    if (cy < 0) cy += GY; if (cy >= GY) cy -= GY;
    return {cx, cy};
}


// --- ASCII renderer (simple dot field) ---
struct Viewport { int w,h; double scale; }; //####//

static void render_ascii(const Viewport& vp, const std::vector<flecs::entity>& asteroids) { //####//
    std::vector<std::string> buf(vp.h, std::string(vp.w, ' '));  //####//

    auto plot = [&](double x, double y, char ch){   //####//
        int cx = int(std::round(vp.w*0.5 + x / vp.scale));
        int cy = int(std::round(vp.h*0.5 - y / vp.scale));
        if (cx>=0 && cx<vp.w && cy>=0 && cy<vp.h) buf[cy][cx] = ch;
    };

    for (auto e : asteroids) {  //####//
        if (e.has<Position>()) {
            Position p = e.get<Position>();
            plot(p.x, p.y, '.');
        }
    }

    std::cout << "\x1b[H";   // Move the cursor to the home of the terminal. //####//
    for (auto& row : buf) std::cout << row << "\n";    // Print out the rows
    std::cout.flush();
}

int main(int argc, char* argv[]) {
    flecs::world world(argc, argv); //####//

    int K = 10;
    if (argc > 1) {
        try { K = std::max(1, std::stoi(argv[1])); }  //####//
        catch (...) { std::cerr << "Invalid K; using default 10\n"; K = 10; }
    }

    world.component<Position>(); //####//
    world.component<Velocity>(); //####//
    world.component<Accel>(); //####//
    world.component<Mass>(); //####//
    world.component<AsteroidTag>(); //####//
```

```
    // ------- Create a random swarm -------
    const int N = 150; //####//

  // std::mt19937 rng(42);  // Initialise a random number generator with a known seed (useful for testing)

    // Tools for picking random numbers
  std::mt19937 rng( std::random_device{}()  ) ; // Initialise a random number generator with random device (for actua

    std::uniform_real_distribution<double> UposX(-W, W); //####//
    std::uniform_real_distribution<double> UposY(-H, H); //####//
    std::uniform_real_distribution<double> Uvel(-0.4, 0.4); //####//
    std::uniform_real_distribution<double> Umass(0.5, 2.0); //####//

    std::vector<flecs::entity> asteroids; //####//
    asteroids.reserve(N); //####//

    for (int i = 0; i < N; ++i) { //####//
        asteroids.push_back( //####//
            world.entity() //####//
                .add<AsteroidTag>() //####//
                .set<Position>({UposX(rng), UposY(rng)}) //####//
                .set<Velocity>({Uvel(rng), Uvel(rng)}) //####//
                .set<Accel>({0.0, 0.0}) //####//
                .set<Mass>({Umass(rng)})); //####//
    } //####//

    // Clamp K
    K = std::min(K, std::max(1, N - 1)); //####//

    // ------- Spatial bins - used as our grid the asteroids sit inside
    // bins[c] holds *indices* into `asteroids`.
    std::vector<std::vector<int>> bins(GX * GY); //####//

    auto rebuild_bins = [&](){ //####//
        for (auto &b : bins) { b.clear(); } //####//

        for (int i = 0; i < (int)asteroids.size(); ++i) { //####//
            if (!asteroids[i].has<Position>()) //####//
                continue; //####//
            Position p = asteroids[i].get<Position>(); //####//
            auto [cx, cy] = pos_to_cell(p); //####//
            bins[cy*GX + cx].push_back(i); //####//
        }
    };

    // ------- Systems -------
    // 0) Clear accelerations
    world.system<Accel>()
        .with<AsteroidTag>()
```

```
    .kind(flecs::PreUpdate)
    .each([](Accel& a){ a.ddx = 0.0; a.ddy = 0.0; });

// 1) Update the accelerations - KNN gravity using only current cell + 8 neighbours
world.system<const Position, Accel, const Mass>()
    .with<AsteroidTag>()
    .kind(flecs::OnUpdate)
    .each([&](flecs::entity self, const Position& pi, Accel& ai, const Mass& /*mi*/){
        // Find my cell
        auto [cx, cy] = pos_to_cell(pi);

        // Gather candidate indices from surrounding neighbourhood (with cell wrap)
        std::vector<int> candidates;
        candidates.reserve(40); // heuristic
        for (int dy = -1; dy <= 1; ++dy) {
            for (int dx = -1; dx <= 1; ++dx) {
                int nx = (cx + dx + GX) % GX;
                int ny = (cy + dy + GY) % GY;
                auto &bucket = bins[ny*GX + nx];
                candidates.insert(candidates.end(), bucket.begin(), bucket.end());
            }
        }

        // Build distances to candidates using minimum-image
        std::vector<std::pair<double,int>> dlist;
        dlist.reserve(candidates.size());
        for (int idx : candidates) {
            if (asteroids[idx] == self) continue;
            if (!asteroids[idx].has<Position>()) continue;
            Position pj = asteroids[idx].get<Position>();
            double dx = min_image(pj.x - pi.x, W, BOX_W);
            double dy = min_image(pj.y - pi.y, H, BOX_H);
            double d2 = dx*dx + dy*dy;
            dlist.emplace_back(d2, idx);
        }

        // Take K smallest
        if ((int)dlist.size() > K) {
            std::nth_element(dlist.begin(), dlist.begin() + K, dlist.end(),
                             [](const auto& a, const auto& b){ return a.first < b.first; });
            dlist.resize(K);
        }

        // Sum gravity from those neighbours (again with minimum-image dx,dy)
        for (auto& [/*d2*/_, idx] : dlist) {
            if (!asteroids[idx].has<Position>() || !asteroids[idx].has<Mass>()) continue;
            Position pj = asteroids[idx].get<Position>();
            Mass     mj = asteroids[idx].get<Mass>();

            double rx = min_image(pj.x - pi.x, W, BOX_W);
```

```
            double ry = min_image(pj.y - pi.y, H, BOX_H);

            double r2 = rx*rx + ry*ry + SOFTEN2;
            double inv_r  = 1.0 / std::sqrt(r2);
            double inv_r3 = inv_r * inv_r * inv_r;
            double s = G * mj.m * inv_r3;

            ai.ddx += rx * s;
            ai.ddy += ry * s;
          }
        });

    // 2) Apply the accelerations
    world.system<Position, Velocity, const Accel>()
        .with<AsteroidTag>()
        .kind(flecs::PostUpdate)
        .each([](Position& p, Velocity& v, const Accel& a){
            v.dx += a.ddx * DT;
            v.dy += a.ddy * DT;
            p.x  += v.dx * DT;
            p.y  += v.dy * DT;
            p.x = wrap_coord(p.x, W);
            p.y = wrap_coord(p.y, H);
        });

    // ------- Run -------
    std::cout << "\x1b[2J\x1b[?25l"; // Clear the screen and hide the cursor
    Viewport vp{100, 30, 0.6};

    const int STEPS = 4000;
    const int SLEEP_MS = 16;

    for (int i = 0; i < STEPS; ++i) {
        // Important: bins correspond to *current* positions, so rebuild before systems run
        rebuild_bins();

        world.progress();

        render_ascii(vp, asteroids);
        std::this_thread::sleep_for(std::chrono::milliseconds(SLEEP_MS));
    }
    std::cout << "\x1b[?25h\n";  // Show the cursor again
    return 0;
}
```