Physics simulation in 2D space

February 14, 2022

Table of contents

Introduction

Task:

- to model some of the physical laws of motion
- for rigid bodies
- in a two-dimensional space

Problems:

- how to represent physical quantities
- how to represent physical systems
- how to solve differential equations
- how to detect collisions
- how to constrain motion
- how to deal with numerical errors

Physical quantities

Scalar: mass, moment of inertia, coefficients - `double`
Vector: position, speed, impulse, force

```
struct Vec2 {
    double x, y;
}; // utility/mathutil.h
```

Matrix: rotation matrix

```
struct Mat2x2 {
    double m00, m10, m01, m11;
}; // utility/mathutil.h
```

Physical system - shape and body

Shape - properties of a rigid body outside of space

```
struct Shape {
    double radius;                    // For circles
    std::vector<Vec2> vert, norm; // For polygons
}; // naphy/shape.h
```

Body - a shape inside space

```
struct PhysBody {
    Shape shape;
    Vec2 pos, vel, force;
    double ang, angvel, torque;
    double m, I, m_inv, I_inv;
}; // naphy/physbody.h
```

Physical system - scene

Scene - physical system

```
struct PhysScene {
    std::vector<PhysBody> bodies;    // Bodies
    std::vector<Arbtier> arbiters;   // Arbiters
    std::vector<Spring> springs;     // Springs
}; // naphy/physscene.h
```

Differential equations

Equations of motion (2^{nd} Newton's law)
Semi-implicit Euler method

```
vel += acc * dt; // Explicit
pos += vel * dt; // Implicit
```

Other methods (e.g. multistep Adams-Bashforth) more precise but stability is the same

Collision detection

circle-circle - compare distance to sum of radii
circle-polygon - *SAT*: normal with minimal projection intersection
polygon-polygon - *SAT*

Optimization - support point function
Polygon clipping - *Sutherland-Hodgman* algorithm

Collision detection - arbiter

Arbiter - all data for a collision of two bodies

```
struct Arbiter {
    PhysBody *A, *B;
    double depth;
    Vec2 normal;
    std::vector<Vec2> contact;
}; // naphy/arbiter.h
```

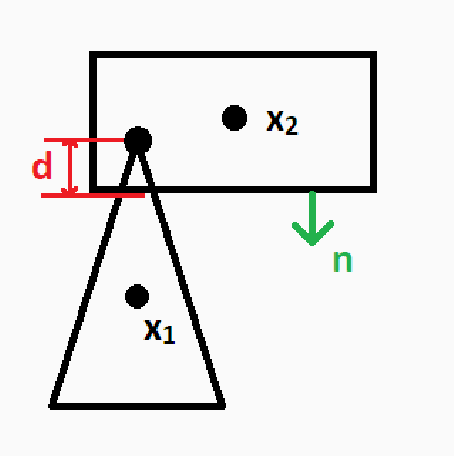
Constraints

$C(x) = 0$
We're constraining position and rotation: $C(x) = C(p, r)$
Non-penetration constraint - we need two bodies: $C(x) = C(p_1, r_1, p_2, r_2)$

Solve for derivative: $C(x) = 0 \Rightarrow \dot{C}(x) = 0$

Constraints - non-penetration constraint

If $\dot{C} \neq 0$, we need to apply force on bodies



$$C(p_1, r_1, p_2, r_2) = d = n \cdot (x_2 - x_1)$$

Constraints - non-penetration constraint

$$
\begin{aligned}
&C = (x_2 - x_1) \cdot n \\
&\dot{C} = \tfrac{d}{dt}(x_2 - x_1) \cdot n + (x_2 - x_1) \cdot \tfrac{d}{dt}n \\
&= (\tfrac{d}{dt}x_2 - \tfrac{d}{dt}x_1) \cdot n + (x_2 - x_1) \cdot \tfrac{d}{dt}n \\
&= (v_2 + \omega_2 \times r_2 - v_1 - \omega_1 \times r_1)n + (x_2 - x_1) \cdot \tfrac{d}{dt}n
\end{aligned}
$$

We only care about velocity, so $(x_2 - x_1) \cdot \frac{d}{dt}n$ is ignored:

$$
\begin{aligned}
&= (v_2 + \omega_2 \times r_2 - v_1 - \omega_1 \times r_1)n \\
&= v_2 n + (\omega_2 \times r_2)n - v_1 n - (\omega_1 \times r_1)n
\end{aligned}
$$

Recall triple product property: $(a \times b) \cdot c = a \cdot (b \times c)$:

$$
\begin{aligned}
&= v_2 n + \omega_2(r_2 \times n) - v_1 n - \omega_1(r_1 \times n) \\
&= \begin{bmatrix} -n & -(r_1 \times n) & n & (r_2 \times n) \end{bmatrix} \begin{bmatrix} v_1 & \omega_1 & v_2 & \omega_2 \end{bmatrix}
\end{aligned}
$$

Constraints - non-penetration constraint

$$\dot{C} = \begin{bmatrix} -n & -(r_1 \times n) & n & (r_2 \times n) \end{bmatrix} \begin{bmatrix} v_1 & \omega_1 & v_2 & \omega_2 \end{bmatrix} = J \cdot V$$

If $\dot{C} \neq 0$, then there exists ΔV such that $\dot{C} = J(V + \Delta V) = 0$
We want the impulse* ($p = m\Delta v, L = r \times p$)
$\Delta V = \begin{bmatrix} -\frac{p}{m_1} & -\frac{r_1 \times p}{I_1} & \frac{p}{m_2} & \frac{r_2 \times p}{I_2} \end{bmatrix}$
We know the direction of the impulse (n), magnitude is unknown: $p = \lambda n$
$\Delta V = \begin{bmatrix} -\frac{p}{m_1} & -\frac{r_1 \times p}{I_1} & \frac{p}{m_2} & \frac{r_2 \times p}{I_2} \end{bmatrix} =$
$\begin{bmatrix} -\frac{\lambda n}{m_1} & -\frac{r_1 \times \lambda n}{I_1} & \frac{\lambda n}{m_2} & \frac{r_2 \times \lambda n}{I_2} \end{bmatrix} =$
$\lambda \begin{bmatrix} -\frac{n}{m_1} & -\frac{r_1 \times n}{I_1} & \frac{n}{m_2} & \frac{r_2 \times n}{I_2} \end{bmatrix}$
Solving the constraint means finding a λ such that $J(V + \Delta V) = 0$

* why impulse?

Sequential impulses

Why impulses?

One collision $\Rightarrow$ one arbiter $\Rightarrow$ one constraint $\Rightarrow$ one λ
Multiple collisions at one moment $\Rightarrow$ multiple λ to solve
System of constraints

(A) Global solver
(Б) Iterative solver

Sequential impulses - iterative method

Sequential impulses

$$\Delta V = \lambda \begin{bmatrix} -\frac{n}{m_1} & -\frac{r_1 \times n}{I_1} & \frac{n}{m_2} & \frac{r_2 \times n}{I_2} \end{bmatrix}$$

Extract the mass:

$$\Delta V = \lambda \begin{bmatrix} \frac{1}{m_1} & 0 & 0 & 0 \\ 0 & \frac{1}{I_1} & 0 & 0 \\ 0 & 0 & \frac{1}{m_2} & 0 \\ 0 & 0 & 0 & \frac{1}{I_2} \end{bmatrix} \begin{bmatrix} -n \\ -(r_1 \times n) \\ n \\ (r_2 \times n) \end{bmatrix} = \lambda M^{-1} J^T$$

$$J(V + \Delta V) = 0$$
$$\Rightarrow \lambda = -\frac{JV}{JM^{-1}J^T}$$

Sequential impulses

$\lambda = -\frac{JV}{JM^{-1}J^T}, \quad p = m\Delta v = \lambda n$

```
// JV = C' = dv * n
Vec2 dv = (B->vel + cross(B->angvel, r2))
         -(A->vel + cross(A->angvel, r1));
Vec2 dvn = dot(dv, n);
// J M^-1 J^T
double m = (A->m_inv + r1n * r1n * A->I_inv)
          +(B->m_inv + r2n * r2n * B->I_inv);
// p = lambda * n
Vec2 impulse = (-dvn / m) * n;
A->vel -= impulse * A->m_inv;
B->vel += impulse * B->m_inv;
// naphy/arbiter.cpp :: solve(), apply_impulse()
```

Sequential impulses

For small dt we have $p \approx Fdt$
Each iteration gives a slightly better final velocity

1) Apply all external forces (e.g. gravity) once:
$v_{(0)} = v_{prev} + m^{-1}F_e dt$
2) In k iterations apply constraint impulses:
$v_{(i)} = v_{(i-1)} + m^{-1}p$

```cpp
const unsigned iterations = 10;
for (unsigned j = 0; j < iterations; j++) {
    for (unsigned i = 0; i < scene->arbiter.size(); i++) {
        scene->arbiter[i].solve();
    }
} // naphy/physscene.cpp :: scene_update_constraints()
```

Friction

Similar process, but tangent to the collision
Static friction and kinetic friction
Coulomb's law: $|F_s| \leq \mu |F_n|$

```
double lambda_t = - dot(dv, t) / m;
if (abs(lambda_t) > u * abs(lambda))
    lambda_t = -kfric * abs(lambda); // kinetic
else
    lambda_t = 0; // static
Vec2 impulse_t = lambda * t;
A->vel -= impulse_t * A->m_inv;
B->vel += impulse_t * B->m_inv;
A->angvel -= cross(r1, impulse_t) * A->I_inv;
B->angvel += cross(r2, impulse_t) * B->I_inv;
// naphy/arbiter.cpp :: solve()
```

Springs

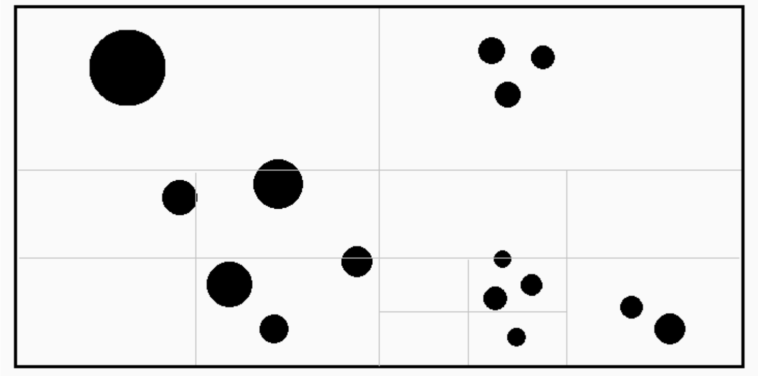
Hooke's law: $F = -k\Delta x$
$\Rightarrow$ penalty function, external force
$F = (-k\Delta x) \cdot (p_A - p_B) + c \cdot (v_A - v_B)$
c - damping factor

```
struct Spring {
    PhysBody *A, *B;
    double rest_length;
    double k;
    double c;
}; // naphy/spring.h
```

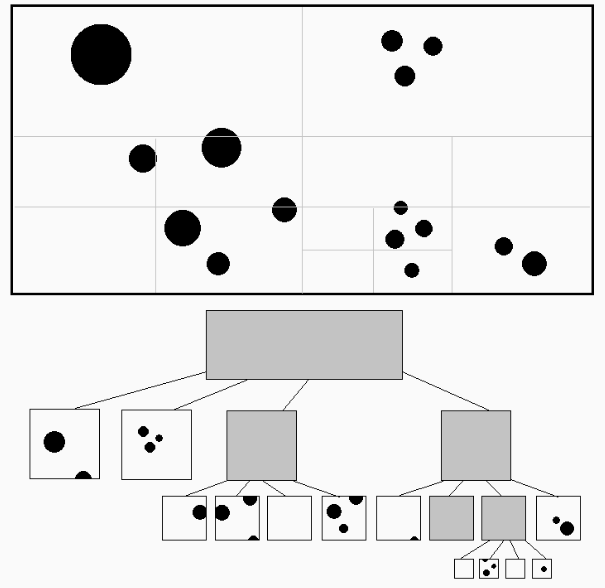
Spatial indexing

Broad-phase, *middle-phase*, *narrow-phase*

Quadtree



Spatial indexing - quadtree



Spatial indexing - quadtree

```
struct QuadNode {
    Vec2 pos, size;
    std::vector<PhysBody*> obj;
    QuadNode* child[4];
    unsigned capacity;
}; // naphy/quadtree.h
for (QuadNode& leaf : leaves) {
    std::vector<PhysBody*>* body = leaf->object;
    for (unsigned i = 0; i < body->size; i++) {
        for (unsigned j = i + 1; j < body->size; j++) {
            PhysBody *A = body[i], *B = body[j];
        }
    }
} // naphy/physscene.cpp :: collision_quadtree()
```

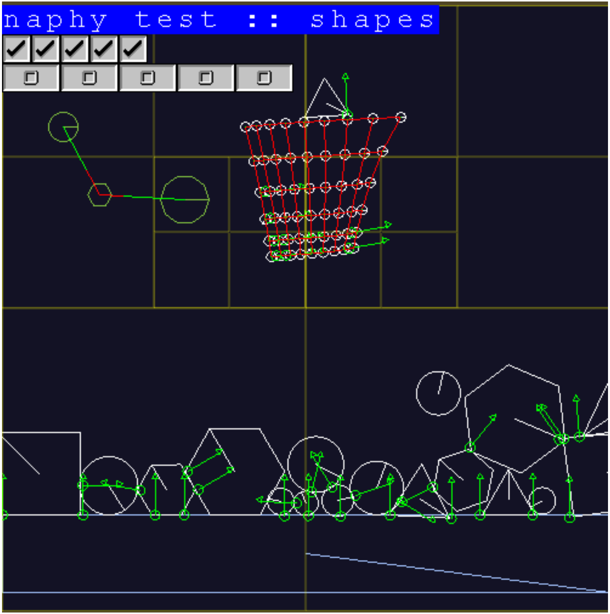
Dealing with numerical errors

Coefficient of restitution: $e \in [0, 1]$
$\lambda = -(1 + e) \cdot \frac{JV}{JM^{-1}J^T}$
Baumgarte stabilization: $\beta \in [0, 1], b = \beta \cdot \frac{C}{dt}$
$\lambda = -(1 + e + b)\frac{JV}{JM^{-1}J^T}$
Clamping the impulse: λ_{acc}: $\lambda`_{(i)} = max(\lambda_{acc} + \lambda_{(i)}, 0) - \lambda_{acc}$
Penetration slop: apply impulse $\lambda_{slop} = \beta_{bias} max(C - \beta_{slop})$

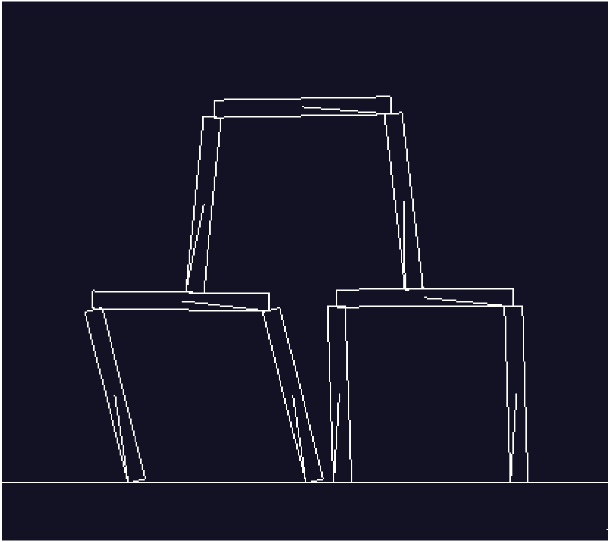
```
double slop = 0.07f;
double bias = 0.6f;
double correction = bias * max(depth - slop, 0.0);
Vec2 pcorr = n * correction / (A->m_inv + B->m_inv);

A->pos -= pcorr * A->m_inv;
B->pos += pcorr * B->m_inv;
// naphy/arbiter.cpp :: post_solve()
```

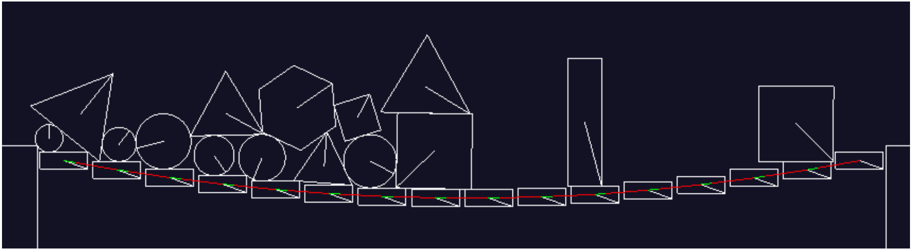
Example - bodies, arbiters, springs, quadtree



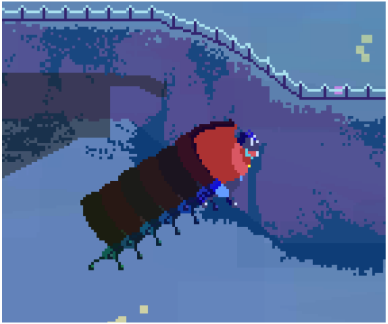
Example - instability

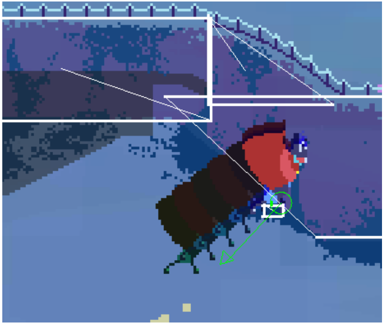


Пример - bridge



Example - something less abstract





Literature

Erin Catto: Iterative Dynamics with Temporal Coherence
Erin Catto: Fast and Simple Physics using Sequential Impulses
Erin Catto: Modeling and Solving Constraints
ImpulseEngine
Improving the stability of your physics
Collision Response
Ming-Lun Chou: Constraints & Sequential Impulse