PhysicsEngine

April 29, 2022

1 Physics Engine

1.1 Why?

In video games, we want to simulate realistic movements. Animations are produced by displaying successive images like in cartoon movies. When the images are displayed quickly, the effect is an apparent smooth and continuous movement of the objects.

To have realistic movement, we need to update state of objects according to the laws of physics.

1.2 Newtonian mechanics

To modelize rigid bodies, we use the Newtonian mechanics founded by Isaac Newton and the three laws of motion.

1.2.1 Inertia

If no force is applied on an object, it continues to move infinitely with the same velocity.

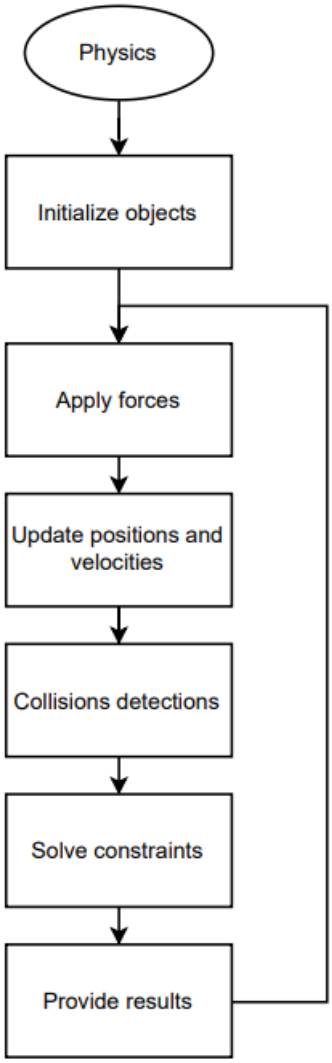
1.2.2 Forces and acceleration

Sum of forces acting on an object is equal to the mass of the object multiplied by its acceleration. $\sum \vec{F_{ext}} = m\vec{a}$

1.2.3 Action and reaction

When a first body applies a force $\vec{F}$ to a second body, the second body applies a force $\vec{R}$ with the same magnitude to the first body in an opposite direction. $\vec{F} = -\vec{R}$.

1.3 Physic Engine loop



1.4 Everything begins with a particle

To understand physic simulation, it's very interesting to begin by particle simulation.

To make the simulation realistic, the time step to simulate must be the same as the real amount of time that passed since the last simulation step. Inside a game engine, this time could be scaled easily by passing the elapsed time.

We have one particle with mass m, position $p(t_i)$ and velocity $v(t_i)$ at a specific time t_i. We apply a force on this particle at this time $F(t_i)$ and we will determine velocity and position at t_{i+1}.

$F(t) = ma(t) \Leftrightarrow a(t) = F(t)/m$

By applying Euler method : $x(t + \Delta t) = x(t) + \dot{x}(t)\Delta t$

Let $\Delta t = t_{i+1} - t_i$,

$$v(t_i + \Delta t) = v(t_{i+1}) = v(t_i) + \dot{v}(t_i)\Delta t \iff \boxed{v(t_{i+1}) = v(t_i) + a(t_i)\Delta t}$$

$$\boxed{p(t_{i+1}) = p(t_i) + v(t_i)\Delta t}$$

1.4.1 Point

[1]:
```cpp
#include <iostream>

namespace Maths
{
    template<typename Type>
    class Point2D
    {
    public:
        explicit Point2D(const Type& _x = 0, const Type& _y = 0)
            : x(_x)
            , y(_y)
        {}

        Type x;
        Type y;
    };

    template<typename Type>
    std::ostream& operator<<(std::ostream& os, const Point2D<Type>& point)
    {
        os << "(" << point.x << ", " << point.y << ")";
        return os;
    }
}
```

1.4.2 Vector

[2]:
```cpp
#include <cmath>

namespace Maths
{
    template<typename Type>
    class Vector2D
    {
    public:
        explicit Vector2D(const Type& _x = 0, const Type& _y = 0);
        Vector2D(const Point2D<Type>& begin, const Point2D<Type>& end);

        Type dotProduct(const Vector2D& lhs);
        Vector2D getNormalized() const;
        Vector2D& normalize() const;
```

```
        Type squareLength() const;
        Type getLength() const;

        Type x;
        Type y;
    };
}
```

[3]:
```
namespace Maths
{
    template<typename Type>
    Vector2D<Type>::Vector2D(const Type& _x, const Type& _y) : x(_x), y(_y) {}

    template<typename Type>
    Vector2D<Type>::Vector2D(const Point2D<Type>& begin, const Point2D<Type>&␣
 ↪end)
        : Vector2D(end.x - begin.x, end.y - begin.y)
    {}

    template<typename Type>
    Type Vector2D<Type>::dotProduct(const Vector2D& lhs) { return x * lhs.x + y␣
 ↪* lhs.y; }

    template<typename Type>
    Vector2D<Type> Vector2D<Type>::getNormalized() const
    {
        const auto length = getLength();
        return Vector2D{ x / length, y / length};
    }

    template<typename Type>
    Vector2D<Type>& Vector2D<Type>::normalize() const
    {
        const auto length = getLength();
        x /= length;
        y /= length;
        return *this;
    }
}
```

[4]:
```
namespace Maths
{
    template<typename Type>
    Type Vector2D<Type>::squareLength() const { return x * x + y * y; }

    template<typename Type>
    Type Vector2D<Type>::getLength() const { return std::sqrt(squareLength()); }
```

```
    template<typename Type>
    Vector2D<Type> operator+(const Vector2D<Type>& lhs, const Vector2D<Type>&␣
 ↪rhs)
    {
        return Vector2D<Type>{ lhs.x + rhs.x, lhs.y + rhs.y };
    }

    template<typename Type>
    Point2D<Type> operator+(const Vector2D<Type>& vec, const Point2D<Type>&␣
 ↪point)
    {
        return Point2D<Type>{vec.x + point.x, vec.y + point.y};
    }
}
```

[5]:
```
namespace Maths
{
    template<typename Type>
    Point2D<Type> operator+(const Point2D<Type>& point, const Vector2D<Type>&␣
 ↪vec)
    {
        return Point2D<Type>{vec.x + point.x, vec.y + point.y};
    }

    template<typename Type>
    Vector2D<Type> operator-(const Vector2D<Type>& lhs, const Vector2D<Type>&␣
 ↪rhs)
    {
        return Vector2D<Type>{ lhs.x - rhs.x, lhs.y - rhs.y };
    }

    template<typename Type>
    Point2D<Type> operator-(const Point2D<Type>& point, const Vector2D<Type>&␣
 ↪vec)
    {
        return Point2D<Type>{-vec.x + point.x, -vec.y + point.y};
    }
}
```

[6]:
```
namespace Maths
{
    template<typename Type>
    Type operator*(const Vector2D<Type>& lhs, const Vector2D<Type>& rhs)
    {
        return lhs.x * rhs.y - lhs.y * rhs.x;
```

```
    }

    template<typename Type>
    Vector2D<Type> operator*(const Type& lhs, const Vector2D<Type>& rhs)
    {
        return Vector2D<Type>{ lhs * rhs.x, lhs * rhs.y };
    }

    template<typename Type>
    Vector2D<Type> operator*(const Vector2D<Type>& lhs, const Type& rhs)
    {
        return Vector2D<Type>{ lhs.x * rhs, lhs.y * rhs };
    }

    template<typename Type>
    std::ostream& operator<<(std::ostream& os, const Vector2D<Type>& vec)
    {
        os << "(" << vec.x << ", " << vec.y << ")";
        return os;
    }
}
```

1.4.3 Particle

[7]:
```
namespace Physics
{
    template<typename Type>
    struct Particle
    {
        Maths::Point2D<Type> position;
        Maths::Vector2D<Type> velocity;
        Type mass;
    };
}
```

1.4.4 Particle system

[8]:
```
#include <random>
#include <vector>
#include <thread>

namespace Physics
{
    using Point2d = Maths::Point2D<double>;
    using Vector2d = Maths::Vector2D<double>;
    using ParticleD = Particle<double>;
```

```
    constexpr double g_gravity = 9.81;

    class ParticleSystem;
    std::ostream& operator<<(std::ostream& os, const ParticleSystem&␣
 ↪particleSystem);

    class ParticleSystem
    {
        friend std::ostream& operator<<(std::ostream& os, const ParticleSystem&␣
 ↪particleSystem);
    public:
        ParticleSystem(const size_t& numberOfParticle);
        void simulate(const double& elapsedTimeMs, const double& stepMs = 1.0);

    private:
        Vector2d computeGravityForce(ParticleD& particle);
        std::vector<ParticleD> m_particles;
    };
}
```

[9]:
```
namespace Physics
{
    std::ostream& operator<<(std::ostream& os, const ParticleSystem&␣
 ↪particleSystem)
    {
        for (size_t i = 0; i < particleSystem.m_particles.size(); ++i)
            os << "[" << i << "] p" << particleSystem.m_particles[i].position
                << " v" << particleSystem.m_particles[i].velocity
                << " m=" << particleSystem.m_particles[i].mass << std::endl;

        return os;
    }

    ParticleSystem::ParticleSystem(const size_t& numberOfParticle)
    {
        m_particles.resize(numberOfParticle);
        std::random_device rd;
        std::mt19937 gen(rd());
        std::uniform_real_distribution<> dis(0.0, 100.0);
        for (auto& particle : m_particles)
        {
            particle.position = Point2d{ dis(gen), dis(gen) };
            particle.velocity = Vector2d{ 0.0, 0.0 };
            particle.mass = 1.0;
        }
    }
```

```
}
```

[10]:
```
namespace Physics
{
    void ParticleSystem::simulate(const double& elapsedTimeMs, const double&␣
 ↪stepMs)
    {
        auto currentTimeMs = 0.0;
        while (currentTimeMs < elapsedTimeMs)
        {
            std::this_thread::sleep_for(std::chrono::milliseconds(1));

            for (auto& particle : m_particles)
            {
                const auto gravityForce = computeGravityForce(particle);
                const auto gravityAcceleration = Vector2d{gravityForce.x /␣
 ↪particle.mass, gravityForce.y / particle.mass};
                particle.position.x += particle.velocity.x * stepMs;
                particle.position.y += particle.velocity.y * stepMs;
                particle.velocity.x += gravityAcceleration.x * stepMs;
                particle.velocity.y += gravityAcceleration.y * stepMs;
            }

            std::cout << *this << std::endl;
            currentTimeMs += stepMs;
        }
    }

    Vector2d ParticleSystem::computeGravityForce(ParticleD& particle)
    {
        return Vector2d{0.0, g_gravity * particle.mass };
    }
}
```

[11]:
```
Physics::ParticleSystem particleSystem(3);
std::cout << particleSystem << std::endl;
particleSystem.simulate(5);
```

```
[0] p(73.5764, 89.5599) v(0, 0) m=1
[1] p(88.4619, 40.5264) v(0, 0) m=1
[2] p(91.4966, 37.9725) v(0, 0) m=1

[0] p(73.5764, 89.5599) v(0, 9.81) m=1
[1] p(88.4619, 40.5264) v(0, 9.81) m=1
[2] p(91.4966, 37.9725) v(0, 9.81) m=1

[0] p(73.5764, 99.3699) v(0, 19.62) m=1
```

```
[1] p(88.4619, 50.3364) v(0, 19.62) m=1
[2] p(91.4966, 47.7825) v(0, 19.62) m=1

[0] p(73.5764, 118.99) v(0, 29.43) m=1
[1] p(88.4619, 69.9564) v(0, 29.43) m=1
[2] p(91.4966, 67.4025) v(0, 29.43) m=1

[0] p(73.5764, 148.42) v(0, 39.24) m=1
[1] p(88.4619, 99.3864) v(0, 39.24) m=1
[2] p(91.4966, 96.8325) v(0, 39.24) m=1

[0] p(73.5764, 187.66) v(0, 49.05) m=1
[1] p(88.4619, 138.626) v(0, 49.05) m=1
[2] p(91.4966, 136.072) v(0, 49.05) m=1
```

1.5 Rigid body simulation

1.5.1 What is a rigid body?

A rigid body is a solid that cannot be deformed. It's a very useful model used by video games developers to represent physics in games and to simplify the dynamics of solids.

Rigid bodies have same caracteristics than particles but with some additions : shape and volume (for 3D).

Moreover, a rigid body could be affect by another transformation: the rotation.

1.5.2 Center of mass

To summarize By default a rigid body rotates around its center of mass, and we place it in the space according to its center of mass. So its position is the position of its center of mass.

Physical consideration The center of mass is the barycenter of the mass distribution of a body.

Discrete representation If we have a rigid body of mass M composed of n tiny particles with mass m_i and relative position r_i inside the body.

The center of mass is:

$$\boxed{c = \frac{1}{M}\sum_{i=1}^{i=n} m_i r_i}$$

So the center of mass is just the average of the particle position weighted by their mass.

Uniform rigid body For uniform rigid body, the center of mass is the same as the geometric center of body shape.

The geometric center of body shape is called **centroid**. In video games we work with uniform density and center of mass and centroid have the same position but we don't forget that it's not the same concepts.

Continuous representation Rigid bodies are not discrete and for each particle that we can select, there will be always a particle between. Indeed there are not a finite number of discrete particles but it's continuous.

We need to use a continuous sum with an integral.

$$\boxed{c = \frac{1}{M}\int_V \rho(r) r dV}$$

- M is the mass of rigid body
- ρ is the density function
- r is the position vector
- V is the continuous volume

1.6 Angular velocity

Given a rigid body could rotate, we could represent the angular velocity ω in radians per second.

1.7 Torque

To accumulate angular velocity, we need to apply to the rigid body a rotational force called **torque** τ. By using the second law of Newton:

$$\boxed{\tau = I\alpha}$$

- α is the angular acceleration
- I is the moment of inertia

1.7.1 Moment of inertia

The moment is like the formula of the center of mass but for rotations.

In 2D, the result is a scalar

$I = \int_V \rho(r) r^2 dV$

- V is the volume
- r is the position vector in the polar space
- r^2 is the result of dot product of $\vec{r}.\vec{r}$
- ρ is the density function

Example with rectangle Let $\int_x \int_y \rho(r) r^2 dydx$

The body is homogeneous so

$\int_x \int_y r^2 dydx = \int_x \int_y (x^2 + y^2) dydx = \int_x \int_y x^2 dydx + \int_x \int_y y^2 dydx$

with * $I_x = \int_x \int_y x^2 dydx$ * $I_y = \int_x \int_y y^2 dydx$ * $J_z = I_x + I_y$

For a rectangle of width w and height h with centroid on the origin, we have the following equations:

$$I_x = \int_{-\frac{w}{2}}^{\frac{w}{2}} \int_{-\frac{h}{2}}^{\frac{h}{2}} x^2 dydx = \int_{-\frac{w}{2}}^{\frac{w}{2}} x^2 \left(y\right)\Big|_{-\frac{h}{2}}^{\frac{h}{2}} dx = \int_{-\frac{w}{2}}^{\frac{w}{2}} hx^2 = h\left(\frac{x^3}{3}\right)\Big|_{-\frac{w}{2}}^{\frac{w}{2}}$$

$$\boxed{I_x = \frac{hw^3}{12}}$$

$I_y = \int_{-\frac{w}{2}}^{\frac{w}{2}} \int_{-\frac{h}{2}}^{\frac{h}{2}} y^2 dy dx = \int_{-\frac{w}{2}}^{\frac{w}{2}} \left(\frac{y^3}{3}\right)\Big|_{-\frac{h}{2}}^{\frac{h}{2}} dx = \frac{1}{3} \int_{-\frac{w}{2}}^{\frac{w}{2}} \frac{h^3}{4}$

$$\boxed{I_y = \frac{wh^3}{12}}$$

$$\boxed{J_z = I_x + I_y = \frac{hw}{12}(w^2 + h^2)}$$

To compute the moment of inertia of our object we have to use ρ function multiply with hw to have the mass M

So $\boxed{I = \frac{M}{12}(w^2 + h^2)}$

1.7.2 Torque and force

When a force $\vec{F}$ is applied to a point M on a rigid body with a center of mass C, it may produce torque.

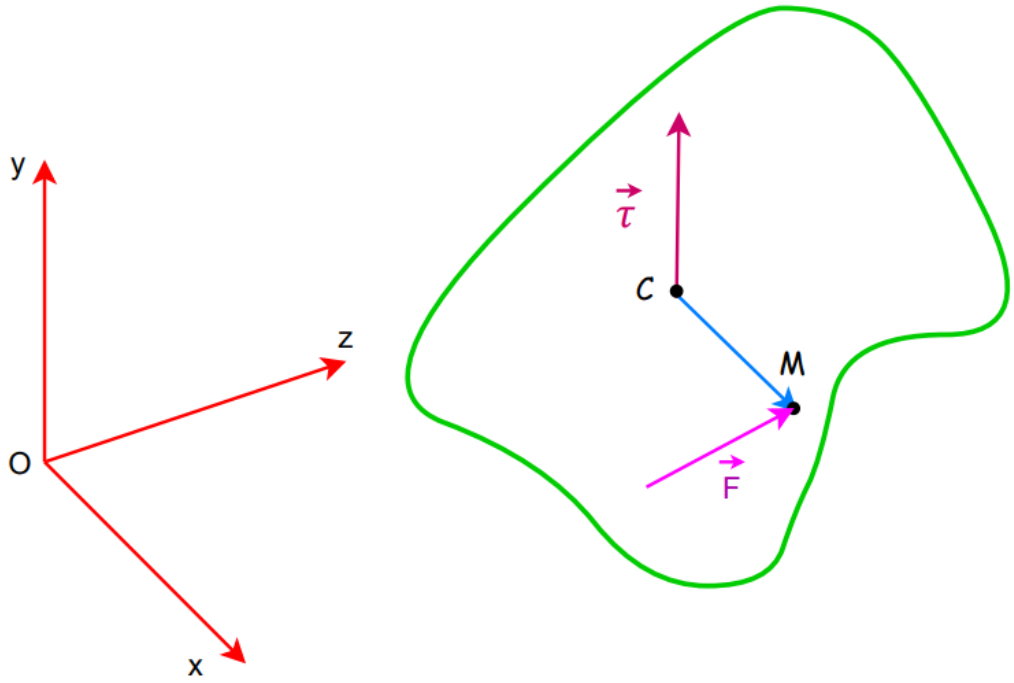
$$\boxed{\vec{\tau} = C\vec{M} \times \vec{F}}$$

In 2D, the result is a scalar and not a vector.

Magnitude and cross product We can compute the torque with the formula according to the cross product definition.

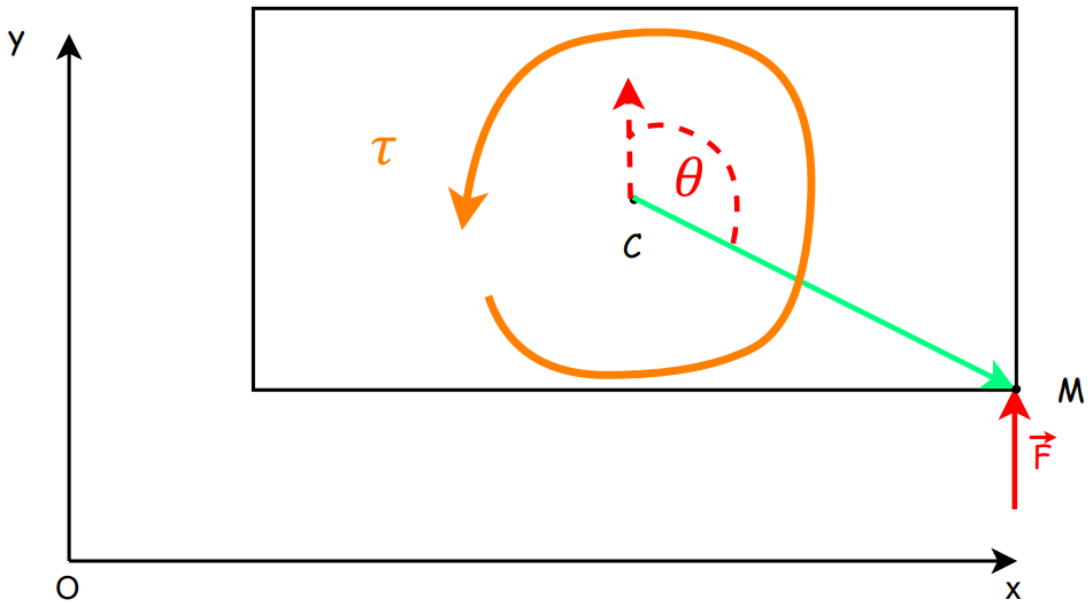
$$\boxed{\tau = ||C\vec{M}|| \times ||\vec{F}|| sin(\theta)}$$

θ is the oriented angle $(C\vec{M}, \vec{F})$



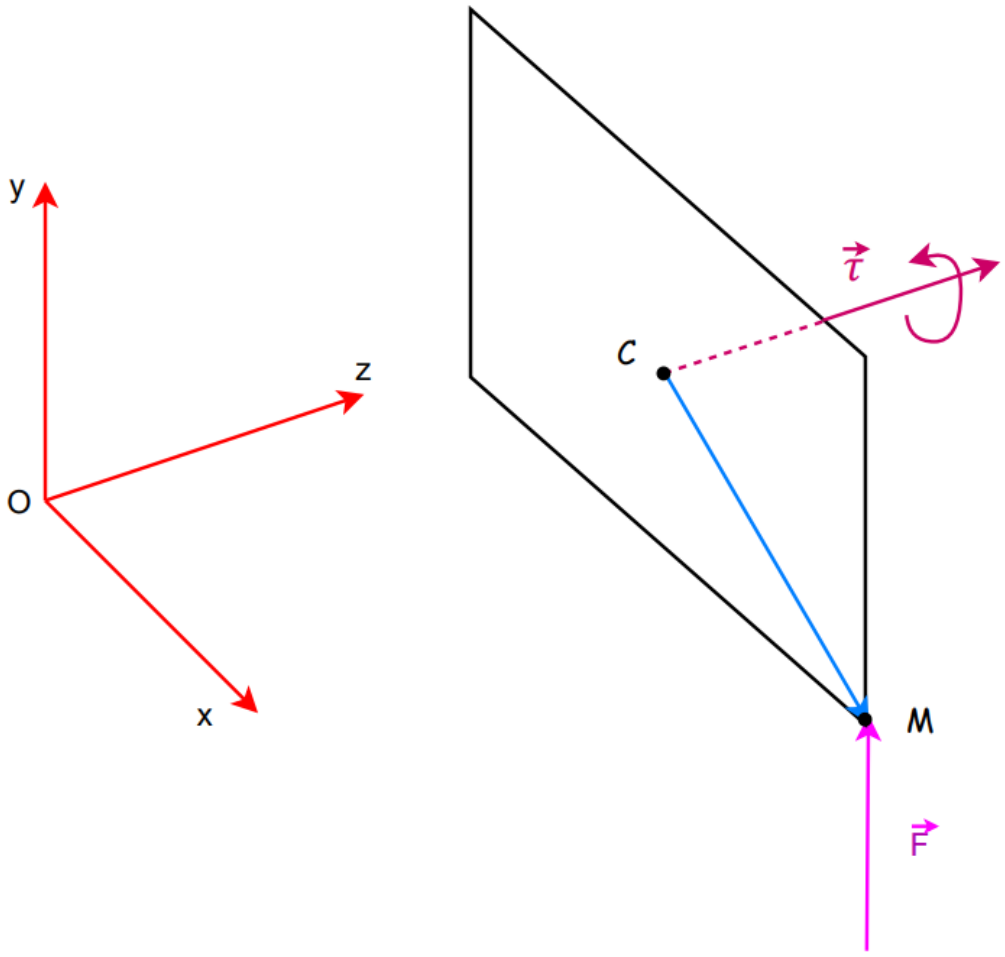
Example in 3D

Exemple in 2D In two dimensions, we have defined that the result is a scalar. We could notice that the scalar product implies a rotation.



Why we have a scalar in 2D?

In fact, the result of a cross product is a vector but in 2D, we have zero x and y component. So we don't show the vector in the third dimension but we could see how it is in the following figure.



[ ]: