HW1

0856631 陳立軒

1 Introduction

Rigid body 2D is the thing that can only move in the XY plane and can only rotate on an axis perpendicular to the plane. In this homework, we are asked to implement rigid body 2D system which conforms to physical properties but not including rotation.

In this homework, rigid body 2D contains Axis Aligned Bounding Box(AABB) and circle, we need to detect collision and resolve impulse between them. Moreover, there are two types of joint constraints we need to apply including spring and distance joint.

Finally, in order to let rigid body 2D move, we need to implement integrator, which includes Explicit Euler method and Runge Kutta Fourth(RK4).

2 Fundamentals

Collision in physics refers to the state of motion of a system, and significant change will occur in a short time interval and the state of the system changes suddenly. In this very short time interval, it is usually difficult to determine the force at each instant. Therefore, we must use Newton's second law of motion.

Newton's second law of motion can be formally stated as follows: The acceleration of an object as produced by a net force is directly proportional to the magnitude of the net force, in the same direction as the net force, and inversely proportional to the mass of the object. This verbal statement can be expressed in equation form as follows: $a = \frac{F_{net}}{m}$, as long as we get acceleration, we can compute velocity by $v = v_0 + a\Delta t$.

3 Implementation

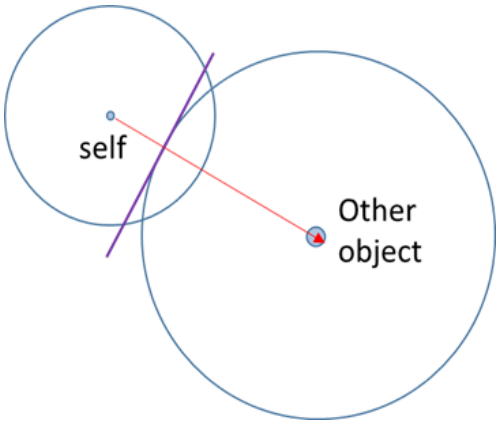
3.1 Collision detection

This topic is that how to detect collision in my physics system, we have three types of collision to detect; Before starting to introduce collision detection, I want to introduce the data structure first.

```
class Manifold
{
    typedef linalg::aliases::float2 float2;
public:
    std::shared_ptr<RigidBody2D> m_body0, m_body1;
    float2 m_normal;
    float m_penetration;

    bool m_isHit;

public:

    Manifold(
        std::shared_ptr<RigidBody2D> _body0,
        std::shared_ptr<RigidBody2D> _body1,
        float2 _normal,
        float _penetration,
        bool _isHit);

    void Resolve() const;
    void PositionalCorrection() const;
};
```

Manifold is the unit of collision detection, m_body0 represents self, and m_body1 represents other object, m_normal represents which direction make the other object exit in the most efficient way, $m\_penetration$ represents how much it penetrates, m_isHit represents whether m_body0 collides with m_body1.

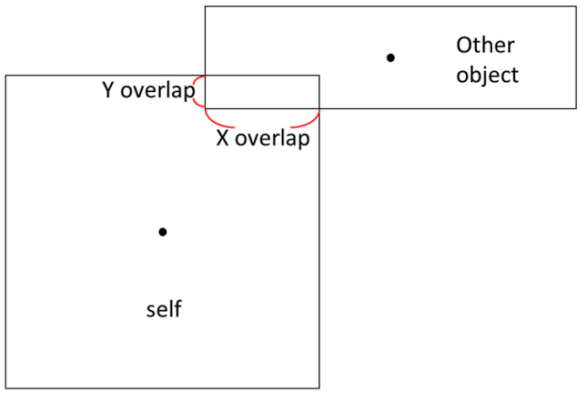
1. Circle to circle:



Because the property of circle is that the distance between center of circle and anywhere in the circumference is the same, so the m_normal is the vector with respect to two center of circles, and $m\_penetration$ is sum of the radius of two circles minus the distance of center of mass, and if $m\_penetration > 0$, two circles collide.

```
Manifold Circle::visitCircle(std::shared_ptr<const Circle> _shape) const
{
    float center_of_mass_distance = linalg::distance(_shape->m_body->GetPosition(), m_body->GetPosition());
    float penertration_depth = m_radius + _shape->m_radius - center_of_mass_distance;

    return Manifold(
        m_body,
        _shape->m_body,
        linalg::normalize(_shape->m_body->GetPosition() - m_body->GetPosition()),
        penertration_depth,
        penertration_depth > 0
    );
}
```

2. AABB to AABB:



Different from above case, m_normal in AABB to AABB has four types depending on relative position of two objects and magnitude comparison between $X - overlap$ and $Y - overlap$. I handle multiple situations simply by calculating the vector of center of mass and determine which overlapped area is more small against X-axis and Y-axis, and if $penetration\_depth$ in X-axis and Y-axis are more than zero, two AABBs collide.

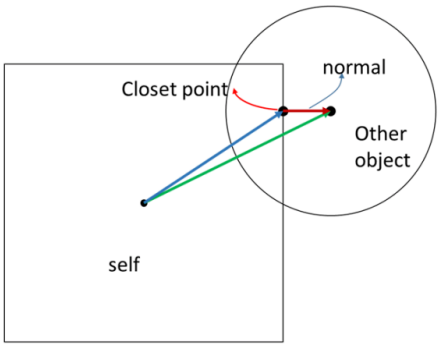
```
Manifold AABB::visitAABB(std::shared_ptr<const AABB> _shape) const
{
    float2 center_of_mass_vector = _shape->m_body->GetPosition() - m_body->GetPosition();
    float2 penetration_depth = (m_extent + _shape->m_extent) / 2 - linalg::abs(center_of_mass_vector);
    float2 nor;

    if (penetration_depth.x > penetration_depth.y) {
        nor = linalg::normalize(float2(0.0, center_of_mass_vector.y));
    }
    else {
        nor = linalg::normalize(float2(center_of_mass_vector.x, 0.0));
    }

    return Manifold(
        m_body,
        _shape->m_body,
        nor,
        (penetration_depth.x > penetration_depth.y) ? penetration_depth.y : penetration_depth.x,
        penetration_depth.x > 0 && penetration_depth.y > 0
    );

}
```

3. AABB to circle/ Circle to AABB:

Collision detection between AABB to circle or circle to AABB is more complicated than above situation, and it can split into two cases; In two cases, we need to calculate the minimum and maximum coordinate in AABB with respect to X-axis and Y-axis (i.e. the left lower and upper right corner), then we need to know whether the center of circle lies in AABB.

```
bool Is_Center_Inside_AABB(float2 c, float2 b_min, float2 b_max)
{
    if (c.x > b_min.x && c.x < b_max.x && c.y > b_min.y && c.y < b_max.y) return true;
    else return false;
}
```

Case1: The center of circle is outside the AABB



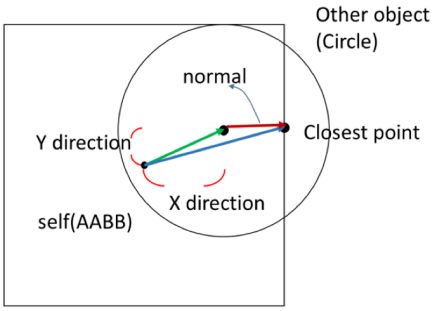
We can calculate the closest point in the AABB boundary with respect to center of circle.

```
float2 AABB_Toward_Outside_Closest_Point(float2 p, float2 b_min, float2 b_max)
{
    float2 q;
    for (int i = 0; i < 2; i++) {
        float v = p[i];
        if (v < b_min[i]) v = b_min[i];
        if (v > b_max[i]) v = b_max[i];
        q[i] = v;
    }

    return q;
}
```

Once we get the closest point with respect to the center of circle, we can calculate contact normal easily, and the $penetration\_depth$ is the radius of circle minus the distance between the center of circle and the closest point, and if the radius of circle bigger than the distance between the center of circle and the closest point, they collide.

```
float2 closest_point = AABB_Toward_Outside_Closest_Point(_b->m_body->GetPosition(), AABB_min, AABB_max);
float2 closest_point_to_centre = _b->m_body->GetPosition() - closest_point;

return Manifold(
    _a->m_body,
    _b->m_body,
    linalg::normalize(closest_point_to_centre),
    _b->m_radius - linalg::length(closest_point_to_centre),
    _b->m_radius > linalg::length(closest_point_to_centre)
);
```

Case2: The center of circle is inside the AABB



We can get the contact normal simply by calculating the vector of the center of circle toward AABB corner, and fetch its X-component or Y-component.

```
float2 Inside_Closest_Point_Toward_AABB(float2 p, float2 b_min, float2 b_max)
{
    float2 Centre_To_Corner = linalg::distance2(p, b_min) > linalg::distance2(p, b_max) ? b_max - p : b_min - p;
    float2 Centre_To_Closest_Point = linalg::abs(Centre_To_Corner.x) > linalg::abs(Centre_To_Corner.y) ?
        linalg::normalize(float2(0.0, Centre_To_Corner.y)) : linalg::normalize(float2(Centre_To_Corner.x, 0.0));
    return Centre_To_Closest_Point;
}
```

The $penetration\_depth$ is the radius of circle add the length of contact normal, and if the center of circle is in the AABB, they collide.

```
if (Is_Center_Inside_AABB(_b->m_body->GetPosition(), AABB_min, AABB_max)) {
    float2 contact_normal = Inside_Closest_Point_Toward_AABB(_b->m_body->GetPosition(), AABB_min, AABB_max);

    return Manifold(
        _a->m_body,
        _b->m_body,
        contact_normal,
        _b->m_radius + linalg::length(contact_normal),
        Is_Center_Inside_AABB(_b->m_body->GetPosition(), AABB_min, AABB_max)
    );
}
```

3.2 Resolve impulse

I resolve the impulse directly by the formula given by TAs, but the two points are worth mentioning, one is that if two objects don't collide with each other, we don't need to resolve them, and the other is that if the direction of relative velocity between two objects is the same as their contact normal, that is to say, two objects are separating now, we also don't need to resolve them.

```
void Manifold::Resolve() const
{
    if (!m_isHit) return;

    float e = std::min(m_body0->m_restitution, m_body1->m_restitution);
    float2 vr = m_body1->GetVelocity() - m_body0->GetVelocity();

    // Solve the same direction of relative velocity and normal
    if (linalg::dot(vr, m_normal) > 0) return;

    float2 I = (1 + e) * m_normal * linalg::dot(vr, m_normal) / (m_body0->GetInvMass() + m_body1->GetInvMass());

    m_body0->AddVelocity(I * m_body0->GetInvMass());
    m_body1->AddVelocity(-I * m_body1->GetInvMass());
}
```

3.3 Joint constraint

3.3.1 Spring joint

$$\Delta l = |\overrightarrow{x_a} - \overrightarrow{x_b}| - restLength\ , \quad \vec{l} = \frac{\overrightarrow{x_a} - \overrightarrow{x_b}}{|\overrightarrow{x_a} - \overrightarrow{x_b}|}$$

$$\overrightarrow{f_a} = -(k_s \Delta l)\vec{l}, \qquad \overrightarrow{f_b} = -\overrightarrow{f_a}$$

And the damper coefficient is elastic coefficient / 30

```
void SpringJoint::ApplyConstriant() const
{
    float center_of_mass_distance = linalg::distance(m_body0->GetPosition(), m_body1->GetPosition());
    float damper_coef = m_stiffness / 30;

    float2 xr = m_body0->GetPosition() - m_body1->GetPosition();
    float2 vr = m_body0->GetVelocity() - m_body1->GetVelocity();
    float2 l = xr / center_of_mass_distance;

    // Spring force
    m_body0->AddForce(-m_stiffness * (center_of_mass_distance - m_restLength) * l);
    m_body1->AddForce(-m_stiffness * (center_of_mass_distance - m_restLength) * -l);

    // Dameper force
    m_body0->AddForce(-damper_coef * linalg::dot(vr, xr) / center_of_mass_distance * l);
    m_body1->AddForce(-damper_coef * linalg::dot(vr, xr) / center_of_mass_distance * -l);

}
```

3.3.2 Distance joint

As long as eliminating the normal velocity and keep the distance between the other the same, we can guarantee the distance constraint being reached.

```
void DistanceJoint::ApplyConstriant() const
{
    float center_of_mass_distance = linalg::distance(m_body0->GetPosition(), m_body1->GetPosition());

    float2 normal = linalg::normalize(m_body1->GetPosition() - m_body0->GetPosition());

    float2 vr = m_body1->GetVelocity() - m_body0->GetVelocity();
    float vr_x = linalg::dot(vr, normal);
    float xr = center_of_mass_distance - m_restLength;
    float remove = vr_x + xr / m_deltaTime;

    float2 l = remove / (m_body0->GetInvMass() + m_body1->GetInvMass()) * normal;

    m_body0->AddVelocity(m_body0->GetInvMass() * l);
    m_body1->AddVelocity(-(m_body1->GetInvMass() * l));

}
```

3.4 Integrator

3.4.1 Explicit Euler Method

In explicit Euler method, we directly use current velocity multiply delta time to predict next position and current net force multiply inverse mass to predict next velocity.

$$\dot{y} = f(t_n, y_n)$$

$$y_{n+1} = y_n + hf(t_n, y_n)$$

```cpp
void ExplicitEulerIntegrator::Integrate(const std::vector<BodyRef>& _bodies, float deltaTime)
{
    const float2 gravity(0.0f, -9.8f);

    for (int i = 0; i < _bodies.size(); i++) {
        if (_bodies[i]->GetInvMass() == 0) {
            _bodies[i]->SetVelocity(float2(0.0, 0.0));
            continue;
        }
        _bodies[i]->AddForce(gravity);
        _bodies[i]->AddPosition(_bodies[i]->GetVelocity() * deltaTime);
        _bodies[i]->AddVelocity(_bodies[i]->GetForce() * _bodies[i]->GetInvMass() * deltaTime);
        _bodies[i]->SetForce(float2(0.0, 0.0));
    }
}
```

3.4.2 Runge Kutta Fourth (RK4)

Runge Kutta methods are a family of implicit and explicit iterative methods, which include the well-known routine called Euler method, used in temporal discretization for the approximate solutions of ordinary differential equations, and the most widely known member of the RK family is generally referred to as “RK4”.

$$\dot{y} = f(t_n, y_n)$$

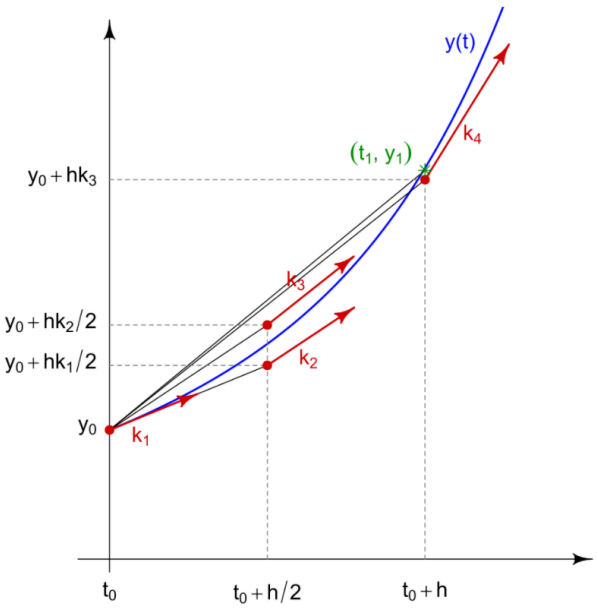
$$k1 = f(t_n, y_n),$$

$$k2 = f\left(t_n + \frac{h}{2}, y_n + h\frac{k1}{2}\right),$$

$$k3 = f\left(t_n + \frac{h}{2}, y_n + h\frac{k2}{2}\right),$$

$$k4 = f(t_n + h, y_n + hk3),$$

$$y_{n+1} = y_n + \frac{1}{6}h(k1 + 2k2 + 2k3 + k4)$$



```cpp
void RungeKuttaFourthIntegrator::Integrate(const std::vector<BodyRef>& _bodies, float deltaTime)
{
    if(scene == nullptr)
    {
        throw std::runtime_error("RungeKuttaFourthIntegrator has no target scene.");
    }

    const float2 gravity(0.0f, -9.8f);

    // The absolute value of position and velocity
    std::vector<StateStep> currentState(_bodies.size());

    // The delta value of each state
    std::vector<StateStep> deltaK1State(_bodies.size());
    std::vector<StateStep> deltaK2State(_bodies.size());
    std::vector<StateStep> deltaK3State(_bodies.size());
    std::vector<StateStep> deltaK4State(_bodies.size());

    for (int i = 0; i < _bodies.size(); i++) {
        if (_bodies[i]->GetInvMass() == 0) {
            _bodies[i]->SetVelocity(float2(0.0, 0.0));
            continue;
        }
        _bodies[i]->AddForce(gravity);

        // Explicit Euler Method
        deltaK1State[i].position = _bodies[i]->GetVelocity();
        deltaK1State[i].velocity = _bodies[i]->GetForce() * _bodies[i]->GetInvMass();

        // Compute K2
        deltaK2State[i].position = _bodies[i]->GetVelocity() + deltaK1State[i].velocity * deltaTime/ 2;
        deltaK2State[i].velocity = deltaK1State[i].velocity;

        // Compute K3
        deltaK3State[i].position = _bodies[i]->GetVelocity() + deltaK2State[i].velocity * deltaTime / 2;
        deltaK3State[i].velocity = deltaK1State[i].velocity;

        // Compute K4
        deltaK4State[i].position = _bodies[i]->GetVelocity() + deltaK3State[i].velocity * deltaTime;
        deltaK4State[i].velocity = deltaK1State[i].velocity;

        _bodies[i]->AddPosition((deltaK1State[i].position / 6 + deltaK2State[i].position / 3 +
            deltaK3State[i].position / 3 + deltaK4State[i].position / 6) * deltaTime);

        _bodies[i]->AddVelocity((deltaK1State[i].velocity / 6 + deltaK2State[i].velocity / 3 +
            deltaK3State[i].velocity / 3 + deltaK4State[i].velocity / 6) * deltaTime);

        _bodies[i]->SetForce(float2(0.0, 0.0));
    }

}
```

4 Conclusion

In this homework, I learn simple rigid body collision detection, impulse resolving, joint constraint, and integrator, it's a good experiment to learn the practice that has not been learned before.