Project: Massive Rigid-Body Simulation

NAME: LIU YIFEI
EMAIL: LIUYF7@SHANGHAITECH.EDU.CN
NAME: YAN YIHENG
EMAIL: YANYH1@SHANGHAITECH.EDU.CN

ACM Reference Format:
Name: Liu Yifei, email: liuyf7@shanghaitech.edu.cn, Name: Yan Yiheng, email: yanyh1@shanghaitech.edu.cn . 2023. Project: Massive Rigid-Body Simulation. 1, 1 (January 2023), 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Contents

1 INTRODUCTION

In this project, we implemented a simple and complete convex polygon physics engine based on Erin Catto's "Iterative Dynamics with Temporal Coherence" and other modern physics engines. In short, we implemented the main modules of mechanics calculation, collision detection, and constraint solving on top of the basic components of the previous homework, and managed to simulate the collision mechanics between objects and object environments with good results.

2 IMPLEMENTATION DETAILS

2.1 Engine pipeline

In the mass spring homework, we have implemented two parts to analyze the forces and update the velocity and position. The overall flow proposed by Erin Catto is shown in Figure

Author's address: Name: Liu Yifei
email: liuyf7@shanghaitech.edu.cn
Name: Yan Yiheng
email: yanyh1@shanghaitech.edu.cn.

XXXX-XXXX/2023/1-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

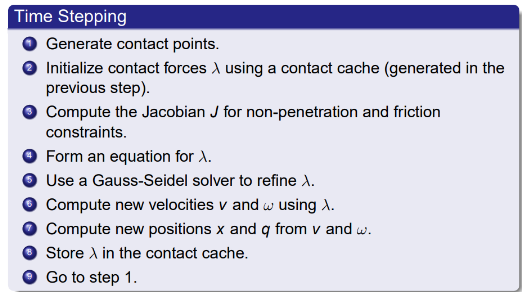

Fig. 1. Erin Catto's pipeline

2.2 Collision pair calculation

Imagine a scene with N objects, if we perform collision detection between every two of them, the computational complexity required is $O(N^2)$, which is obviously unacceptable for a computer. So we divide the collision detection into two phases, Broad-Phase and Narrow-Phase.
Broad-Phase uses some Bounding Volume to represent the collision information of rigid bodies, and then saves these Bounding Volumes in a spatial division, so that the rigid body pairs that may collide with each other can be selected in a short time.

2.2.1 Broad-Phase.

By using some kind of Bounding Volume to represent the collision information of rigid bodies, and then using spatial division to save these Bounding Volumes, it is possible to filter the rigid body pairs that may collide with each other in a short time. The usual methods are AABB (axis-aligned bounding boxes), OBB (oriented bounding boxes), Circle/Sphere.

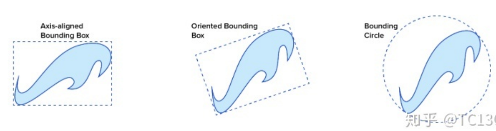

Fig. 2. Broad-Phase VH type

```
std::vector<std::pair<Collider*, Collider*>>&
    BroadPhase::ComputePairs()
{
colliderPairs.clear();
#pragma omp parallel for schedule(dynamic,1)
for (int iA = 0; iA < aabbs.size(); iA++)
```

```
{
    for (int iB = iA + 1; iB < aabbs.size(); iB++)
    {
        if (aabbs[iA]->collider->GetBody()->GetGroup() != 0)
        {
                if (aabbs[iA]->collider->GetBody()->GetGroup() == aabbs[iB]->collider->GetBody()->GetGroup())
                    continue;
        }

        if (aabbs[iA]->collider->GetBody()->GetInvMass() == 0.0f && aabbs[iB]->collider->GetBody()->GetInvMass() == 0.0f)
                continue;

        if (Overlap(aabbs[iA], aabbs[iB]))
                colliderPairs.push_back(std::make_pair(aabbs[iA]->collider, aabbs[iB]->collider));
    }
}

    return colliderPairs;
}

void BroadPhase::Update()
{
    glm::vec3 v[8];
    for (int i = 0; i < aabbsL.size(); i++)
    {
        AABB* aabbL = aabbsL[i];

        if (aabbL->collider->GetShape() == Collider::Sphere)
        {
                aabbs[i]->min = aabbL->min + aabbs[i]->collider->GetBody()->GetPosition();
                aabbs[i]->max = aabbL->max + aabbs[i]->collider->GetBody()->GetPosition();
                continue;
        }

        v[0] = glm::vec3(aabbL->min.x, aabbL->min.y, aabbL->min.z);
        v[1] = glm::vec3(aabbL->max.x, aabbL->min.y, aabbL->min.z);
        v[2] = glm::vec3(aabbL->max.x, aabbL->max.y, aabbL->min.z);
        v[3] = glm::vec3(aabbL->min.x, aabbL->max.y, aabbL->min.z);
        v[4] = glm::vec3(aabbL->min.x, aabbL->max.y, aabbL->max.z);
        v[5] = glm::vec3(aabbL->max.x, aabbL->max.y, aabbL->max.z);
        v[6] = glm::vec3(aabbL->max.x, aabbL->min.y, aabbL->max.z);
        v[7] = glm::vec3(aabbL->min.x, aabbL->min.y, aabbL->max.z);

        for (int i = 0; i < 8; i++)
        {
                v[i] = aabbL->collider->GetBody()->LocalToGlobalPoint(v[i]);
        }

        AABB* aabb = aabbs[i];
        aabb->min = glm::vec3(FLT_MAX);
        aabb->max = glm::vec3(-FLT_MAX);
        for (int i = 0; i < 8; i++)
        {
                aabb->min.x = glm::min(aabb->min.x, v[i].x);
                aabb->min.y = glm::min(aabb->min.y, v[i].y);
                aabb->min.z = glm::min(aabb->min.z, v[i].z);
                aabb->max.x = glm::max(aabb->max.x, v[i].x);
                aabb->max.y = glm::max(aabb->max.y, v[i].y);
                aabb->max.z = glm::max(aabb->max.z, v[i].z);
        }
    }
}
```

2.2.2 *Narrow-Phase.*

A geometry is said to be convex if any two points of the line within the geometry must fall within the geometry. Almost all collision algorithms are based on convex geometry by default. For a concave geometry, a convex geometry can be generated using the QuickHull algorithm, or the concave geometry can be decomposed into several convex geometries using similar algorithms such as V-HACD. In general, convex geometry can satisfy most of our needs. For oddly shaped objects, it is possible to approximate them with some regular collision shapes.

In the physics engine will use a variety of collision body shape, some shapes of the collision calculation is very intuitive and simple, such as between two spheres, determine whether the distance between the two centers of the circle is greater than the sum of the radius, you can directly calculate whether the collision. Separating Axis Theorem (SAT) principle: two convex polygons do not intersect when and only when there must be a straight line, the projection of two convex polygons on this line does not intersect. Alternatively, it can be described as follows: if two convex polygons intersect, the projections on all lines are intersecting.

```
void DetectHullvsHull(std::vector<Manifold>& manifolds, HullCollider* A, HullCollider* B)
{
    FaceQuery faceQueryA;
    if (!QueryFaceAxes(faceQueryA, A, B))
            return;
    assert(faceQueryA.separation <= 0.0f);

    FaceQuery faceQueryB;
    if (!QueryFaceAxes(faceQueryB, B, A))
            return;
    assert(faceQueryB.separation <= 0.0f);

    EdgeQuery edgeQuery;
    if (!QueryEdgeAxes(edgeQuery, A, B))
            return;
    assert(edgeQuery.separation <= 0.0f);

    float maxFaceSep = faceQueryA.separation > faceQueryB.separation ? faceQueryA.separation : faceQueryB.separation;
    float epsilon = 0.05f;

    if (edgeQuery.separation > maxFaceSep + epsilon)
    {
        // edge contact
        CreateEdgeContact(manifolds, A, B, edgeQuery);
    }
    else
    {
        // face contact
        int incidentFace;
        if (faceQueryA.separation > faceQueryB.separation + epsilon)
        {
                incidentFace = FindIncidentFace(B, A, faceQueryA.faceIndex);
                CreateFaceContact(manifolds, B, A, incidentFace, faceQueryA.faceIndex);
        }
        else
        {
                incidentFace = FindIncidentFace(A, B, faceQueryB.faceIndex);
                CreateFaceContact(manifolds, A, B, incidentFace, faceQueryB.faceIndex);
        }
    }
}

void DetectSphereWithSphere(std::vector<Manifold>& manifolds, SphereCollider* A, SphereCollider* B)
{
    glm::vec3 CA = A->GetBody()->LocalToGlobalPoint(A->GetCentroid());
    glm::vec3 CB = B->GetBody()->LocalToGlobalPoint(B->GetCentroid());

    glm::vec3 normal = CB - CA;
    float dist2 = glm::l2Norm(normal);
    float ra = A->GetRadius();
    float rb = B->GetRadius();
    float rSum = ra + rb;

    if (dist2 > rSum)
            return;

    normal = glm::normalize(normal);


    glm::vec3 PA = CA + A->GetRadius() * normal;
    glm::vec3 PB = CB - B->GetRadius() * normal;
    glm::vec3 C = (PA + PB) * 0.5f;
    float penetration = rSum - dist2;

    Manifold m;
    m.contacts.push_back(std::move(Contact(A->GetBody(), B->GetBody(), C, normal, penetration)));
```

```
    manifolds.push_back(m);
}
/// <summary>
/// Check whether A sphere is collided with a Hull(OK)
/// </summary>
/// <param name="manifolds"></param>
/// <param name="A"></param>
/// <param name="B"></param>
void DetectSphereWithHull(std::vector<Manifold>& manifolds, SphereCollider* A, HullCollider* B)
{
    glm::vec3 center = A->GetBody()->GetPosition();          // sphere center

    if (QueryPoint(B, center))
    {
        float maxSeparation = -FLT_MAX;
        float separation = 0.0;
        int bestFitFace = -1;          // should be max separation

        for (int i = 0; i < B->GetFaceCount(); i++)
        {
            glm::vec3 normal = B->GetBody()->LocalToGlobalVec(B->GetFace(i)->normal);
            glm::vec3 origin = B->GetBody()->LocalToGlobalPoint(B->GetFace(i)->edge->tail->position);

            separation = glm::dot(origin - center, normal);
            if (separation > maxSeparation)
            {
                    maxSeparation = separation;
                    bestFitFace = i;
            }
        }

        glm::vec3 normal = B->GetBody()->LocalToGlobalVec(B->GetFace(bestFitFace)->normal);
        glm::vec3 contact = center - maxSeparation * normal;

        Manifold m;
        m.contacts.push_back(std::move(Contact(A->GetBody(), B->GetBody(), contact, -normal, -maxSeparation)));
        manifolds.push_back(m);
    }
    else
    {
        glm::vec3 normal(0);
        glm::vec3 point(0);
        HalfSpace plane;
        float distance = 0.0f;
        float minDistance = FLT_MAX;
        int bestFace = -1;

        for (int i = 0; i < B->GetFaceCount(); i++)
        {
            normal = B->GetBody()->LocalToGlobalVec(B->GetFace(i)->normal);
            point = B->GetBody()->LocalToGlobalPoint(B->GetFace(i)->edge->tail->position);
            plane = HalfSpace(normal, point);
            distance = plane.Distance(center);

            if (distance > 0.0f && distance < minDistance)
            {
                    minDistance = distance;
                    bestFace = i;
            }
        }

        if (minDistance > A->GetRadius())
                return;

        // intersect with closest face plane
        glm::vec3 contact;
        float penetration;
        normal = B->GetBody()->LocalToGlobalVec(B->GetFace(bestFace)->normal);
        contact = center - minDistance * normal;

        std::vector<glm::vec3> verts;
        HEdge* e = B->GetFace(bestFace)->edge;
```

```
        do {
            verts.push_back(B->GetBody()->LocalToGlobalPoint(e->tail->position));
            e = e->next;
        } while (e != B->GetFace(bestFace)->edge);

        if (QueryPoint(contact, verts, normal))
        {
            penetration = A->GetRadius() - minDistance;
            Manifold m;
            Contact c(A->GetBody(), B->GetBody(), contact, -normal, penetration);
            m.contacts.push_back(c);
            manifolds.push_back(m);
            return;
        }
        return;
        // intersect with edges of closest face
        e = B->GetFace(bestFace)->edge;
        glm::vec3 pA, pB;
        do {
            pA = B->GetBody()->LocalToGlobalPoint(e->tail->position);
            pB = B->GetBody()->LocalToGlobalPoint(e->twin->tail->position);
            if (IntersectSegmentSphere(pA, pB, A, contact))
            {
                normal = contact - center;
                float l = glm::length(normal);
                normal /= l;
                penetration = A->GetRadius() - l;
                Manifold m;
                Contact c(A->GetBody(), B->GetBody(), contact, normal, penetration);
                m.contacts.push_back(c);
                manifolds.push_back(m);
                return;
            }
            e = e->next;
        } while (e != B->GetFace(bestFace)->edge);
    }
}
void DetectCollision(std::vector<Manifold>& manifolds, Collider* A, Collider* B)
{
        if (A->GetShape() == Collider::Sphere && B->GetShape() == Collider::Sphere)
        {
                DetectSphereWithSphere(manifolds, static_cast<SphereCollider*>(A), static_cast<SphereCollider*>(B));
        }
        else if (A->GetShape() == Collider::Hull && B->GetShape() == Collider::Hull)
        {
                DetectHullvsHull(manifolds, static_cast<HullCollider*>(A), static_cast<HullCollider*>(B));
        }
        else if (A->GetShape() == Collider::Hull && B->GetShape() == Collider::Sphere)
        {
                DetectSphereWithHull(manifolds, static_cast<SphereCollider*>(B), static_cast<HullCollider*>(A));
        }
        else if (A->GetShape() == Collider::Sphere && B->GetShape() == Collider::Hull)
        {
                DetectSphereWithHull(manifolds, static_cast<SphereCollider*>(A), static_cast<HullCollider*>(B));
        }
        else {
                assert(false);
        }
```

```
}
```

2.3 Constraints

. Constraints are very important in physics engines, just like shader is the core of achieving various rendering effects, Constraints are the core concept in physics engines. In some real-world physics constraint solving, such as robot motion, is very complex and time consuming. In games, due to frame rate requirements, we do not have enough time to get absolutely correct results, and we aim for a reasonably accurate and stable solution.

For example, if a box is placed on a fixed plane, we solve for the P point in the lower left corner of the box. The constraint of the contact constraint is that the contact points of the two objects do not move along the normal direction of the contact. This gives us an easy representation of the position constraint C. It is not enough to have only the position constraint, we derive C to obtain a formula for the velocity constraint:

$$\dot{C} = (\dot{\boldsymbol{p}}_B - \dot{\boldsymbol{p}}_A) \cdot \boldsymbol{n} + (\boldsymbol{p}_B - \boldsymbol{p}_A) \cdot \dot{\boldsymbol{n}}$$

Since we assume that the ground is fixed, we have $\dot{\boldsymbol{p}}_A = 0$, and $\boldsymbol{p}_A$ and $\boldsymbol{p}_B$ are actually equal, so we simplify the equation to get: $\dot{C} = \dot{\boldsymbol{p}}_B \cdot \boldsymbol{n}$

$$\dot{\boldsymbol{p}}_B = \boldsymbol{v}_B + \boldsymbol{\omega}_B \times \boldsymbol{r}$$

$$\dot{C} = (\boldsymbol{v}_B + \boldsymbol{\omega}_B \times \boldsymbol{r}) \cdot \boldsymbol{n}$$

$$\begin{aligned}\dot{C} &= \vec{n}(v'_{p_a} - v'_{p_b}) \\ &= \vec{n}\,(v_a - v_b) \ + \lambda_n \left(\frac{1}{m_a} + \frac{1}{m_b}\right) + \vec{n}\,(w_a \times r_a \ - \ w_b \times r_b) \ + \\ &\lambda_n \left(\frac{\vec{n}\,(r_a \times \vec{n}) \times r_a}{I_a} + \frac{\vec{n}\,(r_b \times \vec{n}) \times r_b}{I_b}\right) \\ &= \vec{n}(v_{p_a} - v_{p_b}) + \lambda_n \left(\frac{1}{m_a} + \frac{1}{m_b} + \frac{\vec{n}\,(r_a \times \vec{n}) \times r_a}{I_a} + \frac{\vec{n}\,(r_b \times \vec{n}) \times r_b}{I_b}\right) \\ &= 0\end{aligned}$$

$$\lambda_n = \frac{-\,\vec{n}\,(v_{p_a} - v_{p_b})}{\frac{1}{m_a} + \frac{1}{m_b} + \frac{\vec{n}(r_a \times \vec{n}) \times r_a}{I_a} + \frac{\vec{n}(r_b \times \vec{n}) \times r_b}{I_b}}$$

```
#define BAUMGARTE 0.2f
#define VELOCITYSLOP 0.5f
#define POSITIONSLOP 0.001f

Contact::Contact(Body* A, Body* B, const glm::vec3& position, const glm::vec3& normal, const float penetration)
        : A(A), B(B), position(position), normal(normal), penetration(penetration), impulseSumN(0.0f)
{
    ComputeBasis(normal, tangent[0], tangent[1]);

    rA = position - A->GetCentroid();
    rB = position - B->GetCentroid();

    CalculateJacobian(JN, normal);
    CalculateBias();

    for (int i = 0; i < 2; i++)
    {
            impulseSumT[i] = 0.0f;
            CalculateJacobian(JT[i], tangent[i]);
            kt[i] = CalculateEffectiveMass(JT[i], A, B);
    }
}
//Normal constraint(OK)
float Contact::CalculateNormalConstraint() const
{
        glm::vec3 vA = A->GetVelocity() + glm::cross(A->GetAngularVelocity(), rA);
        glm::vec3 vB = B->GetVelocity() + glm::cross(B->GetAngularVelocity(), rB);
        return glm::dot((vB - vA), normal);
}
//3-Compute the Jacobian J for non-penetration and friction constraints(OK)
void Contact::CalculateJacobian(Jacobian& J, const glm::vec3& axis)
{
        J = Jacobian(-axis, -glm::cross(rA, axis), axis, glm::cross(rB, axis));
}
//Slop bias(OK)
void Contact::CalculateBias()
{
        // restitution
        float e = (A->GetRestitution() + B->GetRestitution()) * 0.5f;
        float vSep = CalculateNormalConstraint();
        //bias = e * vSep;
        bias = e * (glm::min(vSep - 0.5f, 0.0f));
}

void Contact::SolveVelocities(Velocity& vA, Velocity& vB)
{
        if (CalculateNormalConstraint() >= 0)
```

```
        {
            return;
        }

        float lambda, oldImpulse;
        float uf = (A->GetFriction() + B->GetFriction()) * 0.5f;
        float maxFriction;

        lambda = CalculateLagrangian(JN, vA, vB, kn, bias);

        oldImpulse = impulseSumN;
        impulseSumN = glm::max(0.0f, impulseSumN + lambda);
        lambda = impulseSumN - oldImpulse;

        vA.v += lambda * JN.L1 * A->GetInvMass();
        vA.w += lambda * JN.A1 * A->GetInvInertia();
        vB.v += lambda * JN.L2 * B->GetInvMass();
        vB.w += lambda * JN.A2 * B->GetInvInertia();

        for (int i = 0; i < 2; i++)
        {
            lambda = CalculateLagrangian(JT[i], vA, vB, kt[i], 0.0f);

            oldImpulse = impulseSumT[i];
            maxFriction = uf * impulseSumN; //uf * kt[i] * 9.8f;
            impulseSumT[i] = glm::clamp(impulseSumT[i] + lambda, -maxFriction, maxFriction);
            lambda = impulseSumT[i] - oldImpulse;

            vA.v += lambda * JT[i].L1 * A->GetInvMass();
            vA.w += lambda * JT[i].A1 * A->GetInvInertia();
            vB.v += lambda * JT[i].L2 * B->GetInvMass();
            vB.w += lambda * JT[i].A2 * B->GetInvInertia();
        }
}

void Contact::SolvePositions(Position& pA, Position& pB)
{
        float K = CalculateEffectiveMass(JN, A, B);
        float C = -BAUMGARTE * (glm::max(penetration - POSITIONSLOP, 0.0f));
        float lambda = K > 0.0f ? -C / K : 0.0f;
        glm::vec3 P = lambda * normal;

        pA.c -= P * A->GetInvMass();
        pA.q += 0.5f * glm::quat(0.0f, A->GetInvInertia() * glm::cross(rA, -P)) * pA.q;
        pA.q = glm::normalize(pA.q);
        pB.c += P * B->GetInvMass();
        pB.q += 0.5f * glm::quat(0.0f, B->GetInvInertia() * glm::cross(rB, P)) * pA.q;
        pB.q = glm::normalize(pB.q);
}

void Manifold::SolveVelocities()
{
    Body* A = contacts[0].A;
    Body* B = contacts[0].B;

    if (A->GetInvMass() != 0 && B->GetInvMass() != 0)
    {
        if (!A->IsAwake() || !B->IsAwake())
        {
                B->SetAwake(true);
                A->SetAwake(true);
        }
    }

    glm::vec3 v1 = A->GetVelocity();
    glm::vec3 w1 = A->GetAngularVelocity();
    glm::vec3 v2 = B->GetVelocity();
    glm::vec3 w2 = B->GetAngularVelocity();
    Velocity vA(v1, w1);
    Velocity vB(v2, w2);

    for (int i = 0; i < contacts.size(); i++)
    {
        contacts[i].SolveVelocities(vA, vB);
    }
```

```
    A->SetVelocity(vA.v);
    A->SetAngularVelocity(vA.w);
    B->SetVelocity(vB.v);
    B->SetAngularVelocity(vB.w);
}

void Manifold::SolvePositions()
{
    Body* A = contacts[0].A;
    Body* B = contacts[0].B;
    glm::vec3 c1 = A->GetCentroid();
    glm::quat q1 = A->GetOrientation();
    glm::vec3 c2 = B->GetCentroid();
    glm::quat q2 = B->GetOrientation();
    Position pA(c1, q1);
    Position pB(c2, q2);

    for (int i = 0; i < contacts.size(); i++)
    {
            contacts[i].SolvePositions(pA, pB);
    }

    A->SetCentroid(pA.c);
    A->SetOrientation(pA.q);
    B->SetCentroid(pB.c);
    B->SetOrientation(pB.q);
}
```

3 RESULTS

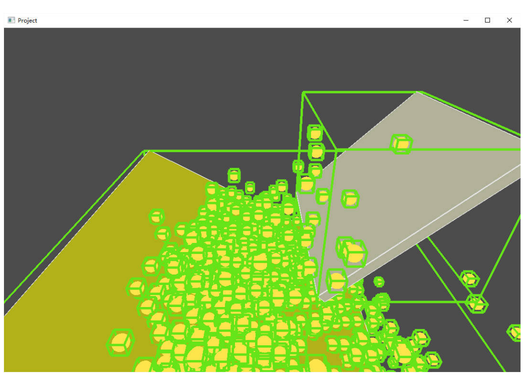

Fig. 3. Massive Balls simulation

Fig. 4. Massive Boxes simulation

3.1 Division of labor

In this project, Liu implemented the physical mechanics simulation system and wrote the report, while Yan implemented the basic environment, including camera, geometry, etc., and created the PPT.