CSCI 522 Milestone 2: Physics Implementation – Rigid Body Dynamics (Cubic)

In Milestone 1, I successfully built a foundational physics engine framework that enabled physical objects to interact and respond to collisions. However, several issues emerged that impacted the functionality and realism of the simulation:

Collision Detection: The system relied on SAT algorithms, which calculated only a single collision point and did not accurately generate collision normal vectors.

Collision Resolution: The resolution phase was tightly coupled with the detection phase, complicating debugging and the addition of more advanced logic.

Accuracy: The previous approach lacked precision. It struggled to handle complex collisions and lacked mechanisms for balancing system properties like speed, position, and friction.

Rotation Updates: The method for updating rotations was incorrect, leading to unrealistic behavior.

These issues culminated in the flawed demo video I submitted previously.

For this milestone, I designed a new framework that addresses these limitations. The core improvement is a revised Tick function in the physics engine, which divides the simulation into distinct stages. This allows each object to update its state in a synchronized manner, ensuring that the engine operates on the most up-to-date information for all objects.

Additionally, I integrated the GJK and EPA algorithms to improve collision detection. These algorithms enable accurate computation of collision normals, points of contact, penetration depths, and other essential parameters, leading to more realistic simulation results. The system now also supports adjustable object properties, such as mass, friction, and restitution, allowing for the simulation of different materials.

Remaining Challenges

Despite these improvements, several challenges remain:

Stability Issues: Since the engine maintains only one collision point per object per frame, stabilizing a box resting on a flat surface is problematic.

Disappearing Objects: There is a persistent bug causing boxes to disappear, which I am actively investigating.

Collision Detection

Add GJK/EPA algorithms

```cpp
bool CollisionDetector::Penetration(PhysicsShape* Shape1, PhysicsShape* Shape2, Vector3& guess, sResults& result)
{
    MinkowskiDiff shape;
    InitializeMinkowskiDiff(Shape1, Shape2, result, shape);

    // gjk
    GJK gjk;
    GJK::eStatus::_ gjk_status = gjk.Evaluate(shape, guess * -1);

    switch (gjk_status)
    {

    case GJK::eStatus::Inside:
    {
        EPA epa;
        EPA::eStatus::_ epa_status = epa.Evaluate(gjk, guess * -1);
        if (epa_status != EPA::eStatus::Failed)
        {
            Vector3 w0 = Vector3(0, 0, 0);
            for (U i = 0; i < epa.m_result.rank; ++i)
            {
                // w0 是 物体1在物体2的最深穿透点在世界坐标下的坐标
                // Support 返回的是全局坐标系的点，所以可以判断w0是全局坐标系下的点。
                w0 = w0 + shape.Support(epa.m_result.c[i]->d, 1) * epa.m_result.p[i];

            }
            Matrix4x4 wtrs1 = Shape1->m_worldTransform.inverse();
            result.status = sResults::Penetrating;
            result.witnessesInFirstLocal[0] = wtrs1 * w0;                          // 物体1在物体2中的最深穿透点在物体1
            Vector3 secondObjectPointInFirstObject = w0 - epa.m_normal * epa.m_depth;
            result.witnessesInFirstLocal[1] = wtrs1 * secondObjectPointInFirstObject;   // 物体2在物体1中的最深穿透点在物体1
            result.witnessInGlobal[0] = w0;
            result.witnessInGlobal[1] = secondObjectPointInFirstObject;
            result.normal = epa.m_normal;   // 全局坐标下，由物体1指向物体2
            result.distance = epa.m_depth;  // 距离为正数
            return true;
        }
        else
        {
            result.status = sResults::EPA_Failed;
        }
```

GJK

```
eStatus::_ Evaluate(const tShape& shapearg, const Vector3& guess)
{
    U iterations = 0;
    float sqdist = 0;
    float alpha = 0;
    Vector3 lastw[4];
    U clastw = 0;

    /* Initialize solver*/
    m_free[0] = &m_store[0];
    m_free[1] = &m_store[1];
    m_free[2] = &m_store[2];
    m_free[3] = &m_store[3];
    m_nfree = 4;
    m_current = 0;
    m_status = eStatus::valid;
    m_shape = shapearg;
    m_distance = 0;

    //Initialize simplex
    m_simplices[0].rank = 0;
    m_ray = guess;
    const float sqrl = m_ray.lengthSqr();
    appendvertice(m_simplices[0], sqrl > 0 ? m_ray * -1 : Vector3(1, 0, 0));
    m_simplices[0].p[0] = 1;
    m_ray = m_simplices[0].c[0]->w;
    sqdist = sqrl;
    lastw[0] =
        lastw[1] =
        lastw[2] =
        lastw[3] = m_ray;

    /* Loop                          */
    do
    {
        const U next = 1 - m_current;
        sSimplex& cs = m_simplices[m_current];
        sSimplex& ns = m_simplices[next];

        //Check zero
        const float rl = m_ray.length();
        if (rl < GJK_MIN_DISTANCE)
```

```
/* Loop                        */
do
{
    const U next = 1 - m_current;
    sSimplex& cs = m_simplices[m_current];
    sSimplex& ns = m_simplices[next];

    //Check zero
    const float rl = m_ray.length();
    if (rl < GJK_MIN_DISTANCE)
    {
        // Touching or inside
        m_status = eStatus::Inside;
        break;
    }

    //Append new vertice in -'v' direction
    appendvertice(cs, m_ray * -1);
    const Vector3& w = cs.c[cs.rank - 1]->w;
    bool found = false;
    for (U i = 0; i < 4; i++)
    {
        Vector3 diff = w - lastw[i];
        if (diff.lengthSqr() < GJK_DUPLICATED_EPS)
        {
            found = true;
            break;
        }
    }
    if (found)
    {
        /* Return old simplex                */
        removevertice(m_simplices[m_current]);
        break;
    }
    else
    {
        /* Update lastw                  */
        lastw[clastw = (clastw + 1) & 3] = w;
    }
    /* Check for termination                */
    const float omega = m_ray.dotProduct(w) / rl;
    alpha = omega > alpha ? omega : alpha;
```

EPA

```
eStatus::_ Evaluate(GJK& gjk, Vector3& guess)
{
    GJK::sSimplex& simplex = *gjk.m_simplex;
    if ((simplex.rank > 1) && gjk.EncloseOrigin())
    {
        /* Clean up              */
        while (m_hull.root)
        {
            sFace* f = m_hull.root;
            remove(m_hull, f);
            append(m_stock, f);
        }
        m_status = eStatus::Valid;
        m_nextsv = 0;
        /* Orient simplex        */
        if (gjk.det(simplex.c[0]->w - simplex.c[3]->w,
            simplex.c[1]->w - simplex.c[3]->w,
            simplex.c[2]->w - simplex.c[3]->w) < 0)
        {
            btSwap(simplex.c[0], simplex.c[1]);
            btSwap(simplex.p[0], simplex.p[1]);
        }
        /* Build initial hull    */
        sFace* tetra[] = { newface(simplex.c[0], simplex.c[1], simplex.c[2], true),
                           newface(simplex.c[1], simplex.c[0], simplex.c[3], true),
                           newface(simplex.c[2], simplex.c[1], simplex.c[3], true),
                           newface(simplex.c[0], simplex.c[2], simplex.c[3], true) };
        if (m_hull.count == 4)
        {
            sFace* best = findbest();
            sFace outer = *best;
            U pass = 0;
            U iterations = 0;
            bind(tetra[0], 0, tetra[1], 0);
            bind(tetra[0], 1, tetra[2], 0);
            bind(tetra[0], 2, tetra[3], 0);
            bind(tetra[1], 1, tetra[3], 2);
            bind(tetra[1], 2, tetra[2], 1);
            bind(tetra[2], 2, tetra[3], 1);
            m_status = eStatus::Valid;

            for (; iterations < EPA_MAX_ITERATIONS; ++iterations)
```

```
for (; iterations < EPA_MAX_ITERATIONS; ++iterations)
{
    if (m_nextsv < EPA_MAX_VERTICES)
    {
        sHorizon horizon;
        sSV* w = &m_sv_store[m_nextsv++];
        bool valid = true;
        best->pass = (U1)(++pass);
        gjk.getsupport(best->n, *w);
        const float wdist = best->n.dotProduct(w->w) - best->d;
        if (wdist > EPA_ACCURACY)
        {
            for (U j = 0; (j < 3) && valid; ++j)
            {
                valid &= expand(pass, w,
                    best->f[j], best->e[j],
                    horizon);
            }
            if (valid && (horizon.nf >= 3))
            {
                bind(horizon.cf, 1, horizon.ff, 2);
                remove(m_hull, best);
                append(m_stock, best);
                best = findbest();
                outer = *best;
            }
            else
            {
                m_status = eStatus::InvalidHull;
                break;
            }
        }
        else
        {
            m_status = eStatus::AccuraryReached;
            break;
        }
    }
    else
    {
        m_status = eStatus::OutOfVertices;
        break;
    }
```

Support Function

```
Vector3 Box::GetSupport(Vector3& dir)
{
    float maxDot = -std::numeric_limits<float>::infinity();

    Vector3 supportPoint;


    for (int i = 0; i < 8; ++i) {

        float dotProduct = TransformedCorners[i].dotProduct(dir);

        if (dotProduct > maxDot) {
            maxDot = dotProduct;
            supportPoint = TransformedCorners[i];
        }
    }

    return supportPoint;
}
```

MinkowskiDiff

```
struct MinkowskiDiff
{
    PhysicsShape* box1;
    PhysicsShape* box2;

    inline Vector3 Support1(Vector3& dir);
    inline Vector3 Support2(Vector3& dir);
    Vector3 Support(Vector3& dir);
    Vector3 Support(Vector3& dir, int idx);
};
```

ContactPoint

```
struct ContactPoint {

    ContactPoint(void)
        : normalImpulseSum(0.0f)
        , tangentImpulseSum1(0.0f)
        , tangentImpulseSum2(0.0f)
        , m_jN(JacobianType::Normal)
        , m_jT(JacobianType::Tangent)
        , m_jB(JacobianType::Tangent)
    {
    }

    // contact point data
    Vector3 globalPositionA;    // Penetration point of object A in global coordinate
    Vector3 globalPositionB;    // Penetration point of object B in global coordinate
    Vector3 localPositionA; // Penetration point of object A in self coordinate
    Vector3 localPositionB; // Penetration point of object B in self coordinate

    // these 3 vectors form an orthonormal basis
    Vector3 normal; // Penetration normal vector
    Vector3 tangent1, tangent2; // two different tangent vectors
    Vector3 rA; // Penetration vector of A
    Vector3 rB; // Penetration vector of B

    // penetration depth
    float penetrationDistance;  // Penetration depth

    // for clamping (more on this later)
    float normalImpulseSum;
    float tangentImpulseSum1;
    float tangentImpulseSum2;

    Jacobian m_jN;
    Jacobian m_jT;
    Jacobian m_jB;
};
```

Collision Resolve

```
void PhysicsManager::Resolve(std::vector<std::shared_ptr<ContactManifold>>& manifolds, float deltaTime)
{

    for each (std::shared_ptr<ContactManifold> manifold in manifolds)
    {
        for (int i = 0; i < manifold->contactPointCount; i++)
        {
            InitContactConstranst(manifold, i, deltaTime);
        }
    }

    for each (std::shared_ptr<ContactManifold> manifold in manifolds)
    {
        for (int i = 0; i < manifold->contactPointCount; i++)
        {
            SolveContactConstranst(manifold, i, deltaTime);
        }
    }
}

void PhysicsManager::InitContactConstranst(std::shared_ptr<ContactManifold> manifold, int idx, float deltaTime)
{
    manifold->contactPoints[idx].m_jN.Init(manifold, idx, JacobianType::Normal, manifold->contactPoints[idx].normal
    manifold->contactPoints[idx].m_jT.Init(manifold, idx, JacobianType::Tangent, manifold->contactPoints[idx].tange
    manifold->contactPoints[idx].m_jB.Init(manifold, idx, JacobianType::Tangent, manifold->contactPoints[idx].tange
}

void PhysicsManager::SolveContactConstranst(std::shared_ptr<ContactManifold> manifold, int idx, float deltaTime)
{
    manifold->contactPoints[idx].m_jN.Solve(manifold, idx, manifold->contactPoints[idx].normal, deltaTime);
    manifold->contactPoints[idx].m_jT.Solve(manifold, idx, manifold->contactPoints[idx].tangent1, deltaTime);
    manifold->contactPoints[idx].m_jB.Solve(manifold, idx, manifold->contactPoints[idx].tangent2, deltaTime);
}
```

```
void Jacobian::Init(std::shared_ptr<ContactManifold> manifold, int idx, JacobianType jt, Vector3 dir, float dt)
{
    jacobinType = jt;

    m_jva = dir * -1;
    m_jwa = manifold->contactPoints[idx].rA.crossProduct(dir) * -1;
    m_jvb = dir;
    m_jwb = manifold->contactPoints[idx].rB.crossProduct(dir);

    m_bias = 0.0f;

    if (jacobinType == JacobianType::Normal)
    {
        float betaA = manifold->colliderA->GetContactBeta();
        float betaB = manifold->colliderB->GetContactBeta();
        float beta = betaA * betaB;

        float restitutionA = manifold->colliderA->GetRestitution();
        float restitutionB = manifold->colliderB->GetRestitution();
        float restitution = restitutionA * restitutionB;

        restitution = 1;

        Vector3 va = manifold->colliderA->GetVelocity();
        Vector3 wa = manifold->colliderA->GetAngularVelocity();
        Vector3 ra = manifold->contactPoints[idx].rA;

        Vector3 vb = manifold->colliderB->GetVelocity();
        Vector3 wb = manifold->colliderB->GetAngularVelocity();
        Vector3 rb = manifold->contactPoints[idx].rB;

        m_bias = 0;
        if (jacobinType == JacobianType::Normal)
        {
            // See
            // http://allenchou.net/2013/12/game-physics-resolution-contact-constraints/
            Vector3 relativeVelocity = vb + wb.crossProduct(rb) - va - wa.crossProduct(ra);
            float closingVelocity = relativeVelocity.dotProduct(dir);
            m_bias = -(beta / dt) * manifold->contactPoints[idx].penetrationDistance + restitution * closingVelocity;
        }
    }
```

```
// http://allenchou.net/2013/12/game-physics-constraints-sequential-impulse/
// https://www.youtube.com/watch?v=pmdYzNF9x34
// effectiveMass

auto rigidA = manifold->colliderA;
auto rigidB = manifold->colliderB;

Vector3 rva = rigidA->GetInverInertiaLocal() * m_jwa;
Vector3 rvb = rigidB->GetInverInertiaLocal() * m_jwb;

float k =
    rigidA->GetInverseMass() + rigidB->GetInverseMass()
    + m_jwa.dotProduct(rva)
    + m_jwb.dotProduct(rvb);

m_effectiveMass = 1 / k;
m_totalLambda = 0;
```

Solve Funtion

```
void Jacobian::Solve(std::shared_ptr<ContactManifold> manifold, int idx, Vector3 dir, float dt)
{
    ContactPoint& point = manifold->contactPoints[idx];

    float jv = m_jva.dotProduct(manifold->colliderA->GetVelocity())
        + m_jwa.dotProduct(manifold->colliderA->GetAngularVelocity())
        + m_jvb.dotProduct(manifold->colliderB->GetVelocity())
        + m_jwb.dotProduct(manifold->colliderB->GetAngularVelocity());

    float lambda = m_effectiveMass * (-(jv + m_bias));
    float oldTotalLambda = m_totalLambda;
    switch (jacobinType)
    {
    case JacobianType::Normal:
        m_totalLambda = std::max(m_totalLambda + lambda, 0.0f);
        break;

    case JacobianType::Tangent:
        float friction = manifold->colliderA->GetFriction() * manifold->colliderB->GetFriction();
        float maxFriction = friction * point.m_jN.m_totalLambda;
        m_totalLambda = Clamp(m_totalLambda + lambda, -maxFriction, maxFriction);
        break;
    }
    lambda = m_totalLambda - oldTotalLambda;

    Vector3 va = manifold->colliderA->GetVelocity();
    Vector3 vadelta = m_jva * manifold->colliderA->GetInverseMass() * lambda;
    manifold->colliderA->SetVelocity(va + vadelta);

    Vector3 wa = manifold->colliderA->GetAngularVelocity();
    Vector3 wadelta = (manifold->colliderA->GetInverseInertiaTensorWorld() * m_jwa) * lambda;
    manifold->colliderA->SetAngularVelocity(wa + wadelta);

    Vector3 vb = manifold->colliderB->GetVelocity();
    Vector3 vbdelta = m_jvb * manifold->colliderB->GetInverseMass() * lambda;
    manifold->colliderB->SetVelocity(vb + vbdelta);

    Vector3 wb = manifold->colliderB->GetAngularVelocity();
    Vector3 wbdelta = (manifold->colliderB->GetInverseInertiaTensorWorld() * m_jwb) * lambda;
    manifold->colliderB->SetAngularVelocity(wb + wbdelta);
}
```

Integrate Stage

```
void PhysicsShape::do_PHYSICS_START(Events::Event* pEvt)
{
    if (!EnablePhysics || !IsDynamic)
    {
        SetVelocity(Vector3(0, 0, 0));
        SetAngularVelocity(Vector3(0, 0, 0));
        return;
    }

    Event_PHYSICS_START* pRealEvent = (Event_PHYSICS_START*)(pEvt);
    float deltaTime = pRealEvent->m_frameTime;

    // gravity
    if (EnableGravity)
    {
        Vector3 gravity = Vector3(0.0f, -9.80665f, 0.0f);
        Vector3 gravityImpulse = gravity * deltaTime;

        SetVelocity(GetVelocity() + gravityImpulse);
    }


    UpdatePosition(deltaTime);

    force = Vector3(0, 0, 0);

    UpdateRotation(deltaTime);

    // 重置接触状态
    isOnGround = false;
}
```

Rotation update. Here I used another approach

```cpp
void Box::UpdateRotation(float deltaTime)
{
    if (!EnablePhysics || !IsDynamic)return;
    //// 更新旋转
    //if (angularVelocity.length() > EPSILON)
    //{
    //  Vector3 axis = angularVelocity.normalized();
    //  float angle = angularVelocity.length() * deltaTime;

    //  // 应用旋转
    //  m_base.turnAboutAxis(angle, axis);

    //  // 正交化旋转矩阵
    //  m_base.orthonormalizeRotation();

    //
    //}

    Vector3 omega = GetAngularVelocity();
    if (omega.lengthSqr() > EPSILON)
    {
        //Quaternion Rotation = m_base.createQuat();
        //Quaternion delta(omega.m_x * deltaTime, omega.m_y * deltaTime, omega.m_z * deltaTime, 1.0f);
        //Quaternion target = delta * Rotation;
        //m_base.setFromQuatAndPos(target, m_base.getPos());

        Quaternion Rotation = m_base.createQuat();

        // 使用轴角公式构造增量四元数
        float angle = omega.length() * deltaTime;
        Vector3 axis = omega.normalized();
        Quaternion delta = Quaternion(axis, angle);

        // 更新旋转
        Quaternion target = delta * Rotation;
        m_base.setFromQuatAndPos(target, m_base.getPos());

        // 正交化旋转矩阵，防止误差累积
        m_base.orthonormalizeRotation();
    }

}
```

Other

InertiaTensor

```
PE::Components::Box::Box(PE::GameContext& context, PE::MemoryArena arena, Handle hMyself, Vector3 _
    :PhysicsShape(context, arena, hMyself), Max(_Max), Min(_Min)
{
    DebugRenderColor = Vector3(1.f, 1.f, 0.f);
    PhysicsShapeType = ShapeType::ST_Box;

    for (int i = 0; i < 8; ++i) {
        Corners[i] = _Corners[i];
    }

    width = Max.m_x - Min.m_x;
    height = Max.m_y - Min.m_y;
    depth = Max.m_z - Min.m_z;

    // 计算局部惯性张量的对角元素
    Ixx = (1.0f / 12.0f) * mass * (height * height + depth * depth);
    Iyy = (1.0f / 12.0f) * mass * (width * width + depth * depth);
    Izz = (1.0f / 12.0f) * mass * (width * width + height * height);

    // 构建局部惯性张量矩阵
    inertiaTensorLocal.clear();
    inertiaTensorLocal.m[0][0] = Ixx;
    inertiaTensorLocal.m[1][1] = Iyy;
    inertiaTensorLocal.m[2][2] = Izz;

    // 构建局部逆惯性张量矩阵
    inverseInertiaTensorLocal.clear();
    inverseInertiaTensorLocal.m[0][0] = (Ixx != 0.0f) ? 1.0f / Ixx : 0.0f;
    inverseInertiaTensorLocal.m[1][1] = (Iyy != 0.0f) ? 1.0f / Iyy : 0.0f;
    inverseInertiaTensorLocal.m[2][2] = (Izz != 0.0f) ? 1.0f / Izz : 0.0f;
}
```

Physics Engine Loop

```
void PhysicsManager::do_PHYSICS_START(Events::Event* pEvt)
{
    Event_PHYSICS_START* pRealEvent = (Event_PHYSICS_START*)(pEvt);

    for (int i = 1; i < m_components.m_size; i++)//since index0 is Log component
    {
        PhysicsShape* shape1 = m_components[i].getObject<PhysicsShape>();
        shape1->do_CALCULATE_TRANSFORMATIONS(nullptr);
    }

    updateCollisions(pRealEvent->m_frameTime, manifolds);

    Resolve(manifolds, pRealEvent->m_frameTime);

    UpdateShapes(pRealEvent->m_frameTime, pEvt);

    manifolds.clear();

    //UpdateContactManifolds();

    /*for (auto& manifold : contactManifolds)
    {
        InitializeContactPoints(manifold);
    }*/

    //SolveContacts(pRealEvent->m_frameTime);

}
```

```cpp
void PhysicsShape::do_CALCULATE_TRANSFORMATIONS(Events::Event* pEvt)
{
    Handle hParentPS = Component::getFirstParentByType<PhysicsShape>();
    if (hParentPS.isValid())
    {
        Matrix4x4 tmp = hParentPS.getObject<PhysicsShape>()->m_worldTransform;
        if (m_inheritPositionOnly)
        {
            Vector3 pos = tmp.getPos();
            tmp.loadIdentity();
            tmp.setPos(pos);
        }
        m_worldTransform = tmp * m_base;
    }
    else
    {
        m_worldTransform = m_base;
    }
    UpdateInverseInertiaTensorWorld();
}
```