Milestone3: Stabilizing and Optimizing Physics Engine

In M3, I implemented Baumgarte Stabilization to prevent physics bodies from further penetrating one another. Additionally, two slope parameters are utilized to mitigate jittering when both speed and penetration depth are low. Currently, our system maintains only a single contact point for each manifold, which complicates the full stabilization of the boxes; ideally, four points should be maintained in total to maximize the area covered by these points. Consequently, during the demonstration, you may observe that the boxes exhibit some jittering—this behavior is expected.

```
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);
    float SlopP = 0.0005, SlopR = 0.5;//stablization terms
    m_bias = -(beta / dt) * fmaxf(manifold->contactPoints[idx].penetrationDistance - SlopP, 0) + restitution * fmaxf(closingVelocity - SlopR, 0);
}
```

```
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;

    if (manifold->colliderA->IsDynamic)
    {
        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);
    }

    if (manifold->colliderB->IsDynamic)
    {
        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);

    }
```

Another optimization implemented is multithreading. In each stage of the physics update substep, I utilized a thread pool to efficiently calculate collisions and apply updates. This approach enables the physics system to iterate at a consistent speed while managing numerous objects simultaneously. Consequently, in the demo video, you can observe many boxes colliding and penetrating one another, all while maintaining stable FPS. This represents a

significant performance enhancement.

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

    if (buttonPressed)
    {

        Vector3 groundCornerMin(-0.5, -0.3, -0.5);
        Vector3 groundCornerMax(0.5, 0.3, 0.5);
        Vector3 groundPosition((rand() % 10) - 5, 1.1 + 3, 0);
        Box* groundBox = createStaticBox(m_pContext, m_arena, groundPosition, groundCornerMin, groundCornerMax, "StaticBox", false);
        groundBox->IsDynamic = true;
        buttonPressed = false;

    }


    int iterations = 5;
    float deltaTime = pRealEvent->m_frameTime / iterations;
    for (int j = 0; j < iterations; j++)
    {
        ParallelCalculateTransformations();


        updateCollisions(deltaTime, manifolds);

        Resolve(manifolds, deltaTime);


        UpdateShapes(deltaTime, pEvt);

        manifolds.clear();
    }

    float deleteThreshold = -10;
```

```
void PhysicsManager::updateCollisions(const float& deltaTime, std::vector<std::shared_ptr<ContactManifold>>& collisions)
{

    if (m_components.m_size <= 1) return;

    unsigned int threadCount = std::thread::hardware_concurrency();
    if (threadCount == 0) threadCount = 4;


    std::vector<std::thread> threads;
    std::vector<std::vector<std::shared_ptr<ContactManifold>>> threadLocalCollisions(threadCount);


    auto worker = [&](int startI, int endI, int threadIndex)
        {
            CollisionDetector CD; // 每个线程有自己独立的CD实例
            for (int i = startI; i < endI; i++)
            {
                PhysicsShape* shape1 = m_components[i].getObject<PhysicsShape>();
                if (!shape1) continue;

                for (int j = i + 1; j < m_components.m_size; j++)
                {
                    PhysicsShape* shape2 = m_components[j].getObject<PhysicsShape>();
                    if (!shape2) continue;


                    if (!shape1->ReadyToCollide || !shape2->ReadyToCollide) continue;
                    if (!shape1->EnableCollision || !shape2->EnableCollision) continue;

                    AABB* AABB_shape1 = shape1->getAABB();
                    AABB* AABB_shape2 = shape2->getAABB();
                    if (!AABB_shape1 || !AABB_shape2) continue;

                    if (!AABB_shape1->Intersects(*AABB_shape2)) continue;


                    CD.CollideDetection(shape1, shape2, threadLocalCollisions[threadIndex]);
                }
            }
        };


    int N = m_components.m_size;
    int workCount = N - 1;
    int chunkSize = workCount / threadCount;
    int remainder = workCount % threadCount;
```

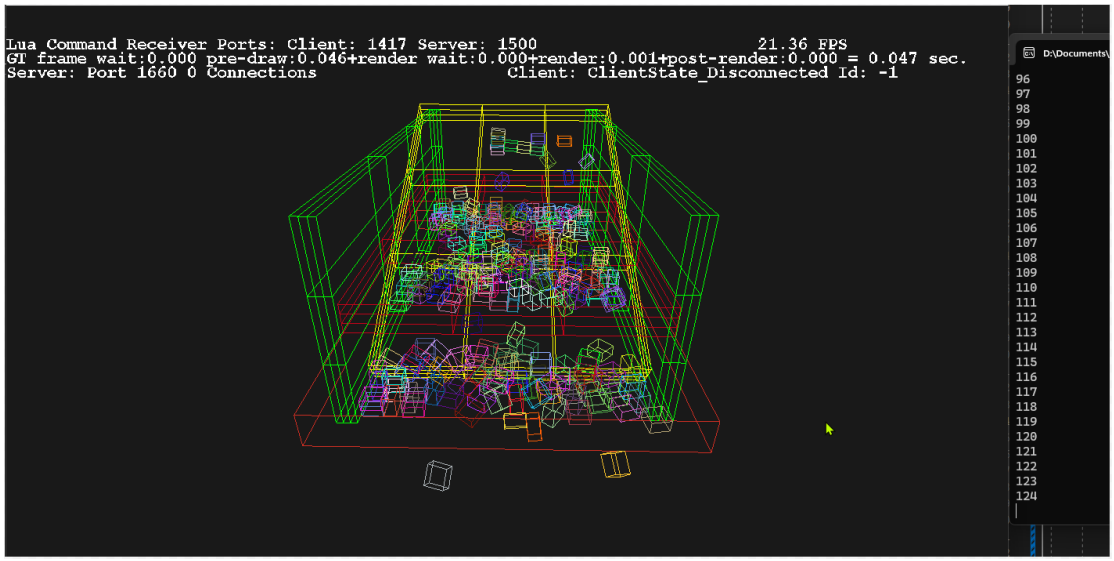
```cpp
int N = m_components.m_size;
int workCount = N - 1;
int chunkSize = workCount / threadCount;
int remainder = workCount % threadCount;

int currentStart = 1;
for (unsigned int t = 0; t < threadCount; t++)
{
    int currentEnd = currentStart + chunkSize + (t < remainder ? 1 : 0);
    if (currentEnd > N) currentEnd = N;
    threads.emplace_back(worker, currentStart, currentEnd, t);
    currentStart = currentEnd;
}


for (auto& th : threads) {
    if (th.joinable()) th.join();
}
```

```cpp
for (auto& localResult : threadLocalCollisions)
{
    std::move(localResult.begin(), localResult.end(), std::back_inserter(collisions));
}
```

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

    if (manifolds.empty()) return;

    unsigned int threadCount = std::thread::hardware_concurrency();
    if (threadCount == 0) threadCount = 4;

    int manifoldCount = static_cast<int>(manifolds.size());
    int chunkSize = manifoldCount / threadCount;
    int remainder = manifoldCount % threadCount;


    auto initWorker = [&](int start, int end) {
        for (int m = start; m < end; ++m) {
            auto& manifold = manifolds[m];
            for (int i = 0; i < manifold->contactPointCount; i++)
            {
                InitContactConstraints(manifold, i, deltaTime);
            }
        }
        };

    std::vector<std::thread> initThreads;
    {
        int currentStart = 0;
        for (unsigned int t = 0; t < threadCount; t++)
        {
            int currentEnd = currentStart + chunkSize + (t < remainder ? 1 : 0);
            if (currentEnd > manifoldCount) currentEnd = manifoldCount;
            initThreads.emplace_back(initWorker, currentStart, currentEnd);
            currentStart = currentEnd;
        }

        for (auto& th : initThreads) {
            if (th.joinable()) th.join();
        }
    }


    for (auto& manifold : manifolds)
    {
        for (int i = 0; i < manifold->contactPointCount; i++)
        {
            SolveContactConstraints(manifold, i, deltaTime);
        }
    }
```

```cpp
void PhysicsManager::UpdateShapes(float deltaTime, Events::Event* pEvt)
{
    if (m_components.m_size <= 1) return;

    unsigned int threadCount = std::thread::hardware_concurrency();
    if (threadCount == 0) threadCount = 4;

    int N = m_components.m_size - 1; //
    int chunkSize = N / threadCount;
    int remainder = N % threadCount;

    auto integrateWorker = [&](int startIndex, int endIndex) {
        for (int i = startIndex; i < endIndex; i++) {

            int actualIndex = i + 1;
            PhysicsShape* shape1 = m_components[actualIndex].getObject<PhysicsShape>();
            if (shape1) {
                shape1->Integrate(deltaTime);
            }
        }
        };


    std::vector<std::thread> threads;
    {
        int currentStart = 0;
        for (unsigned int t = 0; t < threadCount; t++) {
            int currentEnd = currentStart + chunkSize + (t < remainder ? 1 : 0);
            if (currentEnd > N) currentEnd = N;
            threads.emplace_back(integrateWorker, currentStart, currentEnd);
            currentStart = currentEnd;
        }


        for (auto& th : threads) {
            if (th.joinable()) th.join();
        }
    }
}
```

The demonstration features a simulated coin pusher, utilizing cubes in place of actual coins, as the GJK and EPA algorithms are indifferent to the types of colliders used, provided that appropriate support functions are implemented. Players can manipulate the push bar to propel coins onto the stage while simultaneously generating an increasing number of coins within the scene. Observing numerous coins being pushed is quite satisfying; thus, a scoreboard has been incorporated to track the total number of coins successfully pushed off the platform.



Hope you enjoy the demo!