# #1 – Physics Overview

(Can be seen in PhysicsEngine::UpdatePhysics)

//A whole physics engine in 6 simple steps =D

//1. Broadphase Collision Detection (Fast and dirty)

//2. Narrowphase Collision Detection (Accurate but slow)

//3. Initialize Constraint Params (precompute elasticity/baumgarte factor etc)

//4. Update Velocities

//5. Constraint Solver – Solve for velocity based on external constraints

//6. Update Positions (with final 'real' velocities)

# #2 – Integration

PhysicsEngine::UpdatePhysics

Section 4 Add:

for (PhysicsNode\* obj : physicsNodes) obj->IntegrateForVelocity(updateTimestep);

Section 6 Add:

for (PhysicsNode\* obj : physicsNodes) obj->IntegrateForPosition(updateTimestep);

PhysicsNode.cpp

**PhysiceNode::IntegrateForVelocity**

void PhysicsNode::IntegrateForVelocity(float dt)

{

//Apply Gravity

// Technically gravity here is calculated by formula: ( gravity / invMass \* invMass \* dt )

// So even though the divide and multiply cancel out, we still need to handle the possibility of divide by zero.

if (invMass > 0.0f)

linVelocity += PhysicsEngine::Instance()->GetGravity() \* dt;

//Semi-Implicit Euler Intergration

// - See "Update Position" below

linVelocity += force \* invMass \* dt;

//Apply Velocity Damping

// - This removes a tiny bit of energy from the simulation each update to stop slight calculation errors accumulating and adding force from nowhere.

// - In it's present form this can be seen as a rough approximation of air resistance, albeit (wrongly?) making the assumption that all objects

// have the same surface area.

linVelocity = linVelocity \* PhysicsEngine::Instance()->GetDampingFactor();

//Angular Rotation

// - These are the exact same calculations as the three lines above, except for rotations rather than positions.

// - Mass -> Torque

// - Velocity -> Rotational Velocity

// - Position -> Orientation

angVelocity += invInertia \* torque \* dt;

//Apply Velocity Damping

angVelocity = angVelocity \* PhysicsEngine::Instance()->GetDampingFactor();

}

**PhysicsNode::IntegrateForPosition**

void PhysicsNode::IntegrateForPosition(float dt)

{

//Update Position

// - Euler integration; works on the assumption that linearvelocity does not change over time (or changes so slightly it doesnt make a difference).

// - In this scenario, gravity /will/ be increasing velocity over time. The in-accuracy of not taking into account of these changes over time can be

// - visibly seen in tutorial 1.. and thus how better integration schemes lead to better approximations by taking into account of curvature.

position += linVelocity \* dt;

//Update Orientation

// - This is slightly different calculation due to the wierdness of quaternions. It does the same thing as position update

// (with a slight error) but from i've seen, is generally the best way to update orientation.

orientation = orientation + Quaternion(angVelocity \* dt \* 0.5f, 0.0f) \* orientation;

//invInertia = invInertia \* (Quaternion(angVelocity \* dt \* 0.5f, 0.0f) \* orientation).ToMatrix3();

//As the above formulation has slight approximation error, we need to normalize our orientation here to stop them accumulating

// over time.

orientation.Normalise();

}

# #3 – Constraints

**PhysicsEngine.cpp**

**PhysicsEngine::UpdatePhysics()**

Section 3 Add:

for (Constraint\* c : constraints) c->PreSolverStep(updateTimestep);

//Optional pre-computation step, needed for some constraints

Section 5 Add:

for (Constraint\* c : constraints) c->ApplyImpulse();

**DistanceConstraint.h**

**DistanceConstraint::ApplyImpulse**

virtual void ApplyImpulse() override

{

//Compute current constraint vars based on object A/B's position/rotation

Vector3 r1 = pnodeA->GetOrientation().ToMatrix3() \* relPosA;

Vector3 r2 = pnodeB->GetOrientation().ToMatrix3() \* relPosB;

//Get the global contact points in world space

Vector3 globalOnA = r1 + pnodeA->GetPosition();

Vector3 globalOnB = r2 + pnodeB->GetPosition();

//Get the vector between the two contact points

Vector3 ab = globalOnB - globalOnA;

Vector3 abn = ab;

abn.Normalise();

//Compute the velocity of object's A and B at the point of contact

Vector3 v0 = pnodeA->GetLinearVelocity() + Vector3::Cross(pnodeA->GetAngularVelocity(), r1);

Vector3 v1 = pnodeB->GetLinearVelocity() + Vector3::Cross(pnodeB->GetAngularVelocity(), r2);

float abnVel = Vector3::Dot(v0 - v1, abn); //Relative velocity in constraint direction

//Compute the 'mass' of the constraint

// e.g. How difficult it is to move the two objects in the direction of the constraint

float invConstraintMassLin = pnodeA->GetInverseMass() + pnodeB->GetInverseMass();

float invConstraintMassRot = Vector3::Dot(abn,

Vector3::Cross(pnodeA->GetInverseInertia() \* Vector3::Cross(r1, abn), r1) +

Vector3::Cross(pnodeB->GetInverseInertia() \* Vector3::Cross(r2, abn), r2)

);

float constraintMass = invConstraintMassLin + invConstraintMassRot;

if (constraintMass > 0.0f)

{

//Baumgarte Offset (Adds energy to the system to counter slight solving errors that accumulate over time - known as 'constraint drift')

// - Experiment by commenting this out and see how it affects the constraints over time and when you manually move the objects apart.

// - The key is to find a nice value that is small enough not to cause objects to explode but also

// enough to make sure all constraints /will/ be satisfied. This value (0.1) will change based on your physics objects, timestep etc

// and also how many constraints you are chaining together.

float b = 0.0f;

//-Optional-

float distance\_offset = ab.Length() - targetLength;

float baumgarte\_scalar = 0.1f;

b = -(baumgarte\_scalar / PhysicsEngine::Instance()->GetDeltaTime()) \* distance\_offset;

//-Eof Optional-

//Compute velocity impulse (jn)

// In order to satisfy the distance constraint we need to apply forces to ensure

// the relative velocity (abnVel) in the direction of the constraint is zero. So

// we take inverse of the current rel velocity and multiply it by how hard it will be to

// move the objects.

// Note: We also add in any extra energy to the system here, e.g. baumgarte (and later elasticity)

float jn = -(abnVel + b) / constraintMass;

//Apply linear velocity impulse

pnodeA->SetLinearVelocity(pnodeA->GetLinearVelocity() + abn \* (pnodeA->GetInverseMass() \* jn));

pnodeB->SetLinearVelocity(pnodeB->GetLinearVelocity() - abn \* (pnodeB->GetInverseMass() \* jn));

//Apply rotational velocity impulse

pnodeA->SetAngularVelocity(pnodeA->GetAngularVelocity() + pnodeA->GetInverseInertia() \* Vector3::Cross(r1, abn \* jn));

pnodeB->SetAngularVelocity(pnodeB->GetAngularVelocity() - pnodeB->GetInverseInertia() \* Vector3::Cross(r2, abn \* jn));

}

}

# #4 – Collision Detection

**General (and inside PhysicsEngine::Narrowphase)**

Void Narrowphase()

{

For (all possible collisions)

If (collisiondetectionSAT::AreColliding(pair))

//YAY

}

**SphereCollisionShape**

void SphereCollisionShape::GetCollisionAxes(const PhysicsNode\* otherObject, std::vector<Vector3>& out\_axes) const

{

/\* There are infinite possible axes on a sphere so we MUST handle it seperately

- Luckily we can just get the closest point on the opposite object to our centre and use that.

\*/

Vector3 dir = (otherObject->GetPosition() - Parent()->GetPosition()).Normalise();

Vector3 p1 = Parent()->GetPosition();

Vector3 p2 = otherObject->GetCollisionShape()->GetClosestPoint(p1);

out\_axes.push\_back((p1 - p2).Normalise());

}

Vector3 SphereCollisionShape::GetClosestPoint(const Vector3& point) const

{

Vector3 diff = (point - Parent()->GetPosition()).Normalise();

return Parent()->GetPosition() + diff \* m\_Radius;

}

void SphereCollisionShape::GetMinMaxVertexOnAxis(

const Vector3& axis,

Vector3& out\_min,

Vector3& out\_max) const

{

out\_min = Parent()->GetPosition() - axis \* m\_Radius;

out\_max = Parent()->GetPosition() + axis \* m\_Radius;

}

**CollisionDetectionSAT**

bool CollisionDetectionSAT::AreColliding(CollisionData\* out\_coldata)

{

if (!cshapeA || !cshapeB)

return false;

areColliding = false;

possibleColAxes.clear();

//<----- DEFAULT AXES ------->

// - GetCollisionAxes takes in the /other/ object as a parameter here

std::vector<Vector3> axes1, axes2;

cshapeA->GetCollisionAxes(pnodeB, axes1);

for (const Vector3& axis : axes1)

AddPossibleCollisionAxis(axis);

cshapeB->GetCollisionAxes(pnodeA, axes2);

for (const Vector3& axis : axes2)

AddPossibleCollisionAxis(axis);

//<----- EDGE-EDGE CASES ----->

// Handles the case where two edges meet and the final collision direction is mid way between two default collision axes

// provided above. (This is only needed when dealing with 3D collision shapes)

// As mentioned in the tutorial, this should be the edge vector's not normals we test against, however for a cuboid

// example this is the same as testing the normals as each normal /will/ match a given edge elsewhere on the object.

// For more complicated geometry, this will have to be changed to actual edge vectors instead of normals.

for (const Vector3& norm1 : axes1)

{

for (const Vector3& norm2 : axes2)

{

AddPossibleCollisionAxis(

Vector3::Cross(norm1, norm2).Normalise()

);

}

}

//Seperating axis theorem says that if a single axis can be found where the two objects are not colliding, then they cannot be colliding.

// - So we have to check each possible axis until we either return false, or return the best axis (one with the least penetration) found.

CollisionData cur\_colData;

bestColData.\_penetration = -FLT\_MAX;

for (const Vector3& axis : possibleColAxes)

{

//If the collision axis does NOT intersect then return immediately as we know that atleast

// in one direction/axis the two objects do not intersect

if (!CheckCollisionAxis(axis, cur\_colData))

return false;

if (cur\_colData.\_penetration >= bestColData.\_penetration)

{

bestColData = cur\_colData;

}

}

if (out\_coldata) \*out\_coldata = bestColData;

areColliding = true;

return true;

}

bool CollisionDetectionSAT::CheckCollisionAxis(const Vector3& axis, CollisionData& out\_coldata)

{

//Overlap Test

// Points go:

// +-------------+

// +-------|-----+ 2 |

// | 1 | | |

// | +-----|-------+

// +-------------+

// A ------C --- B ----- D

//

// IF A < C AND B > C (Overlap in order object 1 -> object 2)

// IF C < A AND D > A (Overlap in order object 2 -> object 1)

Vector3 min1, min2, max1, max2;

//Get the min/max vertices along the axis from shape1 and shape2

cshapeA->GetMinMaxVertexOnAxis(axis, min1, max1);

cshapeB->GetMinMaxVertexOnAxis(axis, min2, max2);

float A = Vector3::Dot(axis, min1);

float B = Vector3::Dot(axis, max1);

float C = Vector3::Dot(axis, min2);

float D = Vector3::Dot(axis, max2);

//Overlap Test (Order: Object 1 -> Object 2)

if (A <= C && B >= C)

{

out\_coldata.\_normal = axis;

out\_coldata.\_penetration = C - B; //Smallest overlap distance is between B->C

//Compute closest point on edge of the object

out\_coldata.\_pointOnPlane = max1 + out\_coldata.\_normal \* out\_coldata.\_penetration;

return true;

}

//Overlap Test (Order: Object 2 -> Object 1)

if (C <= A && D >= A)

{

out\_coldata.\_normal = -axis; //Invert axis here so we can do all our resolution phase as Object 1 -> Object 2

out\_coldata.\_penetration = A - D; //Smallest overlap distance is between D->A

//Compute closest point on edge of the object

out\_coldata.\_pointOnPlane = min1 + out\_coldata.\_normal \* out\_coldata.\_penetration;

return true;

}

return false;

# }

# #5 – Manifolds

**PhysicsEngine.cpp**

**PhysicsEngine::NarrowPhaseCollisions()**

After:

bool okA = cp.pObjectA->FireOnCollisionEvent(cp.pObjectA, cp.pObjectB);

bool okB = cp.pObjectB->FireOnCollisionEvent(cp.pObjectB, cp.pObjectA);

Insert:

if (okA && okB)

{

// Build full collision manifold that will also handle the collision response between the two objects in the solver stage

Manifold\* manifold = new Manifold();

manifold->Initiate(cp.pObjectA, cp.pObjectB);

// Construct contact points that form the perimeter of the collision manifold

colDetect.GenContactPoints(manifold);

if (manifold->contactPoints.size() > 0)

{

// Add to list of manifolds that need solving

manifolds.push\_back(manifold);

}

else

delete manifold;

}

**CollisionDetectionSAT.cpp**

**CollisionDetectionSAT::GenContactPoints()**

if (!out\_manifold || !areColliding)

return;

if (bestColData.\_penetration >= 0.0f)

return;

//Get the required face information for the two shapes around the collision normal

std::list<Vector3> polygon1, polygon2;

Vector3 normal1, normal2;

std::vector<Plane> adjPlanes1, adjPlanes2;

cshapeA->GetIncidentReferencePolygon( bestColData.\_normal, polygon1, normal1, adjPlanes1);

cshapeB->GetIncidentReferencePolygon(-bestColData.\_normal, polygon2, normal2, adjPlanes2);

//If either shape1 or shape2 returned a single point, then it must be on a curve and thus the only contact point to generate is already availble

if (polygon1.size() == 0 || polygon2.size() == 0)

{

return; //No points returned, resulting in no possible contact points

}

else if (polygon1.size() == 1)

{

out\_manifold->AddContact(

polygon1.front(), //Polygon1 -> Polygon 2

polygon1.front() + bestColData.\_normal \* bestColData.\_penetration,

bestColData.\_normal,

bestColData.\_penetration);

}

else if (polygon2.size() == 1)

{

out\_manifold->AddContact(

polygon2.front() - bestColData.\_normal \* bestColData.\_penetration,

polygon2.front(), //Polygon2 <- Polygon 1

bestColData.\_normal,

bestColData.\_penetration);

}

else

{

//Otherwise use clipping to cut down the incident face to fit inside the reference planes using the surrounding face planes

//First we need to know if have to flip the incident and reference faces around for clipping

bool flipped = fabs(Vector3::Dot(bestColData.\_normal, normal1)) < fabs(Vector3::Dot(bestColData.\_normal, normal2));

if (flipped)

{

std::swap(polygon1, polygon2);

std::swap(normal1, normal2);

std::swap(adjPlanes1, adjPlanes2);

}

//Clip the incident face to the adjacent edges of the reference face

if (adjPlanes1.size() > 0)

SutherlandHodgmanClipping(polygon2, adjPlanes1.size(), &adjPlanes1[0], &polygon2, false);

//Finally clip (and remove) any contact points that are above the reference face

Plane refPlane = Plane(-normal1, -Vector3::Dot(-normal1, polygon1.front()));

SutherlandHodgmanClipping(polygon2, 1, &refPlane, &polygon2, true);

//Now we are left with a selection of valid contact points to be used for the manifold

for (const Vector3& point : polygon2)

{

//Compute distance to reference plane

Vector3 pointDiff = point - GetClosestPointPolygon(point, polygon1);

float contact\_penetration = Vector3::Dot(pointDiff, bestColData.\_normal);

//Set Contact data

Vector3 globalOnA = point;

Vector3 globalOnB = point - bestColData.\_normal \* contact\_penetration;

//If we flipped incident and reference planes, we will need to flip it back before

// sending it to the manifold. e.g. turn it from talking about object2->object1

// into object1->object2

if (flipped)

{

contact\_penetration = -contact\_penetration;

globalOnA = point + bestColData.\_normal \* contact\_penetration;

globalOnB = point;

}

//Just make a final sanity check that the contact point is actual a point of contact

// not just a clipping bug

if (contact\_penetration < 0.0f)

{

out\_manifold->AddContact(

globalOnA,

globalOnB,

bestColData.\_normal,

contact\_penetration);

}

}

}

# #6 – Collision Response

**Manifold.cpp**

**Manifold::UpdateConstraint()**

//After c.sumImpulseFriction = (This is used in the next tutorial)

//Baumgarte Offset (Adds energy to the system to counter slight solving errors that accumulate over time - known as 'constraint drift')

//Very slightly different to the one we used to prevent the distance constraints breaking, as we now allow a minimum allowed error.

// In this case, we allow the objects to overlap by 1mm before adding in correctional energy to counteract the error,

// this is a little dirty trick that results in us always getting a manifold for constantly colliding objects which normally we would get collide

// one frame and not the next, endlessly jittering. If your interested, the \_slop part (and this issue in general) is usually handled in physics

// engines by pretending the collision volume is larger than it is during the narrowphase, so manifolds are generated for colliding, and almost colliding pairs.

const float baumgarte\_scalar = 0.1f;

const float baumgarte\_slop = 0.001f;

const float penetration\_slop = min(c.colPenetration + baumgarte\_slop, 0.0f);

c.b\_term += -(baumgarte\_scalar / PhysicsEngine::Instance()->GetDeltaTime()) \* penetration\_slop;

//Compute Elasticity Term

// This is the total velocity going into the collision relative to the collision normal,

// as elasticity is 'adding' energy back into the system we can attach it to our 'b' term

// which we already add to 'jt' when solving the contact constraint.

const float elasticity = pnodeA->GetElasticity() \* pnodeB->GetElasticity();

const float elatisity\_term = Vector3::Dot(c.colNormal,

pnodeA->GetLinearVelocity()

+ Vector3::Cross(c.relPosA, pnodeA->GetAngularVelocity())

- pnodeB->GetLinearVelocity()

- Vector3::Cross(c.relPosB, pnodeB->GetAngularVelocity())

);

c.b\_term += (elasticity \* elatisity\_term) / contactPoints.size();

**Manifold::ApplyImpulse()**

void Manifold::ApplyImpulse()

{

for (ContactPoint& contact : contactPoints)

{

SolveContactPoint(contact);

}

}

**Manifold::SolveContactPoint()**

void Manifold::SolveContactPoint(ContactPoint& c)

{

Vector3 r1 = c.relPosA;

Vector3 r2 = c.relPosB;

Vector3 v0 = pnodeA->GetLinearVelocity() + Vector3::Cross(pnodeA->GetAngularVelocity(), r1);

Vector3 v1 = pnodeB->GetLinearVelocity() + Vector3::Cross(pnodeB->GetAngularVelocity(), r2);

Vector3 dv = v1 - v0;

//Collision Resolution

float constraintMass = (pnodeA->GetInverseMass() + pnodeB->GetInverseMass()) +

Vector3::Dot(c.colNormal,

Vector3::Cross(pnodeA->GetInverseInertia()\*Vector3::Cross(r1, c.colNormal), r1) +

Vector3::Cross(pnodeB->GetInverseInertia()\*Vector3::Cross(r2, c.colNormal), r2));

if (constraintMass > 0.0f)

{

float jn = max(-Vector3::Dot(dv, c.colNormal) + c.b\_term, 0.0f);

jn = jn / constraintMass;

pnodeA->SetLinearVelocity(pnodeA->GetLinearVelocity() - c.colNormal\*(jn \* pnodeA->GetInverseMass()));

pnodeB->SetLinearVelocity(pnodeB->GetLinearVelocity() + c.colNormal\*(jn \* pnodeB->GetInverseMass()));

pnodeA->SetAngularVelocity(pnodeA->GetAngularVelocity() - pnodeA->GetInverseInertia()\* Vector3::Cross(r1, c.colNormal \* jn));

pnodeB->SetAngularVelocity(pnodeB->GetAngularVelocity() + pnodeB->GetInverseInertia()\* Vector3::Cross(r2, c.colNormal \* jn));

}

//Friction

Vector3 tangent = dv - c.colNormal \* Vector3::Dot(dv, c.colNormal);

float tangent\_len = tangent.Length();

if (tangent\_len > 1e-6f)

{

tangent = tangent / tangent\_len;

float frictionalMass = (pnodeA->GetInverseMass() + pnodeB->GetInverseMass()) +

Vector3::Dot(tangent,

Vector3::Cross(pnodeA->GetInverseInertia()\* Vector3::Cross(r1, tangent), r1) +

Vector3::Cross(pnodeB->GetInverseInertia()\* Vector3::Cross(r2, tangent), r2));

if (frictionalMass > 0.0f)

{

float frictionCoef = (pnodeA->GetFriction() \* pnodeB->GetFriction());

float jt = -Vector3::Dot(dv, tangent) \* frictionCoef;

jt = jt / frictionalMass;

pnodeA->SetLinearVelocity(pnodeA->GetLinearVelocity() - tangent\*(jt\*pnodeA->GetInverseMass()));

pnodeB->SetLinearVelocity(pnodeB->GetLinearVelocity() + tangent\*(jt\*pnodeB->GetInverseMass()));

pnodeA->SetAngularVelocity(pnodeA->GetAngularVelocity() - pnodeA->GetInverseInertia()\* Vector3::Cross(r1, tangent\*jt));

pnodeB->SetAngularVelocity(pnodeB->GetAngularVelocity() + pnodeB->GetInverseInertia()\* Vector3::Cross(r2, tangent\*jt));

}

}

}

PhysicsEngine.cpp

PhysicsEngine::UpdatePhysics()

Add to section 3:

for (Manifold\* m : manifolds) m->PreSolverStep(updateTimestep);

Add to section 5:

for (Manifold\* m : manifolds) m->ApplyImpulse();

# #7 - SOLVER

PhysicsEngine.cpp

//REPLACE

//4. Constraint Solver

for (Manifold\* m : manifolds) m->ApplyImpulse();

for (Constraint\* c : constraints) c->ApplyImpulse();

//WITH

//4. Constraint Solver

for (size\_t i = 0; i < SOLVER\_ITERATIONS; ++i)

{

for (Manifold\* m : manifolds) m->ApplyImpulse();

for (Constraint\* c : constraints) c->ApplyImpulse();

}

Manifold.cpp – SolveContactPoint

Changes here are needed to ensure that :

1. The Collision never solves itself backwards (e.g. pushes itself all the way through the other side of the other object ). As now if an object is colliding on both sides it could theoretically solve it by doing the above scenario .

2. The Friction force never exceeds the force applied by the main collision force . As now the main collision force will change over time .

//Add after float jn=…

//and before jn = jn / constraintMass;

float oldSumImpulseContact = c.sumImpulseContact;

c.sumImpulseContact = max(c.sumImpulseContact + jn, 0.0f);

jn = c.sumImpulseContact - oldSumImpulseContact;

//Add after float jt=…

//and before jt = frictionCoef / frictionalMass;

//Stop Friction from ever being more than frictionCoef \* normal resolution impulse

//

Similar to above for SumImpulseContact, except for friction the direction of friction is potentially changing each time we

// solve the contact (as other objects are bouncing this object in different directions). So instead of a scalar,

we need to clamp the total tangential impulse vector to the normal resolution impulse. This way, we can

// solve friction in any direction during the solving step without having to precompute and check a single tangent vector.

Vector3 oldImpulseFriction = c.sumImpulseFriction;

c.sumImpulseFriction = c.sumImpulseFriction + tangent \* jt;

float len = c.sumImpulseFriction.Length();

if (len > 0.0f && len > c.sumImpulseContact)

{

c.sumImpulseFriction = c.sumImpulseFriction / len \* c.sumImpulseContact;

}

tangent = oldImpulseFriction - c.sumImpulseFriction;

jt = 1.0f;

Solver – Final note on improving accuracy:-

As you have probably realised by now, a lot of making a ‘stable’ physics engine is about either solving without errors entirely (ideally) or dealing with errors in a way that doesn’t explode the rest of the system. So we have to accept the fact that our solver still has errors, partly because we don’t run it till it’s fully solved but only for a fixed number of iterations and partly because of the way we solve constraints although as you can hopefully see they are very small. However, if you watch the stacking scene example with the above solver code you may see the stack very slowly drift and fall over. This is because those tiny errors are all being solved in the same way each collision, and each collision (on the pyramid example) is colliding every frame, so the result is a consistent error in the same direction every frame. To allieviate this, we can randomise the order we solve the contact points in the manifold and also randomise the order we solve the manifolds/constraints each frame.

For those of you keeping up with the above notes on Ax=B matrix form, this is fine as long as we don’t randomise anything inside the solving of the above equation. It is equivalently randomising the row order.

PhysicsEngine.cpp

Directly after the Narrowphase in PhysicsEngine::UpdatePhysics()

std::random\_shuffle(manifolds.begin(), manifolds.end());

std::random\_shuffle(constraints.begin(), constraints.end());

Manifold.cpp

void Manifold::PreSolverStep(float dt)

{

std::random\_shuffle(contactPoints.begin(), contactPoints.end()); //NEW

for (ContactPoint& contact : contactPoints)

{

UpdateConstraint(contact);

}

}