My milestone aims to do physics simulation with impulse and angular velocity.

However it works well with only friction and linear impulse, but if I put everything together, things start to look different and incorrect. I need more time to make this right, maybe another milestone.

Following are major changes I made for this physics engine:  
PhysicsManager: Aim to handle collision and solve physics problems

#include "PhysicsManager.h"

#include "PrimeEngine/Events/StandardEvents.h"

#include "PrimeEngine/Lua/LuaEnvironment.h"

//#include <PrimeEngine/Scene/MeshInstance.h>

#include "PrimeEngine/Scene/DebugRenderer.h"

//#include <PrimeEngine/Scene/SkeletonInstance.h>

//#include <CharacterControl/Characters/SoldierNPC.h>

//#include <CharacterControl/Characters/SoldierNPCMovementSM.h>

float Clamp(float value, float minVal, float maxVal)

{

return std::max(minVal, std::min(value, maxVal));

}

namespace PE {

namespace Components {

PE\_IMPLEMENT\_CLASS1(PhysicsManager, Component);

using namespace PE::Events;

//using namespace CharacterControl::Components;

PhysicsManager::PhysicsManager(PE::GameContext& context, PE::MemoryArena arena, Handle hMyself)

:Component(context, arena, hMyself)

{

}

void PhysicsManager::do\_PHYSICS\_START(Events::Event\* pEvt)

{

Event\_PHYSICS\_START\* pRealEvent = (Event\_PHYSICS\_START\*)(pEvt);

updateCollisions(pRealEvent->m\_frameTime);

UpdateContactManifolds();

for (auto& manifold : contactManifolds)

{

InitializeContactPoints(manifold);

}

SolveContacts(pRealEvent->m\_frameTime);

}

void PhysicsManager::do\_PRE\_RENDER\_needsRC(PE::Events::Event\* pEvt)

{

for (int i = 0; i < m\_components.m\_size; i++)

{

Handle& h = m\_components[i];

PhysicsShape\* pShape = h.getObject<PhysicsShape>();

if (pShape->isInstanceOf<PhysicsShape>()) pShape->DebugRender();

}

}

void PhysicsManager::SolveContacts(float deltaTime)

{

const int iterations = 10; // Number of iterations, adjustable as needed

for (int i = 0; i < iterations; ++i)

{

for (ContactManifold& manifold : contactManifolds)

{

for (ContactPoint& contact : manifold.contacts)

{

SolveContact(manifold.shapeA, manifold.shapeB, contact, deltaTime);

}

}

}

}

void PhysicsManager::do\_START\_SIMULATION(PE::Events::Event\* pEvt)

{

for (int i = 0; i < m\_components.m\_size; i++)

{

Handle& h = m\_components[i];

PhysicsShape\* pShape = h.getObject<PhysicsShape>();

if (pShape->isInstanceOf<Sphere>())

{

pShape->EnableGravity = true;

pShape->EnablePhysics = true;

}

if (pShape->isInstanceOf<Box>())

{

pShape->EnableGravity = true;

pShape->EnablePhysics = true;

}

}

}

void PhysicsManager::addDefaultComponents()

{

Component::addDefaultComponents();

PE\_REGISTER\_EVENT\_HANDLER(Event\_PHYSICS\_START, PhysicsManager::do\_PHYSICS\_START);

PE\_REGISTER\_EVENT\_HANDLER(Event\_PRE\_RENDER\_needsRC, PhysicsManager::do\_PRE\_RENDER\_needsRC);

PE\_REGISTER\_EVENT\_HANDLER(Event\_START\_SIMULATION, PhysicsManager::do\_START\_SIMULATION);

//PE\_REGISTER\_EVENT\_HANDLER(Events::Event\_CALCULATE\_TRANSFORMATIONS, PhysicsManager::do\_CALCULATE\_TRANSFORMATIONS);

}

bool PhysicsManager::CheckSphereCollision(Sphere\* sphere1, Sphere\* sphere2, Vector3& collisionPoint, float& PenetrationDepth)

{

// Calculate the directional vector between the centers of the two spheres

Vector3 direction = sphere2->TransformedCenter - sphere1->TransformedCenter;

// Calculate the distance between centers

float distance = direction.length();

// Check for collision

float radiusSum = sphere1->radius + sphere2->radius;

if (distance <= radiusSum)

{

// Normalize direction vector

Vector3 collisionNormal = direction / distance;

// Calculate the collision point (located between the surfaces of the two spheres)

collisionPoint = sphere1->TransformedCenter + collisionNormal \* sphere1->radius;

PenetrationDepth = (sphere1->radius + sphere2->radius) - (sphere1->GetPosition() - sphere2->GetPosition()).length();

return true;

}

else

{

return false;

}

}

bool PhysicsManager::CheckBoxCollision(Box\* box1, Box\* box2, Vector3& collisionPoint, float &PenetrationDepth)

{

// Check for collision

bool xOverlap = (box1->TransformedMin.m\_x <= box2->TransformedMax.m\_x) && (box1->TransformedMax.m\_x >= box2->TransformedMin.m\_x);

bool yOverlap = (box1->TransformedMin.m\_y <= box2->TransformedMax.m\_y) && (box1->TransformedMax.m\_y >= box2->TransformedMin.m\_y);

bool zOverlap = (box1->TransformedMin.m\_z <= box2->TransformedMax.m\_z) && (box1->TransformedMax.m\_z >= box2->TransformedMin.m\_z);

if (xOverlap && yOverlap && zOverlap)

{

// Calculate the minimum and maximum points of the overlap region

float overlapMinX = std::max(box1->TransformedMin.m\_x, box2->TransformedMin.m\_x);

float overlapMaxX = std::min(box1->TransformedMax.m\_x, box2->TransformedMax.m\_x);

float overlapMinY = std::max(box1->TransformedMin.m\_y, box2->TransformedMin.m\_y);

float overlapMaxY = std::min(box1->TransformedMax.m\_y, box2->TransformedMax.m\_y);

float overlapMinZ = std::max(box1->TransformedMin.m\_z, box2->TransformedMin.m\_z);

float overlapMaxZ = std::min(box1->TransformedMax.m\_z, box2->TransformedMax.m\_z);

// Calculate the overlap amount on each axis

float overlapX = overlapMaxX - overlapMinX;

float overlapY = overlapMaxY - overlapMinY;

float overlapZ = overlapMaxZ - overlapMinZ;

// Calculate penetration depth by taking the minimum overlap amount

PenetrationDepth = std::min({ overlapX, overlapY, overlapZ });

// Determine the collision normal direction, based on the minimum overlap axis

Vector3 collisionNormal(0.0f, 0.0f, 0.0f);

if (PenetrationDepth == overlapX)

{

// Minimum overlap in X-axis direction

if (box1->GetPosition().m\_x < box2->GetPosition().m\_x)

collisionNormal = Vector3(-1.0f, 0.0f, 0.0f); // box1 is to the left of box2

else

collisionNormal = Vector3(1.0f, 0.0f, 0.0f); // box1 is to the right of box2

}

else if (PenetrationDepth == overlapY)

{

// Minimum overlap in Y-axis direction

if (box1->GetPosition().m\_y < box2->GetPosition().m\_y)

collisionNormal = Vector3(0.0f, -1.0f, 0.0f); // box1 is below box2

else

collisionNormal = Vector3(0.0f, 1.0f, 0.0f); // box1 is above box2

}

else // penetrationDepth == overlapZ

{

// Minimum overlap in Z-axis direction

if (box1->GetPosition().m\_z < box2->GetPosition().m\_z)

collisionNormal = Vector3(0.0f, 0.0f, -1.0f); // box1 is in front of box2

else

collisionNormal = Vector3(0.0f, 0.0f, 1.0f); // box1 is behind box2

}

// Calculate the center point of the overlap region as the collision point

collisionPoint.m\_x = (overlapMinX + overlapMaxX) \* 0.5f;

collisionPoint.m\_y = (overlapMinY + overlapMaxY) \* 0.5f;

collisionPoint.m\_z = (overlapMinZ + overlapMaxZ) \* 0.5f;

return true;

}

else

{

return false;

}

}

bool PhysicsManager::CheckSphereBoxCollision(Sphere\* sphere, Box\* box, Vector3& collisionPoint, float& PenetrationDepth)

{

// Get the sphere center coordinates

Vector3 sphereCenter = sphere->TransformedCenter;

// Initialize the closest point to the sphere center

Vector3 closestPoint = sphereCenter;

// For each axis, find the closest point within the box's range

if (sphereCenter.m\_x < box->TransformedMin.m\_x) closestPoint.m\_x = box->TransformedMin.m\_x;

else if (sphereCenter.m\_x > box->TransformedMax.m\_x) closestPoint.m\_x = box->TransformedMax.m\_x;

if (sphereCenter.m\_y < box->TransformedMin.m\_y) closestPoint.m\_y = box->TransformedMin.m\_y;

else if (sphereCenter.m\_y > box->TransformedMax.m\_y) closestPoint.m\_y = box->TransformedMax.m\_y;

if (sphereCenter.m\_z < box->TransformedMin.m\_z) closestPoint.m\_z = box->TransformedMin.m\_z;

else if (sphereCenter.m\_z > box->TransformedMax.m\_z) closestPoint.m\_z = box->TransformedMax.m\_z;

// Calculate the squared distance between the sphere center and the closest point

Vector3 diff = sphereCenter - closestPoint;

float distanceSquared = diff.dotProduct(diff);

// Check for collision

if (distanceSquared <= (sphere->radius \* sphere->radius))

{

float distance = sqrt(distanceSquared);

// Calculate penetration depth

PenetrationDepth = sphere->radius - distance;

// The collision point is the closest point

collisionPoint = closestPoint;

return true;

}

else

{

return false;

}

}

void PhysicsManager::updateCollisions(const float& deltaTime)

{

for (int i = 1; i < m\_components.m\_size; i++) //since index0 is Log component

{

PhysicsShape\* shape1 = m\_components[i].getObject<PhysicsShape>();

for (int j = i + 1; j < m\_components.m\_size; j++)

{

PhysicsShape\* shape2 = m\_components[j].getObject<PhysicsShape>();

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

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

bool collision = false;

AABB AABB\_shape1 = shape1->getAABB();

AABB AABB\_shape2 = shape2->getAABB();

if (!AABB\_shape1.Intersects(AABB\_shape2)) continue;

Vector3 CollidePoint;

float PenetrationDepth;

if (shape1->isInstanceOf<Sphere>() && shape2->isInstanceOf<Sphere>())

{

collision = CheckSphereCollision(static\_cast<Sphere\*>(shape1), static\_cast<Sphere\*>(shape2), CollidePoint, PenetrationDepth);

}

else if (shape1->isInstanceOf<Box>() && shape2->isInstanceOf<Box>())

{

collision = CheckBoxCollision(static\_cast<Box\*>(shape1), static\_cast<Box\*>(shape2), CollidePoint, PenetrationDepth);

}

else if (shape1->isInstanceOf<Sphere>() && shape2->isInstanceOf<Box>())

{

collision = CheckSphereBoxCollision(static\_cast<Sphere\*>(shape1), static\_cast<Box\*>(shape2), CollidePoint, PenetrationDepth);

}

else if (shape1->isInstanceOf<Box>() && shape2->isInstanceOf<Sphere>())

{

collision = CheckSphereBoxCollision(static\_cast<Sphere\*>(shape2), static\_cast<Box\*>(shape1), CollidePoint, PenetrationDepth);

}

if (collision)

{

// Handle collision response

shape1->OnOverlap(shape2, CollidePoint, deltaTime);

shape2->OnOverlap(shape1, CollidePoint, deltaTime);

// Resolve collision

ResolveCollisionAngular(shape1, shape2, CollidePoint, PenetrationDepth, deltaTime);

}

}

}

}

void PhysicsManager::ResolveCollision(PhysicsShape\* shapeA, PhysicsShape\* shapeB, const Vector3& collisionPoint, float deltaTime)

{

// Calculate the collision normal

Vector3 normalA = shapeA->ComputeCollisionNormal(collisionPoint);

Vector3 collisionNormal = -normalA; // From shapeA to collision point, then inverted

// Check if both objects have physics enabled

bool A\_isDynamic = shapeA->EnablePhysics && shapeA->mass > 0;

bool B\_isDynamic = shapeB->EnablePhysics && shapeB->mass > 0;

// If both objects are static, no need to process

if (!A\_isDynamic && !B\_isDynamic)

return;

// Calculate relative velocity

Vector3 relativeVelocity = shapeA->velocity - shapeB->velocity;

// Calculate the component of velocity along the collision normal

float velocityAlongNormal = relativeVelocity.dotProduct(collisionNormal);

// If objects are separating, skip collision

if (velocityAlongNormal > 0)

return;

// Calculate restitution (take minimum of both objects)

float e = std::min(shapeA->restitution, shapeB->restitution);

// Calculate impulse scalar

float j = -(1 + e) \* velocityAlongNormal;

float inverseMassSum = (A\_isDynamic ? (1 / shapeA->mass) : 0.0f) + (B\_isDynamic ? (1 / shapeB->mass) : 0.0f);

if (inverseMassSum == 0)

return; // Avoid division by zero

j /= inverseMassSum;

const float contactThreshold = 0.01f;

bool isContact = std::abs(velocityAlongNormal \* deltaTime) < contactThreshold;

if (isContact)

{

e = 0.0f;

j = -velocityAlongNormal / inverseMassSum;

}

Vector3 impulse = j \* collisionNormal;

if (A\_isDynamic) shapeA->velocity += impulse \* (1 / shapeA->mass);

if (B\_isDynamic) shapeB->velocity -= impulse \* (1 / shapeB->mass);

if (isContact)

{

shapeA->isOnGround = true;

shapeB->isOnGround = true;

}

}

};

};

Sphere/Box: Inherited from PhysicsShape, updating them self every frame.

#include "Box.h"

#include "PrimeEngine/Lua/LuaEnvironment.h"

#include "PrimeEngine/Events/StandardEvents.h"

#include "PrimeEngine/Scene/DebugRenderer.h"

namespace PE

{

namespace Components

{

PE\_IMPLEMENT\_CLASS1(Box, PhysicsShape);

void Box::addDefaultComponents()

{

Component::addDefaultComponents();

PE\_REGISTER\_EVENT\_HANDLER(Events::Event\_CALCULATE\_TRANSFORMATIONS, Box::do\_CALCULATE\_TRANSFORMATIONS);

PE\_REGISTER\_EVENT\_HANDLER(Events::Event\_PHYSICS\_START, Box::do\_PHYSICS\_START);

}

void Box::DebugRender()

{

const static int numEdges = 12;

const static int numPts = numEdges \* 2;

Vector3 linepts[numPts \* 2];

int iPt = 0;

for (int i = 0; i < numEdges; ++i)

{

Vector3 start = TransformedCorners[edges[i][0]];

Vector3 end = TransformedCorners[edges[i][1]];

linepts[iPt++] = start;

linepts[iPt++] = DebugRenderColor;

linepts[iPt++] = end;

linepts[iPt++] = DebugRenderColor;

}

bool hasTransform = true;

DebugRenderer::Instance()->createLineMesh(

hasTransform,

m\_worldTransform,

&linepts[0].m\_x,

numPts,

0.f);

PhysicsShape::DebugRender();

}

AABB Box::calculateAABB()

{

// 初始化 AABB 的最小和最大点

Vector3 minPoint = TransformedCorners[0];

Vector3 maxPoint = TransformedCorners[0];

// 遍历所有变换后的顶点，计算最小和最大坐标值

for (int i = 1; i < 8; ++i)

{

const Vector3& point = TransformedCorners[i];

// 更新最小点

if (point.m\_x < minPoint.m\_x) minPoint.m\_x = point.m\_x;

if (point.m\_y < minPoint.m\_y) minPoint.m\_y = point.m\_y;

if (point.m\_z < minPoint.m\_z) minPoint.m\_z = point.m\_z;

// 更新最大点

if (point.m\_x > maxPoint.m\_x) maxPoint.m\_x = point.m\_x;

if (point.m\_y > maxPoint.m\_y) maxPoint.m\_y = point.m\_y;

if (point.m\_z > maxPoint.m\_z) maxPoint.m\_z = point.m\_z;

}

// 返回计算得到的 AABB

return AABB(minPoint, maxPoint);

}

void Box::UpdatePosition(float deltaTime)

{

if (!EnablePhysics || !IsDynamic)return;

// 更新盒子的位置

// 假设您有一个表示位置的成员变量，例如 position

Vector3 newPos = m\_worldTransform.getPos() + velocity \* deltaTime;

// 更新世界变换矩阵

SetPosition(newPos);

}

void Box::UpdateRotation(float deltaTime)

{

if (!EnablePhysics || !IsDynamic)return;

//angularVelocity = Vector3(2, 0, 0);

// 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 Box::ComputeCollisionNormal(const Vector3& collisionPoint)

{

// 将碰撞点转换到盒子的局部空间

Matrix4x4 invTransform = m\_worldTransform.inverse();

Vector3 localPoint = invTransform \* collisionPoint;

// 获取盒子的半尺寸

Vector3 halfExtents = (Max - Min) \* 0.5f;

// 计算盒子的局部中心

Vector3 localCenter = (Min + Max) \* 0.5f;

// 计算从中心到局部碰撞点的偏移

Vector3 d = localPoint - localCenter;

// 计算到每个面的距离

float dx = halfExtents.m\_x - fabsf(d.m\_x);

float dy = halfExtents.m\_y - fabsf(d.m\_y);

float dz = halfExtents.m\_z - fabsf(d.m\_z);

// 确定哪个面最近

Vector3 localNormal;

if (dx <= dy && dx <= dz)

{

localNormal = Vector3((d.m\_x > 0) ? 1 : -1, 0, 0);

}

else if (dy <= dx && dy <= dz)

{

localNormal = Vector3(0, (d.m\_y > 0) ? 1 : -1, 0);

}

else

{

localNormal = Vector3(0, 0, (d.m\_z > 0) ? 1 : -1);

}

// 将局部法线转换回世界空间

Vector3 worldNormal = m\_worldTransform.transformDirection(localNormal);

return worldNormal.normalized();

}

void Box::UpdateInverseInertiaTensorWorld()

{

// 提取物体的旋转矩阵

Matrix3x3 rotationMatrix = m\_worldTransform.GetRotationMatrix();

// 计算世界坐标系下的逆惯性张量

inverseInertiaTensorWorld = rotationMatrix \* inverseInertiaTensorLocal \* rotationMatrix.transpose();

}

void Box::do\_CALCULATE\_TRANSFORMATIONS(Events::Event\* pEvt)

{

PhysicsShape::do\_CALCULATE\_TRANSFORMATIONS(pEvt);

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

{

TransformedCorners[i] = m\_worldTransform \* Corners[i];

}

// 初始化 TransformedMin 和 TransformedMax

TransformedMin = TransformedCorners[0];

TransformedMax = TransformedCorners[0];

// 遍历所有的 TransformedCorners，计算 TransformedMin 和 TransformedMax

for (int i = 1; i < 8; ++i)

{

Vector3& corner = TransformedCorners[i];

if (corner.m\_x < TransformedMin.m\_x) TransformedMin.m\_x = corner.m\_x;

if (corner.m\_y < TransformedMin.m\_y) TransformedMin.m\_y = corner.m\_y;

if (corner.m\_z < TransformedMin.m\_z) TransformedMin.m\_z = corner.m\_z;

if (corner.m\_x > TransformedMax.m\_x) TransformedMax.m\_x = corner.m\_x;

if (corner.m\_y > TransformedMax.m\_y) TransformedMax.m\_y = corner.m\_y;

if (corner.m\_z > TransformedMax.m\_z) TransformedMax.m\_z = corner.m\_z;

}

ReadyToCollide = true;

if (m\_isTransformDirty)

{

//m\_isTransformDirty = false;

}

}

void Box::do\_PHYSICS\_START(Events::Event\* pEvt)

{

PhysicsShape::do\_PHYSICS\_START(pEvt);

if (!EnablePhysics)return;

}

PE::Components::Box::Box(PE::GameContext& context, PE::MemoryArena arena, Handle hMyself)

:PhysicsShape(context, arena, hMyself)

{

DebugRenderColor = Vector3(1.f, 1.f, 0.f);

PhysicsShapeType = ShapeType::ST\_Box;

}

PE::Components::Box::Box(PE::GameContext& context, PE::MemoryArena arena, Handle hMyself, Vector3 \_Max, Vector3 \_Min, Vector3 \_Corners[8])

: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;

}

}

}

#include "PrimeEngine/APIAbstraction/APIAbstractionDefines.h"

#include "PrimeEngine/Lua/LuaEnvironment.h"

#include "Sphere.h"

#include "../Events/Component.h"

#include "PrimeEngine/Events/StandardEvents.h"

#include "PrimeEngine/Scene/DebugRenderer.h"

namespace PE

{

namespace Components

{

PE\_IMPLEMENT\_CLASS1(Sphere,PhysicsShape);

void Sphere::addDefaultComponents()

{

Component::addDefaultComponents();

PE\_REGISTER\_EVENT\_HANDLER(Events::Event\_CALCULATE\_TRANSFORMATIONS, Sphere::do\_CALCULATE\_TRANSFORMATIONS);

PE\_REGISTER\_EVENT\_HANDLER(Events::Event\_PHYSICS\_START, Sphere::do\_PHYSICS\_START);

}

Sphere::Sphere(PE::GameContext& context, PE::MemoryArena arena, Handle hMyself)

:PhysicsShape(context, arena, hMyself)

{

DebugRenderColor = Vector3(0.0f, 1.0f, 0.0f);

PhysicsShapeType = ShapeType::ST\_Shpere;

}

void Sphere::DebugRender()

{

const int numSegments = 12; // 渲染精度

const static int numPts = numSegments \* 3 \* 2; // 每个纬线圈和经线圈各有 numSegments 条线段，乘以 3（XY、XZ、YZ 三个平面）

Vector3 linepts[numPts \* 2];

int iPt = 0;

// (XY, XZ, YZ)

for (int j = 0; j < 3; ++j)

{

for (int i = 0; i < numSegments; ++i)

{

float theta1 = (float(i) / numSegments) \* 2.0f \* 3.14159265f;

float theta2 = (float(i + 1) / numSegments) \* 2.0f \* 3.14159265f;

Vector3 start, end;

switch (j)

{

case 0: // XY plane

start = Vector3(TransformedCenter.m\_x + radius \* cos(theta1), TransformedCenter.m\_y + radius \* sin(theta1), TransformedCenter.m\_z);

end = Vector3(TransformedCenter.m\_x + radius \* cos(theta2), TransformedCenter.m\_y + radius \* sin(theta2), TransformedCenter.m\_z);

break;

case 1: // XZ plane

start = Vector3(TransformedCenter.m\_x + radius \* cos(theta1), TransformedCenter.m\_y, TransformedCenter.m\_z + radius \* sin(theta1));

end = Vector3(TransformedCenter.m\_x + radius \* cos(theta2), TransformedCenter.m\_y, TransformedCenter.m\_z + radius \* sin(theta2));

break;

case 2: // YZ plane

start = Vector3(TransformedCenter.m\_x, TransformedCenter.m\_y + radius \* cos(theta1), TransformedCenter.m\_z + radius \* sin(theta1));

end = Vector3(TransformedCenter.m\_x, TransformedCenter.m\_y + radius \* cos(theta2), TransformedCenter.m\_z + radius \* sin(theta2));

break;

}

linepts[iPt++] = start;

linepts[iPt++] = DebugRenderColor;

linepts[iPt++] = end;

linepts[iPt++] = DebugRenderColor;

}

}

bool hasTransform = true;

DebugRenderer::Instance()->createLineMesh(

hasTransform,

m\_worldTransform,

&linepts[0].m\_x,

numPts,

0.f);

PhysicsShape::DebugRender();

}

AABB Sphere::calculateAABB()

{

Vector3 min = TransformedCenter - Vector3(radius, radius, radius); // 最小点

Vector3 max = TransformedCenter + Vector3(radius, radius, radius); // 最大点

return AABB(min, max);

}

void Sphere::UpdatePosition(float deltaTime)

{

if (!EnablePhysics)return;

// 更新球心位置

TransformedCenter += velocity \* deltaTime;

// 更新世界变换矩阵（如果有）

// 根据新的中心位置更新 m\_worldTransform

SetPosition(TransformedCenter);

}

void Sphere::UpdateRotation(float deltaTime)

{

// 更新旋转

float angularSpeed = angularVelocity.length();

// 更新旋转

if (angularSpeed > EPSILON)

{

// 计算旋转轴和角度

Vector3 axis = angularVelocity.normalized();

float angle = angularSpeed \* deltaTime;

// 围绕轴旋转角度

m\_base.turnAboutAxis(angle, axis);

// 规范化基向量，保持正交性和单位长度

m\_base.normalizeUVN();

}

}

Vector3 Sphere::ComputeCollisionNormal(const Vector3& collisionPoint)

{

return(collisionPoint - GetPosition()).normalized();

}

void Sphere::UpdateInverseInertiaTensorWorld()

{

// 假设球体的质量和半径已知

float mass = this->mass;

float radius = this->radius;

// 计算惯性张量的标量部分

float inertiaScalar = (2.0f / 5.0f) \* mass \* radius \* radius;

// 计算惯性张量的逆标量

float inverseInertiaScalar = (inertiaScalar != 0.0f) ? (1.0f / inertiaScalar) : 0.0f;

// 构建逆惯性张量矩阵（单位矩阵乘以逆标量）

Matrix3x3 inverseInertiaTensor;

inverseInertiaTensor.setIdentity();

inverseInertiaTensor = inverseInertiaTensor \* inverseInertiaScalar;

// 对于球体，局部和世界惯性张量是相同的

this->inverseInertiaTensorWorld = inverseInertiaTensor;

}

void Sphere::do\_CALCULATE\_TRANSFORMATIONS(Events::Event\* pEvt)

{

//if (!ReadyToCollide)SetPosition(m\_base.getPos() + Vector3(0, 0, radius));

PhysicsShape::do\_CALCULATE\_TRANSFORMATIONS(pEvt);

TransformedCenter = m\_worldTransform \* center;

ReadyToCollide = true;

if (m\_isTransformDirty)

{

//m\_isTransformDirty = false;

}

}

void Sphere::do\_PHYSICS\_START(Events::Event\* pEvt)

{

PhysicsShape::do\_PHYSICS\_START(pEvt);

if (!EnablePhysics)return;

}

}

}

#include "PhysicsShape.h"

#include "PrimeEngine/Lua/LuaEnvironment.h"

#include "PrimeEngine/Events/StandardEvents.h"

#include "PrimeEngine/APIAbstraction/APIAbstractionDefines.h"

#include "PrimeEngine/Scene/DebugRenderer.h"

using namespace PE::Events;

namespace PE

{

namespace Components

{

PE\_IMPLEMENT\_CLASS1(PhysicsShape, Component);

PhysicsShape::PhysicsShape(PE::GameContext& context, PE::MemoryArena arena, Handle hMyself)

:Component(context, arena, hMyself),

mass(1.0f),

velocity(Vector3(0, 0, 0)),

angularVelocity(Vector3(0,0,0)),

acceleration(Vector3(0, 0, 0)),

force(Vector3(0, 0, 0)),

restitution(0.5f),

friction(0.5f)

{

}

void PhysicsShape::OnOverlap(PhysicsShape\* OtherShape, Vector3 CollidePoint, const float& deltaTime)

{

if (!EnablePhysics)return;

DebugRenderColor = Vector3((rand() % 255) / 255.0f, (rand() % 255) / 255.0f, (rand() % 255) / 255.0f);

}

void PhysicsShape::OnOverlap(PhysicsShape\* OtherShape)

{

if (OtherShape->PhysicsShapeType != this->PhysicsShapeType)

{

DebugRenderColor = Vector3(1.f, 0.f, 0.f);

}

}

void PhysicsShape::DebugRender()

{

AABB myAABB = getAABB();

Vector3 AABBLineMin = myAABB.min;

Vector3 AABBLineMax = myAABB.max;

Vector3 AABBLine[4] = { AABBLineMin ,Vector3(1,1,1),AABBLineMax ,Vector3(1,1,1) };

DebugRenderer::Instance()->createLineMesh(

false,

m\_worldTransform,

&AABBLine[0].m\_x,

2,

0.f);

}

AABB PhysicsShape::getAABB()

{

m\_cachedAABB = calculateAABB();

if (m\_isTransformDirty)

{

}

return m\_cachedAABB;

}

void PhysicsShape::addDefaultComponents()

{

PE\_REGISTER\_EVENT\_HANDLER(Events::Event\_MOVE, PhysicsShape::do\_MOVE);

PE\_REGISTER\_EVENT\_HANDLER(Events::Event\_PHYSICS\_START, PhysicsShape::do\_PHYSICS\_START);

PE\_REGISTER\_EVENT\_HANDLER(Events::Event\_CALCULATE\_TRANSFORMATIONS, PhysicsShape::do\_CALCULATE\_TRANSFORMATIONS);

}

void PhysicsShape::SetPosition(Vector3& newPosition)

{

Matrix4x4& m = m\_base;

m.setPos(newPosition);

m\_isTransformDirty = true;

}

void ResolveCollision(PhysicsShape\* shapeA, PhysicsShape\* shapeB, const Vector3& collisionPoint, const Vector3& collisionNormal, const float& deltaTime)

{

}

void PhysicsShape::ApplyForce(const Vector3& newForce)

{

force += newForce;

}

void PhysicsShape::do\_MOVE(Events::Event\* pEvt)

{

Events::Event\_MOVE\* pRealEvent = (Events::Event\_MOVE\*)(pEvt);

Matrix4x4& m = m\_base;

m.setPos(m.getPos() + pRealEvent->m\_dir);

m\_isTransformDirty = true;

}

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();

}

void PhysicsShape::do\_PHYSICS\_START(Events::Event\* pEvt)

{

if (!EnablePhysics || !IsDynamic)return;

Event\_PHYSICS\_START\* pRealEvent = (Event\_PHYSICS\_START\*)(pEvt);

float deltaTime = pRealEvent->m\_frameTime;

if (mass > 0 && EnableGravity && !isOnGround)

{

Vector3 gravityForce = Vector3(0, -9.81f \* mass, 0); // 重力加速度为 9.81 m/s^2

Vector3 torque = Vector3(0,0,0).crossProduct(gravityForce);

angularVelocity += inverseInertiaTensorWorld \* torque \* deltaTime;

force += gravityForce;

}

ApplyForce(force);

// 计算加速度：a = F / m

acceleration = force / mass;

// 更新速度：v = v0 + a \* dt

velocity += acceleration \* deltaTime;

// 更新位置：x = x0 + v \* dt

// 对于 Sphere 和 Box，需要更新它们的中心或位置

UpdatePosition(deltaTime);

// 清除作用力，准备下一帧

force = Vector3(0, 0, 0);

UpdateRotation(deltaTime);

// 重置接触状态

isOnGround = false;

}

}

}