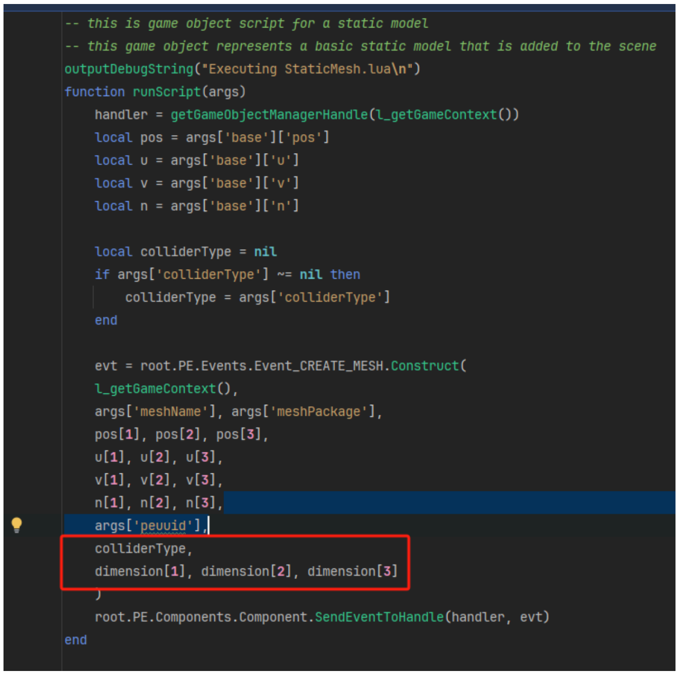
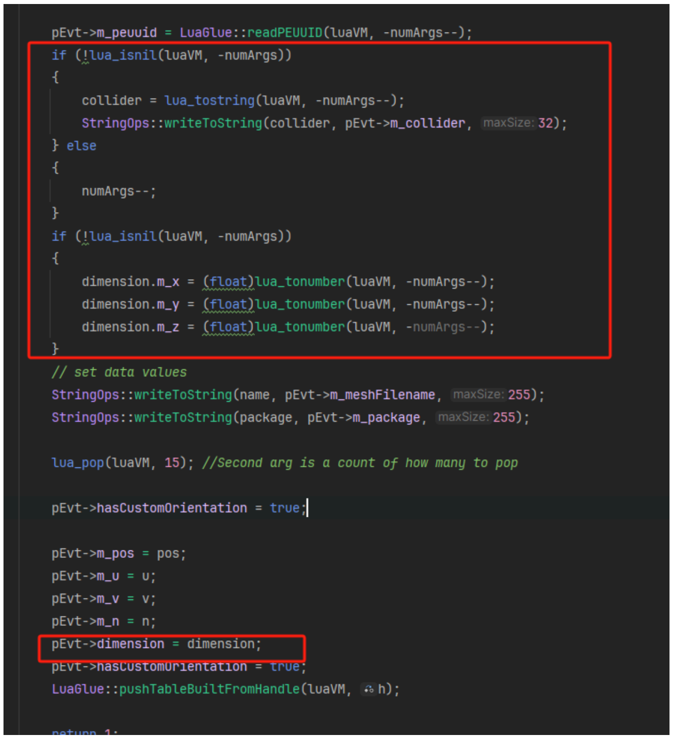
Video Link:
https://drive.google.com/file/d/1whxoY08qiknsIxLjXkzS6fCTwWo8Vgeo/view?usp=sharing

I first modified Lua environment for Event_CREATE_MESH, I added a colliderType variable to specify what collider to add to the mesh



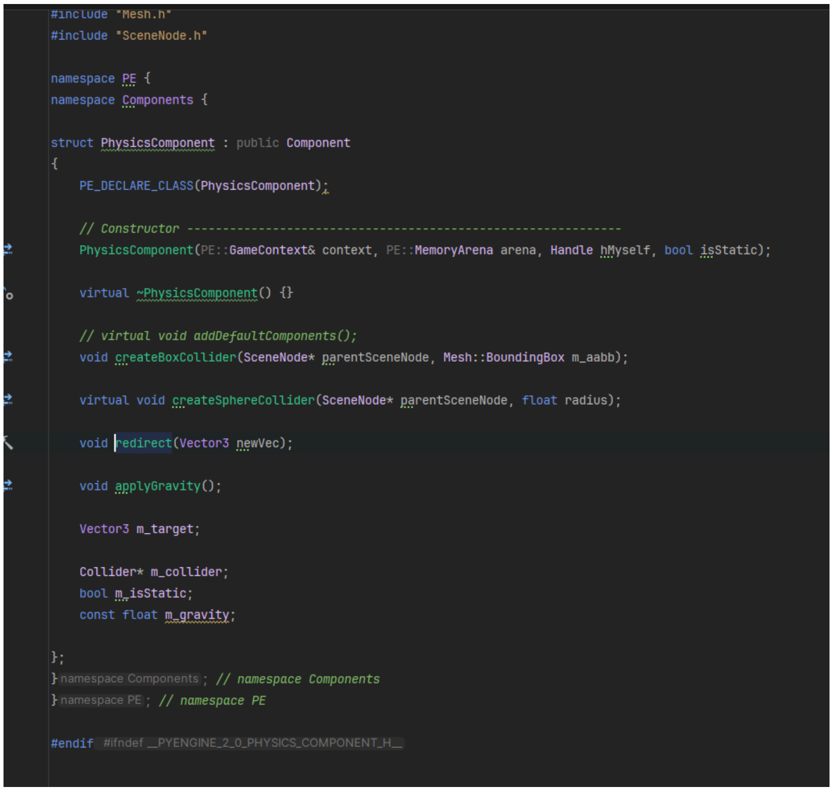
```lua
-- this is game object script for a static model
-- this game object represents a basic static model that is added to the scene
outputDebugString("Executing StaticMesh.lua\n")
function runScript(args)
    handler = getGameObjectManagerHandle(l_getGameContext())
    local pos = args['base']['pos']
    local u = args['base']['u']
    local v = args['base']['v']
    local n = args['base']['n']

    local colliderType = nil
    if args['colliderType'] ~= nil then
        colliderType = args['colliderType']
    end

    evt = root.PE.Events.Event_CREATE_MESH.Construct(
    l_getGameContext(),
    args['meshName'], args['meshPackage'],
    pos[1], pos[2], pos[3],
    u[1], u[2], u[3],
    v[1], v[2], v[3],
    n[1], n[2], n[3],
    args['peuuid'],
    colliderType,
    dimension[1], dimension[2], dimension[3]
    )
    root.PE.Components.Component.SendEventToHandle(handler, evt)
end
```



```cpp
pEvt->m_peuuid = LuaGlue::readPEUUID(luaVM, -numArgs--);
if (!lua_isnil(luaVM, -numArgs))
{
    collider = lua_tostring(luaVM, -numArgs--);
    StringOps::writeToString(collider, pEvt->m_collider, maxSize: 32);
} else
{
    numArgs--;
}
if (!lua_isnil(luaVM, -numArgs))
{
    dimension.m_x = (float)lua_tonumber(luaVM, -numArgs--);
    dimension.m_y = (float)lua_tonumber(luaVM, -numArgs--);
    dimension.m_z = (float)lua_tonumber(luaVM, -numArgs--);
}
// set data values
StringOps::writeToString(name, pEvt->m_meshFilename, maxSize: 255);
StringOps::writeToString(package, pEvt->m_package, maxSize: 255);

lua_pop(luaVM, 15); //Second arg is a count of how many to pop

pEvt->hasCustomOrientation = true;

pEvt->m_pos = pos;
pEvt->m_u = u;
pEvt->m_v = v;
pEvt->m_n = n;
pEvt->dimension = dimension;
pEvt->hasCustomOrientation = true;
LuaGlue::pushTableBuiltFromHandle(luaVM, h);

return 1;
```

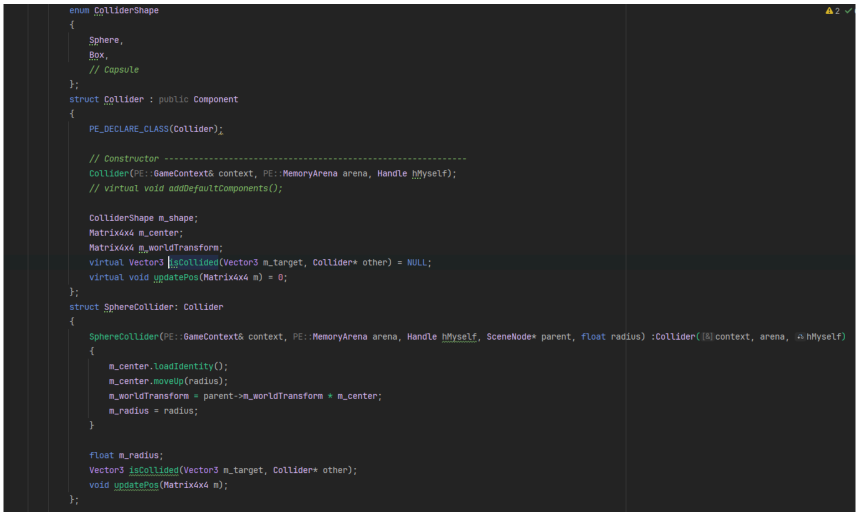
Inside GameObjectManager::do_CREATE_MESH, I added a default physical component creator when no custom collider is specified

I created an Entity-component style physics system, where a physics component may have different physical properties, but currently only a collider.

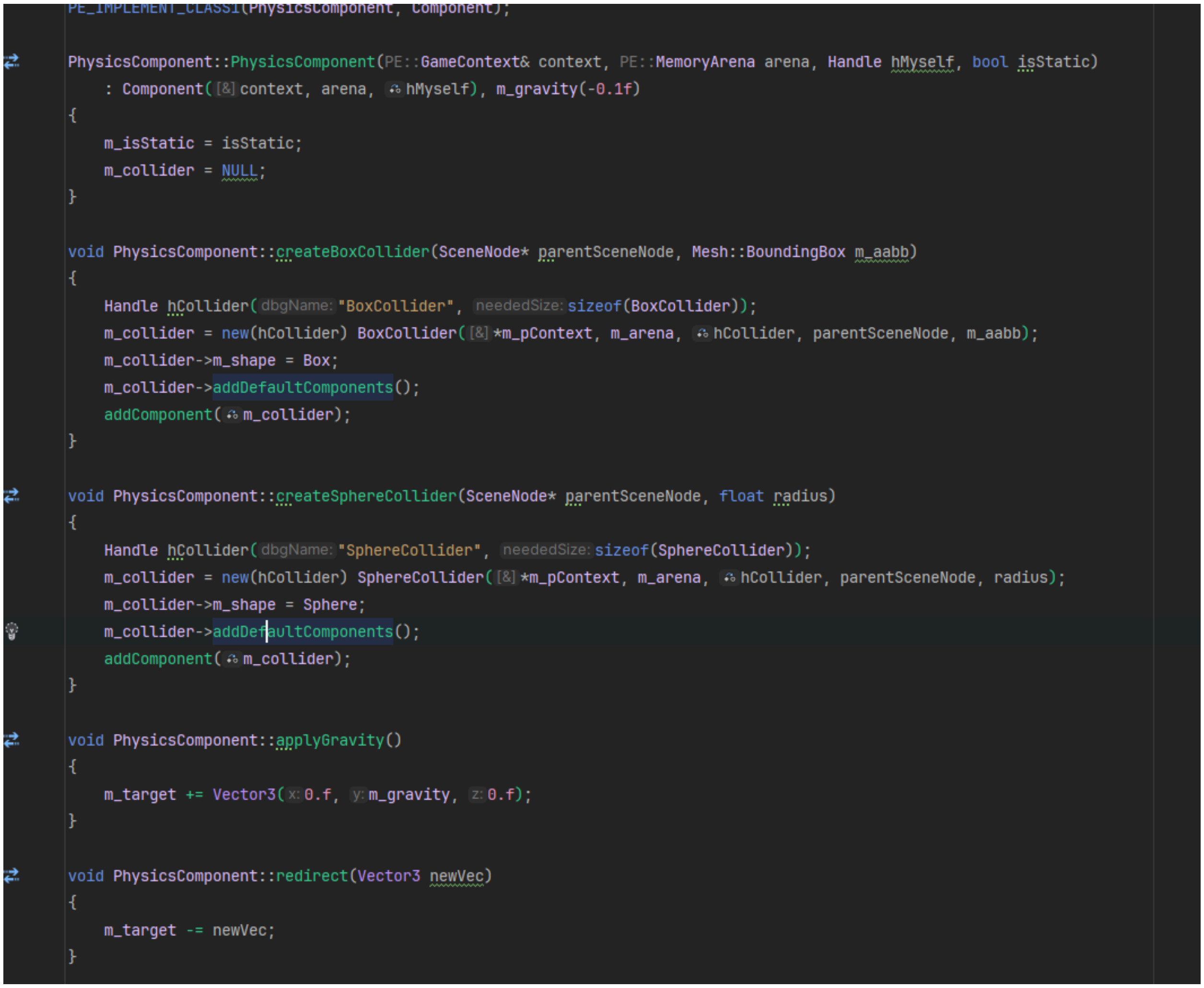


```cpp
#include "Mesh.h"
#include "SceneNode.h"

namespace PE {
namespace Components {

struct PhysicsComponent : public Component
{
    PE_DECLARE_CLASS(PhysicsComponent);

    // Constructor -------------------------------------------------------------
    PhysicsComponent(PE::GameContext& context, PE::MemoryArena arena, Handle hMyself, bool isStatic);

    virtual ~PhysicsComponent() {}

    // virtual void addDefaultComponents();
    void createBoxCollider(SceneNode* parentSceneNode, Mesh::BoundingBox m_aabb);

    virtual void createSphereCollider(SceneNode* parentSceneNode, float radius);

    void redirect(Vector3 newVec);

    void applyGravity();

    Vector3 m_target;

    Collider* m_collider;
    bool m_isStatic;
    const float m_gravity;

};
}namespace Components ; // namespace Components
}namespace PE ; // namespace PE

#endif #ifndef __PYENGINE_2_0_PHYSICS_COMPONENT_H__
```

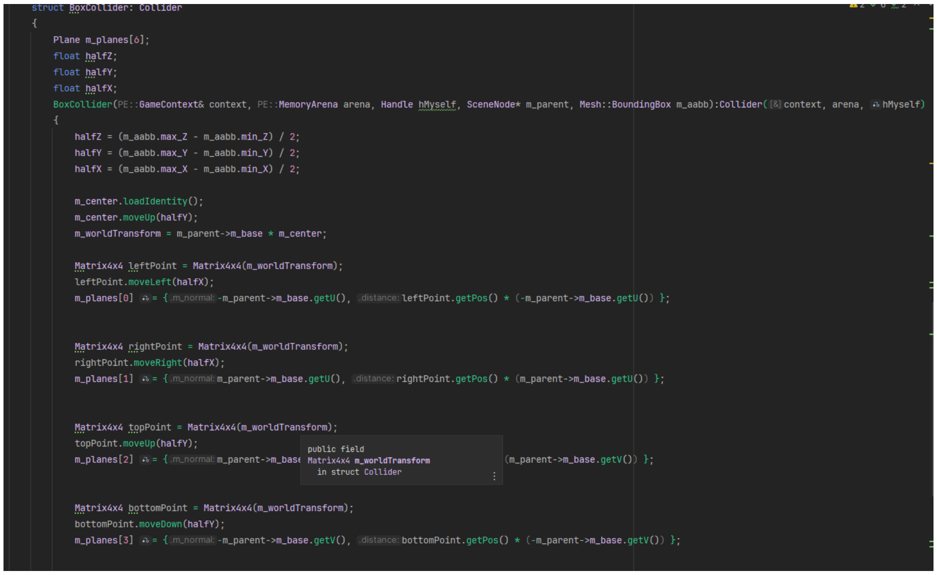
I created a parent class of collider and a BoxCollider and a SphereCollider that inherits the Collider class. For easier future calculations, BoxCollider is defined by 6 planes generated by the bounding box data defined in assignment 3, and sphere collider is defined by a radius. Both colliders have a center and world transform data for updating collider data.Colliders are created by PhysicsComponent when initializing game objects.



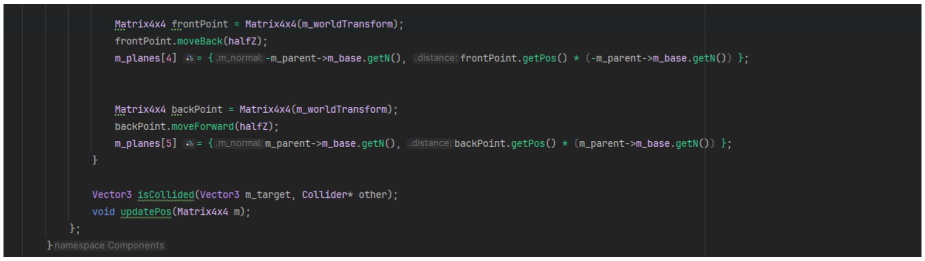
```cpp
enum ColliderShape
{
    Sphere,
    Box,
    // Capsule
};
struct Collider : public Component
{
    PE_DECLARE_CLASS(Collider);

    // Constructor -------------------------------------------------------------
    Collider(PE::GameContext& context, PE::MemoryArena arena, Handle hMyself);
    // virtual void addDefaultComponents();

    ColliderShape m_shape;
    Matrix4x4 m_center;
    Matrix4x4 m_worldTransform;
    virtual Vector3 isCollided(Vector3 m_target, Collider* other) = NULL;
    virtual void updatePos(Matrix4x4 m) = 0;
};
struct SphereCollider: Collider
{
    SphereCollider(PE::GameContext& context, PE::MemoryArena arena, Handle hMyself, SceneNode* parent, float radius) :Collider([&]context, arena, hMyself)
    {
        m_center.loadIdentity();
        m_center.moveUp(radius);
        m_worldTransform = parent->m_worldTransform * m_center;
        m_radius = radius;
    }

    float m_radius;
    Vector3 isCollided(Vector3 m_target, Collider* other);
    void updatePos(Matrix4x4 m);
};
```



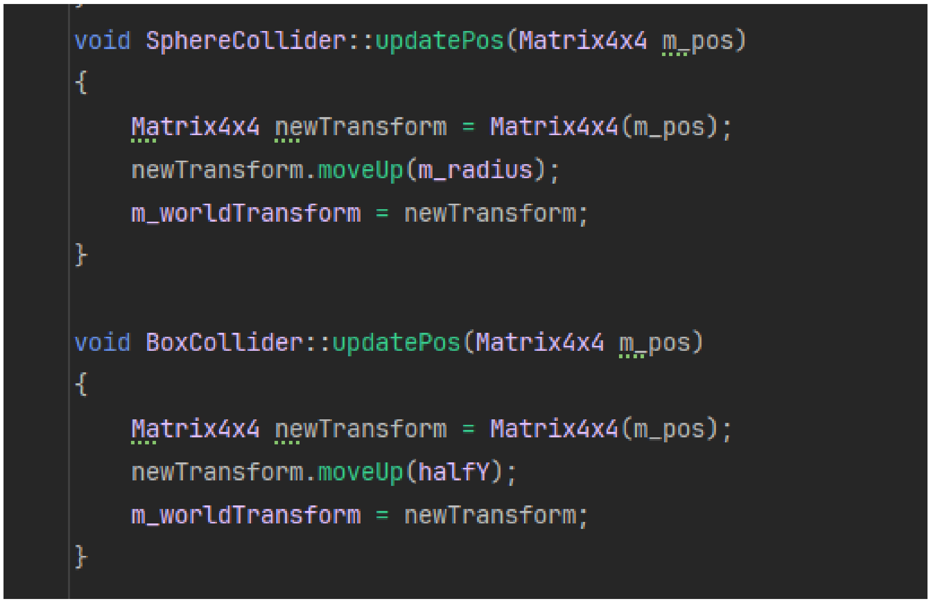
```cpp
PE_IMPLEMENT_CLASS1(PhysicsComponent, Component);

PhysicsComponent::PhysicsComponent(PE::GameContext& context, PE::MemoryArena arena, Handle hMyself, bool isStatic)
    : Component([&]context, arena, hMyself), m_gravity(-0.1f)
{
    m_isStatic = isStatic;
    m_collider = NULL;
}

void PhysicsComponent::createBoxCollider(SceneNode* parentSceneNode, Mesh::BoundingBox m_aabb)
{
    Handle hCollider(dbgName: "BoxCollider", neededSize: sizeof(BoxCollider));
    m_collider = new(hCollider) BoxCollider([&]*m_pContext, m_arena, hCollider, parentSceneNode, m_aabb);
    m_collider->m_shape = Box;
    m_collider->addDefaultComponents();
    addComponent(m_collider);
}

void PhysicsComponent::createSphereCollider(SceneNode* parentSceneNode, float radius)
{
    Handle hCollider(dbgName: "SphereCollider", neededSize: sizeof(SphereCollider));
    m_collider = new(hCollider) SphereCollider([&]*m_pContext, m_arena, hCollider, parentSceneNode, radius);
    m_collider->m_shape = Sphere;
    m_collider->addDefaultComponents();
    addComponent(m_collider);
}

void PhysicsComponent::applyGravity()
{
    m_target += Vector3(x: 0.f, y: m_gravity, z: 0.f);
}

void PhysicsComponent::redirect(Vector3 newVec)
{
    m_target -= newVec;
}
```



```
struct BoxCollider: Collider
{
    Plane m_planes[6];
    float halfZ;
    float halfY;
    float halfX;
    BoxCollider(PE::GameContext& context, PE::MemoryArena arena, Handle hMyself, SceneNode* m_parent, Mesh::BoundingBox m_aabb):Collider([&]context, arena, hMyself)
    {
        halfZ = (m_aabb.max_Z - m_aabb.min_Z) / 2;
        halfY = (m_aabb.max_Y - m_aabb.min_Y) / 2;
        halfX = (m_aabb.max_X - m_aabb.min_X) / 2;

        m_center.loadIdentity();
        m_center.moveUp(halfY);
        m_worldTransform = m_parent->m_base * m_center;

        Matrix4x4 leftPoint = Matrix4x4(m_worldTransform);
        leftPoint.moveLeft(halfX);
        m_planes[0] = {.m_normal: -m_parent->m_base.getU(), .distance: leftPoint.getPos() * (-m_parent->m_base.getU()) };


        Matrix4x4 rightPoint = Matrix4x4(m_worldTransform);
        rightPoint.moveRight(halfX);
        m_planes[1] = {.m_normal: m_parent->m_base.getU(), .distance: rightPoint.getPos() * (m_parent->m_base.getU()) };


        Matrix4x4 topPoint = Matrix4x4(m_worldTransform);
        topPoint.moveUp(halfY);
        m_planes[2] = {.m_normal: m_parent->m_base                                (m_parent->m_base.getV()) };


        Matrix4x4 bottomPoint = Matrix4x4(m_worldTransform);
        bottomPoint.moveDown(halfY);
        m_planes[3] = {.m_normal: -m_parent->m_base.getV(), .distance: bottomPoint.getPos() * (-m_parent->m_base.getV()) };
```

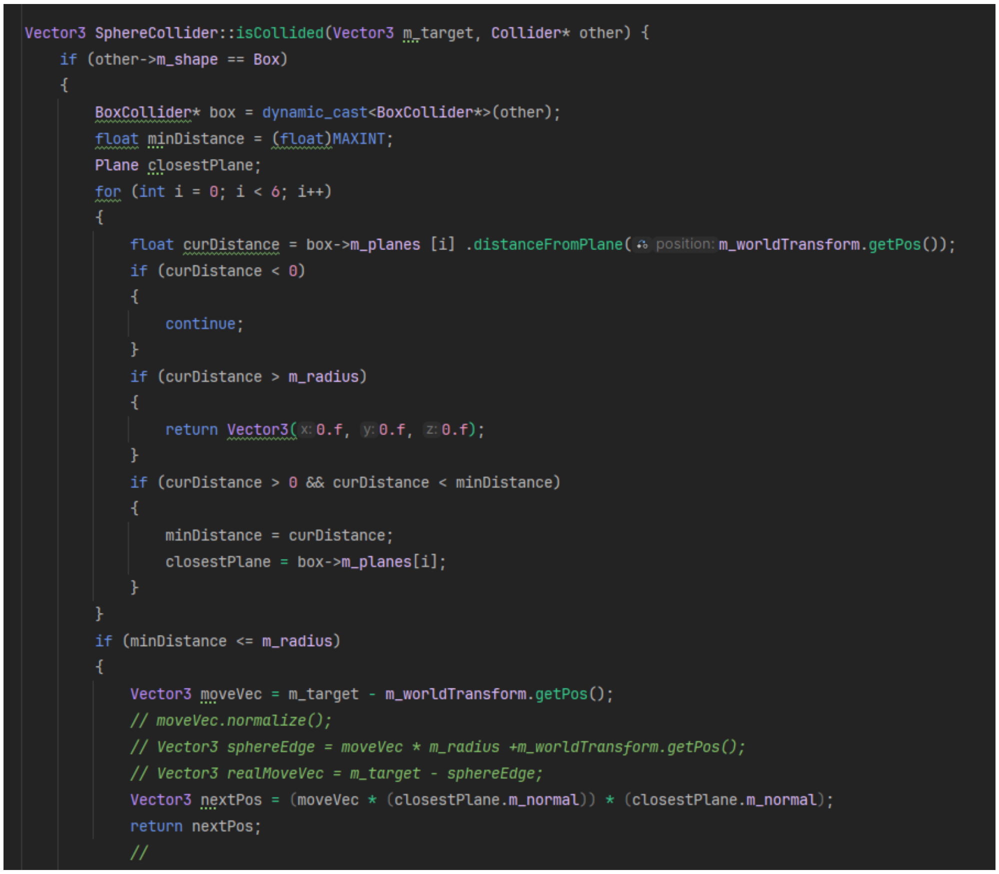


```
        Matrix4x4 frontPoint = Matrix4x4(m_worldTransform);
        frontPoint.moveBack(halfZ);
        m_planes[4] = {.m_normal: -m_parent->m_base.getN(), .distance: frontPoint.getPos() * (-m_parent->m_base.getN()) };


        Matrix4x4 backPoint = Matrix4x4(m_worldTransform);
        backPoint.moveForward(halfZ);
        m_planes[5] = {.m_normal: m_parent->m_base.getN(), .distance: backPoint.getPos() * (m_parent->m_base.getN()) };
    }

    Vector3 isCollided(Vector3 m_target, Collider* other);
    void updatePos(Matrix4x4 m);
};
}namespace Components
```

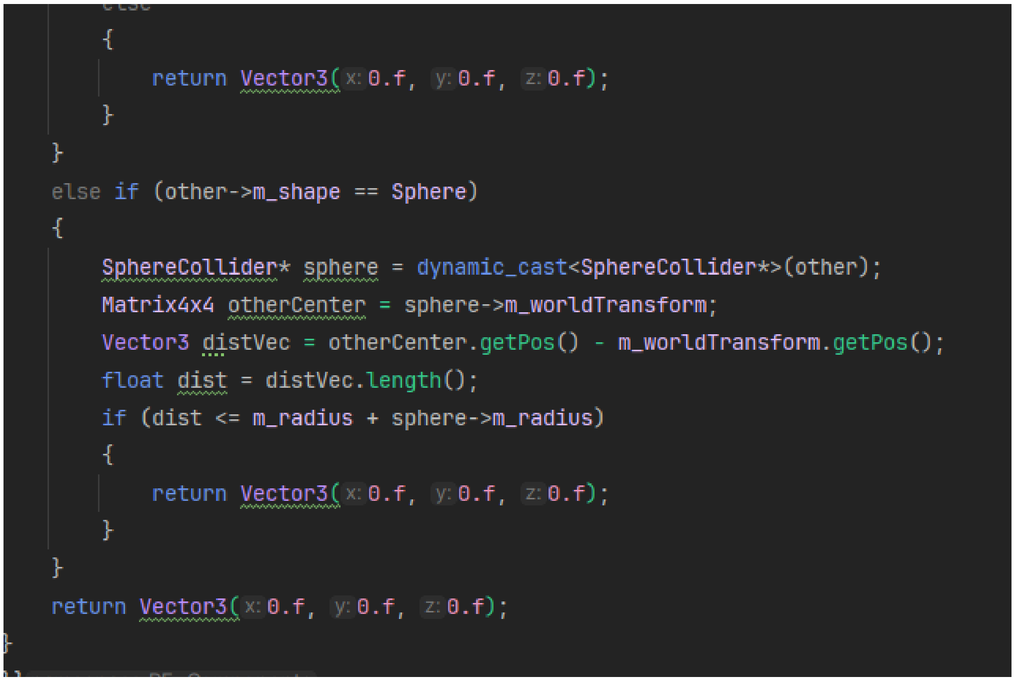


```
void SphereCollider::updatePos(Matrix4x4 m_pos)
{
    Matrix4x4 newTransform = Matrix4x4(m_pos);
    newTransform.moveUp(m_radius);
    m_worldTransform = newTransform;
}

void BoxCollider::updatePos(Matrix4x4 m_pos)
{
    Matrix4x4 newTransform = Matrix4x4(m_pos);
    newTransform.moveUp(halfY);
    m_worldTransform = newTransform;
}
```

In the collision detection method, I used a plane-sphere collision detection method and calculated the redirection offset for each collided object.

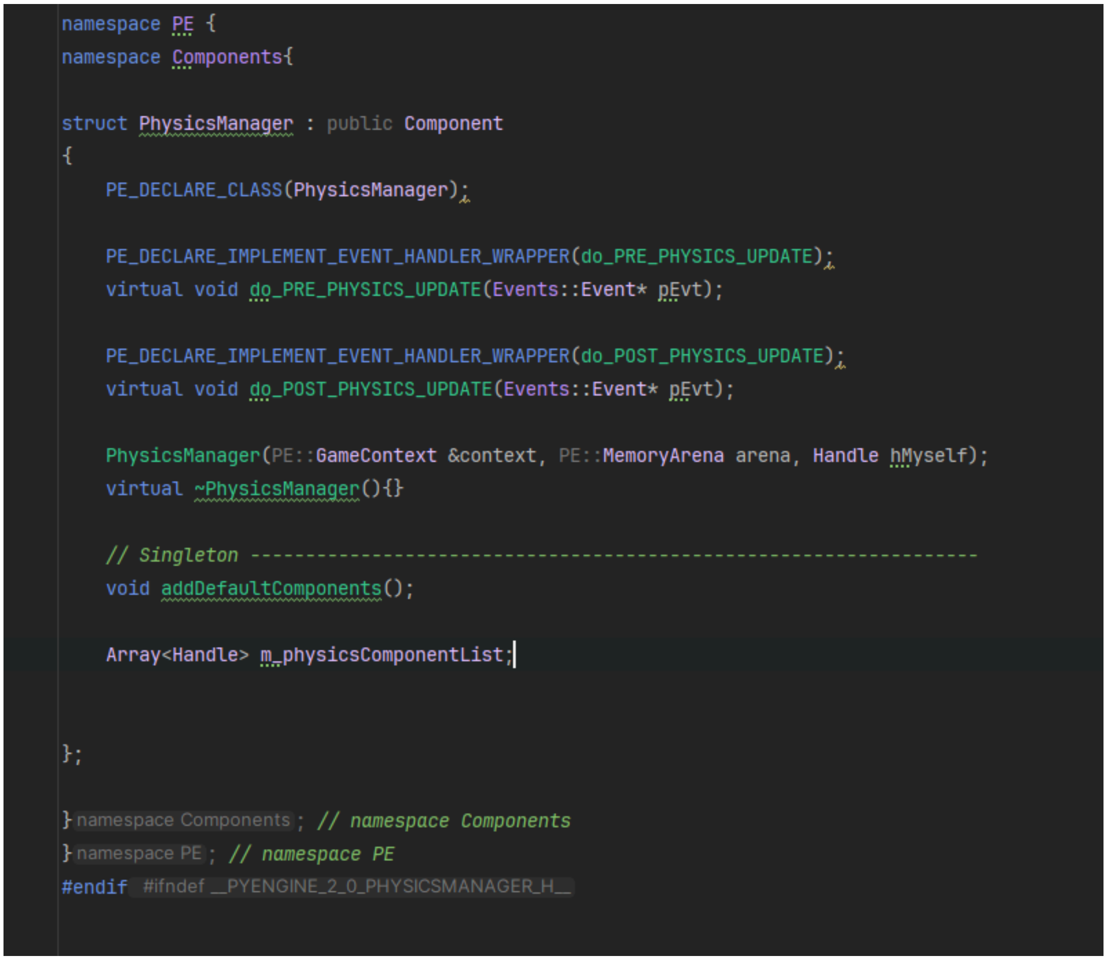


```
Vector3 SphereCollider::isCollided(Vector3 m_target, Collider* other) {
    if (other->m_shape == Box)
    {
        BoxCollider* box = dynamic_cast<BoxCollider*>(other);
        float minDistance = (float)MAXINT;
        Plane closestPlane;
        for (int i = 0; i < 6; i++)
        {
            float curDistance = box->m_planes [i] .distanceFromPlane( position: m_worldTransform.getPos());
            if (curDistance < 0)
            {
                continue;
            }
            if (curDistance > m_radius)
            {
                return Vector3( x: 0.f,  y: 0.f,  z: 0.f);
            }
            if (curDistance > 0 && curDistance < minDistance)
            {
                minDistance = curDistance;
                closestPlane = box->m_planes[i];
            }
        }
        if (minDistance <= m_radius)
        {
            Vector3 moveVec = m_target - m_worldTransform.getPos();
            // moveVec.normalize();
            // Vector3 sphereEdge = moveVec * m_radius +m_worldTransform.getPos();
            // Vector3 realMoveVec = m_target - sphereEdge;
            Vector3 nextPos = (moveVec * (closestPlane.m_normal)) * (closestPlane.m_normal);
            return nextPos;
            //
```



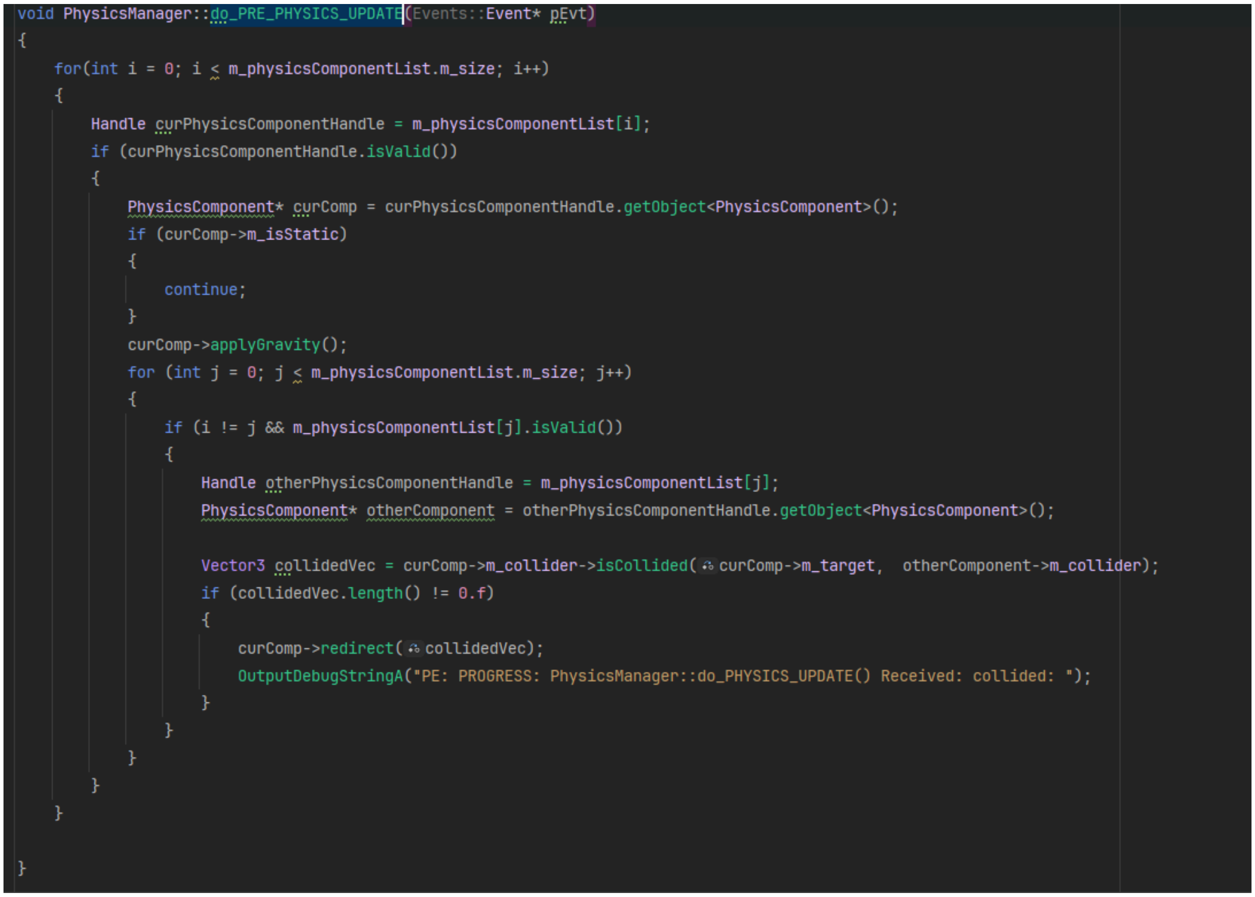
```
        else
        {
            return Vector3( x: 0.f,  y: 0.f,  z: 0.f);
        }
    }
    else if (other->m_shape == Sphere)
    {
        SphereCollider* sphere = dynamic_cast<SphereCollider*>(other);
        Matrix4x4 otherCenter = sphere->m_worldTransform;
        Vector3 distVec = otherCenter.getPos() - m_worldTransform.getPos();
        float dist = distVec.length();
        if (dist <= m_radius + sphere->m_radius)
        {
            return Vector3( x: 0.f,  y: 0.f,  z: 0.f);
        }
    }
    return Vector3( x: 0.f,  y: 0.f,  z: 0.f);
}
```

PhysicsManger simply keeps a list of physical components, and calculates all interaction between them during Pre_Physics_Update and Post_Pysics_Update event.



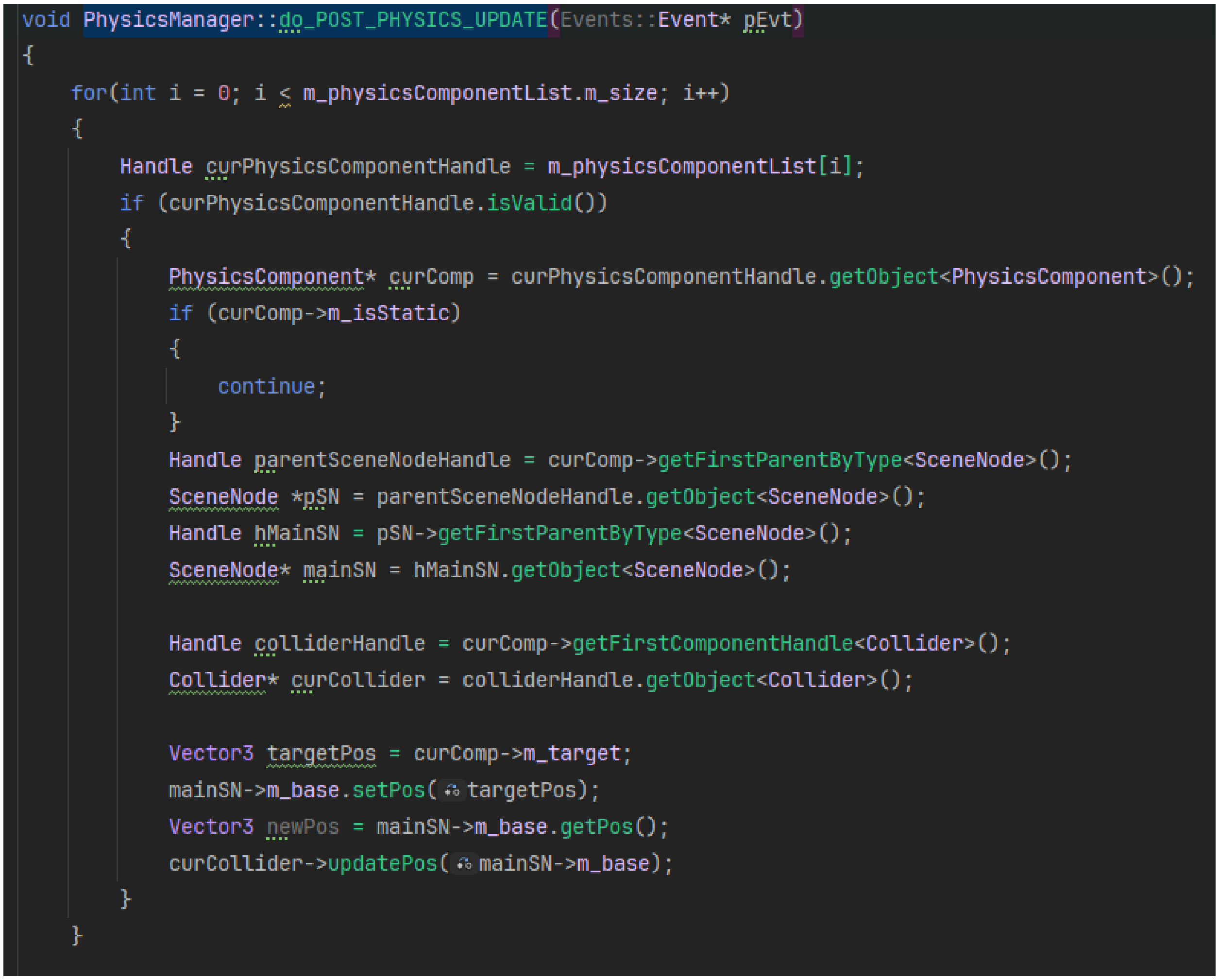
```
namespace PE {
namespace Components{

struct PhysicsManager : public Component
{
    PE_DECLARE_CLASS(PhysicsManager);

    PE_DECLARE_IMPLEMENT_EVENT_HANDLER_WRAPPER(do_PRE_PHYSICS_UPDATE);
    virtual void do_PRE_PHYSICS_UPDATE(Events::Event* pEvt);

    PE_DECLARE_IMPLEMENT_EVENT_HANDLER_WRAPPER(do_POST_PHYSICS_UPDATE);
    virtual void do_POST_PHYSICS_UPDATE(Events::Event* pEvt);

    PhysicsManager(PE::GameContext &context, PE::MemoryArena arena, Handle hMyself);
    virtual ~PhysicsManager(){}

    // Singleton ------------------------------------------------------------------
    void addDefaultComponents();

    Array<Handle> m_physicsComponentList;


};

} namespace Components ; // namespace Components
} namespace PE ; // namespace PE
#endif #ifndef __PYENGINE_2_0_PHYSICSMANAGER_H__
```

Every calculation is triggered in physicsManager::do_PRE_PHYSICS_UPDATE. For every dynamic physics component, apply gravity first. Given the current target to move, calculate and redirect the target position when collided.



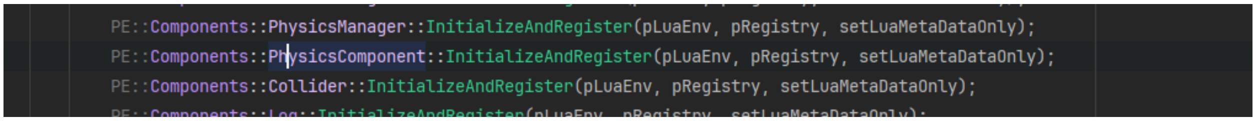
```
void PhysicsManager::do_PRE_PHYSICS_UPDATE(Events::Event* pEvt)
{
    for(int i = 0; i < m_physicsComponentList.m_size; i++)
    {
        Handle curPhysicsComponentHandle = m_physicsComponentList[i];
        if (curPhysicsComponentHandle.isValid())
        {
            PhysicsComponent* curComp = curPhysicsComponentHandle.getObject<PhysicsComponent>();
            if (curComp->m_isStatic)
            {
                continue;
            }
            curComp->applyGravity();
            for (int j = 0; j < m_physicsComponentList.m_size; j++)
            {
                if (i != j && m_physicsComponentList[j].isValid())
                {
                    Handle otherPhysicsComponentHandle = m_physicsComponentList[j];
                    PhysicsComponent* otherComponent = otherPhysicsComponentHandle.getObject<PhysicsComponent>();

                    Vector3 collidedVec = curComp->m_collider->isCollided(curComp->m_target,  otherComponent->m_collider);
                    if (collidedVec.length() != 0.f)
                    {
                        curComp->redirect(collidedVec);
                        OutputDebugStringA("PE: PROGRESS: PhysicsManager::do_PHYSICS_UPDATE() Received: collided: ");
                    }
                }
            }
        }
    }

}
```

In PhysicsManager::do_POST_PHYSICS_UPDATE, update the new transform and center position after updating the parent sceneNode position.

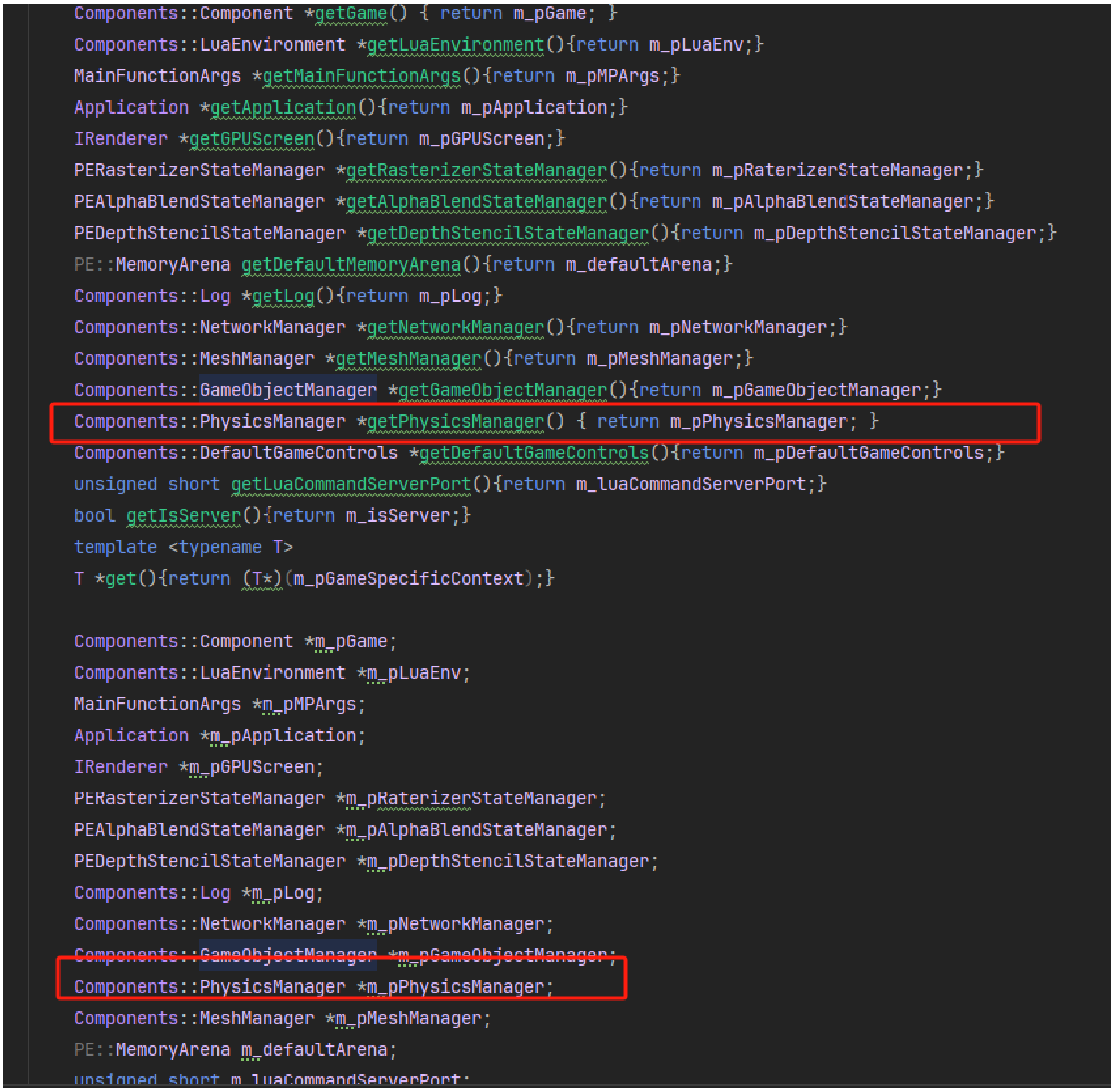


```
void PhysicsManager::do_POST_PHYSICS_UPDATE(Events::Event* pEvt)
{
    for(int i = 0; i < m_physicsComponentList.m_size; i++)
    {
        Handle curPhysicsComponentHandle = m_physicsComponentList[i];
        if (curPhysicsComponentHandle.isValid())
        {
            PhysicsComponent* curComp = curPhysicsComponentHandle.getObject<PhysicsComponent>();
            if (curComp->m_isStatic)
            {
                continue;
            }
            Handle parentSceneNodeHandle = curComp->getFirstParentByType<SceneNode>();
            SceneNode *pSN = parentSceneNodeHandle.getObject<SceneNode>();
            Handle hMainSN = pSN->getFirstParentByType<SceneNode>();
            SceneNode* mainSN = hMainSN.getObject<SceneNode>();

            Handle colliderHandle = curComp->getFirstComponentHandle<Collider>();
            Collider* curCollider = colliderHandle.getObject<Collider>();

            Vector3 targetPos = curComp->m_target;
            mainSN->m_base.setPos(targetPos);
            Vector3 newPos = mainSN->m_base.getPos();
            curCollider->updatePos(mainSN->m_base);
        }
    }
```

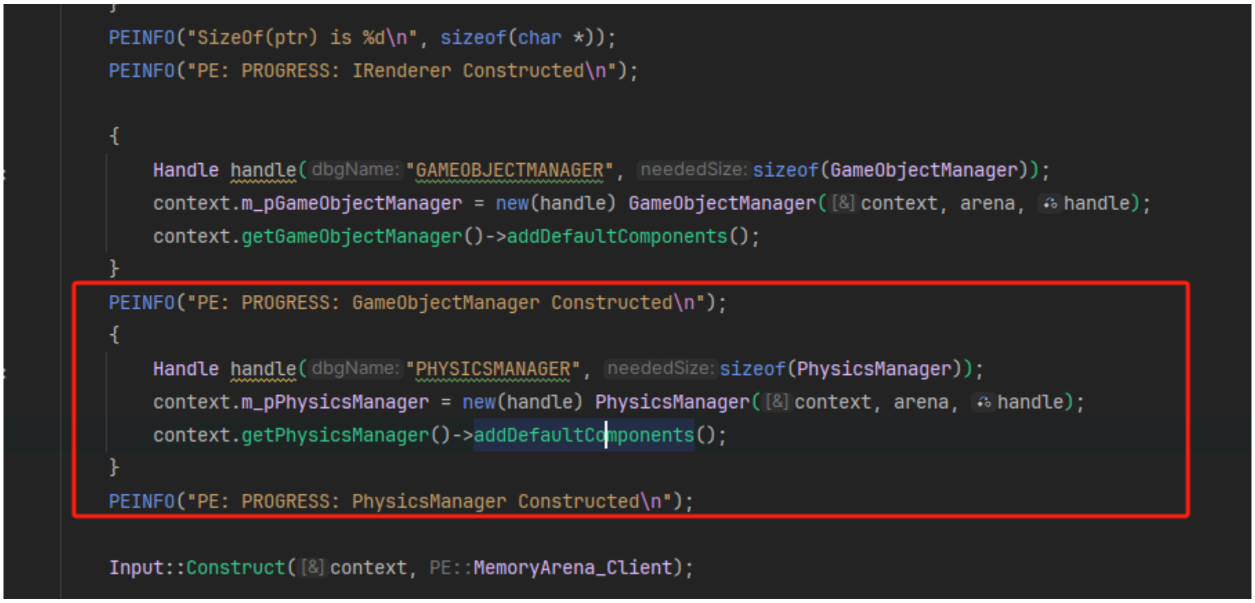
After creating all physical elements used in the system, register these components and add PhysicsManager to the GameContext during ClientGame initialization.



```
PE::Components::PhysicsManager::InitializeAndRegister(pLuaEnv, pRegistry, setLuaMetaDataOnly);
PE::Components::PhysicsComponent::InitializeAndRegister(pLuaEnv, pRegistry, setLuaMetaDataOnly);
PE::Components::Collider::InitializeAndRegister(pLuaEnv, pRegistry, setLuaMetaDataOnly);
PE::Components::Log::InitializeAndRegister(pLuaEnv, pRegistry, setLuaMetaDataOnly);
```



```
Components::Component *getGame() { return m_pGame; }
Components::LuaEnvironment *getLuaEnvironment(){return m_pLuaEnv;}
MainFunctionArgs *getMainFunctionArgs(){return m_pMPArgs;}
Application *getApplication(){return m_pApplication;}
IRenderer *getGPUScreen(){return m_pGPUScreen;}
PERasterizerStateManager *getRasterizerStateManager(){return m_pRaterizerStateManager;}
PEAlphaBlendStateManager *getAlphaBlendStateManager(){return m_pAlphaBlendStateManager;}
PEDepthStencilStateManager *getDepthStencilStateManager(){return m_pDepthStencilStateManager;}
PE::MemoryArena getDefaultMemoryArena(){return m_defaultArena;}
Components::Log *getLog(){return m_pLog;}
Components::NetworkManager *getNetworkManager(){return m_pNetworkManager;}
Components::MeshManager *getMeshManager(){return m_pMeshManager;}
Components::GameObjectManager *getGameObjectManager(){return m_pGameObjectManager;}
Components::PhysicsManager *getPhysicsManager() { return m_pPhysicsManager; }
Components::DefaultGameControls *getDefaultGameControls(){return m_pDefaultGameControls;}
unsigned short getLuaCommandServerPort(){return m_luaCommandServerPort;}
bool getIsServer(){return m_isServer;}
template <typename T>
T *get(){return (T*)(m_pGameSpecificContext);}

Components::Component *m_pGame;
Components::LuaEnvironment *m_pLuaEnv;
MainFunctionArgs *m_pMPArgs;
Application *m_pApplication;
IRenderer *m_pGPUScreen;
PERasterizerStateManager *m_pRaterizerStateManager;
PEAlphaBlendStateManager *m_pAlphaBlendStateManager;
PEDepthStencilStateManager *m_pDepthStencilStateManager;
Components::Log *m_pLog;
Components::NetworkManager *m_pNetworkManager;
Components::GameObjectManager *m_pGameObjectManager;
Components::PhysicsManager *m_pPhysicsManager;
Components::MeshManager *m_pMeshManager;
PE::MemoryArena m_defaultArena;
unsigned short m_luaCommandServerPort;
```



```
    }
    PEINFO("SizeOf(ptr) is %d\n", sizeof(char *));
    PEINFO("PE: PROGRESS: IRenderer Constructed\n");

    {
        Handle handle( dbgName: "GAMEOBJECTMANAGER",  neededSize: sizeof(GameObjectManager));
        context.m_pGameObjectManager = new(handle) GameObjectManager( [&] context, arena,  handle);
        context.getGameObjectManager()->addDefaultComponents();
    }
    PEINFO("PE: PROGRESS: GameObjectManager Constructed\n");
    {
        Handle handle( dbgName: "PHYSICSMANAGER",  neededSize: sizeof(PhysicsManager));
        context.m_pPhysicsManager = new(handle) PhysicsManager( [&] context, arena,  handle);
        context.getPhysicsManager()->addDefaultComponents();
    }
    PEINFO("PE: PROGRESS: PhysicsManager Constructed\n");

    Input::Construct( [&] context, PE::MemoryArena_Client);
```

Finally, in the npc state machine SoldierNPCMovementSM::do_UPDATE, modify the setPos step for each npc. Make sure that the new target position is passed into the physical component for physics system calculation.