Step1: PhysicsComponent and PhysicsManager

PhysicsComponent:

```
struct PhysicsComponent : public Component {
PE_DECLARE_CLASS(PhysicsComponent);
protected:
public:
    MeshInstance* m_pMeshInstance;
    PhysicsComponent(PE::GameContext& context, PE::MemoryArena arena, Handle hMyself, MeshInstance* meshInstance);
    virtual ~PhysicsComponent() {}
};
```

Struct in the PhysicsManager Class

BoxCollider and SphereCollider inherits the PhysicsComponent

BoxCollider uses the Mesh's AABB as the boundingBox, the 6 planeNormals are used later in collision detection. And the planePointIndex are just used for a faster index reference. Other functions are used in calculating the distance between other sphereCollider.

```
struct BoxCollider : public PhysicsComponent {
    PE_DECLARE_CLASS(BoxCollider);
    //Matrix4x4 m_base;
    Vector3 m_corners[8];

    Vector3 m_planeNormals[6];
    float m_planePointIndices[6][4];
    float m_radius;

    BoxCollider(PE::GameContext& context, PE::MemoryArena arena, Handle hMyself, Matrix4x4 base, Array<Vector3> aabbs, MeshInstance* meshInstance);
    float getBoundingWidth() { return abs(m_corners[0].getX() - m_corners[4].getX()); }
    float getBoundingHeight() { return abs(m_corners[0].getY() - m_corners[3].getY()); }
    float getBoundingDepth() { return abs(m_corners[0].getZ() - m_corners[1].getZ()); }
    Vector3 getCenter();

};
```

Instantiate BoxCollider in GameObjectManager

```
// Create static box collider for meshes that is not soldier
Mesh* pMesh = pMeshInstance->m_hAsset.getObject<Mesh>();

PhysicsManager* pPhysicsManager = m_pContext->getPhysicsManager();


Handle hBoxCollider("BOX_COLLIDER", sizeof(BoxCollider));
BoxCollider* pBoxCollider = new(hBoxCollider) BoxCollider(*m_pContext, m_arena, hBoxCollider, pSN->m_base, pMesh->m_AABB, pMeshInstance);

pPhysicsManager->addStaticCollider(pBoxCollider);
pMeshInstance->addComponent(hBoxCollider);
```

SphereCollider

It's center position and radius of the sphere.

```
struct SphereCollider : public PhysicsComponent {
    PE_DECLARE_CLASS(SphereCollider);
    Vector3 m_center;
    float m_radius;
    Vector3 m_offset;
    SphereCollider(PE::GameContext& context, PE::MemoryArena arena, Handle hMyself, Vector3 center, float radius, MeshInstance* meshInstance);
};
```

Instantiate sphereCollider in Soldier class

```
    PE::Handle hSphereCollider("SPHERE_COLLIDER", sizeof(SphereCollider));
    SphereCollider* pSphereCollider = new(hSphereCollider)SphereCollider(*m_pContext, m_arena, hSphereCollider, pMainSN->m_base.getPos(), 0.5f, pMeshInstance);

    pPhysicsManager->addDynamicCollider(pSphereCollider);
    pMeshInstance->addComponent(hSphereCollider);
```

Update the Soldier position in SoldierMovementSM:

```
// instantaneous turn
pSN->m_base.turnInDirection(dir, 3.1415f);

Vector3 targetPos = curPos + dir * dist;
//Update the collider's position Update pos by physics
ClientGameObjectManagerAddon* pGameObjectManagerAddon = (ClientGameObjectManagerAddon*)(m_pContext->get<CharacterControlContext>()->getGameObjectManagerAddon());

if (pGameObjectManagerAddon) {
    SoldierNPC* pSol = getFirstParentByTypePtr<SoldierNPC>();
    PE::Components::SphereCollider* pCollider = pGameObjectManagerAddon->getSoldierNPCCollider(pSol);
    assert(pCollider);

    PhysicsManager* pPhysicsManager = m_pContext->getPhysicsManager();
    Vector3 resPos;
    pPhysicsManager->MoveCollider(pCollider, targetPos, resPos);

    pSN->m_base.setPos(resPos);
}
```

PhysicsManager Class
Stores lists of dynamic(sphere) and static(box) colliders in the scene.
addCollider functions are called in other classes when the object is instantiated.

MoveCollider is used to do collision detection between the Box and Sphere.

```
struct PhysicsManager : public Component
{
    PE_DECLARE_CLASS(PhysicsManager);
    // Singleton ---------------------------------------------------------------
    // Constructor -------------------------------------------------------------
    PhysicsManager(PE::GameContext& context, PE::MemoryArena arena, Handle hMyself);
    virtual ~PhysicsManager() {}
    // Methods      ------------------------------------------------------------
    void addStaticCollider(BoxCollider* collider);
    void addDynamicCollider(SphereCollider* collider);

    const std::vector<BoxCollider*>& getStaticColliders() { return m_staticColliders; };
    const std::vector<SphereCollider*>& getDynamicColliders() { return m_dynamicColliders; };

    void MoveCollider(SphereCollider* collider, Vector3 targetPos, Vector3& resPos);
    // Component ---------------------------------------------------------------
    virtual void addDefaultComponents();
    // Individual events -------------------------------------------------------
private:
    std::vector<BoxCollider*> m_staticColliders;
    std::vector<SphereCollider*> m_dynamicColliders;
};
```

MoveCollider:

Check the distance between the sphere collider with all static collider in the scene, if it's too far away just skip it.

```
void PhysicsManager::MoveCollider(SphereCollider* collider, Vector3 targetPos, Vector3& resPos)
{
    // For test, just move down
    // simulate gravity
    targetPos.m_y -= 0.03;

    // Avoid collision with floor
    Vector3 originColliderPos = collider->m_center + collider->m_offset;
    Vector3 targetColliderPos = targetPos + collider->m_offset;

    for (auto* staticCollider : m_staticColliders) {
        // Too far away, ignore
        if ((originColliderPos - staticCollider->getCenter()).length() > staticCollider->m_radius + collider->m_radius) {
            continue;
        }
```

The code then verifies if the sphere is intersecting the plane by checking whether the sphere's center-to-plane distance is less than its radius. It then calculates the intersection point on the plane by subtracting the plane's normal (multiplied by that distance) from the sphere center.

```
    for (int i = 0; i < 6; i++) {
        int index0 = staticCollider->m_planePointIndices[i][0];
        int index1 = staticCollider->m_planePointIndices[i][1];
        int index2 = staticCollider->m_planePointIndices[i][2];
        int index3 = staticCollider->m_planePointIndices[i][3];

        Vector3 normal = staticCollider->m_planeNormals[i];
        Vector3 point = staticCollider->m_corners[index0];

        // the mesh maybe plane with no y normal
        if (normal.length() != 1) {
            continue;
        }

        // Ignore if move away from plane
        if (normal.dotProduct(moveDir) >= 0) {
            continue;
        }

        float distance = (targetColliderPos - point).dotProduct(normal);

        if (distance < 0) {
            continue;
        }
```

Next, it uses cross-product checks on each edge of the quadrilateral to confirm the intersection point is inside the shape. Finally, if the point is indeed inside, the code concludes a collision and shifts the sphere's position along the plane's normal to resolve the overlap.

```
        if (distance < collider->m_radius) {
            Vector3 intersetionPoint = targetColliderPos - normal * distance;
            Vector3 v0 = staticCollider->m_corners[index0];
            Vector3 v1 = staticCollider->m_corners[index1];
            Vector3 v2 = staticCollider->m_corners[index2];
            Vector3 v3 = staticCollider->m_corners[index3];

            Vector3 cross0 = (v1 - v0).crossProduct(intersetionPoint - v0);
            Vector3 cross1 = (v2 - v1).crossProduct(intersetionPoint - v1);
            Vector3 cross2 = (v3 - v2).crossProduct(intersetionPoint - v2);
            Vector3 cross3 = (v0 - v3).crossProduct(intersetionPoint - v3);

            if (cross0.dotProduct(cross1) >= 0 &&
                cross1.dotProduct(cross2) >= 0 &&
                cross2.dotProduct(cross3) >= 0 &&
                cross3.dotProduct(cross0) >= 0) {
                // Collision
                //Vector3 backOffset = normal * (collider->m_radius - distance); // Move back
                Vector3 backOffset = normal * moveDir.dotProduct(normal);
                targetPos -= backOffset;
            }
        }
    }
```