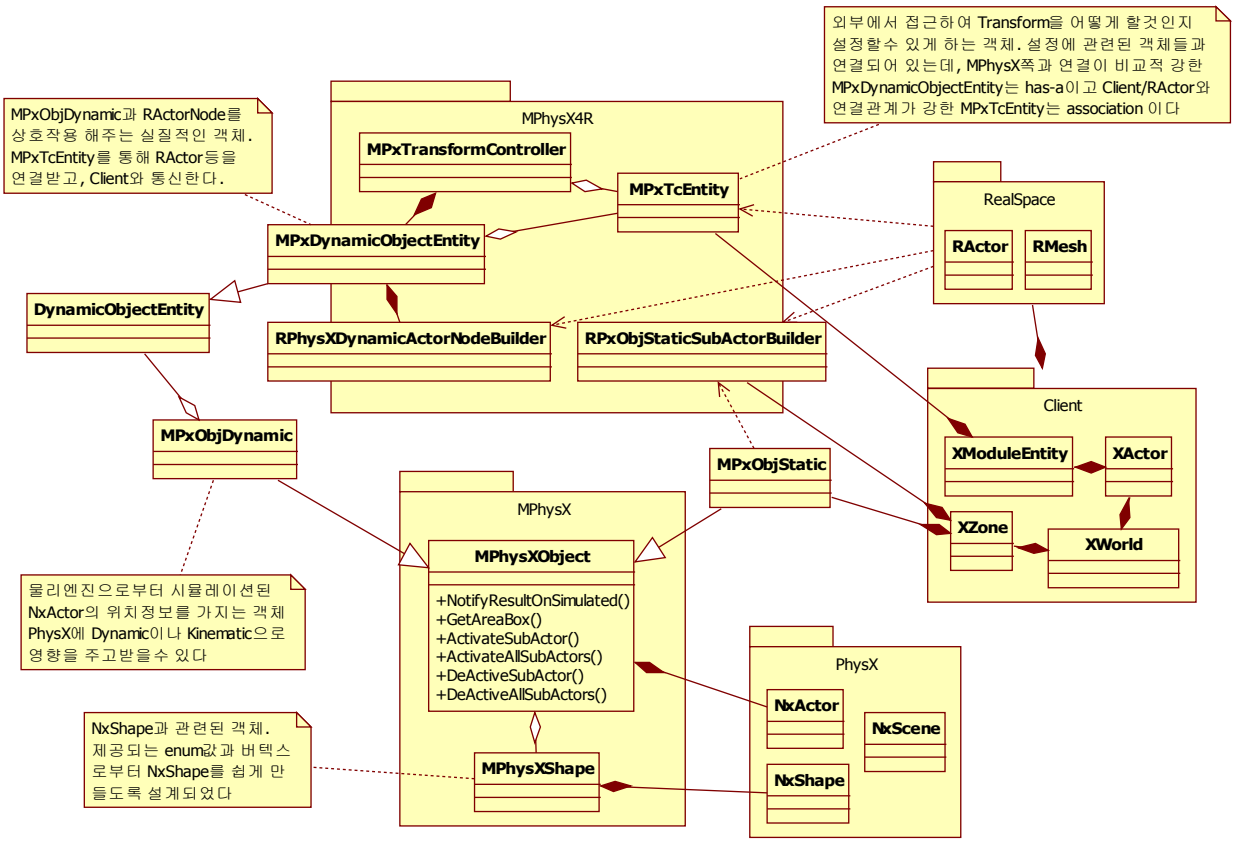
Physics Simulation for RaiderZ



MPhysX

- PhysX의 Wrapper / 싱글턴
- 물리 시뮬레이션에 관련된 Nx 객체 : NxScene, NxActor
- NxActor를 NxShape을 이용해 만든후 NxScene에 삽입, 이후 NxScene::simulate 호출
- NxScene::createActor 메쏘드를 호출시에 NxActor를 생성하면서 씬에 삽입하는 구조
- NxActor에 대응하는 MPhysX 객체 : MPhysXObject
- NxShpae에 대응하는 MPhysX 객체 : MphysXShape
- 충돌 바닥면등을 위한 MPxObjStatic과 브레이커블 파츠에 쓰이는 MpxObjDynamic
- 그외의 MPxObject는 현재 사용하는 곳이 없다.

RActorNode : RMeshNode = NxActor : NxShape

- 시뮬레이션된 NxActor 객체의 좌표는 RActorNode의 Transform에 대응
- NxActor가 사용하는 Shape은 RMeshNode에 대응
- RActor는 RActorNode들의 집합체(씬노드), RMesh는 RMeshNode들의 집합체(리소스)

Notification

- 충돌여부등의 이벤트를 수신할 객체 필요
- RealSpace가 이러한 notification을 받을 이유가 없다
- MPhysX로부터 notification을 수신할 객체 필요.
- MPhysX와 RealSpace를 연결해줄 객체도 필요

MPhysX4R

- Builder
  - RealSapce 객체로 부터 MPhysX(PhysX)에 적합한 객체 생성
  - MPxObjDynamic : RPhysXDynamicActorNodeBuilder

```
RPhysXDynamicActorNodeBuilder builder;
...
// 빌더에더하기
builder.AddActorNodeInfo(
        new RphysXDynamicActorNodeInfo(
                m_ActorNodeInfos.m_vecActorNodesList[i], pNxActor ) );
...
// building
m_pPhysXActorController->BuildNxObject(&builder);
```

  - MPxObjStatic : RPxObjStaticSubActorBuilder

```
RPxObjStaticSubActorBuilder rs3MeshBuilder;
rs3MeshBuilder.SetTerrain( pTerrain, nDivisionCount );


...
// 주변10미터를하나의섹터단위로한다.
rs3MeshBuilder.SetCollisionTreeRootNode(
GetCollisionTree()->GetRootNode(), 10000.f);

// 피직스스태틱바디빌드
m_pPhysXStatic->AddStaticSubActorsFromBuilder(rs3MeshBuilder);

// 현재위치에서지연생성을하면랙에쩔수있으니까한번활성화해준다.
MBox vCurrentBox( vMyPos - 5000.f, vMyPos + 5000.f );
std::vector< const MBox* > vList;
vList.push_back(&vCurrentBox);
m_pPhysXStatic->ActivateStaticSubActorsByAreaList(vList);
```

- TransformController (통칭, TC)
  - RealSpace/Client 와 MPhysX간의 통신을 위한 객체
  - 실제적으로 통신이 필요한 MPxObjDynamic과 연계
  - 두가지 타입의 Entity
    - Client/RActor와 강한 관계에 있는 MPxTcEntity
    - MPxObjDynamic과 연결이 강한 MPxDynamicObjectEntity
  - TransformController는 클라이언트가 가진다

```
XModuleEntity::XModuleEntity(XObject* pOwner)
: Xmodule(pOwner)
...
{
        ...
        pTc = new MPxDynamicActorNodeTC(m_pTcEntity);
        MPxDynamicActorNodeTC::SetID(nTcID++);
        m_vecTransformControllers.push_back(pTc);
        ...
```

  - MPxTcEntity도 클라이언트가 가지고 TC들과 연결지어진다

```
XModuleEntity::XModuleEntity(XObject* pOwner)
: Xmodule(pOwner)
...
{
        ...
        class XPxTcEntity : public MPxTcEntity
        {
        public:
                XPxTcEntity(XModuleEntity* pOwnerEntity) :
                                                m_pOwnerEntity(pOwnerEntity) {}
                virtual ~XPxTcEntity(){}

                virtual rs3::RActor* GetActor()
                {
                        return m_pOwnerEntity->GetActor();
                }
        ...
        };

        m_pTcEntity = new XpxTcEntity(this);
        ...
```

- MPxDynamicObjectEntity
  - MPxDynamicObjectEntity는 TC가 가지고 MPxTcEntity들과 연결지어진다.
  - MPxDynamicObjectEntity는 MPxObjDynamic::DynamicObjectEntity를 상속
  - MPxDynamicObjectEntity는 MPxObjDynamic으로부터 호출되어진다.

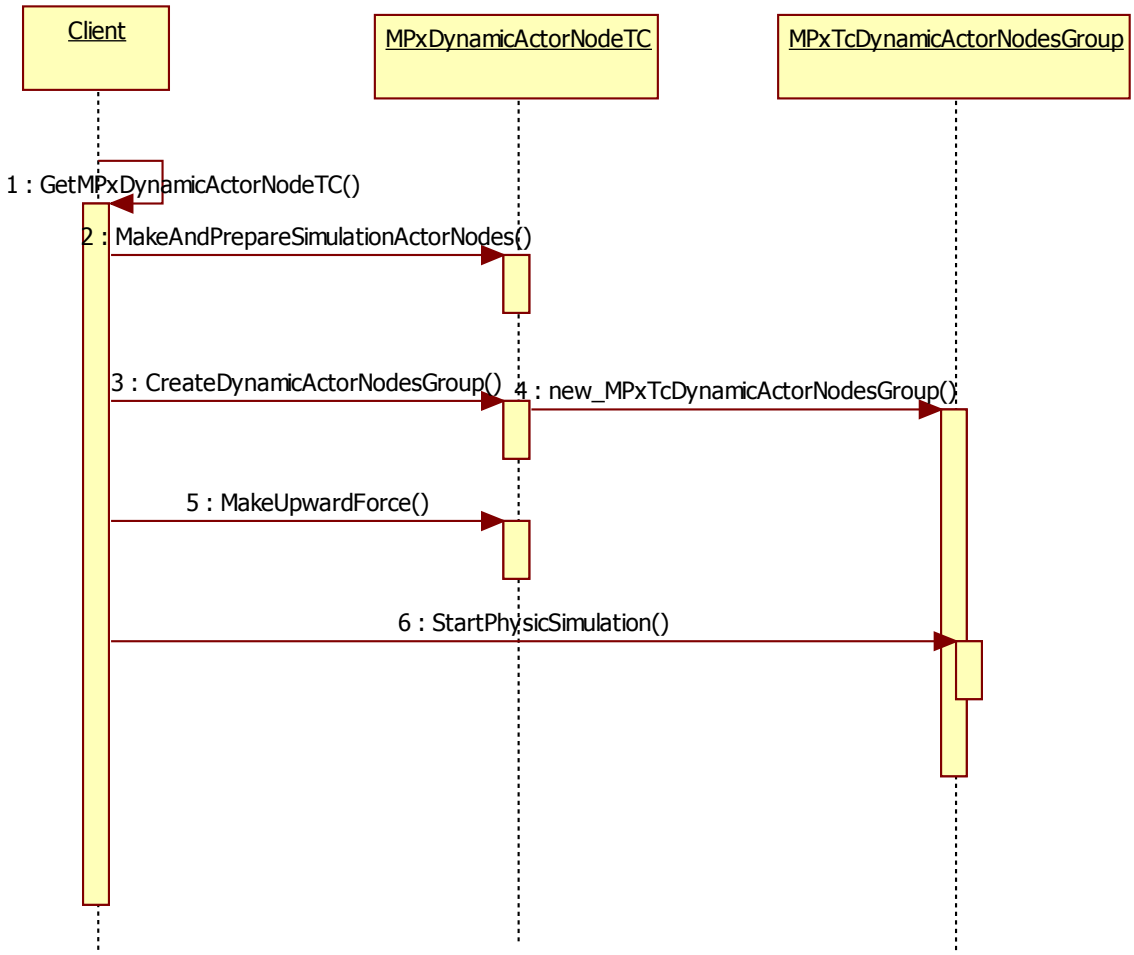
```
void MPxObjDynamic::ActivateSubActorsAll()
{
	ActivateAllSubActors();
	if (m_pDynamicObjectEntity)
	{
		_ASSERT(m_pPhysX);
		MMatrix matWorld;
		m_pDynamicObjectEntity->OnActivatedSubActorsAll(matWorld);
		DoSetWorldTransform(matWorld);
	}
	DoUpdateTransform();
	m_bActive = true;
}
...
void MPxObjDynamic::NotifyResultOnSimulated()
{
	// Entity 와관련된Notify 시작
	if (NULL == m_pDynamicObjectEntity)
		return;
	if(!m_bObservingResultOnSimulated)
		return;

	// SubActor 단위Notify
	const list<int>& listUsedHandle = m_NxInfos.GetUsedHandleList();
	for (list<int>::const_iterator itr = listUsedHandle.begin();
		itr != listUsedHandle.end(); ++itr)
	{
		NX_INFO& info = m_NxInfos.Get( *itr );
		if (NULL == info.m_pActor)
			continue;
		m_pDynamicObjectEntity->OnSubActorSimulated(info.m_pActor);
	}

	// PxObject (Dynamic) 단위Notify
	m_pDynamicObjectEntity->OnSimulated();
}
```

Breakable Parts

- Get MPxDynamicActorNodeTC / Prepare simulation
- Create MPxDynamicActorNodeGroup(파츠 번호에 대응) / Start physics simulation




```
void MPxTcDynamicActorNodesGroup::StartPhysicSimulation( const std::vector<std::string>& _rActorNodeName, const vec3& _rForce,
                                                         const vec3& _rForceWorldPosition, const char* _pSzDeletionChunck)
{
        StopPhysicSimulation();

        m_pPxObjectDynamicBody = MPhysX::GetInstance()->CreatePxObject<physx::MPxObjDynamic>();
        m_pDynamicBodyEntity->ResetDynamicBodyEntity(m_pActor, _rActorNodeName, _pSzDeletionChunck);
        m_pPxObjectDynamicBody->RegisterDynamicObjectEntity(m_pDynamicBodyEntity);
        if (m_pPxObjectDynamicBody->AddSubActorFromEntity()) {
                // set collision group
                bool bSelfGroupCollision = false;
                if (m_pOwnerActorNodeTC->IsLinkedAtDynamicActorNode()) bSelfGroupCollision = true;
                //int nCollisionGroup = MPhysX::GetInstance()->GetDefaultDynamicObjectCollisionGroupId(bSelfGroupCollision);
                int nCollisionGroup = MPhysX::GetInstance()->GetDefaultDynamicObjectCollisionGroupId(true);
                m_pDynamicBodyEntity->SetCollisionGroup(nCollisionGroup);

                // activate subactor
                m_pPxObjectDynamicBody->ActivateSubActorsAll();
                m_pPxObjectDynamicBody->ApplyForce(_rForce, _rForceWorldPosition);
        }else{
                mlog("failed adding shape in physx₩n");
                StopPhysicSimulation();
        }
}
```