Autonóm robotok és járművek laboratórium 1

3. mérés
Autonóm dinamikus rendszerek számítógépes grafikával támogatott megjelenítése

Mérési útmutató

A mérés során egy fizikai szimulációs szoftver-könyvtár (PhysX) segítségével modellezünk egy szabadon mozgó, illetve összekapcsolt merev testekből álló fizikai rendszert, valamint megjelenítjük az OpenGL grafikus könyvtár segítségével. Az operációs rendszer és az OpenGL összekötésében a GLUT könyvtár lesz a segítségünkre.

A következő feladatokat kell megoldanunk:
- grafikus ablak megnyitása és az OpenGL környezet létrehozása GLUT-tal [készen rendelkezésre fog állni]
- a GLUT callback függvényei segítségével a felhasználói- és rendszeresemények kezelése (billentyűzet és egér) [készen rendelkezésre fog állni]
- a virtuális kamera megvalósítása [készen rendelkezésre fog állni]
- a kamera vezérlésének megvalósítása [készen rendelkezésre fog állni]
- a virtuális világot (vagy színteret) reprezentáló heterogén adatszerkezet felépítése [a váza meglesz, a többi a mérésen megvalósítandó]
- a színtér objektumainak összekapcsolása a fizikai szimulációval [a mérésen megvalósítandó]
- a színtér objektumainak megjelenítése a fizikai szimuláció során számított helyzetben [a mérésen megvalósítandó]

Környezet

A fejlesztői környezet a Visual Studio 2003. Egy PhysXlab nevű solution-ben egy szintén PhysXlab nevű projecten fogunk dolgozni. Ez egy Win32 console application, vagyis a lehető legegyszerűbb C++ program, melynek belépési pontja a _tmain függvény a PhysXlab.cpp fileban.

Az OpenGL, GLUT, illetve PhysX könyvtárak a project beállításaiban, mint include, illetve linker input adatok, már szerepelnek.

Az alkalmazás osztály

A program belépési pontján kívül nincs globális függvény, objektum-orientáltan dolgozunk. A PhysXApp osztály, illetve annak egyetlen példánya fogja a globális szintű feladatokat ellátni (a Singleton nevű objektum-orientált tervezési mintát követve).

Minden osztályhoz tartozni fog egy .h kiterjesztésű header file és egy .cpp kiterjesztésű forrásfile.

A PhysXApp run metódusa egy statikus metódus, amely létrehoz egy PhysXApp példányt, majd átadja a vezérlést a GLUT-nak. A PhysXApp konstruktora regisztrálja GLUT callback függvényként a PhysXApp statikus eseménykezelő metódusait, melyek aztán az egyetlen példány nem statikus metódusait hívják. (Lásd a következő fejezetet a GLUT-ról.)

A GLUT

Az OpenGL grafikus könyvtár a virtuális világ primitívjeinek (ezek általában háromszögek) és a grafikus csővezeték állapotának (pl. kamera adatai) megadása után képes képet alkotni. (Részletesen ennek miként jéről az OpenGL részben). Hogy ezt a képet hova rakjuk a képernyőn, illetve mikor kell azt megrajzolni, az OpenGLnek nem feladata meghatározni. Ez többnyire az operációs rendszertől, illetve annak ablakozó alrendszerétől függ. Egy ablak tartalmát pl. akkor kell újrarajzolni, ha fölötte egy másik ablakot elhúztak. Másrészt közvetlen felhasználói eseményeket (egérklikk, billentyű-nyomás) is szeretnénk a programunkban kezelni. Ezek szintén oprendszer-függőek. Tehát vagy rendszerfüggő programot írunk (pl. a Windows üzenet-kezelését, a Windows API wgl függvényeit haszánljuk), vagy a GLUT könyvtárat, ami elfedi előlünk ezeket.

Pl. ablakot, amibe a későbbi OpenGL hívások majd rajzolhatnak, a következő GLUT hívássokkal nyithatunk:

```
glutInitWindowSize(600, 600);
glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
glutCreateWindow("PhysXLab");
```

Egy GLUT-os programban mindig szerepel a:

```
glutMainLoop();
```

Ez a függvény a program végéig soha sem tér vissza. Végtelen ciklusban dolgozza fel a felhasználói (egér, billentyű) és rendszereseményeket (ablak újrarajzolás, átméretezés). Ha van valamilyen esemény, akkor meghív egy **általunk írt függvényt**. Ez a callback. Tehát a programunk futása eseménykezelések egymásutánjából fog állni.

Ahhoz, hogy a GLUT tudja, milyen függvényt kell hívnia, függvénypointereket használ. Ezek globális, __cdecl hívási konvenciójú függvényekre mutatnak, tehát nem mutathatnak egy objektum-példány valamely metódusára. Ezért statikus metódusokat definiálunk (ami gyakorlatilag megegyezik egy globális függvénnyel):

```
static void __cdecl displayCallback();
```

Ezeket a statikus metódusokat állítjuk be a GLUT-nak:

```
glutDisplayFunc(displayCallback);
```

A statikus metódus pedig meghívhatja egy példány egy metódusát. A példában, ha az ablakot újra kell rajzolni, akkor a megjelenítési funkciót megvalósító render metódust hívjuk.

```
void PhysXApp::displayCallback()
{
	singletonApp->render();
}
```

Fontos még az idle callback. Ez akkor hívódik, ha épp nincs más feladat: ismétlődően, de rendszertelenül. Ebben végezhetjük el a fizikai szimuláció szükséges lépéseit, de figyelnünk kell arra, hogy mindig más lehet az eltelt idő két idle callback hívás között. A szimulációs

lépés végén a virtuális világ megváltozott, újra kell rajzolni a képet. Ezt a GLUT felé így jelezzük:

```
glutPostRedisplay();
```

Az OpenGL

A grafikus könyvtár feladata: egy virtuális világmodell (színtér) alapján valószerű képet alkotni. A világmodell olyan elemekből kell álljon, aminek megjelenítésére van egyszerű algoritmus: pont, vonal, háromszög. Bonyolultabb alakzatokat ezekre kell visszavezetni (ezt hívják tesszellációnak).

Egy modell-elem megadása a csúcspontjaival történik. Pl. egy háromszög három csúcsával:

```
glBegin(GL_TRIANGLES);
glVertex3d(0, 10, 0);
glVertex3d(1, 2, 1000);
glVertex3d(10, 1, 0);
glEnd();
```

Ezek azonban úgynevezett modellezési koordináták. A képernyőre az OpenGL nem ide fogja rajzolni a háromszöget, hanem megadhatjuk neki, hogy:

- Hol van a tárgy a virtuális világban? Pl. ha pörgő kockát rajzolunk, ne kelljen nekünk újraszámolni a koordinátákat. Ez a számítás a **modellezési transzformáció**. MODEL
- Hol van a kamerához képest? A kamera mozoghat. Ez a számítás a **nézeti transzformáció**. VIEW
- Hova kerül a képernyőn? Ez a perspektív vetítés. Ez a számítás a **vetítési transzformáció**. PROJECTION

A második két transzformációt a kamera jellemzőivel szoktuk megadni, illetve a kamera adataiból számoljuk. Az OpenGL az első két számítást végzi egy lépésben, ez a MODELVIEW transzformáció. Az összes transzformáció megvalósítása egy-egy mátrix-szorzással történik. Mi ezt a mátrixot állíthatjuk be. Amikor a fenti kódrészlethez hasonlóan valamit rajzolunk, akkor a koordinátákat a MODELVIEW, majd a PROJECTION mátrixokkal megszorozza az OpenGL, és a kapott képernyő-koordinátáknak megfelelő képpontokat színezi be (raszterizáció). Ha már van egy pixelben érték, és az közelebbi felületi ponthoz tartozik, mint az éppen rajzolt felület, azt nem írja felül (z-buffer algoritmus).

A rajzolás menete tehát:

- PROJECTION beállító módba váltunk

```
glMatrixMode(GL_PROJECTION);
```

- a kamera látószögének, stb. (ezek nem szoktak változni) megfelelően beállítjuk a PROJECTION transzformációt

```
glLoadIdentity();
gluPerspective(45.0f, aspectRatio, 1.0f, 10000.0f);
```

- MODELVIEW beállító módba váltunk

```
glMatrixMode(GL_MODELVIEW);
```

- a kamera pozíciójának és orientációjának megfelelően beállítjuk a VIEW transzformációt:

```
glLoadIdentity();
gluLookAt(
      pos.x, pos.y, pos.z,
      pos.x + forward.x, pos.y + forward.y, pos.z + forward.z,
      0.0f, 1.0f, 0.0f);
```

- elmentjük az aktuális állapotot

```
glPushMatrix();
```

- hozzáfűzzük (hozzászorozzuk) a MODELVIEW-hoz a éppen aktuális kirajzolandó tárgy pozíciója és orientációja alapján a MODEL transzformációt
- visszaállítjuk a csak a kamerát tartalmazó MODELVIEW állapotot

```
glPopMatrix();
```

- a `glPushMatrix();`-tól addig ismételjük amíg van objektum

A transzformációk beállítása fordított sorrendben történik, mint a végrehajtásuk!

Az OpenGL számos egyéb képessége között számunkra még fontos az, hogy képes árnyalást számolni. Ez a számítás a MODELVIEW transzformáció után és a PROJECTION előtt történik (ezért van tulajdonképpen szétválasztva, különben egyetlen lépésben lehetne modellezési koordinátákból képernyő-koordinátát számolni). Ehhez be kell állítani a legalább egy fényforrást:

```
glEnable(GL_LIGHTING);
float AmbientColor[]    = { 0.0f, 0.1f, 0.2f, 0.0f };
glLightfv(GL_LIGHT0, GL_AMBIENT, AmbientColor);
float DiffuseColor[]    = { 0.2f, 0.2f, 0.2f, 0.0f };
glLightfv(GL_LIGHT0, GL_DIFFUSE, DiffuseColor);
float SpecularColor[]   = { 0.5f, 0.5f, 0.5f, 0.0f };
glLightfv(GL_LIGHT0, GL_SPECULAR, SpecularColor);
float Position[]        = { 100.0f, 100.0f, -400.0f, 1.0f };
glLightfv(GL_LIGHT0, GL_POSITION, Position);
glEnable(GL_LIGHT0);
```

Ezt a kamera transzformáció beállítása után érdemes megtenni. (Mivel az árnyalás számítás a MODELVIEW után, kamera-koordináta rendszerben történik, az OpenGL a fények pozícióit ebbe a rendszerbe transzformálja, vagyis beszorozza a MODELVIEW-val, és így jegyzi meg. Ha ezután a kamerát elmozgatjuk, és a fényeket nem adjuk újra meg, úgy fog tűnni, mintha fények együtt mozognának a kamerával.)

Megjegyzés: a fények és az árnyalás bekapcsolása nem jelenti azt, hogy árnyékok is lesznek. Az sokkal bonyolultabb feladat, és az OpenGL-nek nem része a megoldás.

Funkciók egy interaktív grafikus rendszerben

Három fő funkciót szeretnénk megvalósítani:

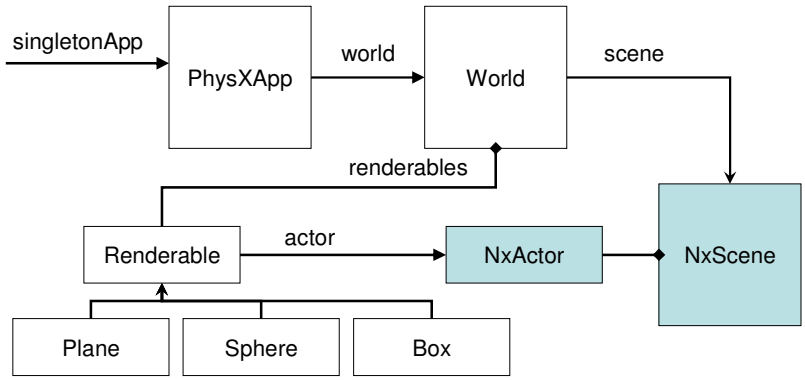
- **render**, megjelenítés: a fentiek használatával, OpenGL-lel. Az aktuális modellezési transzformációt a PhysX-ből kell kinyernünk.
- **control**, vezérlés: a GLUT eseménykezelők révén megismert felhasználói akarat szerint, illetve a mesterséges intelligencia döntései (itt most ilyen nem lesz) vagy a fizikai törvények alkalmazásából a virtuális világ elemeire ható hatások kiszámítása. Itt a PhysX könyvtárnak kell beállítani a szabályszerűségeket és hatásokat.
- **animate**, animáció: a vezérlés során meghatározott hatások érvényesítése, az objektumok helyzetének az eltelt idő függvényében történő megváltoztatása. Ezt a PhysX-re fogjuk bízni.

A GLUT eseménykezelők ezeket a funkciókat fogják beindítani.
- Az egéresemény-kezelők közvetlenül vezérlik a kamerát.
- A gombnyomásesemény-kezelők módosítják az input állapotot, amit egy bool tömbben tárolunk (amelyik gomb le van nyomva, ahhoz a tömbben az érték true).
- A display esemény rajzol: **render**.

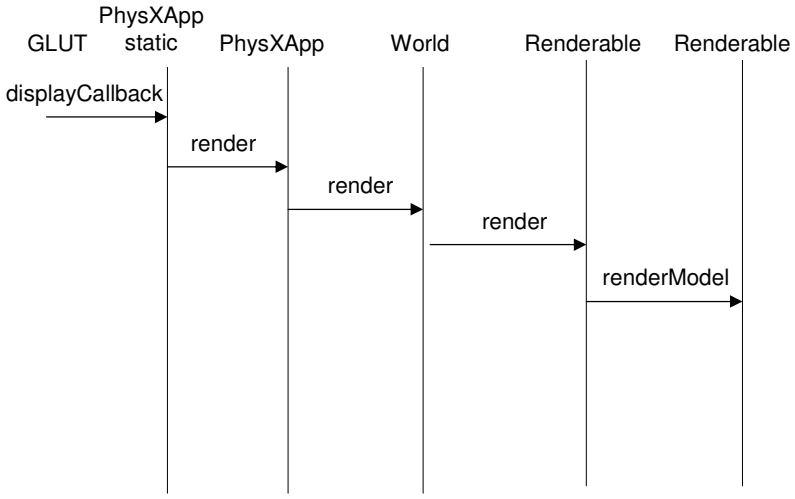
- Az idle esemény szimulációt futtat: **animate** és **control**. Miért ebben a sorrendben? Mert a **control** a PhysX-hez fordul a számítások elvégzésére, a következő **animate** pedig a PhysXtől kéri el az eredményeket. Ha pl. a fizikai számításokat PPU (Physical Processing Unit) fizikai gyorsítókártyával végezzük, érdemes időt hagyni a PhysX-nek a számítások elvégzésére. Tehát mindig az egy szimulációs lépéssel korábbi eredményeket rajzoljuk.

A heterogén virtuális világ

A színtér különböző objektumokból áll: talajsík, gömbök, hasábok. Ezekben a közös, hogy mindegyiket ki lehet rajzolni, illetve animálni és vezérelni lehet őket, csak mindegyiknél máshogy kell. A két utolsó funkciót a PhysX **Actor** objektumai tudják. Marad a render: az osztály neve Renderable. Absztrakt ősosztályként működik, a Plane, Sphere és Box ennek a leszármazott osztályai lesznek, amiben a rajzolást más-más módon fogjuk implementálni. Minden Renderable-höz tartozik egy NxActor. A Renderable-leszármazottnak példányosításakor egy megfelelő NxActort kell létrehoznia.

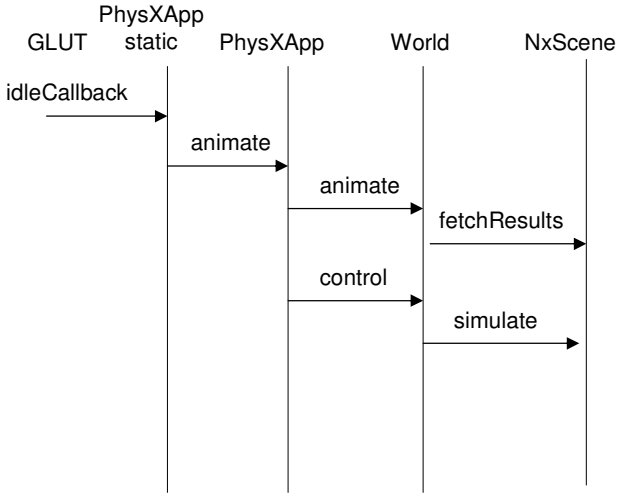


A virtuális világ legfontosabb adatszerkezete egy Renderable* vektor (dinamikus méretű tömb). Ebben a mutatók Plane, Sphere és Box példányokra mutatnak. A virtuális világ kirajzolásakor minden Renderable-nek a render metódusát kell hívni. A render metódus beállítja a megfelelő modellezési transzformációt, majd a virtuális RenderModel metódust hívja, ami kirajzolja az alakzatot. Ez a különböző Renderable-ből származtatott osztályokban (Plane, Sphere, Box) különböző módon van implementálva.



A PhysX színtér

Hasonlóan ahhoz, ahogy mi a Renderable objektumainak egy tárolóba gyűjtöttük, a PhysX is az NxActorokat egy NxScene nevű osztályba foglalja. A vezérlési és animációs feladatokat gyakorlatilag ennek a színtér objektumnak fogjuk átadni. A vezérlési funkciónak a simulate metódus felel meg: a control-ban feldolgozzuk a felhasználói inputokat, ennek megfelelően beállítjuk a PhysX paramtéreit, majd elindítjuk a szimulációt. Az animate funkciót (vagyis a vezérlésnek megfelelő mozgások kiértékelését) ezután úgy valósítjuk meg, hogy lekérjük a PhysX eredményeket. Kirajzoláskor így a szükséges adatok az NxActor objektumokból elérhetőek lesznek.



Egy színtér felépítése PhysXben

Indulásképp létre kell hozni egy SDK objektumot, azon belül egy színteret:

```
physicsSDK = NxCreatePhysicsSDK(NX_PHYSICS_SDK_VERSION);

// Set the physics parameters
physicsSDK->setParameter(NX_SKIN_WIDTH, 0.01);

// Create the scene
NxSceneDesc sceneDesc;
sceneDesc.simType = NX_SIMULATION_SW;
sceneDesc.gravity = NxVec3(0,-9.8,0);
scene = physicsSDK->createScene(sceneDesc);
```

Kell egy anyagot is definiálni, mechanikai tulajdonságaival. A színtér minden objektuma ebből az anyagból lesz.

```
// Create the default material
NxMaterial* defaultMaterial = scene->getMaterialFromIndex(0);
defaultMaterial->setRestitution(0.5);
defaultMaterial->setStaticFriction(0.5);
defaultMaterial->setDynamicFriction(0.5);
```

Egy actor létrehozása egy actor leíró létrehozásával és a színtér objektum createActor metódusával történik:

```
NxPlaneShapeDesc planeDesc;
NxActorDesc actorDesc;
actorDesc.shapes.pushBack(&planeDesc);
actor = gScene->createActor(actorDesc);
```

Az ilyen létrehozásokat azonban meghagyjuk a különböző alakzatok (Plane, Sphere, Box) konstruktorainak. Így pl. egy sík létrehozása és tárolása a saját renderables vektorunkban és a PhysX színterében mindössze ennyi lesz:

```
      // ez létrehozza az actor a scenben:
Plane* groundPlane = new Plane(scene);
      // berakjuk a saját vektorunkba is:
renderables.push_back(groundPlane);
```

Összekapcsolt elemek létrehozása

Egyes elemek között kapcsolatokat (ízületeket, csuklókat) definiálhatunk az NxScene::createJoint metódussal. Ehhez egy joint leíró adatszerkezetet kell kitöltenünk, amiben a legfontosabb, hogy melyik két objektumot és milyen módon kötjük össze. Mi forgó kapcsolatokat fogunk használni, itt a forgástengely egy pontja és a tengely iránya számít.

Robotkar építése és vezérlése

A mérés során a fent leírt keretrendszert kell kiegészíteni hasáb (Box) objektumokkal. Ezeket össze kell kötni motoros kapcsolatokkal, így egy robotkart kialakítva. A robotkar vezérlését a billentyűzethez kell kötni.

További információk

www.iit.bme.hu/~szecsi/PhysXlab.html (PhysX dokumentáció, illetve a PhysXlab solution váza)