Ion-atom ütközések megjelenítése három dimenzióban

Témavezető:

Tőkési Károly

tudományos főmunkatárs

MTA-ATOMKI

Konzulens:

Halász Gábor

egyetemi docens

Debreceni Egyetem

Informatika Kar

Készítette: **Budai László**

programozó matematikus hallgató

Debrecen

2007

Tartalomjegyzék

1. fejezet

Bevezetés

A szakdolgozat nem a 3D-s programozás grafikai alapjait akarja bemutatni, nem is egy konkrét 3D Application Programmer Interface(API)-t(pl. OpenGL), de nem is az ion-atom ütközések fizikáján van a fő hangsúly. A cél egy program elkészítése volt, amely ion-atom ütközési eseményeket modellez, real-time jeleníti meg a pályákat három dimenzióban és lehetővé teszi az események rögzítését is. A dolgozat tulajdonképp ezen program megírásához szükséges fizikai és informatikai ismereteket tartalmazza. Numerikus módszerekben szándékosan nem mélyül el a dolgozat, hisz a program megírása során nem erre volt a legnagyobb szükség. Fontosabbnak találtam a C++ template metaprogramming lehetőségének bemutatását és a Qt keretrendszer rövid ismertetését. Ez volt az a két technológia, ami nélkül a program nem készült volna el. A fizikai rész két nagy fejezetre tagolódik, az első a tömegpont és tömegpontrendszerekre vonatkozó fizikai alapfogalmakat foglalja össze, a második pedig az ion-atom ütközések klasszikus szimulációjáról szól és tartalmazza a program által megoldott mozgásegyenleteket. Az informatikáról szóló részben a program tervének leírása található meg a fent említett két technológia bemutatása mellett.

rész I

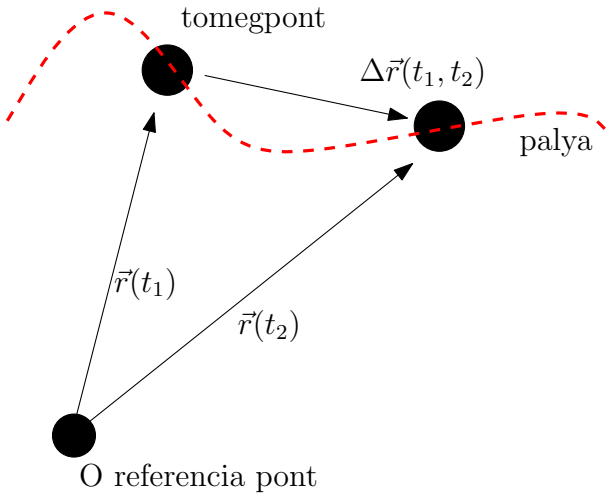
Fizika

2. fejezet

Tömegpont, tömegpontrendszerek

2.1. Tömegpont mozgása

Az anyagi pont vagy tömegpont olyan, tömeggel rendelkező test, amely pontszerűnek tekinthető, vagyis méretei a mozgás más jellemzőihez képest elhanyagolhatók. A tömegpont aktuális helyét a térben az $\vec{r}$ helyvektorral írhatjuk le, amelyet egy O referenciaponttól mérünk. A tömegpont mozgását ismerjük, ha ismerjük az $\vec{r}(t)$ függvényt, amelyet pályagörbének vagy trajektóriának nevezünk.



2.1. ábra. tömegpont pálya

2.1.1. Néhány fizikai fogalom definíciója

Egy három dimenziós térbeli pont helyét egyértelműen meghatározza az (x, y, z) hármas. Ennek a pontnak a **helyvektora** (jele: $\vec{\mathbf{r}}$) a koordináta rendszer kezdőpontját köti össze magával a ponttal, és $\vec{\mathbf{r}}$ a pont irányába mutat.

- **Elmozdulás:** Ha a pont $\vec{\mathbf{r}}_1$-ből $\vec{\mathbf{r}}_2$-be mozog, a pont $\Delta\vec{\mathbf{r}}$ *elmozdulást* végez.

$$\Delta\vec{\mathbf{r}} = \vec{\mathbf{r}}_2 - \vec{\mathbf{r}}_1 \tag{2.1}$$

  Az elmozdulás irányított mennyiség, azaz vektor.

- **Sebesség:** Ha egy tömegpont $\Delta\vec{\mathbf{r}}$ elmozdulásához Δt időre van szükség, a pont **átlagsebessége**(jele:$\bar{v}$):

$$\bar{\mathbf{v}} = \frac{\Delta\vec{\mathbf{r}}}{\Delta t} \qquad \textit{átlagsebesség} \tag{2.2}$$

  $\Delta\vec{\mathbf{r}}$ és $\bar{v}$ iránya megegyezik. Egy pont **pillanatnyi sebessége**(jele $\vec{\mathbf{v}}$)

$$\vec{\mathbf{v}} = \lim_{\Delta t \to 0}\left(\frac{\Delta\vec{\mathbf{r}}}{\Delta t}\right) = \frac{d\vec{\mathbf{r}}}{dt} \tag{2.3}$$

  A tömegpont t pillanatbeli sebessége ($\vec{\mathbf{v}}(t)$) tehát a pályafüggvény differenciálhányadosa:

$$\vec{\mathbf{v}}(t) = \frac{d\vec{\mathbf{r}}(t)}{dt}; \qquad \vec{\mathbf{v}}(t) = \dot{\vec{\mathbf{r}}}(t) \tag{2.4}$$

  $\vec{\mathbf{v}}$ vektormennyiség, és a $\Delta\vec{r}$ elmozdulás irányába mutat. A sebesség v abszolútértéke $\vec{\mathbf{v}}$-nek a koordináta tengelyek mentén vett komponenseiből tevődik össze:

$$v_x = \frac{dx}{dt}; \qquad v_y = \frac{dy}{dt}; \qquad v_z = \frac{dz}{dt}$$

- **Út:** A $t_1 < t_2$ időpillanatok között megtett $s(t_1, t_2)$ út az $\vec{\mathbf{r}}(t)$ függvény $\vec{\mathbf{r}}(t_1)$ és $\vec{\mathbf{r}}(t_2)$ pontjai közé eső ívének a hossza, azaz

$$s(t_1, t_2) = \int_{t_1}^{t_2} \left|\frac{d\vec{r}(t)}{dt}\right| dt \tag{2.5}$$

$$s(t_1, t_2) = \int_{t_1}^{t_2} \left|\vec{\mathbf{v}}(t)\right| dt \tag{2.6}$$

- **Gyorsulás:** Tömegpont gyorsulása(pillanatnyi gyorsulás):

$$\vec{a} = \frac{d^2\vec{r}(t)}{dt^2} \qquad \text{, azaz} \qquad \vec{a} = \dot{\vec{v}}(t) \tag{2.7}$$

Három határesetet lehet megkülönböztetni:

1. $\vec{a}$ párhuzamos $\vec{v}$-vel, és irányuk megegyezik. A gyorsulás pozitív, a sebesség az idővel növekszik. Ez a szűkebb értelemben vett gyorsulás.
2. $\vec{a}$ párhuzamos $\vec{v}$-vel, de irányuk ellentétes. A gyorsulás negatív, a sebesség az idővel csökken, a mozgás *lassuló*.
3. $\vec{a}$ merőleges $\vec{v}$-re. A sebesség nagysága ekkor az időtől független, ám iránya változik(pl.:körmozgás).

- **Lendület:** Egy tömegpont impulzusa(mozgásmennyisége, lendülete):

$$\vec{p} = m\vec{v}, \qquad \text{impulzus} \tag{2.8}$$

$\vec{p}$ vektormennyiség, iránya megegyezik $\vec{v}$ irányával. Ha egy $\vec{F}$ külső erő hat a tömegpontra, annak impulzusa megváltozik:

$$\frac{d\vec{p}}{dt} = \vec{F} \tag{2.9}$$

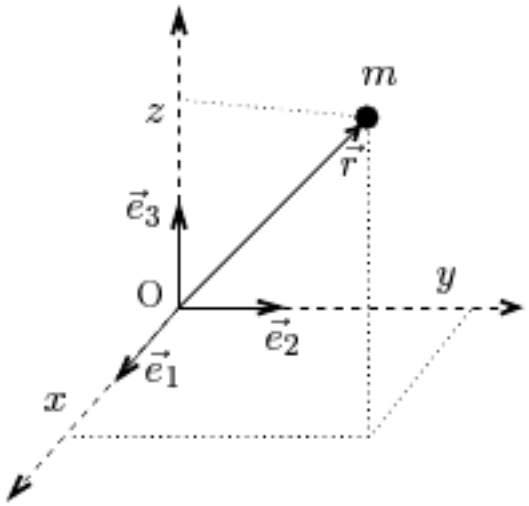
1. Impulzusmegmaradás törvénye. *Ha egy tömegpontra nem hat külső erő, akkor impulzusa állandó és az időtől független.*

- **Mozgási energia:**

$$E_{mozg} = \frac{1}{2}mv^2 \tag{2.10}$$

- **Koordináta-rendszer**

Koordinátarendszer megadása:kijelölünk egy O-referenciapontot(origó) és ezen ponthoz rögzítsük hozzá az $\vec{e}_i, i = 1, 2, 3$ bázisvektorokat. (2.1.1) A koordinátarendszerben az $\vec{r}(t)$ vektort az



2.2. ábra. Koordináta-rendszer

$$\vec{r}(t) = \begin{pmatrix} x(t) \\ y(t) \\ z(t) \end{pmatrix}$$

koordináták írják le, amelyek az idő függvényei. A sebesség és gyorsulás komponensei:

$$\vec{v}(t) = \begin{pmatrix} v_x(t) = \frac{dx(t)}{dt} \\ v_y(t) = \frac{dy(t)}{dt} \\ v_z(t) = \frac{dz(t)}{dt} \end{pmatrix}, \qquad \vec{a}(t) = \begin{pmatrix} a_x(t) = \frac{d^2x(t)}{dt^2} \\ a_y(t) = \frac{d^2y(t)}{dt^2} \\ a_z(t) = \frac{d^2z(t)}{dt^2} \end{pmatrix}$$

2.1.2. A dinamika alaptörvénye

1. Newton 2. törvénye - A dinamika alaptörvénye. *Egy pontszerű test* $\vec{\mathbf{a}}$ *gyorsulása egyenesen arányos a testre ható, a gyorsulással azonos irányú* $\vec{\mathbf{F}}$ *erővel, és fordítottan arányos a test* $\mathbf{m}$ *tömegével.*

Tehát a tömegpont mozgását annak környezete határozza meg úgy, hogy megadja a tömegpont lendületének változási gyorsaságát.

$$\dot{\vec{p}} = \vec{F} \ , \tag{2.11}$$

ahol $\vec{F} = \vec{F}(\vec{r}, \vec{v}, t)$ a környezet testre kifejtett hatását jellemző függvény. Helyettesítsük be a lendület definícióját:

$$\dot{\vec{p}} = \frac{d\vec{p}}{dt} = \frac{d(m\vec{v})}{dt} = m\frac{d\vec{v}}{dt}, \tag{2.12}$$

$$\boxed{m\frac{d^2\vec{r}}{dt^2} = \vec{F}.} \tag{2.13}$$

Az (2.13) egyenletet *mozgásegyenletnek* hívjuk, mert belőle meghatározható a test mozgása, amennyiben ismerjük a környezet hatását($\vec{F}$) a vizsgált testre. Az (2.13) az $\vec{r}(t)$ függvényre egy vektoriális, közönséges másodrendű differenciálegyenlet:

$$m\frac{d^2\vec{r}(t)}{dt^2} = \vec{F} \qquad \text{, azaz} \tag{2.14}$$

$$m\ddot{\vec{r}} = \vec{F}(t, \vec{r}, \dot{\vec{r}}) \tag{2.15}$$

Derékszögű koordinátarendszerben (2.15) a következő három skaláris differenciálegyenlet formájában írható fel:

$$m\ddot{x} = F_x(t, x, y, z, v_x, v_y, v_z),$$

$$m\ddot{y} = F_y(t, x, y, z, v_x, v_y, v_z),$$

$$m\ddot{z} = F_z(t, x, y, z, v_x, v_y, v_z),$$

$$\text{ahol } v_x = \dot{x}, v_y = \dot{y}, v_z = \dot{z}.$$

Az $\vec{r}(t)$ általános megoldásában szereplő integrálási állandók értékét a *kezdeti feltételekből* határozzuk meg. Ehhez meg kell adnunk a tömegpont egy t_0 időpontbeli helyzetét és sebességét(másodrendű differenciálegyenlet : szükség van a nulladik és első deriváltakra), azaz az

$$\vec{r}_0 = \vec{r}(t_0) \qquad \text{és}$$

$$\vec{v}_0 = \vec{v}(t_0) = \dot{\vec{r}}(t_0)$$

vektorokat.

2.2. Tömegpontrendszerek mechanikája

- Tömegpontrendszer: egymással kölcsönhatásban lévő tömegpontok összessége. jelölése:

  $$\{\vec{r}_i, m_i\}_N,$$

  ahol $\vec{r}_i, m_i$ az i-edik tömegpont helyvektora és tömege, N jelöli a tömegpontrendszer elemeinek számát.

- Belső erő: Egy tömegpontra ható belső erő a többi tömegpont által rá kifejtett erők összege:

  $$\vec{F}_i^B = \sum_{j \neq i = 0}^{N} \vec{F}_{j \to i},$$

  ahol $\vec{F}_i^B$ az i-edik tömegpontra ható belső erő, és $\vec{F}_{j \to i}$ jelöli a rendszer j-edik eleme által az i-edikre kifejtett erőt.

- Külső erő: Az $\{\vec{r}_i, m_i\}_N$ tömegpontrendszeren kívül eső testekkel való kölcsönhatást jellemző $\vec{F}_i^K$ erők.

- Zárt tömegpontrendszer: Ha elemeire külső erők nem hatnak.

2.2.1. Tömegpontrendszer mozgásegyenletei

Egy tömegpontrendszer mozgását ismerjük, ha ismerjük minden elemének mozgását. Az N tömegpont közül az i-edik mozgásegyenlete:

$$m_i\ddot{\vec{r}}_i = \sum_{j\neq i=0}^{N} \vec{F}_{j\to i} + \vec{F}_i^K = \vec{F}_i^B + \vec{F}_i^K, \qquad i = 0, \ldots, N \tag{2.16}$$

ahol $\vec{F}_i^B$ és $\vec{F}_i^K$ egy N elemű tömegpontrendszer i-edik tömegpontjára ható belső és külső erők eredője.

2.16 egy $3N$ darab másodrendű differenciálegyenletből álló csatolt differenciálegyenlet rendszer, amely megoldásához szükséges kezdőfeltételek:

$$\vec{r}_{i0} = \vec{r}_i(t_0), \qquad \text{és} \qquad \vec{v}_{i0} = \vec{v}_i(t_0).$$

(azaz meg kell adni a részecske helyét és sebességét egy kezdeti t_0 időpontban).

- Az $\{\vec{r}_i, m_i\}$ tömegpontrendszer *tömegközéppontja(súlypontja)* az

$$\vec{r}_c = \frac{\sum_i m_i\vec{r}_i}{m}, \qquad \text{ahol} \qquad m = \sum_i m_i \tag{2.17}$$

vektorú pont. A tömegközéppont *sebessége*:

$$\vec{v}_c = \frac{\sum_i m_i\vec{v}_i}{m} \tag{2.18}$$

Tömegpontrendszer *teljes impulzusa*:

$$\vec{p}_{teljes} = \sum_i m_i\vec{v}_i = m\vec{v}_c. \tag{2.19}$$

- *A tömegközéppont mozgása:*

$$\sum_{i=1}^{N} m_i\ddot{\vec{r}}_i = \underbrace{\sum_i \vec{F}_i^B}_{=0} + \sum_i \vec{F}_i^K, \tag{2.20}$$

$$\sum_{i=1}^{N} m_i \ddot{\vec{r}}_i = \underbrace{\frac{d}{dt} \sum_{i=1}^{N} \vec{p}_i}_{\sum_{i=1}^{N} \dot{\vec{p}}_i} = \sum_i \vec{F}_i^K \qquad , \tag{2.21}$$

tehát

$$\boxed{\dot{\vec{p}}_{teljes} = \sum_i \vec{F}_i^K} \qquad , \tag{2.22}$$

azaz ha a rendszerre nem hat külső erő, akkor annak tömegközéppontja nyugalomban marad (impulzusmegmaradás törvénye tömegpontrendszerre vonatkoztatva).

3. fejezet

Ion-atom ütközések klasszikus szimulációja

3.1. Bevezetés

Az atomi ütközési folyamatok klasszikus elméleti módszerekkel történő tanulmányozását Thomson 1912-ben az atomok elektronbombázással kiváltott ionizációjának vizsgálatával indította el. A klasszikus elmélet a kvantummechanikai számítások előretörése miatt feledésbe merült, egészen Gryzinski 1959-ben megjelent munkájáig. Gryzinski megmutatta, hogy klasszikus modellfeltevésekkel egyszerű és használható analitikus formulákat lehet adni az atomi folyamatok széles spektrumának leírására, azaz klasszikus közelítést az ütközés egész folyamatára, ahol is az ütközésben résztvevő részecskék a Newton-törvények szerint mozognak. Hatáskeresztmetszetek [1] számítása ebben a modellben a Monte Carlo módszer segítségével lehetséges, ami - statisztikus megfontolások alapján - nagyszámú egyedi pályák meghatározását és kiértékelését jelenti. Ezt az igen nagy számítógép-kapacitást igénylő eljárást klasszikus pályájú Monte Carlo módszernek (Classical Trajec-

[1] Felület jellegű mennyiség, két részecske kölcsönhatásának valószínűségét jellemzik vele : az a felület, amelyet az ütköző molekulák egymásnak célfelületként nyújtanak. mértékegysége m^2,1 barn(1 b) = $10^{-28}m^2$)

tory Monte Carlo, CTMC) nevezik. A módszer alkalmazásának úttörői vegyészek voltak, akik molekulaionok képződésének hatáskeresztmetszeteit határozták meg. A hatvanas években Abrinesnek és Percivalnak több összefoglaló közleménye jelent meg a CTMC módszerről. Ezeknek a cikkeknek a megjelenése a klasszikus elméletek újraéledését jelezte az atomfizikában, amelynek fénykora a mai napig tart. A CTMC módszer az ütközésben résztvevő részecskékre vonatkozó klasszikus (Newton vagy Hamilton) mozgásegyenletek numerikus megoldásán alapszik. Az atomi rendszereket mint parányi naprendszereket kezeli, azaz az elektronok megfelelően kiválasztott Kepler-pályákon mozognak az atom magja körül. A CTMC módszer nem perturbatív, és számba tudja venni az összes résztvevő partner lehetséges végállapotait (gerjesztés, ionizáció, átrendeződés stb.).

3.2. Klasszikus atommodell

Rutherford 1911-ben végzett kísérletei alapján ismert volt, hogy az atomok atommagból és elektronokból állnak. Így speciálisan a hidrogénatomot egy protonból és egy elektronból építhetjük fel. Sejtették azt is, hogy a hidrogénatom különböző energiájú állapotai, amelyeket a hidrogénatom színképének elemzése után azonosítottak, attól függnek, hogy az elektron milyen pályán mozog a proton körül. Ezek után kézenfekvő volt a gondolat, hogy az elektron hasonló módon kering az atommag körül, mint ahogy a bolygók keringenek a Nap körül. Ezt erősítette meg az is, hogy az m_1 és m_2 tömegű testek között lévő tömegvonzás és a $+e_1$ és $-e_2$ elektromos töltések közötti vonzás hasonló. Annál nagyobb a vonzerő a pozitív töltésû proton és a negatív töltésû elektron között, minél közelebb van az elektron a maghoz. Nagy különbség van azonban a Naprendszerben a Nap körül keringő bolygók és a proton körül keringő elektron esete között. Nevezetesen az, hogy a bolygóknak nincs töltésük, az elektronnak pedig van. Mivel az elektrodinamikai törvények szerint minden változó irányú (és nagyságú) sebességgel mozgó elektromos töltés energiát sugároz, az atommag körül keringő elektronnak, a folyamatos energiavesztés miatt, bele kellene esni az atommagba. Ez ellentétben van a tapasztalati megfigyelésekkel. Ezért a

klasszikus atom felépítésekor eltekintünk attól, hogy az elektronnak elektromos töltése révén energiát kellene sugároznia akkor, amikor a proton körül kering. Egy példaként vizsgáljuk meg a modell hidrogénatomot, mint mikroszkópikus naprendszert. Az elektron tömege $1\,a.u.$[2] és töltése $-1\,a.u.$, a proton tömege $1836\,a.u.$ és töltése $+1\,a.u.$. Az elektron ellipszis pályán kering a proton körül, a proton az ellipszis egyik gyújtó pontjában van. Az így jellemzett hidrogénatom alapállapoti kötési energiája $0.5\,a.u.$. A klasszikus atom kezdeti paramétereit az egyenletes eloszlású hatdimenziós fázistérből választhatom.

3.3. Az ütközés dinamikája

Tételezzünk fel három makroszkópikus, tömeggel ($m_i, i = 1,2,3$) és töltéssel ($Z_i, i = 1,2,3$) rendelkező részecskét, melyek a Newton-törvények szerint mozognak. A három részecske Lagrange-függvénye:

$$L = \frac{1}{2}m_1v_1^2 + \frac{1}{2}m_2v_2^2 + \frac{1}{2}m_3v_3^2 - \frac{Z_1Z_2}{|\vec{r}_1 - \vec{r}_2|} - \frac{Z_1Z_3}{|\vec{r}_1 - \vec{r}_3|} - \frac{Z_2Z_3}{|\vec{r}_2 - \vec{r}_3|}, \tag{3.1}$$

ahol $\vec{r}_i$ az i-edik részecske helyvektora és v_i a sebessége.

A Newton-féle mozgásegyenleteket a következő alakban írhatjuk fel:

$$m_i\frac{d^2\vec{r}_i}{dt^2} = \sum_{j=1,j\neq i}^{3} Z_iZ_j\frac{\vec{r}_i - \vec{r}_j}{|\vec{r}_i - \vec{r}_j|^3}, \qquad (i = 1,2,3) \tag{3.2}$$

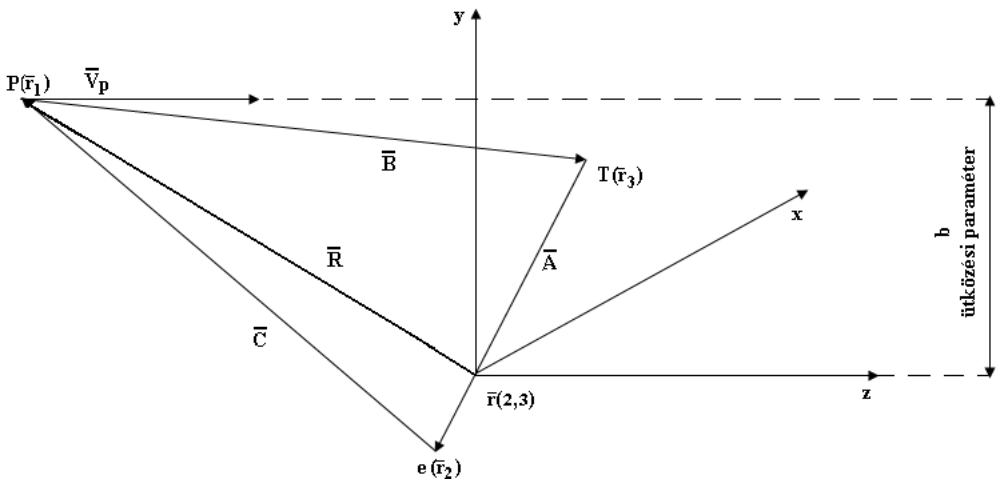
Szeparáljuk a három részecske tömegközéppontjának a mozgását és térjünk át az $\vec{A}, \vec{B}, \vec{C}$ vektorokkal jellemzett relatív helyvektorokra. Ekkor a mozgásegyenletek a következő formába írhatók:

$$\ddot{\vec{A}} = \left[\frac{(N_2+N_3)Z_2Z_3}{|\vec{A}|^3} + \frac{N_2Z_1Z_2}{|\vec{A}+\vec{B}|^3}\right]\vec{A} + \left[\frac{N_2Z_1Z_2}{|\vec{A}+\vec{B}|^3} - \frac{N_3Z_1Z_3}{|\vec{B}|^3}\right]\vec{B} \tag{3.3}$$

$$\ddot{\vec{B}} = \left[-\frac{N_3Z_2Z_3}{|\vec{A}|^3} + \frac{N_1Z_1Z_2}{|\vec{A}+\vec{B}|^3}\right]\vec{A} + \left[\frac{N_1Z_1Z_2}{|\vec{A}+\vec{B}|^3} + \frac{(N_1+N_3)Z_1Z_3}{|\vec{B}|^3}\right]\vec{B} \tag{3.4}$$

3.3-3.4 egyenletrendszer integrálásával kapjuk meg a rendszert alkotó részecskék koordinátáit az idő függvényében, azaz magát a pályát.

[2]atomic unit



3.1. ábra. Az ütközésben résztvevő három részecske relatív helyvektorai. 1,2, és 3 rendre a lövedéket, az elektront és a célatom magját jelölik.$\vec{r}(2,3)$ a célatom tömegközéppontjának koordinátája. $\vec{A} = \vec{r}_2 - \vec{r}_3, \vec{B} = \vec{r}_3 - \vec{r}_1, \vec{C} = \vec{r}_1 - \vec{r}_2$

rész II

Informatika

4. fejezet

Programterv

4.1. Bevezetés

Programozói szemszögből nézve minden ion-atom ütközési probléma megoldási sémája ugyanaz. Van egy tömegpontrendszer rendszer, annak vannak paraméterei és egyenletei. A cél az, hogy megjelenítsük grafikusan a rendszer mozgásegyenletének megoldása során előálló trajektóriákat. A megoldáshoz a pályaegyenletek numerikus integrálását kell elvégeznünk. Ehhez különféle numerikus módszerek állnak rendelkezésünkre, mint pl.: Euler, Runge-Kutta(másodrendű, negyedrendű, alkalmazkodó lépésközű), Verlet, prediktor-korrektor, stb ... A program tervét célszerű platformfüggetlenre készíteni(ne ,,drótozzunk" be semmit, ami platformfüggő lenne) és a tervezés során osztályokkal modellezni minden ,,fogalmat"(fogalomnak minősül a numerikus módszer, a paraméterek, a differenciálegyenlet, a kezdeti érték probléma vagy kezdetiérték probléma, stb...), a ,,sima" osztályok mellett célszerű még sablonokban is gondolkodni, ugyanis ezzel válik lehetővé az, hogy bármilyen ion-atom ütközési problémát minimális erőfeszítéssel leprogramozhassunk. A teendő mindössze annyi, hogy a problémára jellemző paramétereket leírjuk, megadjuk a mozgásegyenleket és ezekkel, mint sablonparaméterekkel felparaméterezzük a meglévő forráskódot, lefordítjuk, így előáll az adott problémát megoldó és megjelenítő

program(tulajdonképp két fő részből áll a programterv: 1- problémák mindegyikére jellemző bázisosztályok és sablonok, 2-az osztályhierarchia levélelemei, azaz a konkrét problémákat leíró osztályok, amelyekre mint cserélhető modulokra tekinthetünk). A választott programozási nyelv a C++, mivel többféle programozási paradigmát is támogat(többek közt az objektum-orientált és generikus programozási paradigmákat), támogatja az operátortúlterhelést továbbá az egyik leghatékonyabb és legelegánsabb programozási nyelv(utóbbi természetesen szubjektív vélemény).

4.2. Fizika és programozás

A generikus és objektumorientált programozási paradigmák felhasználásával lehetőségünk nyílik fizikai problémákat absztrakt szinten kezelni, oly módon, hogy minimális programozással elő tudjunk állítani az absztrakt rendszerből egy konkrét fizikai problémát megoldó és megjelenítő programot. A fizikai problémákat osztályozhatjuk, amely során az egy osztályba tartozó problémákat onnan ismerhetjük fel, hogy a fogalmi rendszerük megegyezik. Ezen fogalmak által válik lehetővé az, hogy a problémával absztrakt szinten foglalkozhassunk. A programtervezés során minden fizikai fogalomhoz egy osztályt rendelünk, ami tulajdonképp az adott fizikai fogalom modellje, pontosabban egy azzal ekvivalens informatikai interpretáció(lényegében egy másik nyelven fogalmazzuk meg a problémát) Mi most a klasszikus ion-atom ütközésekre végeztük el ezt a ,,lefordítást". Az előnye a hagyományos, struktúrált programozással szemben igen nagy: egyetlen programmal lényegében egy egész problémaosztály megoldására nyílik lehetőség, az ezen problémaosztályhoz tartozó feladatok legtöbbször csak annyi programozást igényelnek, hogy leprogramozzuk az egyenleteket és megadjuk a paramétereket, minden egyebet(integrálás, megjelenítés, stb...) a problémaosztály közös jellemzői révén úgymond ,,magától tud" a program.

4.2.1. Az objektum orientált paradigmáról (OOP)

Középpontjában az osztály, ill. objektum áll. Az osztály egy fogalom leírását teszi lehetővé azáltal, hogy egységbe zárja a statikus adatmodellt és a dinamikus funkcionális(avagy viselkedés-) modellt. Az objektum pedig az adott fogalom konkretizációja, amely a fogalmat leíró osztályban található leírás alapján jön létre. Ezt a fajta gondolkodásmódot a következőképp is megközelíthetjük: olyan, hogy ,,ion" nincs, csak pl. olyan, hogy *hidrogén* ion stb. . . .(más hasonlattal élve: olyan sincs, hogy ,,fa", csak olyan, hogy almafa, stb. . .). Az osztályon kívül fontos fogalom még az interface(felület) is, amelyet a viselkedés absztrakciójának tekinthetünk. Ez pl. akkor lehet hasznos, ha a mozgásegyenletekről akarunk konkrét problémától függetlenül beszélni. Az OOP igen fontos lehetősége az *öröklődés* és *bezárás*. Általában három szintű bezárás fogalommal dolgoznak az OOP mentén felépülő nyelvek: private, protected és public, amelyek rendre a következőt jelentik: csak az adott osztály látja, az adott osztály és a leszármazottai látják, mindenki látja. Öröklődés esetén van egy(vagy több) ősosztály(vagy szülő osztály), amiből leszármaztatunk egy vagy több gyermekosztályt(vagy alosztály), amely örökli az ősosztály azon tagjait(adat- és függvénytagok), amiket a bezárás nem tilt meg. Vegyünk pl. egy tekercset, ennek egy igen fontos jellemzője a menetszáma, ebből leszármaztatunk egy olyan osztályt, ami a vasmagos tekercset írja le, de örökli a tekercstől a menetszámot leíró adattagot. Itt jól látszik az ősosztály és leszármazottja között az ,,is-a" kapcsolat, amit magyarul ,,az-egy" kapcsolatnak fordíthatunk: a vasmagos tekercs **az egy** tekercs, továbbá elmondhatjuk azt is, hogy minden vasmagos tekercs az tekercs, de nem minden tekercs vasmagos tekercs(ez a helyettesíthetőségre utal: az ősosztály helyettesíthető az alosztállyal, de fordítva nem). Az OOP további fontos lehetősége a *polimorfizmus*, ami többalakúságot jelent, és az egymással szülő-leszármazott kapcsolatban álló osztályok esetén van szerepe. Vegyünk pl. egy mozgásegyenlet nevű osztályt, annak legyen több leszármazottja, öröklik az ősosztálytól a számolást végző függvényt, ami nem lett megadva, hogy mit jelent, ezt az alosztályoknak kell elvégezni. Ekkor minden alosztályban mást jelent ez

a függvény. Megtehetjük azt, hogy megadjuk az ősosztály nevét paraméternek(ami helyett majd állhat ugye akármelyik leszármazottja), meghívhatjuk a számolófüggvényt, és a konkrét paramétertől függ, hogy a számolófüggvény melyik alakja fog meghívódni.

4.2.2. A generikus paradigmáról

A generikus paradigma lehetővé teszi a típussal történő paraméterezést. Mit lehet típussal paraméterezni? Többek közt függvényeket és osztályokat is(megj.: az osztály típusnak minősül). Ezekre sablonokokra tekinthetünk, a paraméterként megadott típusok a sablonparaméterek. Amennyiben osztályokhoz adunk meg sablonparamétert, akkor sablonosztályokról beszélünk. Ezt egy fizikai probléma során hogyan tujduk hasznosítani? Vegyük pl. a numerikus módszereket, amelyekkel függvényeket integrálunk, annak érdekében, hogy előállítsuk a mozgásegyenletek alapján a pályákat. Ha az integrálást, mint fogalmat egy sablonosztállyal írjuk le, amelynek sablonparamétere egy olyan osztály, ami egy tetszőleges mozgásegyenletet modellez, akkor egy egyszer megírt kód alapján elő tudunk állítani egy olyan kódot, ami a sablonparaméterként megadott egyenlet integrálását végzi el, anélkül, hogy ezen függvényre ,,kézzel" meghívnánk az integrálórutint. Kicsit egyszerűbben fogalmazva: előre megírjuk a mozgásegyenletek kódjait, majd attól függően, hogy milyen problémát megoldó programot akarunk előállítani, kiválasztjuk valamelyiket és azzal a ,,beállítással" fordítjuk le a programot(a sablonból olyan osztályt állít elő a fordító, amiben a sablonparaméter helyére beírja megadott osztályt). A C++ nyelvben lehetőségünk van ún. template metaprogramozásra is, ez a lehetőség egy külön részben kerül bemutatásra. A template metaprogramming teszi lehetővé azt, hogy hatékonyabb numerikus programokat írhassunk, C++-ban, mint Fortranban.

4.2.3. Miért C++ ?

A C++ támogatja többek közt a generikus és az objektum orientált paradigmát is, továbbá törekszik a hatékonyságra, kellően magas szintű, és fizikai problémák esetén a kód

olvashatóságát nagymértékben javítja az operátortúlterhelés támogatása is(amennyiben mátrix-szorzást kell elvégeznünk, akkor a forráskódban ugyanúgy néz ki, mintha papíron írnánk le) A generatív programozás [1] napjaink egyre népszerűbb programozási paradigmája. Egyes részei (aspektus-orientált programozás, szándék-alapú programozás és generikus programozás) már kiléptek a kísérleti stádiumból, és elfogadott technológiává válnak. A C++ template metaprogramozás [2],[3] is most éli ezt a szakaszát. A template metaprogramok használata során a futási idejű algoritmusok egy részét fordítási idejű tevékenységgel helyettesítjük, és segítségükkel hatékonyabb kódot (expression templates), könnyebben bővíthető könyvtárakat és fordítási idejű program-adaptációt hozhatunk létre.

4.3. A programtervről absztrakt szinten

A kivonatban említett, minden ion-atom ütközési problémára jellemző bázisosztályok és sablonok. Itt párhuzamosan kerül bemutatásra a fizikai rendszer absztrakt szinten és az azt leíró programterv, valamint az, hogy ezek hogyan kapcsolódnak egymáshoz.

4.3.1. Egy fizikai probléma fogalmi rendszere és a hozzá kidolgozott absztrakt programterv

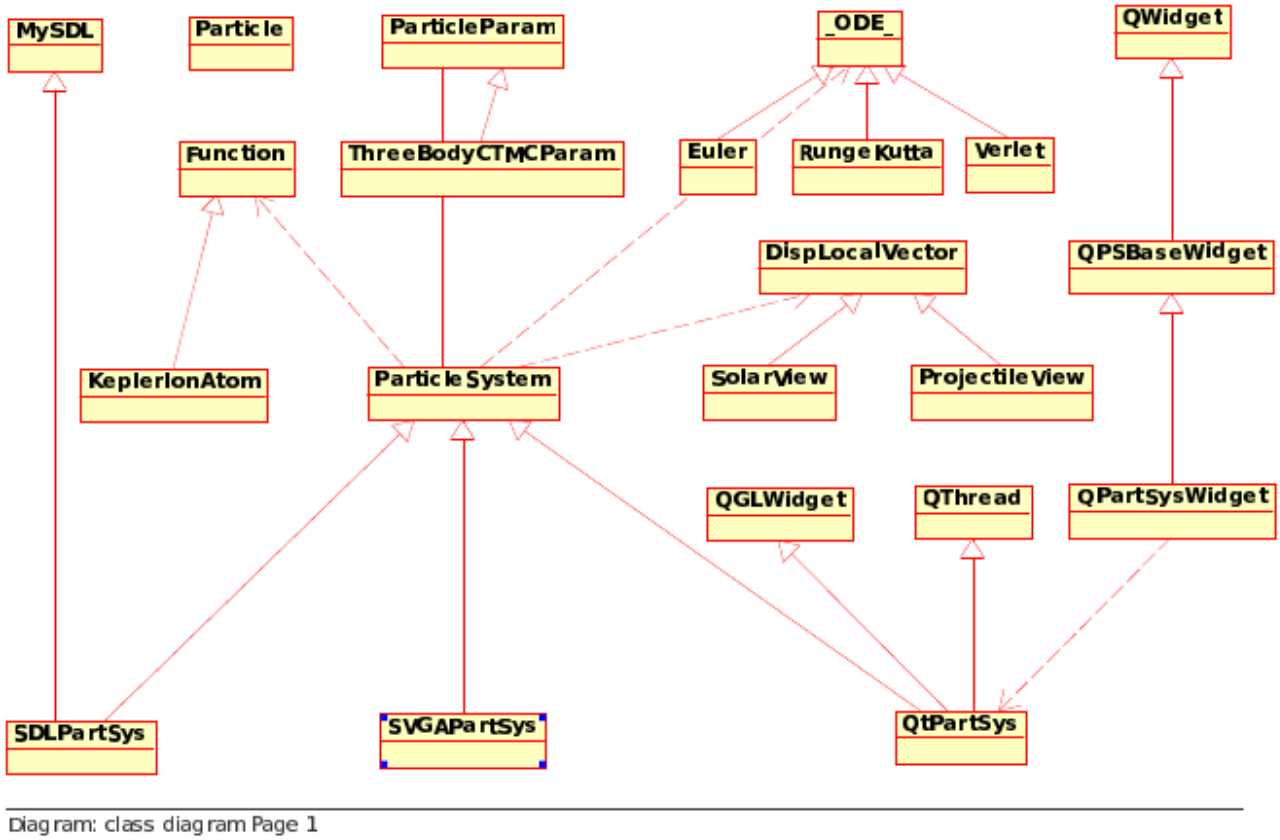
Ion-atom ütközések klasszikus szimulációja

- Paraméterek
- Részecskék rendszere

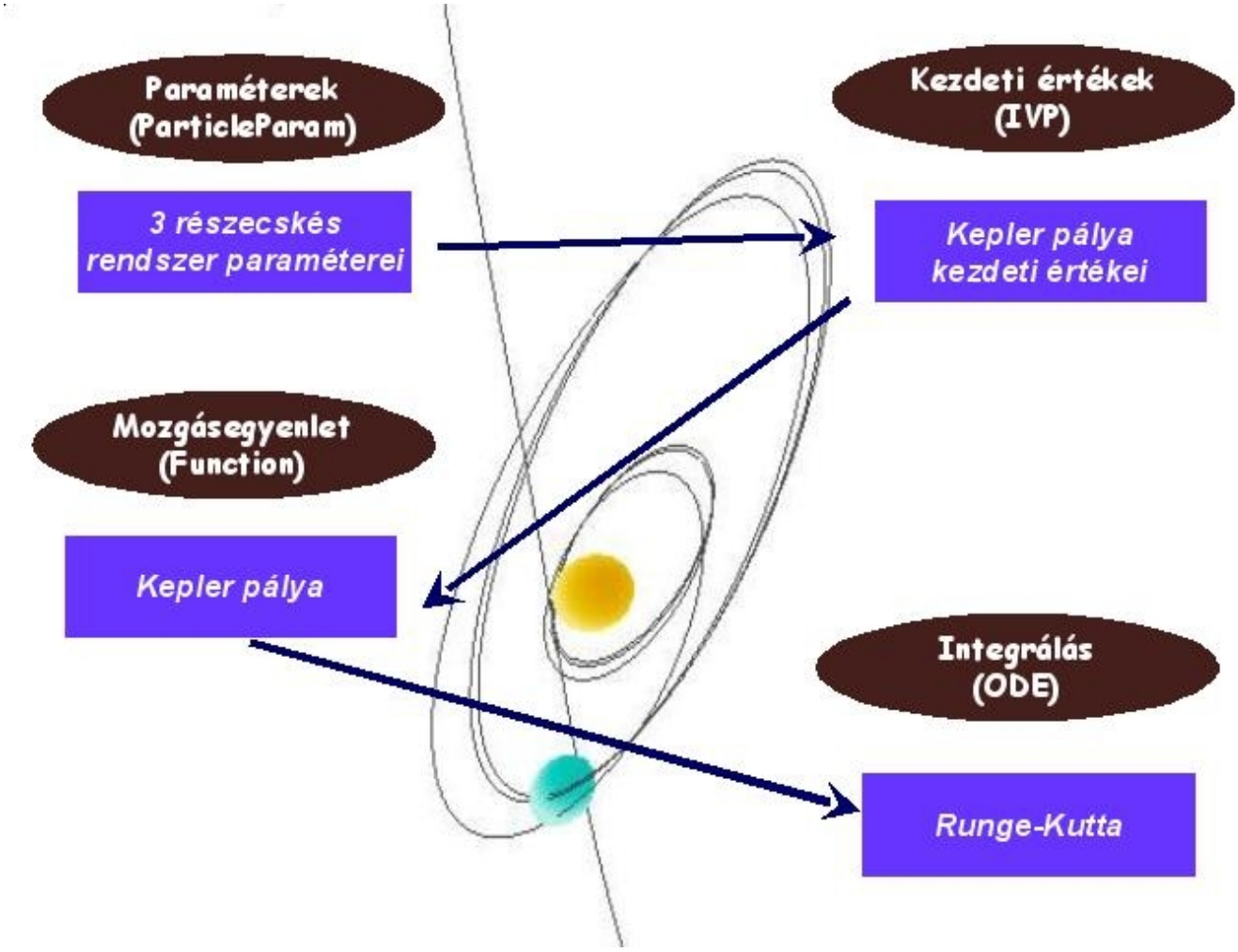
[1] Krzysztof Czarnecki, Ulrich W. Eisenecker, Generative Programming: Methods, Tools and Applications, Addison-Wesley, 2000

[2] Todd Veldhuizen, http://osl.iu.edu/ tveldhui/papers/Template-Metaprograms

[3] Andrei Alexandrescu, Modern C++ design, Addison-Wesley, 2001, http://www.moderncppdesign.com/



4.1. ábra. A programterv egy részlete



4.2. ábra. A rendszer megoldásának felépítése

- Kezdeti érték probléma
- Mozgásegyenlet
- Numerikus integrálómódszer
- Megfigyelési rendszer

Az absztrakt programterv; a fizikai rendszer fogalmainak informatikai interpretációja

Hat kisebb, fogalmi szinten diszjunkt osztályhierarchia van, amelyek gyökérelemei rendre ParticleParam, ParticleSystem, Function, IVP, ODE és DispLocalVector.

- ParticleParam:
  Fizikai rendszerek paramétereinek homogén kezelését lehetővé tevő bázisosztály. Tartalmaz egy olyan eszközt, amely segítségével lehetőségünk nyílik változók deklarációjának emulálására.Ezen eszközzel definiáljuk a fizikai rendszer konstansait(amiket megadhatunk egy paraméterállományban és amilyen nevet ott adunk neki, lényegében a programban is olyan névvel hivatkozhatunk rá), illetve a kezdeti érték probléma megoldásakor előálló értékek is ide kerülnek be, névvel ellátva. Ha valamely osztályban szükségünk van ezen értékekre, akkor elég annyit tenni, hogy átadunk egy ParticleParam típusú referenciát. A ParticleParam-ból leszármazó osztályok úgynevezett ,,környezetleíró" változókat is bevezethetnek, mint pl. grafikus rendszer számára fontos változók, leállási feltétel, stb...
- ParticleSystem:
  Ez egyben sablonosztály(a teljes fizikai rendszert modellezi, sablonparaméterei a mozgásegyenletek, a kezdeti érték probléma és a numerikus módszer) illetve a megjelenítőréteg bázisosztálya, absztrakciója. Tartalmazza a fizikai rendszert leíró adatokat, továbbá azon objektumokat(vagy azokra hivatkozást), amelyek ezen adatok segítségével előállítják a trajektóriákat. A megjelenítőréteg kirajzolja a kapott

eredményt.(tehát tartalmaz egy IVP, egy Function, egy ODE és egy DispLocalVector objektumot; tartalmaz adatokat, amelyek alapján kiszámítja a kezdeti értékeket az IVP, elkészül az integrálandó függvény, ami egy Function objektum, ezt pedig az ODE fogja integrálni, a kapott eredményeket pedig a DisplLocalVector által leírt nézetmódnak megfelelően) transzformáljuk és a megjelenítést rábízzuk az ezen bázisosztályból származtatott osztályra, ami már egy konkrét grafikus könyvtárra épül).

- IVP (Init Value Problem)

  A kezdeti érték probléma absztrakciója. A konkrét fizikai rendszertől függ és attól, hogy a programmal akarjuk kiszámíttatni a kezdeti értékeket vagy pedig inkább a paraméterállományban adjuk meg ezeket is. Amennyiben ez utóbbit választjuk, akkor annyi a teendő, hogy beolvasást végzünk számolás helyett, ezt a rugalmasságot a polimorf függvények(C++ terminológia : virtuális metódusok) biztosítják, egyébként előfordulhat az is, hogy numerikus módszereket kell megadnunk, mert csak úgy oldható meg a kezdeti érték probléma.

- Function

  A mozgásegyenletek absztrakciója. Tisztán virtuális metódusa révén interface-ként(közös felületként) szolgál a részecskék mozgását leíró differenciálegyenletek, egyenletrendszerek számára. Másképp mondva : a részecskék mozgásának absztrakciója.

- ODE (Ordinary Differential Equation)

  A numerikus módszerek absztrakciója, bázisosztálya a numerikus módszereknek.

- DispLocalVector

  A megfigyelési rendszer absztrakciója.

4.4. A programtervről konkrét szinten

4.4.1. Egy fizikai probléma konkrét leírása és a hozzá kidolgozott programterv

Ion-atom ütközések klasszikus szimulációja - 3 részecskés CTMC

- Paraméterek
- Részecskék rendszere
- Kezdeti érték probléma
- Mozgásegyenlet
- Numerikus integrálómódszer
- Megfigyelési rendszer

A konkrét programterv; a fizikai rendszer informatikai interpretációja

- ThreeBodyCTMCParam: ParticleParam osztály leszármazottja.
- ParticleSystem leszármazottai:A megjelenítendő pontok koordinátáit a ParticleSystem bázisosztályból öröklik ezen osztályok. Egyes programkönyvtárak lehetőséget adnak a párhuzamos programozásra, így a program terve olyan, hogy kihasználhassuk ezt.
  - SVGAPartSys:
    A unix rendszereken elérhető SVGAlib alacsonyszintű grafikus programkönyvtáron alapuló megjelenítő osztály. Létezik hozzá MesaGL-es kiterjesztés, ami az OpenGL egy változata és segítségével hatékony 3D-s megjelenítést lehet írni. Hátránya: nem platformfüggetlen és a szálkezelést sem támogatja alapból, erre a POSIX szálakat használhatjuk unix rendszereken. További hátránya, hogy

csak root jogosultsággal futtathatóak az SVGAlib-et használó programok, ami egy igen komoly biztonsági veszélyforrás, viszont nagyon hatékony. Mi csak a sima SVGAlib-et használjuk 2D-s megjelenítésre, demonstrálás céljából.

- SDLPartSys:

  A Simple Direct media Layer nevű programkönyvtárat használja megjelenítésre, amelyet implementáltak unix és windows rendszerekre is. Támogatja a szálak használatát és az OpenGL API is használható. Itt is csak 2D-s megjelenítőt készítettünk.

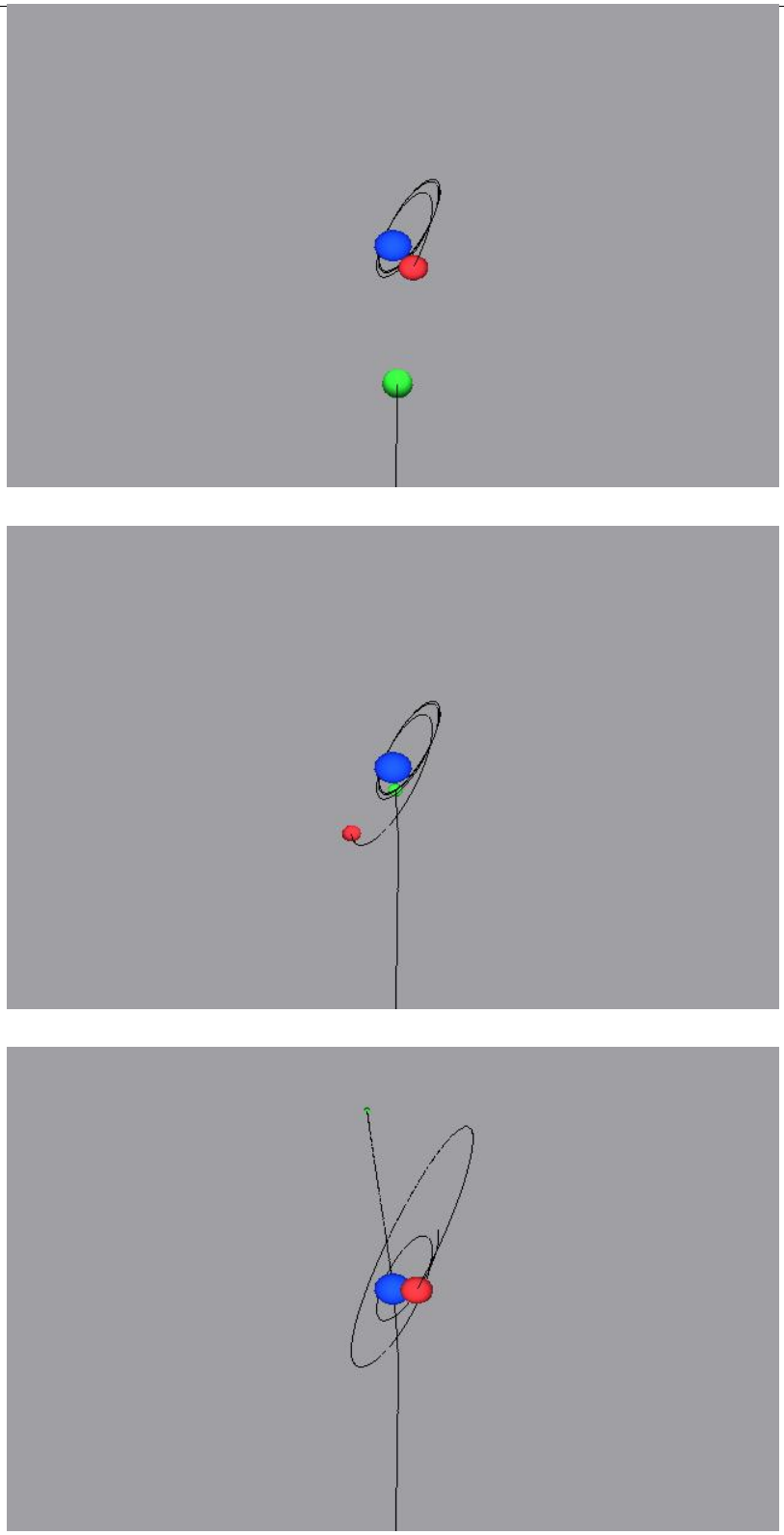
- QPartSys:

  A QPartSys a Qt programkönyvtárat használja, ami egy multiplatformos objektum orientált programkönyvtár, a C++ nyelv egyfajta kiterjesztéseként tekinthetünk rá. Tulajdonképp a statikus objektummodellű C++-t kiterjeszti egy dinamikus objektummodellű rendszerré, ami grafikus megjelenítésre nagyon jól használható. Választásunk több okból is erre a programkönyvtárra esett. Támogatja a szálakat, az OpenGL-t is és hatékony. A Qt fontossága miatt egy külön részben részletesebben is bemutatásra kerül.

- Function leszármazottjai:
  KeplerIonAtom

- ODE leszármazottjai:
  Euler, Runge-Kutta

- DispLocalVector leszármazottjai:
  SolarView, ProjectileView



4.5. ábra. Ütközés után

5. fejezet

C++ template metaprogramozás

5.1. Bevezetés

A generatív programozási paradigma napjaink egyik népszerű, aktív programozási paradigmája. Egyik ága a template metaprogramozás, amely lehetővé teszi algoritmusok, számítások fordítási idejű végrehajtását. A template metaprogramok így csökkentik a futási idejű műveletvégzést. A template metaprogramozás az az eszköz, aminek a segítségével hatékonyabb numerikus számítást végző programkönyvtárakat írhatunk C++-ban, mint Fortranban. Legfontosabb alkalmazási területei:

- expression templates - fordítási idejű sablonkifejezések
- concept checking - avagy típusok tulajdonságainak ellenőrzése
- aktív programkönyvtárak

  Nem csak eljárások és objektumok passzív gyűjteménye, aktív szerepük van a kód generálásában, azaz fordítási időben mutatnak dinamikus viselkedést. Aktív könyvtárak segítségével pl. kiterjeszthetjük a fordítóprogramot(Blitz++).

5.2. Template-ek

A c++ nyelv által biztosított fordítási időben paraméterezhető konstrukció, amely lehetővé teszi a kód(valódi) újrafelhasználását, illetve egy magasabb absztrakciós szintet biztosít.A template-ek nem csak típusokkal paraméterezhetők, hanem akár egész értékekkel, függvénypointerekkel, akár másik template-tel is. Egy adott probléma típusfüggetlen ábrázolását végezhetjük el template-ekkel. Olyan osztályokat és függvényeket írhatunk le, amelyeknek a váza megegyezik. Elkészítjük ezt a sablont, majd a felhasználás helyén megadjuk(argumentumként adjuk át a típust - template argument), hogy ezen sablonból milyen típusú verziót szeretnénk használni.Egy típus-sablonból konkrét típus létrehozás („kód-példányosítás" - instantiation) kétféle módon történhet:

- implicit : fordító által
  pl.: STL konténerek használata, $vector < int >$
  A fordító akkor példányosít egy template-t, ha az abból létrejövő típusra szükség van-igény szerinti, lusta.

- explicit : $templateclassvector < MyType >$;
  A fenti utasítással utasítjuk a fordítót, hogy az adott globális névtérbeli(!) ponton hozza létre az adott típust(ha addig még nem hozott létre ilyen típust). Ez lehetőséget ad a program hatékonyságának növelésére, azáltal, hogy több explicit példányosítást összegyűjtve a fordítónak nem kell megszakítania egy-egy kifejezés fordítását, hogy létrehozza a benne lévő típust és a template metaprogramozásban is jelentős szerepet kap.

5.2.1. Sablon-szerződés modell

Az aktuális és formális paraméterek egyeztetésének módjára szolgál. Lehetővé teszi a fordítási idejű ellenőrzést, így példányosításkor biztosak lehetünk abban, hogy a példányosítandó sablon önmagában helyes. A sablon specifikációja fogalmazza meg implicit módon a

szerződést a sablon törzse és a példányosítás között. Tehát előírást adhatunk meg a template argumentumra, azaz a sablonnak átadható típusra. Az Ada támogatja ezen modellt, a C++ nem - szándékosan nem, ugyanis kizárja annak lehetőségét, hogy a sablonnak csak azon részét használjuk, amelyet az argumentum teljesít.

5.2.2. Template-ből generált típusok ekvivalenciájáról:

Két, ugyanabból a template-ből legyártott típus csak akkor egyezik meg, ha minden template argumentum rendre megegyezik.

5.2.3. Specializáció, részleges specializáció

A template-ek csak sémák arra nézve, hogy a fordító hogyan generáljon futtatható kódot. Ha valamilyen oknál(pl.hatékonysági) fogva szeretnénk bizonyos típusok esetén eltérni az alapértelmezett sablontól, akkor készíthetünk ezen típusok számára specializációkat. *Részleges specializációnak* nevezzük az olyan specializációt, amelyben konkrét típusokkal együtt template paraméterek ugyan továbbra is lehetnek, de ezekre tett szintaktikai megkötésekkel szűkítjük a lehetséges argumentumok körét.

5.2.4. typedef és enum

A **typedef** kulcsszóval egy új álnevet(*alias*) adhatunk egy adott típusnak, nem hoz létre új típust, nem vizsgálja annak szintaktikai helyességét(tehát egy egy még definiálatlan, de már deklarált típusnak is lehet *alias*-t adni). Az **enum** kulcsszóval pedig új felsorolási típust hozhatunk létre. A metaprogramok általában ebben tárolják az adatokat(lévén a felsorolási típus fordítási időben létrejövő konstrukció, mint pl. egy static const változó).

5.3. Metaprogramok

5.3.1. Története

A C++ nyelv 1994 óta támogatja a template-eket, akkor került bele a templte specifikáció a nyelv szabványtervezetébe. A fő cél az egyszerűbben karbantartható program volt, amelyet a típussal való paraméterezés útján értek el. Ugyanebben az évben írt Erwin Unruh egy speciális programot, amely ugyan soha nem fordult le, prímszámokat generált: 1-től egy megadott felső határig ellenőrzött int típusú számokat és típuskonverziós hibát dobott azon, fordítás közben létrejövő típusok esetén, amelyek argumentuma egy prím volt. Ez volt az első C++ template metaprogram.

5.3.2. Működése

A fordító abban az esetben fog példányosítani egy sablon alapján, ha egy adott kifejezés vagy deklaráció kiértékelésében számára ismeretlen típus van, és a típus létrehozható valamelyik template definícióból. Megfelelően megfogalmazott deklarációkkal meghatározhatjuk, hogy egy adott template a program mely pontján példányosuljon. Példa:

```
template <int N>
struct Factorial
{
    enum { value = N*Factorial <N-1>::value };
};

template<>
struct Factorial<1>
{
    enum { value = 1 };
};

int main ()
{
    int r=Factorial<5>::value;
}
```

A két template definíció elemzése után a fordító megpróbálja kiértékelni a főprogramban lévő értékadó kifejezést. Ahhoz. hogy a $Factorial < 5 >:: value$ kifejezés értékét meghatározza, a számára egyenlőre definiálatlan $Factorial < 5 >$ típust kell létrehoznia. Megkeresi, hogy van-e ilyen nevű template, amiből a gyártást elkezdheti. A template létezik, N helyébe 5-öt írva lefordítja az adott kódrészletet. Ahhoz, hogy az ebben elhelyezkedő $Factorial < N-1 >:: value$-t, azaz $Factorial < 4 >:: value$-t ki tudja számítani, példányosítania kell a $Factorial < 4 >$ típust, stb... Végül a már létező $Factorial < 1 >$ típushoz ér, ekkor áll le. Vagyis a fordítót rekurzív template példányosításra kényszerítjük, specializációt használva állítjuk le a rekurziót. A kódba a fordító már a végeredményt fogja beírni, tehát futási időben nem fogunk már faktoriálist számítani. A template metaprogramming Turing-teljes, a Bohm-Jacopini-tétel szerint minden elágazást és rekurziót tartalmazó nyelv Turing-teljes.

A rekurziót már láttuk, most jöjjön egy példa fordítási idejű elágazásra:

```
template <bool Cond, class TrueType, class FalseType>
struct IF
{
    typedef TrueType type;
};

template <class TrueType, class FalseType>
struct IF<false,TrueType,FalseType>
{
    typedef FalseType type;
}
```

6. fejezet

A Qt rövid ismertetése

6.1. Bevezetés

A Qt a norvég Trolltech cég terméke, egy multiplatformos C++ programkönyvtár, amely már több, mint 400 osztályt tartalmaz. Az osztályok több területet is lefednek, mint pl. GUI, hálózati, párhuzamos, adatbázis programozás, támogatja az XML-t, OpenGL-t is, továbbá lehetőséget ad a nemzetközi programozásra is. Támogatja a Unix(AIX, BSDI, FreeBSD, HP-UX, Irix, Solaris, Linux, stb. . .), MacOsX, Windows(9x, NT4, ME, 2000, XP) rendszereket.

6.2. A Qt története

A Qt története a 80'-as évek végén, 90'-es évek elején indult, két frissen végzett norvég egyetemista (Haavard Nord, Eirik Chambe-End) együtt dolgozott egy ultrahang képek számára készülő C++ adatbázis-alkalmazáson. Ennek a rendszernek olyannak kellett lennie, ami több platformon fut(Unix, Macintosh és Windows) és grafikus kezelői felülete is van. A történet szerint egy nyári napon kimentek egy parkba napozni és Haavard azt találta mondani, hogy ,,Egy objektum-orientált megjelenítő rendszerre van szükségünk."

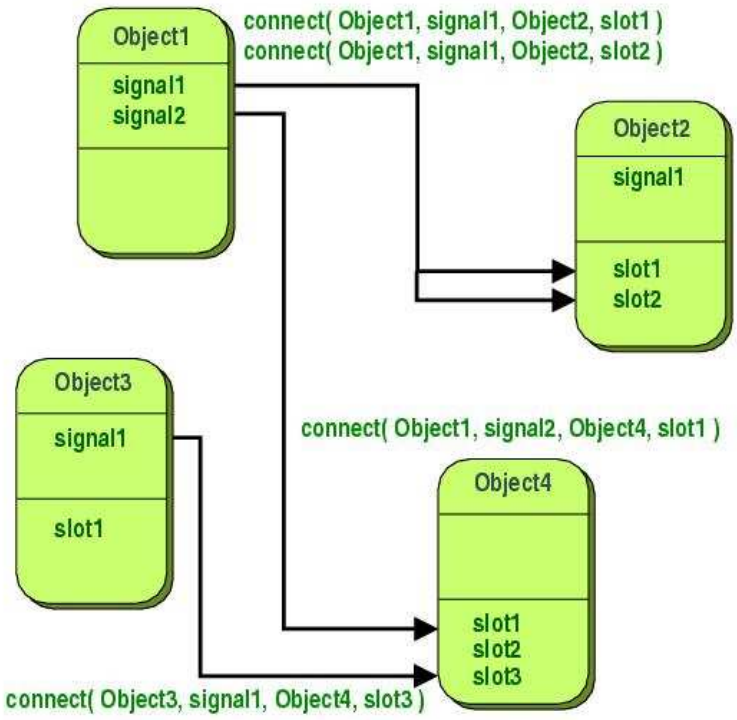
Aztán megírt egy jópár osztályt, ami végülis a Qt alapja lett. Eirik kitalálta a signal-slot modellt, 1993-ra megírták a Qt első grafikus magját és megcsinálták hozzá a saját grafikus elemkészletüket(widget). Honnan jön az elnevezés? Ez érdekes, a ,,Q" ugyanis azért Q, mert jól mutatott Haavard Emacs betűkészletében. A ,,t" hozzáadása pedig az Xt(X toolkit) elnevezés alapján történt. 1994-ben megalapították a Quasar Technologies-t, ami később Troll Tech lett, majd végül Trolltech. 1995-ben jelent meg az első verzió (0.9). Első vásárlóik között volt (egész pontosan a második) az ESA (European Space Agency). 1996-ban indult a KDE project Matthias Ettrich vezetésével, aki a Qt-t választotta alapkönyvtárnak. Jelenleg a 4.x.y verziónál járunk.

6.3. A Qt fogalmi rendszere

6.3.1. signal-slot modell

kulcsszavak:Q_OBJECT, signals, slots, connect, disconnect, emit

A legegyszerűbben úgy lehet ezt a modellt megközelíteni, hogy elképzelünk egy aljzatot, amire rácsatlakoztathatunk eszközöket, amennyiben van nekik mivel rácsatlakozni. Minden osztály definiálhatja a saját aljzatát, ami minden egyes objektumához elkészül és ezen aljzathoz csatlakozhatnak egy (vagy több) olyan osztály objektumai, amelyek rendelkeznek csatlakozóval, sőt, saját maga is csatlakozhat saját magára(nem lesz rövidzárlat, más hasonlattal élve: hermafroditák szaporodása). Egy aljzatba csak a megfelelő csatlakozó dugható be, az aljzat alakjával analóg fogalom a szignatúra. Egy kicsit pontosítva: minden osztály megadhat signal-okat és slot-okat. Egy signal kiváltódásakor a hozzákapcsolt slot lefut. Ha a signal-nak vannak paraméterei, akkor azok átadódnak a slot-nak. Fontos az, hogy a signal formális paraméterlistájának(szignatúra) kompatibilisnek kell lennie a slot formális paraméterlistájával, azaz vagy ugyanannyi, ugyanolyan típusú paraméterekből állnak vagy a slot-nak lehet kevesebb is, hogy kiszűrje a



számára érdektelen paramétereket - így a signal-slot mechanizmus típusbiztos szemben más rendszerek callback-mechanizmusával.(aljzatos analógia : van egy olyan aljzat, amihez pl. max. 8 villás csatlakozót lehet illeszteni, de ugyanúgy kezeli a 7,6, 2 villás csatlakozókat is) Tehát a signal-slot mechanizmus lehetővé teszi, hogy összekössünk objektumokat egymással anélkül, hogy azok bármit is tudnának egymásról. Tulajdonképp úgy is fogalmazhatnánk, hogy van egy objektumunk, ami csinál valamit, aztán a cselekvése során eljut egy olyan állapotba, amikor az addigi tevékenységének eredményéről értesít egy másik objektumot és esetleg át is ad neki adatokat, amivel adott esetben a másik objektum is elkezd valamit csinálni, majd szintén értesít egy harmadik objektumok, ami felhasználja a kapott adatokat(pl.: van egy fizikai problémánk - ion-atom ütközés, CTMC módszerrel számítjuk a trajektóriát, szálként fut a számoló rész, van egy időzítőnk, ami ha leszámolta a megadott időtartamot, kivált egy signal-t, amiben elküldi a megjelenítő objektumnak a koordinátákat és megjeleníti azt - a program futása így nem más, mint egymással diszjunkt objektumok interakciója)

Technikai részletek:

Ha egy C++ osztályhoz signal-okat, slot-okat akarunk hozzáadni, ahhoz először is szerepeltetni kell az osztálydefinícióban a Q_OBJECT makrónevet(az osztálynak a QObject-nek kell a leszármazottjának lennie, pontosabban a QObject-ből kiinduló öröklési lánc valamely elemének), ezután már adhatunk meg egy signals: részt, ide kerülnek a signal-ok specifikációi, majd adhatunk meg egy public—private—protected slots: részt is, ide jönnek a slot-függvények. A signal-ok hozzáférését nem szabályozhatjuk, azokat nem is kell implementálnunk. A slot-ok viszont 1-2 megszorítással ugyanolyan C++ - függvények, mint a többi(ez azt jelenti, hogy akár virtuálisak is lehetnek, vagyis polimorf slotok). A megszorítások a következők: ha egy függvény slot-függvény, akkor csak void lehet a visszatérési értéke(vagyis eljárás jellegű függvény, nincs visszatérési értéke), a másik pedig az, hogy a formális paraméterlista egyetlen tagja sem kaphat alapértelmezett kezdőértéket. (ill. van még egy ,,apróság", ami ugyan nem megszorítás, de jó, ha figyelembe vesszük: egy slot függvény csak annyit csináljon, amit feltétlen muszáj, tehát ne tegyünk bele pl. egy alkalmazkodó lépésközű Runge-Kutta módszert, ugyanazon ok miatt, ami miatt egy megszakítási rutinban sem teszünk bele ilyesmit- hatékonyság). Signal-ok kiváltása történhet implicit és explicit(emit makró) módon is. Egy adott signal-t csak az öt definiáló osztály (illetve az abból származtatott osztályok) példányai válthatnak ki. A signal-okhoz kapcsolt slot-ok előre nem meghatározott sorrendben futnak le, nem determinisztikusan.

signal-ok, slot-ok összekapcsolása

a connect makróval történik, ami a következőképp néz ki:
connect(sender,SIGNAL(signal), receiver, SLOT(slot)); vagy
connect(sender,SIGNAL(signal), receiver, SIGNAL(signal)); vagy
, ahol a *sender* és a *receiver* egy-egy QObjectre hivatkozó mutató típusú változó, a *signal* és a *slot* pedig a signal és slot függvények szignatúrái, paraméternevek nélkül(tehát csak a típusokat kell szerepeltetni). A SIGNAL() és SLOT() makrók lényegében string-gé

konvertálják az argumentumukat.

Kapcsolódási lehetőségek:

- Egy signal kapcsolódhat több slot-hoz
  Ha kiváltódik a signal, a hozzákötött slot-ok egymás után futnak le, tetszőleges sorrendben, tehát nem biztos, hogy a felírás sorrendjében.
- Több signal kapcsolódhat egy slot-hoz
  Ha valamelyik signal kiváltódik, a slot lefut.
- Egy signal kapcsolódhat egy másik signal-hoz
  Amikor a signal-láncolatban első signal kiváltódik, akkor kiváltódik a többi is.

kapcsolatok eltávolítása:

disconnect(sender,SIGNAL(signal), receiver, SLOT(slot));

Erre ritkán van szükség, mert ha egy objektum törlődik, akkor a Qt automatikusan megszünteti azon kapcsolódásokat, amelyek erre az objektumra vontakoztak.

a hatékonyságról:

A callback-metódusok hívása kicsit gyorsabb, mint a signal-slot mechanizmus, de azok nem olyan flexibilisek, mint ez a megoldás(típusbiztonság, stb. . .). Általában egy néhány slot-hoz kötött signal kiváltása tízszer lassabb, mint ha direkt hívnánk meg a fogadó objektum metódusait(nem virtuális metódusok esetén - virtuális metódusok hívása lassabb!). Ezalatt az idő alatt a Qt nem alszik, hanem meghatározza a kapcsolódó objektumokat, biztonságos módon végigmegy rajtuk(ellenőrzi, hogy a fogaó objektumok léteznek-e még, avagy időközben megszűntek), illetve átadja a szükséges paramétereket. Viszonyításképp egy **new** vagy **delete** operátor jóval lassabban fejti ki hatását, mint a signal-slot mechanizmus által előirt műveletek, szóval annyira azért nem vészes a helyzet. A Trolltech adatai szerint egy i586-500 gépen egy másodpercen belül 2,000,000 signal váltható ki, amelyet egy fogadó obejtktum figyel, illetve 1,200,000 signal, amennyiben két fogadó objektum

kapcsolódik hozzá(nem lineárisan csökken a sebesség!).

6.4. Qt & C++

6.4.1. MOS - Qt Meta-Object System

A Qt egyik nagy eredménye, hogy kiterjeszti a C++-t egy olyan mecanizmussal, ami lehetővé teszi független program-komponensek összekapcsolását anélkül, hogy az összekötött komponensek bármit is tudnának egymásról - ez nemcsak GUI esetén hasznos. Ezt a lehetőséget a meta-object system biztosítja. A MOS két fő szolgáltatása: a signal-slot modell és az önelemzés(*reflection*). Itt fontos megjegyezni, hogy a Qt egy statikus objektummodellű programozási nyelvet terjeszt ki dinamikus objektummodellű rendszerré, mint amilyen az Objective-C vagy a Java, C# is. Az önelemző képességre szükség van a signal-slot modell miatt, ez a funkcioanlitás lehetővé teszi azt is, hogy a programozó meta-információkhoz férjen hozzá egy QObject-ből származtatott osztály esetében futási időben. Ezen információ pl. lehet az, hogy milyen signal-jai, slot-jai vannak az adott osztálynak. Illetve támogat ún. dinamikus tulajdonságokat(Dynamic Property System), amit pl. a Qt Designer(GUI builder) használ vagy a nemzetközi programozás során a tr() függvény használja. QObject osztályok példányai esetén használhatjuk a qobject_cast<T>() -ot, ami a C++-beli dynamic_cast<T>-hoz hasonlítható funkcióját tekintve, ám nem igényli a natív C++ RTTI támogatást. Ha az objektum típusa megfelelő, akkor a qobject_cast<T>() 0-t ad vissza, egyébként nem. Ha a QObject egy leszármazottjában nem szerepeltetjük a Q_OBJECT makrót(nem adunk meg signal-okat, slot-okat, property-ket, stb...), akkor nem rendelődik meta-object kód az osztályhoz. Ez a MOS szemszögéből azt jelenti, hogy ezen osztály ekvivalens a hozzá legközelebbi elődjével az osztályhierarchiában, amely rendelkezik meta-object kóddal. A Qt számára a C++ statikus objektummodellje hátrány, hogy mégis tudja biztosítani a fenti dinamikusságot, az a *moc (Meta Object Compiler)*-nak köszönhető. A moc egy előfordító, azon

osztályokhoz generál meta-object kódot, amelyekben szerepel a Q_OBJECT makró. A moc ,,tiszta" C++ kódot generál, így a Qt MOS minden C++ fordítóval működni fog, magyarul egy szabványos C++ fordító és a Qt moc elérhetővé teszi C++-ban a dinamikus objektummodell használatát. Hogyan működik?

- A Q_OBJECT makró deklarál néhány önelemző függvényt , amiket implementálnia kell minden QObject leszármazottnak.
- A moc legenerálja az önelemző függvények és a signal-ok implementációját
- a QObject függvénytagok mint pl. a connect() vagy diconnect() az önelemző függvényeket használva végzik el feladatukat

Mindezt automatikusan kezeli a Qt a qmake,moc eszközeivel.

6.4.2. Qt vs. C++ template

A generikus paradigma mentén történő programozás sok probléma esetén igen jól használható (pl. fordítási időben kiértékelhető sablonkifejezések), ám egy olyan rendszer esetén, ahol pl. dinamikusan épül föl egy felhasználói felület egy XML file-ból, ott nem igazán használható. Qt-ben nem adhatunk meg olyan sablonosztályt, amelyben szerepel a Q_OBJECT makró. Ha mégis, akkor a moc kiírja, hogy a Qt nem támogatja azon sablonosztályokat, amelyekben signal-ok, slot-ok vannak avagy Q_OBJECT. Miért nem támogatja a Qt az ilyen sablonokat? Ennek legfőbb oka az, hogy a különböző sablonparaméterekkel történő instanciációk száma nem meghatározható és mindegyik instanciáció más és más meta-object kódot igényelne. Vannak azonban olyan esetek, amikor szeretnénk signal-slot mechanizmussal bővíteni a sablonosztályunkat. Ekkor mi a teendő? Két dolgot tehetünk:

- megadunk egy bázisosztályt, amelyben szerepel a Q_OBJECT makrónév, megadjuk a signal-okat, slot-okat, majd készítünk egy olyan sablonosztályt, ami ennek a leszármazottja. Ez a megoldás akkor használható, ha a signal-oknak és a slot-oknak nincs szükségük a sablonparaméterekre(nem függnek tőlük).

- Figyelő (Observer) programtervezési minta (design pattern) használata. A Figyelő minta arra szolgál, hogy egy adott objektum belső állapotának válozásáról értesítsen egy vagy több másik objektumot. A Qt erre ad is lehetőséget az ún. eseményszűrőkön(eventFilter) keresztül.

7. fejezet

Összefoglalás

A dolgozatban bemutatásra kerültek mindazon ismeretek, amelyek egy ion-atom ütközést *hatékonyan* szimuláló és 3D-ban megjelenítő program elkészüléséhez szükségesek: fizikai alapismeretek, a szimulált rendszer mozgásegyenletei, C++ template metaprogramozás a Fortran-tól is hatékonyabb numerikus számítások elvégzéséért, a megjelenítésért felelő és a felhasználói felületet leíró Qt programkönyvtár. A program C++-ban íródott és objektum orientált , illetve generikus paradigmák mentén épül fel, kihasználja mindkét paradigma által nyújtott előnyöket. Az elkészült program valós időben képes megjeleníteni a teljes szimulációt, lehetőséget ad tetszőleges t_i időpillanat rögzítésére(JPEG formátumú file-ként), de video(AVI) file-t is képes készíteni a teljes szimulációról(mindeközben persze a felhasználó számára is látható a szimuláció). Többféle irányban is lehetséges továbbfejleszteni, mind informatikai(absztrakt megjelenítő réteget és számító réteget készítve akár egy bluetooth-os telefonon is megjeleníthetnénk a szimulációt, SOAP service-ek formájában elérhetővé lehet tenni a számolást végző alrendszer szolgáltatásait,stb. . .) mind fizikai(Coulumb-potenciál helyett modelpotenciál alkalmazása, Fermi-gyorsítás szimulálása) területen.

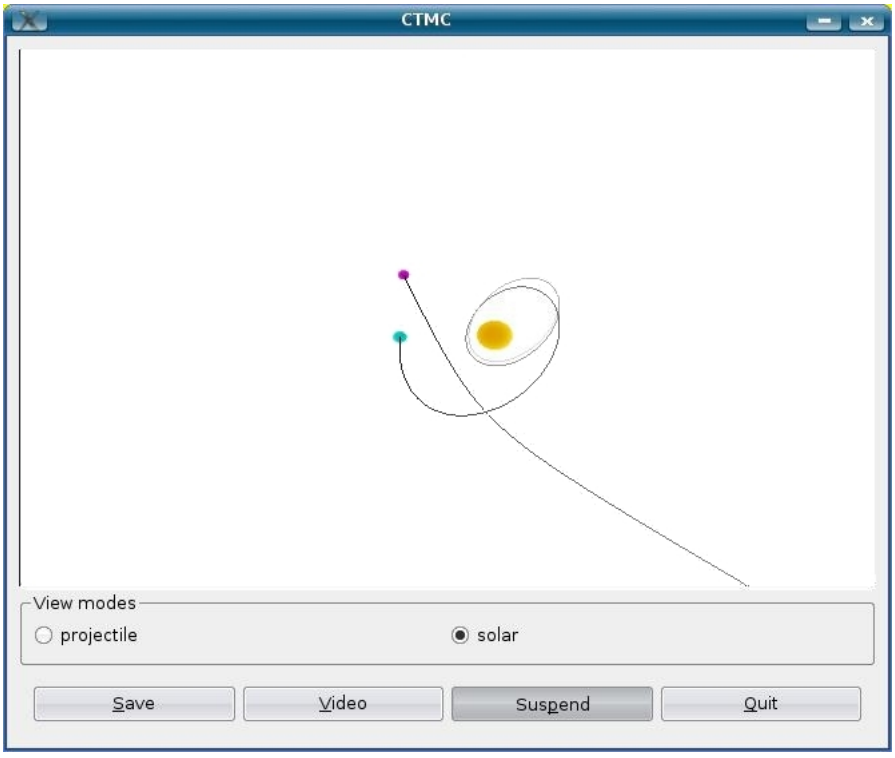
8. fejezet

Köszönetnyilvánítás

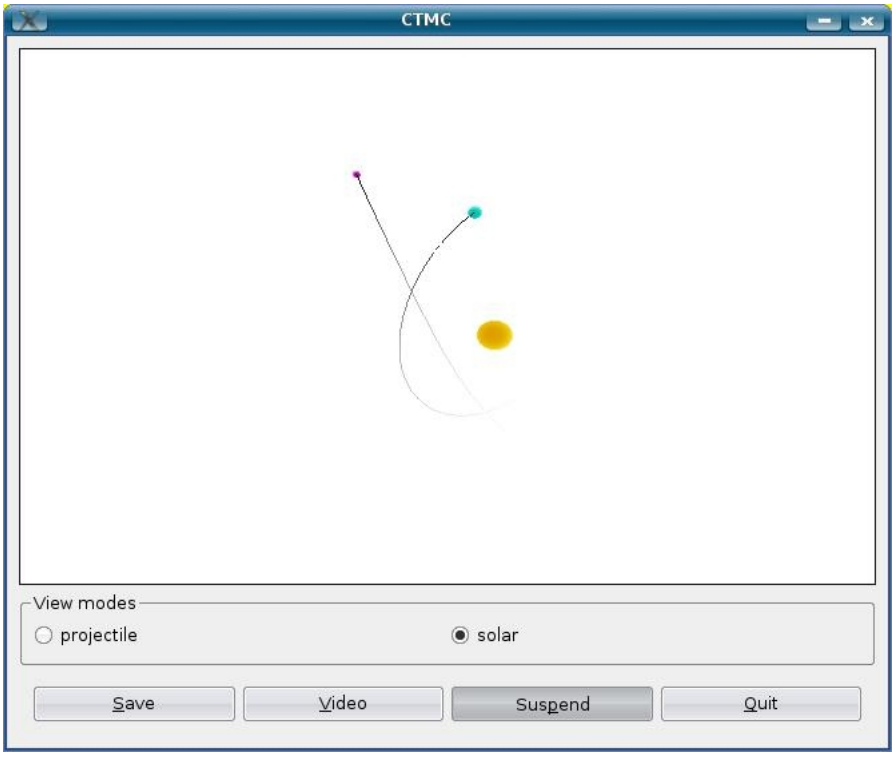
Ezúton szeretnék köszönetet mondani témavezetőmnek, Dr. Tőkési Károlynak a szakdolgozat elkészítésében nyújtott segítségéért, éjszakába nyúló ellenőrző munkájáért. Köszönettel tartozom Dr. Halász Gábornak, amiért segített a dolgozat megszületését akadályozó külső tényezők megszüntetésében.

9. fejezet

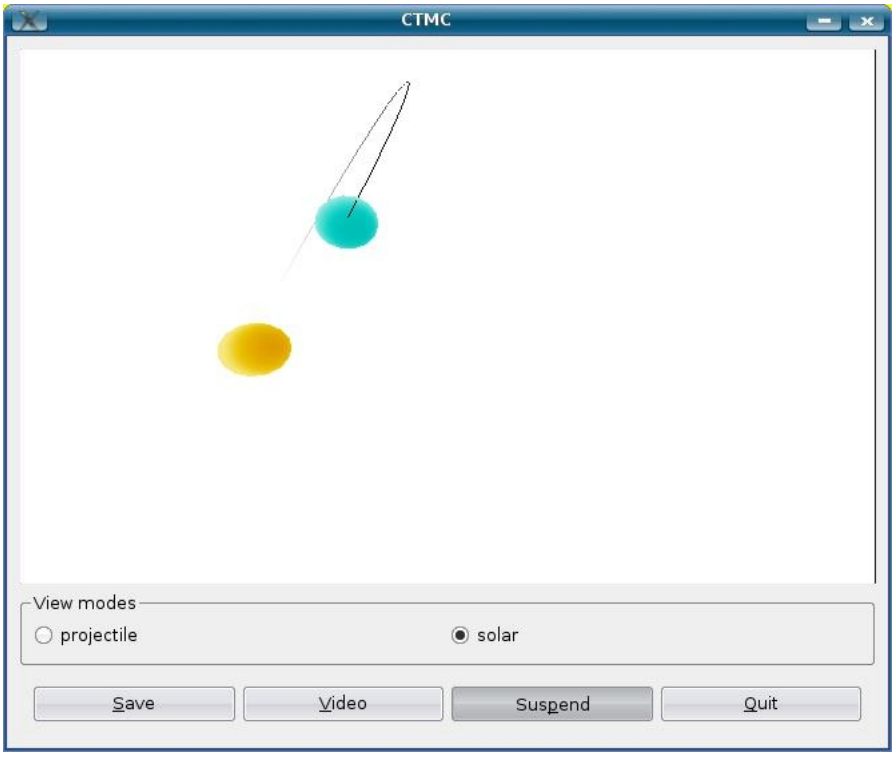
Képek a programról



9.1. ábra. A program futás közben



9.2. ábra. A program futás közben



9.3. ábra. A program futás közben

Irodalomjegyzék

[1] Tőkési Károly :*Ion-atom ütközési folyamatok modellezése Monte Carlo módszerrel. Ph.D értekezés. Témavezetô: Pálinkás J.* MTA-ATOMKI, 1998

[2] K. Tőkési, G. Hock, *Nucl. Insrtum Meth. in Phys. Res.* **B86**(1994)201

[3] Kun Ferenc :*Determinisztikus folyamatok számítógépes modellezése egyetemi előadás* Debreceni Egyetem, Elméleti Fizika Tanszék, 2001

[4] L. Budai, A. Pénzes, Á. Korda, J. Somodi, K. Tőkési :*3D visualization of classical trajectories in ion-atom collisions* ATOMKI Annual Report 2005 8.16 (2005)

[5] *Design Patterns*, Gamma, et al., Addison-Wesley 1995

[6] Veldhuizen, T: *Expression Templates* C++ report **Vol.7**(5), p26-31 (1995)

[7] Veldhuizen, T: *Using C++ Template Metaprograms* C++ report Vol. **Vol.7**(4) p36-43 (1995)

[8] Veldhuizen, T: *Will C++ be faster than Fortran?* Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97)

[9] Volker Hilsheimer : *Academic Solutions to Academic Problems*, Issue 15,Qt Quarterly, Q3 2005

[10] *C++ GUI Programming with Qt 3*, Jasmine Blanchette, Mark Summerfield, Prentice Hall PTR (2004)

[11] *The C++ programming language(special edition)*, Bjarne Stroustrup, Addison-Wesley (2000)