Latvijas Universitāte
Fizikas un matemātikas fakultāte
Datorikas nodaļa

Collision Detection:
Cieto ķermeņu kustības un sadursmes modelēšana

Kursa darbs

Autors
Andrejs Ivanovs

Vadītājs
LU profesors
Paulis Ķikusts

Rīga, 2005.

Anotācija

Fizisko procesu modelēšana sākas no datorspēlēm un aptumsumu pareģošanas līdz konstrukcijas izturības pārbaudei. Jebkura modelēšana sevī paredz atsevišķu parametru ignorēšanu viņu realizācijas sarežģītības dēļ vai arī izskaitļojot resursu nepietiekamību.

Manā kursa darbā es pētu vienu no metodēm, kuras būtība sastāv no priekšstata par objekta izteiksmi caur daudzstūriem. Tiek izskatītas objektu kustības problēmas, esības pārbaude un kolīzijas apstrāde (objektu sadursme). Objekti reaģē uz sadursmi pēc fizikas likumiem, ņemot vērā enerģiju, impulsu, rotācijas momentu un ķermeņu savstarpējo pievilkšanos. Galvenie noteikumi – kustībai ir jābūt reālistiskai un sistēmai noteiktā laikā jāapstrādā liels daudzums objektu.

Abstract

Modeling of physical processes spreads from computer games and predictions of eclipses to checking the durability of designs. Any modeling means neglecting some parameters because of their complexity in realization or lack of computing resources.

In my work I describe one of the methods which consists of representation of objects in the form of polygons. Problems of movement of these objects, collision detection and processing are described. Objects react to collisions according to the laws of physics, considering energy, an impulse and a mutual gravitation. The main requirements - described movement should be realistic and the system should process plenty of objects in real time.

Аннотация

Моделирование физических процессов простирается от компьютерных игр и предсказания затмений до проверки прочности конструкций. Любое моделирование подразумевает в себе пренебрежение некоторых параметров из-за их сложности в реализации или недостатка вычислительных ресурсов.

В моей курсовой работе я рассматриваю один из таких методов, суть которого состоит в представлении объектов в виде многоугольников. Рассмотрены проблемы движения этих объектов, проверка наличия и обработка коллизий (столкновений объектов). Объекты реагируют на столкновения по законам физики, учитывая энергию, импульс, момент вращения и взаимное притяжение. Главные требования к работе – движение должно быть реалистичным и система должна обрабатывать большое количество объектов в реальном времени.

Autoreferāts

Šajā kursa darbā autors pēta fizisko procesu modelēšanas problēmas. Konkrēti ir apskatīta ķermeņu kustība divdimensiju telpā. Jebkura modelēšana sevī paredz atsevišķu parametru ignorēšanu viņu realizācijas sarežģītības dēļ vai arī izskaitļojot resursu nepietiekamību. Šajā darba ķermeņi ir attēloti ka divdimensiju daudzstūri, kuri var kustieties, griezties un sadursties viens ar otru.

Ķermeņu kustība ir sadalīta laika kvantos (Eilera metode) lai vienkāršot rēķinus un izvadu uz ekrāna. Ir apskatīti dažādi kolīzijas noteikšanas algoritmi. Visslābākais no tiem ir pakāpeniskais algoritms, kur izmanto faktu, ka divu augšupejošu kustību kompozīcija arī ir augšupejoša kustība. Autors izstrādāja kolīzijas apstrādes algoritmu augšupejošai kustībai, kurš ir bāzēts uz impulsa atdošanas un enerģijas saglabāšanas principa.

Ir realizēti datu struktūras, pakāpeniskais algoritms un kolīzijas apstrādātājs C++ valodā ar STL un OpenGL palīdzību. Kursa darbs var noderēt pamatam vienkāršam datorspēles divdimensiju fiziskam modelim bez rotācijas.

Saturs

IEVADS **7**
Particle system *7*
Constrain based *7*
Daudzstūri uz impulsu metodes pamata *8*
PIELIETOJUMS 8
VĒSTURE (RELATED WORK) 8
DATU STRUKTŪRAS **10**
GLABĀŠANA 10
MASA 11
MASAS CENTRS 11
INERCES MOMENTS 11
KINEMĀTIKA **14**
AUGŠUPEJOŠA KUSTĪBA 14
Eilera metode *14*
ĶERMEŅA ROTĒJOŠĀ KUSTĪBA APKĀRT NEKUSTĪGAI ASIJ 15
KUSTĪBA 16
KOLĪZIJAS NOTEIKŠANA **17**
TRIVIĀLĀ METODE 17
PAKĀPENISKAIS ALGORITMS 17
Nogriežņa un daudzstūra krustošanās noteikšana *18*
CILINDRU METODE – CĪŅA AR ROTĀCIJU 20
PAKĀPENISKAIS ALGORITMS + ... 21
KOLĪZIJAS APSTRĀDE **25**
ĶERMEŅU ĀTRUMA NOTEIKŠANA PĒC KOLĪZIJAS 27
Teorētiskais pamatojums *27*
Kolīzijas apstrāde pie augšupejošas kustības *28*
PRODUKTIVITĀTES PAAUGSTINĀŠANAS METODES **32**
KINEMĀTIKA 32
KOLĪZIJAS NOTEIKŠANA 32
KOLĪZIJAS APSTRĀDE 34
NOSLĒGUMS **35**
LITERATŪRAS SARAKSTS **36**
PIELIKUMS (PROGRAMMAS TEKSTS) **38**

Ievads

Modeļu radīšana, lai aprakstītu fiziskās norises ir sarežģīta lieta [M1]. Ir nepieciešama zināma pieredze, lai ātri izprastu kādi faktori ietekmē doto fizisko procesu. Citu procesu vidū jāizceļ tie procesi, kuri ir galvenie un kurus var neņemt vērā. Kādēļ dažus fiziskos procesus var neņemt vērā? Lieta ir tāda, ka visu paredzēt ir neiespējami un bezjēdzīgi. Nav jēgas radīt realitāti, censties izveidot modeli, kurš izskatīsies labi, lai cilvēka intuīcija nepretotos tam, ko redz acs. Realitāte ir neatkārtojama, bezjēdzīgi modelēt pasauli pilnībā tādu kāda tā ir. Teorētiski nav iespējams atveidot mūsu pasauli pilnībā, kaut vai tikai pēc Heizenberga nenoteiktības principa [F3]. Piemēram, modelējot akmens krišanu uz zemes no viena metra augstuma, var neņemt vērā brīvā kritiena paātrinājuma izmaiņas attiecībā ar augstumu, kā arī mēness pievilkšanās spēkus un citus faktorus. Modelējot paisumu un bēgumu nedrīkst ignorēt mēness pievilkšanos spēkus, jo tas ir galvenais spēku faktors.

Modelējot fiziku spēlēs ir nepieciešams rūpēties par to, lai fiziskais modelis neizmantotu pārmērīgi daudz datora resursu. Pirmkārt, ir nepieciešams ierobežot pasauli, noteikt, kas būs šai pasaulē un kā nebūs. Piemēram, ir iespējams ignorēt speciālu relatīvisma teoriju balstoties uz to, ka Visums palielinās vājajiem un stiprajiem mijiedarbojoties, kā arī ar to, ko vēl nav atklājusi mūsu zinātne. Parasta cilvēka pasaules modelī nav saprašanas par tādiem aspektiem, tātad šo informāciju var ignorēt, neizjaucot reālisma principu. No citas puses, var nezināt impulsa saglabāšanas likumu, bet tā neesamība modelējamajā pasaulē būs „sist pa acīm”. Ignorēt pienāksies arī to, ko sākotnēji neņemt vērā nedrīkst, iespējams būs nepieciešams aizvietot dažus faktorus uz tiem ekvivalentiem faktoriem. Piemēram, tas, ka zeme ir apaļa, zina visi, bet ikdienas dzīvē mēs to neievērojam. Pat Latvijas robežās, virsma liekas plakana (pastāv daudz un dažādu Latvijas karšu, bet nav Latvijas globusa). Citiem vārdiem sakot, modelējamajā pasaulē zemes virsmu var izveidot par plakanu un aizvietot pievilkšanās spēku zemes centram uz spēku, kas iedarbojas uz leju, perpendikulāri virsmai, kuras spēks jebkurā augstumā ir nemainīgs. Pastāv citi ierobežojumi, tīri tehniski. Neiespējami glabāt precīzas objekta koordinātas, jo dators operē tikai ar ierobežotas precizitātes skaitļiem.

Cits ierobežojums ir saistīts ar programmas darba laiku. Neiespējami pārrēķināt objektu koordinātas bezgalīgi daudz reizes, tādēļ pienākas dalīt laiku uz laika kvantiem (tālāk nedalāmie laika intervāli). Var uzskatīt, ka starp laika kvantiem, ķermeņi nepārvietojas, bet gan teleportējas. Iespējams, šāds modelis neizskatīsies labi, jo ķermeņi varēs pārvietoties cits caur citu. Bet mikro pasaulē teleportēšanās ir pilnīgi parasta parādība (nenoteiktības princips, radioaktivitāte).

Cilvēks ir pieradis uztvert ķermeni kā diskrētu objektu, kā vienu veselu, dalāmu tikai ārkārtējās situācijās. Var iedomāties, ka absolūti cieti ķermeņi nesastāv no molekulām, bet gan attēlo nepārtrauktu vielu. Tā ir laba abstrakcija, jo daudzas reizes novienkāršo izskaitļošanu.

Lai aprakstīt cietos ķermeņus (angliski - *solid*), pastāv dažādi paņēmieni.

Particle system

Pirmais un pats reālistiskākais modelis – tā ir daļiņu sistēma. Pastāv tieša analoģija starp reālo pasauli un modeli – katra daļiņa sistēmā veido atsevišķu molekulu vai atomu. Šīs metodes trūkums ir pārāk lielas prasības pēc datora resursiem. Atmiņas nepieciešamība ir O(n), kur n – daļiņu daudzums, bet apstrādes laiks – $O(n^2)$, katra daļiņa reaģē savā starpā. Šīs metodes priekšrocības – lielā precizitāte un darbības reālisms [M1]

Constrain based

Metodes būtība slēpjas ķermeņu attēlošanā deformācijā, elastīgajā daudzstūrī. Ja noteiktā laika intervālā nav iespējams izskaitļot precīzu ķermeņa formu un atrašanos vietu, tad tas nav nekas bīstams, jo nākamajā laika intervālā sistēmas stāvoklis normalizēsies un kļūs precīzāks.

Daudzstūri uz impulsu metodes pamata

Šīs metodes pamatā ir visu ķermeņu saglabāšana ģeometrisku figūru veidā, tādi kā daudzstūris, daudzskaldnis, lode u.c. Notiek absolūti cietu ķermeņu kustība. Absolūti ciets ķermenis, kam piemīt tikai augšupejoša un rotējoša līmeņa brīvība mehānikā nozīmē, to, ka ķermenis nevar tikt deformēts, proti, ķermenim nevar piešķirt citu enerģiju, izņemot kinētisko enerģiju augšupejošā vai rotējošā kustībā. Dabā absolūti cietu ķermeņu nav, vienīgais, kad ķermeņa deformēšanas spējas ir mazas, reālais ķermenis var tikt aizvietots ar absolūti cietu ķermeni. Cieta ķermeņa kustība ir līdzvērtīga materiālā punkta kustībām šī ķermeņa masas centrā, bet trajektorijas izmaiņas notiek tikai spēka pielietošanas laikā, tādā kā gravitācijas vai divu ķermeņu sadursme. Trajektorijas izskaitļošanai pēc sadursmes (kolīzija, collision) tiek izmantoti impulsa saglabāšanas likums. Šī metode ir laba ar to, ka tā darbojas visātrāk nekā citas metodes un nav pretrunā cilvēciskai intuīcijai, kas arī ir nepieciešams. [CD1, CD2].

Manā kursa darbā tiks izskatīta tieši šī metode. Visa modelēšana tiek dalīta atsevišķos etapos:

1. Kinemātika (Kinematics) – ķermeņa trajektoriju izskaitļošana, ja tās nesadurtos;
2. Kolīzijas atklāšana (Collision Detection) – pārbaida, vai ķermeņi ir sadūrušies vai nē;
3. Kolīzijas apstrāde (Collision Handling) – ja notika kolīzija, tad nepieciešams pareizi izmantot ķermeņu trajektoriju, lai tas labi izskatītos.
4. Pēc tam var attēlot uz monitora to, kas tika izskaitļots un atkārtot ciklu.

Pielietojums

Šis darbs tiks izmantots kā teorētiskais pamatojums datorspēles fiziskajam modelim. Spēle attēlo arkādi ar skatu no sāna. Pārvietoties var tikai pa labi-pa kreisi un uz augšu-uz leju, šajā ziņā var ignorēt trešo dimensiju, kas ļoti novienkāršos realizāciju, kaut gan teorētiskais pamatojums principiāli neatšķirsies no trīsdimensiju modeļa pamatojuma.

Ķermeņi spēlē tiek attēloti plakanu daudzstūru veidā. Programmas kodola (*engine*) fiziskai daļai reālā laikā ir jāapstrādā ne mazāk kā simts ķermeņu. Galvenā prasība – reālistiskums. Ievērojot dažus fizikas likumus (konkrēti Ņūtona likumi) tiek garantēts reālistiskums. Ir vēlams panākt to, lai pie mazāka kadru skaita sekundē (frames per second, FPS) ķermeņi neizietu viens caur otru, nešķērsotu un neraustītos stāvot uz grīdas.

Otrā prasība – sistēmā ir adekvāti un laikus jāapstrādā spēlētāja reakcija. Tādēļ visi ķermeņu aprēķini ir jāveic reālā laikā.

Vēsture (Related work)

Pirmajās spēlēs tika izmantota vienkārša fizika. Objekti nevarēja iziet caur sienām un ar to pietika, jo nebija veiktspējas, lai izveidotu un apstrādātu sarežģītas ķermeņa formas un rotāciju. Tika lietota AABB (*Axis-Aligned Bounding Box*) tehnoloģija. Metode būtība slēpjas katra ķermeņa nodalīšanā ar taisnstūra paralēlskaldni vai taisnstūri. Sadursmes pārbaude nonāk pie divu taisnstūru krustojumu pārbaudes. Ķermeņa formu tuvinājumā var izmantot ne tikai taisnstūrus, bet arī sfēras, cilindrus, taisnstūru komplektus. Tipiski pārstāvji ir: *Doom, Mario*.

Pastāvēja arī cita tehnoloģija. Tiek iedarbināts divdimensiju masīvs, un katrā masīva šūnā tiek glabāts objekts, kas atrodas šajā vietā; viens objekts var aizņemt tikai vienu šūnu datu masīvā. Tā nosaukums ir divdimensiju jaukšana (hešešana, 2D-hash). Kolīzija tiek apstrādāta tikai dēļ ķermeņiem, kas atrodas blakus masīva šūnās. Šīs metodes priekšrocības ir sadursmes apstrādes ātrums, bet tas prasa lielu atmiņu. Tipisks pārstāvis: *Warcraft*.

Nākamais solis tika izdarīts fiziskā karkasa sarežģīšanai. Lai noteiktu kolīziju tiek noteikts, vai ir daudzskaldņu vai daudzstūru krustošanās. Sākotnēji tika radīts fizisks modelis bez rotācijas iespējām, piem., *Quake*. Pēc tam jau ar rotāciju, piem., *Hitman, havoc*.

Dažreiz gadās, ka reālus likumus aizvieto ar tiem līdzīgiem. Piem., likumu par kinētiskās enerģijas saglabāšanu var aizvietot ar ātruma saglabāšanas likumu. Eksistē spēles arī ar tādiem fiziskiem likumiem.

Spēles personas tiek attēlotas kā skeletveidīgi karkasi, kuriem virsū tiek likti poligoni lai attēlot tos ekrānā. Pēdējā laikā kļūst moderni datorspēlēs mironi modelēt kā lupatas lelli. Katrs ķermeņa loceklis pārvietojas tā, lai viss ķermenis būtu kā viens vesels. Dzīvo parādību skeleti (karkasi) pārvietojas pa jau iepriekš paredzētām trajektorijām. Viens no tehnoloģijas attīstības faktiem ir atteikšanās no iepriekš izskaitļotas animācijas un pārejā uz lupatu lellēm, kurām pašām jāizlemj, kuru locītavu saliekt, lai izdarītu soli uz priekšu. Bet tādas spēles vel nav.

Nākošais etaps, kuru vēl nepanāca – jebkuru objektu sasmalcināšana un iznīcināšana, kā arī sarežģītu sistēmu radīšana, līdzīgu pulksteņa mehānismam vai ceļamkrānam, kuri strādās ne pēc iepriekš noteiktiem kanoniem, bet tikai pēc universāliem spēles fizikas likumiem.

Progress nestāv uz vietas – katru gadu spēļu izstrādātāji izgudro kaut ko jaunu. Reālistiskā fizika kopā ar reālistisko grafiku, kā arī augstas veiktspējas datori, agri vai vēlu ļaus radīt reālistisku spēli, kas nepiekāpsies pat filmai. Tikai galvenais varonis nebūs fantastisks varonis, bet gan tu pats.

Datu struktūras

„Algoritmi+datu struktūra=programma”
/N. Virts/

Projektēšanas sākumā, pirmkārt, ir nepieciešams noteikt datu struktūru. Zinot datu struktūru var sākt izstrādāt algoritmu.

Fizika modelējamajā pasaulē ir divdimensionāla. Tas nozīmē, ka visi objekti, kuri būs spēlē, var būt tikai viendimensionāli un divdimensionāli. Jebkurš ķermenis reālajā pasaulē sastāv no molekulām. Jebkuru ķermeni var sašķelt uz molekulām un saglabāt šo molekulu stāvokli. Mūsu gadījumā šī metode neder, jo pieprasa lielu atmiņu molekulu saglabāšanai.

Alternatīvais variants – saglabāt visu ķermeņu kontūras. Zinot ķermeņa kontūras, var izskaitļot, kur vajadzētu atrasties molekulām, zinot, ka attālums starp molekulām ir patstāvīgs un ķermenis ir vienmērīgi aizpildīts ar molekulām un ķermeņa kontūra ir noslēgta līkne. Šo līniju var pietuvināt līdz slēgtai daudzstūra līnijai. Jo vairāk būs malu dotajā daudzstūrī, jo precīzāks sanāks tuvinājums.

Ņemot vērā to, ka daudzi algoritmi algoritmiskajā ģeometrijā darbojas tikai uz izliektiem daudzstūriem, kā arī apstrādāt izliektos daudzstūrus ir vieglāk; uzskatīsim, ka ķermeņi modelējamajā pasaulē tiks saglabāti kā izliektie daudzstūri (*polygon*). Neizliektu ķermeņu modelēšana netiks izskatīta.

Ņemot vērā specialus gadījumus varam ieviest papildiespējas ķermeņu saglabāšanā – tā būs riņķa līnija. Šāda daudzveidība pieprasa papildkoda ieviešanu, sadursmes pārbaudei starp divām riņķa līnijām, riņķa līniju un poligonu. Bez tam izvest uz ekrānu līnijas un poligonus iznāk ātrāk, nekā riņķa līnijas. Punktu var saglabāt kā ļoti mazu trijstūrīti.

Metode, kas paredz visu attēlot ar poligonu palīdzību ir universālāka un efektīvāka nekā katra objekta attēlošana atsevišķi. Tā nosaukums ir poligonālā modelēšana.

Glabāšana

Poligons sastāv no bezgalīgi daudziem nogriežņiem, katram no tiem ir pa vienam kopējam punktam ar diviem blakus nogriežņiem. Nogriežņu aprakstīšanas kārtība ir svarīga. Nogriežņu glabāšanas vietā var glabāt tikai daudzstūru virsotnes.

Visas poligona koordinātas tiks glabātas ņemot vērā smaguma centru (tālāk tekstā tiks minēts kā objekta centrs). Tātad smaguma centrs atradīsies punktā (0,0). Ar ko ir labs smaguma centrs, to izskatīsim nākamajā sadaļā.

Virsotnes ierakstītas kā plaknes apiešana pozitīvā virzienā (pretī pulksteņrādītāja virzienam). Minimālais virsotņu skaits – 3.

Datu struktūras piemērs, lai saglabātu plakni uz C++

```
struct Vector2 // 2-dimension vector
{
    double x,y; // koordinates
};
struct Polygon
{
    vector<Vector2> vertex;
    real square; //laukums
    ...
};
```

Turpmākajās nodaļās tiks izskatīta augšupejoša un rotējoša kustība, kā arī šo kustību saglabāšanas paņēmieni.

Lai savienotu visus lielumus ar fiziskajiem lielumiem un formulām, visi lielumi tiks glabāti starptautiskajā sistēmā SI [F4].

- Koordināta – metrs (m);
- Laiks – sekunde (s);

- Ātrums – metrs sekundē (m/s);
- Masa – kilograms (kg);
- Paātrinājums – m/c2;
- Spēks – Ņūtons (N, kg*m/s2);
- Stūris – radinās.

Visiem saturā augstāk minētajiem mainīgajiem derēs datu tips ar peldošo punktu (float, double). Lai saglabātu pāru lielumus, tādus kā stāvoklis vai ātrums, kuri sastāv no diviem komponentiem: x un y, tiek izmantots divdimensiju vektors. Izmantojot operatoru pārslogošanu C++ (operator overloading) sistēmā ļauj saskaitīt un reizināt vektoru uz skaitli, kas ir izdevīgāks un saprotamāks paņēmiens. Piemēram:

```
Vector2 position, velocity;
double time;

position += velocity*time
```

Masa

Mums ir nepieciešams zināt visu ķermeņu masu, lai pareizi apstrādātu kolīziju. Uzskatīsim, ka mūsu modelī visi ķermeņi ir viendabīgi. Pretējā gadījumā masu, masas centru un ķermeņa inerces momentu var piešķirt arī ar rokām.

Jebkuru ķermeni divdimensiju pasaulē var attēlot cilindra veidā, kura pamatā būs divdimensiju objekts. Poligona A masa divdimensiju pasaulē tiks saukta par cilindra masu, kura pamatā ir poligons A ar augstumu 1 m. Trīsdimensiju pasaulē, palielinot lodes rādiusu 2 reizes, masa palielināsies 8 reizes. Divdimensiju pasaulē palielinot apļa rādiusu, ķermeņa masa palielināsies 4 reizes. Ķermeņa masa (m) pielīdzinās tilpumam (V), kas tiek reizināts uz blīvumu (ρ).

$$m = V\rho$$

Masa divdimensiju pasaulē proporcionāls laukumam (V~S).

Kā atrast poligona laukumu? Sadalīsim poligonu uz trīsstūriem un saskatīsim visus trīsstūru laukumus. Trīsstūra laukums – tā ir puse no pseidoskalārā reizinājuma, kas virza vektorus no divām blakus malām. Principā uz punktu, no kā notiks triangulācija, netiek likts ierobežojums (šim punktam jābūt visos trīsstūros, pārējie divi punkti ir daudzstūra blakusvirsotnes). Ja izvēlētais punkts atrodas ārpus daudzstūra, tad vajag ņemt vērā to faktu, ka trīsstūra laukums var būt negatīvs, par cik mēs uzskatām laukumu par vektoru pseidoskalāro reizinājumu. Izveidojušos laukumus nepieciešams salikt algebriski, ņemot vērā zīmes.

Masas centrs

Trīsstūra masas centrs – tā mediānu krustpunkts. Praktiskā darbībā vieglāk atrast trīsstūra virsotņu koordinātu vidējo aritmētisko, kas arī ir masas centrs.

Tāpat kā līdzšinējā gadījumā, sadalīsim poligonu trīsstūros un saskaitīsim vidējo svērtu no trijstūru masas centriem.

Inerces moments

Lai pareizi apstrādātu kolīziju, ņemot vērā ķermeņa rotāciju, nepieciešams risināt momentu, kurā notiek inerce. Inerces moments ir masas analogs pie augšupejošas kustības. Inerces moments raksturo ķermeņa pretestību rotācijai, tāpat kā masa – pretestība augšupejošai kustībai.

Materiālās punkta inerces moments attiecīgi rotācijas asij tiek saukts materiālā punkta masas reizinājums ar attālumam līdz tā asij kvadrātā. Ķermeņa inerces laiks ir visu materiālo ķermeņa punktu inerces momentu summa [F2]. Piemēram, sportiste rotācijā ienāk samērā lēni, plaši izplešot kājas un rokas, pēc tam viņa savāc savu ķermeņa masu tuvāk rotācijas asij,

piespiežot rokas un kājas arvien tuvāk ķermenim, rotācijas ātrums daudzkārt pieaug, samazinot inerces momentu pie rotācijas momenta saglabāšanas.

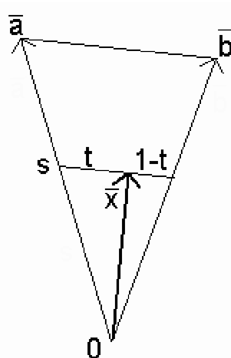
Ķermeņa inerces moments tiek aprēķināts pēc formulas [F1]

$$J = \sum_k m_k r_k^2 \qquad (1)$$

un, ja ir zināms ķermeņa masu izkārtojums atbilstoši asij, tas var tikt atrasts ar tiešu izskaitļošanu. Bet šis uzdevums, īpaši neviendabīgajiem ķermeņiem var izrādīties par sarežģītu. Tā, visdrīzāk, noved pie integrēšanas. Protams, ar datora palīdzību integrāli var izskaitļot, bet analītiski inerces momentus parasti izskaitļo tikai viendabīgām ķermenim.

Skatoties uz formulu (1) var secināt, ka mēs varam sadalīt mūsu ķermeni daļās un saskaitīt inerces laiku, katrai daļai atsevišķi. Ķermeņa inerces laiks būs tā atsevišķo daļu inerces momentu summa.

Lai atrastu daudzstūra inerces momentu, sadalīsim poligonu trīsstūros, kuros viena virsotne atradīsies poligona masas centrā. Uzdevums ir reducēts uz trīsstūra, kur viena virsotne ir rotācijas ass, inerces momenta atrašanai.



Zīm. 3: trīsstūris, kuram ir jāatrod inerces momenta apkārt zemākajam punktam. Vektori a un b – tas ir trijstūra malas. Parametri s un t mainās no 0 līdz 1.

Sadalīsim trijstūru uz nogriežņiem, kuri ir paralēli tālākai trijstūra malai. Trīsstūra inerces moments J ir vienāda summai no nogriežņa inerces momentiem. Tā ka masa ir sadalīta vienmērīgi, tad blīvumu var iznest ārpus formulas (1).

$J = \int_0^1 J(s)ds$, kur s ir attālums no nogriežņa līdz masas centra procentos (0-1). J(s) – tas ir nogriežņa $(s\vec{a}; s\vec{b})$ inerces moments.

Lai izrēķināt nogriežņa inerces momentu, sadalīsim to uz punktiem

$J(s) = \int_0^1 m\,\vec{x}(s,t)^2\,dt$, kur x – punkta rādiuss-vektors.

Punkta rādiuss-vektoru var izrēķināt caur vektoriem a un b: $\vec{x}(s,t) = s(\vec{a}t + \vec{b}(1-t))$

Tātad, trijstūra inerces moments $J = m\int_0^1\int_0^1 \left(s\left(\vec{a}t + \vec{b}(1-t)\right)\right)^2 dtds$. Vektora kvadrāts – tas ir vektora skalārais reizinājums uz to pašu vektoru.

$$\vec{v}^2 = v_y^2 + v_y^2$$

$$
\begin{aligned}
J &= m\int_0^1\int_0^1 \left(\left(s\left(a_x t + b_x(1-t)\right)\right)^2 + \left(s\left(a_y t + b_y(1-t)\right)\right)^2\right) dt\,ds \\
&= m\int_0^1 s^2 \int_0^1 \left(a_x^2 t^2 + a_x b_x (1-t)t + b_x^2 (1-t)^2 + a_y^2 t^2 + a_y b_y (1-t)t + b_y^2 (1-t)^2\right) dt\,ds \\
&= m\int_0^1 s^2 \left(\left(a_x^2 + a_y^2\right)\int_0^1 t^2 dt + \left(a_x b_x + a_y b_y\right)\int_0^1 (1-t)t\,dt + \left(b_x^2 + b_y^2\right)\int_0^1 (1-t)^2\,dt\right) ds \\
&= m\int_0^1 s^2 \left(\frac{1}{3}\left(a_x^2 + a_y^2\right) + \frac{1}{6}\left(a_x b_x + a_y b_y\right) + \frac{1}{3}\left(b_x^2 + b_y^2\right)\right) ds \\
&= \frac{1}{9} m\left(\left(a_x^2 + a_y^2\right) + \frac{1}{2}\left(a_x b_x + a_y b_y\right) + \left(b_x^2 + b_y^2\right)\right)
\end{aligned}
$$

Lai atrast daudzstūra inerces momentu varam sasummēt visas trīsstūru inerces momentus.

Kinemātika

Viena ķermeņa stāvokļa maiņa relatīvi otram ķermenim sauc par *mehānisko kustību*.
Fizikas sadaļa, kas pēta mehāniskās kustības, tiek saukta par mehāniku.
Mehānikas sadaļa, kas pēta kustības un sakarības starp lielumiem, kuri raksturo šīs kustības, sauc par *kinemātiku* [F1]

Par absolūti cietu ķermeni saucas ķermenis, kura deformācijas pēc uzdevuma noteikumiem var tikt ignorētas. Absolūti cietam ķermenim attālums starp jebkuriem tā punktiem laika gaitā nemainās. Absolūti cieta ķermeņa stāvoklis divdimensiju pasaulē tiek attēlot ar trīs koordinātēm. Absolūti cieta ķermeņa stāvoklis pilnīgi fiksējas uz diviem punktiem, kas stingri saistīti ar ķermeni. Divu punktu atrašanās vieta tiek apzīmēta ar četršām koordinātēm, bet attālums starp punktiem ir nemainīgs, tad četras koordinātes tiks saistītas viena vienādojumā. Tātad, neatkarīgas koordinātes, kas nosaka cietā ķermeņa stāvokli telpā būs trīs. Neatkarīgo koordinātu skaitam atbilst neatkarīgo kustību veidu daudzums, uz kura varētu būt izkārtota patvaļīga ķermeņa kustība. Absolūti cietam ķermenim tādas kustības ir trīs. Saka, ka absolūti cietam ķermenim piemīt trīspakāpju brīvība. Neatkarīgos ķermeņu pārvietošanās veidus var izvēlēties dažādi. Piemēram, rīkosimies šādi. Sasiesim cieto ķermeni „stingri" ar vienu punktu un sekosim tā kustībām, kā arī ķermeņa kustībām apkārt punktam. Viena punkta kustība tiek aprakstīta ar divām koordinātēm, t.i. ietver sevī divas brīvības pakāpes. Tās sauc par augšupejošām brīvības pakāpēm. Atlikusī brīvības pakāpe tiek attiecināta uz rotējošā ķermeņa kustību apkārt izvēlētajam punktam. Šo brīvības pakāpi sauc par rotējošo.

Tādā veidā, patvaļīgas cietā ķermeņa kustības var tikt sadalītas uz augšupejošām un rotējošām kustībām apkārt nekustīgam punktam.

Augšupejoša kustība

Par augšupejošu kustību sauc tādu kustību, pie kuras jebkura taisne, stingri saistīta ar ķermeni, pārvietojas paralēli pati sev. Pie augšupejošas kustības visi ķermeņa punkti kustas absolūti vienādi: tiem ir vienādas, bet nobidītas trajektorijas attiecīgi vienam pret otru, vienādas jebkurā laika sprīža ātrumā un vienādi paātrinājumi. Absolūti cieta ķermeņa augšupejoša kustība ir ekvivalenta viena punkta kustībai un augšupejošas kustības kinemātika noved pie punkta kinemātikas. Lai modelētu augšupejošu kustību, var izvēlēties jebkuru ķermeņa punktu. Izvēlēsimies daudzstūra centru (masas centru).

Ja uz ķermeni neiedarbojas spēks, tad ķermenis kustas bez paātrinājuma. Šajā gadījumā ķermeņa kustību var aprakstīt ar formulu

$$P = P_0 + Vt \qquad (1)$$

kur P_0 – ķermeņa stāvoklis nulles laika momenta (laika atskaites sākumā)
V – ātrums,
t – laiks, kas ir pagājis no kustības sākuma.

Eilera metode

Mūsu modelī uz ķermeni iedarbojas spēki (gravitācijas, impulsu apmaiņa pie trieciena), tas nozīmē, ka kustība būs nevienmērīga, ar paātrinājumu. Lielam ķermeņu daudzumam izskaitļot tā koordinātes jebkurā laika sprīdī ir ļoti sarežģīti, ņemot vērā nevienmērīgu paātrinājumu pie sadursmes. Tādēļ varam izmantot Eilera metodi, lai atrisinot diferenciālus vienādojumus, atrastu ķermeņa trajektoriju.

Eilera metode paredz laika sadalīšanu kvantos, tālāk nedalāmiem laika sprīžiem. Paātrinājums ietekmē kustību tikai kvanta sākumā (vai beigas), bet kvanta iekšienē kustība ir vienmērīga. Laika kvantēšana ir saistīta arī ar informācijas attēlošanu uz ekrāna. Mūs interesē ķermeņa stāvoklis katrreiz, kad mēs parādam informāciju uz ekrāna. Izskaitļot ķermeņa stāvokli retāk, nekā izslēgt – bezjēdzīgi, tā kā jaunu informāciju caur ekrānu mēs nesaņemsim. Lai radītu

kustības ilūziju nepieciešams kā minimums 24 reizes sekundē izvadīt informāciju uz ekrāna. Viens attēlojums uz ekrāna ir kā viens *kadrs*. Jo vairāk kadru sekundē (*frames per second, FPS*), jo reālistiskāka ir kustība. Tāpēc vajag tiekties pēc maksimāla FPS. No citas puses, izvadīt informāciju uz monitora biežāk, nekā stars attēlo ekrānu uz CRT monitora, arī ir bezjēdzīgi. Mēs neredzēsim to, kam vajadzētu būt attēlotam uz ekrāna.

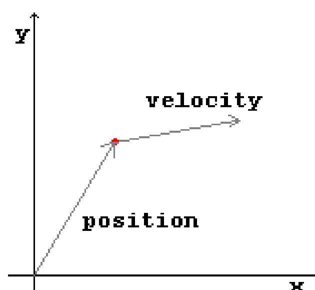
Izvadīšana uz ekrāna nav obligāta. Ja mēs gribam izrēķināt ķermeņu kustību skaitliski vai pārbaudīt konstrukcijas noturīgumu, pietiek ar to, ka eksperimenta beigās mēs vienreiz apskatīsim rezultātu.

Var veikt ķermeņu stāvokļa pārrēķināšanu biežāk, nekā informācija tiks izvadīta uz monitora. Jo vairāk būs kvantu (kvantu būs mazāk), jo kustības modelēšana būs precīzāka.

Principā, laika kvanti var atšķirties viens no otra pēc izmēra. Ja mēs gribam redzēt ķermeņa stāvokli reālā laikā un gribam panākt maksimālu kadru skaitu, tad laika kvanti būs dažāda izmēra, tādēļ ka dažādos laika sprīžos operētājsistēma var tikt pārslogota ar citu uzdevumu risināšanu, iespējama ķermeņu skaita maiņa un kolīzija starp tiem no kvanta kvantā. Viss tas var novest pie tā , ka kvanta laika ilgums nesakritīs ar reālo kvanta apstrādes laiku. Tādēļ kustības var likties saraustītas un nevienmērīgas. Priekšlaicīgi nevar paredzēt, cik laika būs nepieciešams uz šī laika kvanta apstrādi. Optimālais risinājums – tas ir skatīties uz iepriekšējā kadra apstrādes laiku, cerot, ka apstrādes laiks strauji nemainīsies, bet, ja izmainās – tad tas ir jākompensē nākamajā kvantā. Par sākotnējo laika momentu uzskatīsim ($t = 0$) tekošā laika kvanta sākumu un tad formulā (1) t apzīmēs tekošā kvanta laiku (kas līdzinās reālam laikam, kad tika apstrādāts iepriekšējais kvantu laiks).

Ņemot vērā, ka augšupejošai kustībai ir divas brīvības pakāpes, apzīmēsim šīs brīvības pakāpes ar koordinātēm x un y. Lai saglabātu ķermeņa vietu (*position*) varam izmantot *Vector2* tipu, kas tiek aprakstīts iepriekšējā nodaļā. Bez tam mums ir nepieciešams saglabāt koordinātu izmaiņas. Par ātrumu (*velocity*) tiek nosaukts koordinātes pirmais atvasinājums pēc laika. Ātrumu ir izdevīgi glabāt vektora formā. Tādā veidā radīsies jauna struktūra, lai saglabātu cietus ķermeņus.

```
struct solid
{
      Polygon perimeter;
      Vector2 position, velocity;
};
```



Ķermeņa rotējošā kustība apkārt nekustīgai asij

Absolūti cieta ķermeņa stāvoklis šajā gadījumā tiek raksturots ar vienu vienīgu koordinātu: ķermeņa pagrieziena leņķi apkārt asij. Leņķis tiek atskaitīts no ķermeņa stāvokļa noteiktā pusē, rezultātā pagrieziena leņķim tiek piešķirta zīme. Algoritmiska ģeometrijā kustību pozitīvā virzienā (pretēji pulksteņrādītāja virzienam) pieņemts apzīmēt ar pozitīvu skaitli. Brīvā stāvoklī ķermenis rotēs ap masas centru. Rotācijas ass ir perpendikulāra pasaules plaknei un iet cauri masas centram.

Galvenais ķermeņa kustības raksturojums šajā gadījumā ir *leņķiskais ātrums*. Par leņķisko ķermeņa ātrumu tiek saukts atvasinājums no leņķa pagrieziena pēc laika:

$$\omega = \frac{d\varphi}{dt}$$

Analoģiski augšupejošajai kustībai, rotācijas formula ir:

$$\Phi = \Phi_0 + \omega t$$

Zinot pagrieziena leņķi un masas centra stāvokli katrā laika sprīdī var izskaitļot jebkura ķermeņa punkta stāvokli. Mūsu gadījumā pietiekoši izskaitļot daudzstūra virsotņu koordinātes.

$$X_i = x_0 + x_i \cos\Phi - y_i \sin\Phi$$

$$Y_i = y_0 + y_i \cos\Phi - x_i \sin\Phi$$

Koordinātu atrašanas formula daudzstūra virsotnēm. x_0 *un* y_0 *– centra koordinātes,* Φ *– ķermeņa pagrieziena leņķis atbilstoši sākotnējam stāvoklim;* x_i *un* y_i *– daudzstūra virsotņu koordinātas atbilstoši rotācijas asij (masas centram) sākotnējā stāvoklī.* X_i *un* Y_i *– jaunās absolūtās daudzstūra virsotņu koordinātes.*

Ņemot vērā augstākminēto, *solid* struktūra tiks papildināta ar skalāriem lielumiem *w* (leņķa ātrums) un *angle* (ķermeņa pagrieziena leņķis).

Kustība

Ķermeņu stāvokļa pārskaitīšanas pseidokods izskatās šādi:

```
position += time*velocity;
angle    += time*w;
```

Ir ķermeņi, kuriem nevajadzētu kustēties, neskatoties uz to kādi spēki uz to iedarbojas. Par tādu ķermeņu piemēru var kalpot sienas un grīda, kas norobežo mūsu pasauli. Grīdai nekad nedrīkst kustēties, uz viņu nevajadzētu iedarboties zemes pievilkšanās spēkam, savādāk pievilkšanās nebūs jūtama vispār, tāpēc kā visi ķermeņi līdz ar grīdu kritīs lejā.

Kolīzijas noteikšana

Kolīzija (no latīņu *collisio* – sadursme, atduršanās), pretējo spēku sadursme, uzskatu, interešu, tiekšanos. Mūsu gadījumā daudzstūru sadursme.

Otra fizikas modelēšanas problēma – pārbaudīt vai ir sadursme. Varam uzskatīt, ka kolīzijas notiek tikai starp diviem objektiem. Nevar notikt sadursme uzreiz starp trim vai vairāk ķermeņiem, kāda no sadursmēm notiks agrāk. Reālajā pasaulē kolīzija rodas, kad divi ķermeņi atrodas nelielā attālumā viens no otra. Jo attālums ir mazāks, jo atgrūšanās spēks ir lielāks un lielāk deformējas ķermenis. Pēc laika daļa no kinētiskas enerģijas pariet uz iekšējo enerģiju. Pēc tam daļā no enerģijas pariet atpakaļ uz kinētisko. Sadursmē piedalās visas abu ķermeņu molekulas, nododot viens otram kustības impulsu. Bet mūsu modeļi ķermeni nevar deformēties.

Par divu cietu ķermeņu kolīziju sauksim to stāvokli, kad divi ķermeņi saskaras viens ar otru vai pat krustojas, jeb tiem ir kopējs punkts vienā un tai pašā laika sprīdī. Pats galvenais – neizlaist nevienu kolīziju, savādāk ķermeņi atradīsies viens otrā, un vēlāk būs ļoti sarežģīti dabiskā veidā tos atdalīt.

Vēlams noteikt precīzu kolīzijas rašanās laiku. Tas būs nepieciešams kolīzijas apstrādes procesā, lai pareizā secībā apstrādātu kolīziju.

Triviālā metode

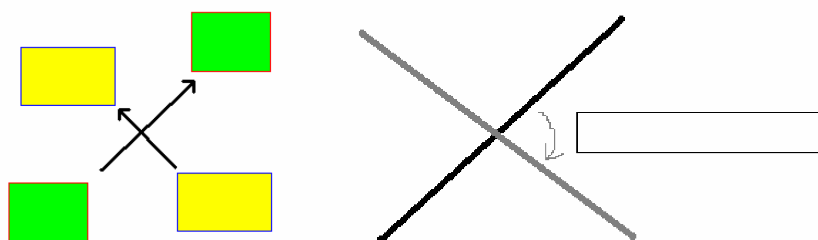
Ar triviālo metodi domāta kustību neesamība un ķermeņu sadursme laika kvantā. Ķermeņi vienkārši teleportējas un momentāni griežas.

Triviālā metode tuvīnajās pie visu ķermeņu koordinātes pārskaitām un ķermeņu pārbaudi pa pāriem, vai tie krustojas vai nē. Ja krustojas, tad ir kolīzija.

Lai pārbaudītu divu ķermeņu krustošanos var izmantot divu daudzstūru krustošanas algoritmu. Triviāls algoritms darbības laiks ir O(nm), kur n un m ir malas skaits divas daudzstūros. Eksistē cits algoritms - Izliektu daudzstūru šķēluma atrašana ar vēziena metodi [AG1], kura ātrdarbība ir O(n+m), bet šis algoritms strādā tikai ar izliektiem daudzstūriem.

Veiktspējas palielināšanas metodes tiks apskatītas nākošajās nodaļās.

Šādas metodes trūkums ir tajā, ka tā ne noteica visas kolīzijas. Ķermeņi var sadurties vienā kvanta laika iekšienē. Piemēram, ķermenis var iziet caur sienu, ja tas veic attālumu kvantu laikā lielāku par saviem izmēriem.



Zīm. 1: pa kreisi nepareizas algoritma darbības piemēri – ķermeņi iziet viens otram cauri, pa labi – pie lielas rotācijas ātrumiem (dotajā gadījumā 80 grādi vienā kvantā) ķermenis iet cauri citam ķermenim.

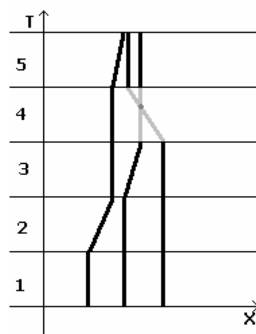
Šīs metodes priekšrocība ir realizācijas vienkāršībā. Nepieciešams tikai daudzstūru krustošanās algoritms.

Pakāpeniskais algoritms

Reālajā pasaulē visi objekti kustas vienlaicīgi (paralēli laikā). Bet uz viena procesora datora ir grūti izkārtot aprēķinus. Apskatīsim metodi, kura palīdz noteikt lielāku kolīziju daudzumu, nekā pirmā metode.

Visi ķermeņi kustas pakāpeniski, viens pēc otra. Tātad, koordinātu pārrēķins notiek pakāpeniski, sākumā pirmajam ķermenim, pēc tam otrajam, un tā visiem pārējiem. Sanāk, ka

mēs pārbaudām kolīzijas starp kustīgiem un nekustīgiem ķermeņiem. Vienā laika kvantā kustas tikai viens ķermenis.



Zīm. 2: secīgas pārrēķināšanas piemērs viendimensijas telpā. Ar pelēkam līnijām apzīmēta kolīzija. T – laika kvanta numurs.

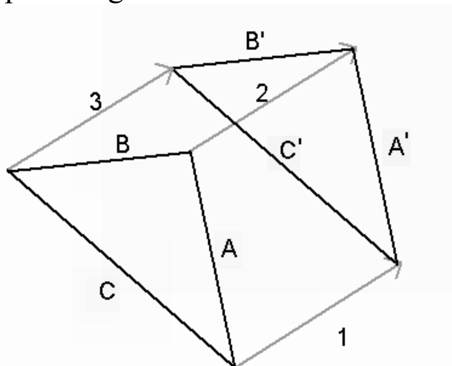
Lai vienkāršot aprēķinus var uzskatīt, ka ķermenis sākumā kustas uz priekšu, pēc tam rotē (vai arī otrādi). Ķermeņus, kuri kustas ļoti lēni un nesaduras var apstrādāt retāk, nekā citus. Ja ķermeņi stāv viens uz otra (abi ķermeni ir nekustīgie), visi spēki ir līdzsvaroti, tad ķermenis arī turpinās stāvēt uz vietas, nemainot savas koordinātes. Tādu ķermeni var vispār neapstrādāt līdz tiek izjaukts līdzsvars. Analogu ķermeņu sasaldēšanas metodi izmanto daudzās spēlēs, lai palielinātu veiktspēju. Ķermeni, kurš izjūt daudz kolīziju, var apstrādāt biežāk, nekā citus ķermeņus. Šiem mērķiem var izmantot virkni ar prioritātēm (piemēram, kaudze, heap). Visi objekti kustas uz priekšu laikā. Vajag bīdīt un apstrādāt tā objekta kolīzijas, kurš visvairāk atpaliek no pārējiem laikā.

Iespējams, ka tas izskatīsies vienpusīgi, par cik vienam ķermenim ir prioritātes citu vidū – viņa kustība tiek apstrādāta agrāk.

Ja objekti kustēsies nevienmērīgi (ar dažādu laiku intervālu), tad iespējamas tādas pašas kļūdas kā triviālā metodē (ķermeņi izies viens otram cauri). Lai novērstu šādas kļūdas, nosakot kolīziju, var pievērst uzmanību kopējām ķermeņa trajektorijām laikā.

Kā ar doto metodi pārbaudīt vai ir notikusi kolīzija? Varam iedomāties, ka vienā laika kvantā ķermenis kustoties atstāj trasi plaknē. Ja šī trase krusto citus ķermeņus – notiek kolīzija.

Apskatīsim tikai augšupejošu kustību. Trases krustošanās pārbaudei ar citu ķermeni varam sadalīt mūsu daudzstūrus uz pusēm un pārbaudīt kolīziju ar katru no daudzstūra pusēm. Trase, kuru atstāj viena puse – tas ir paralelograms.



Zīm. 3:tristuris ABC, kas veido trīs paralelogramus: A1A’2, B2B’3 un C1C’3. Vajag pārbaudīt krustojumus tikai ar A1A’2 un B2B’3, jo nogrieznis C kustējās ar kreiso malu priekšā.

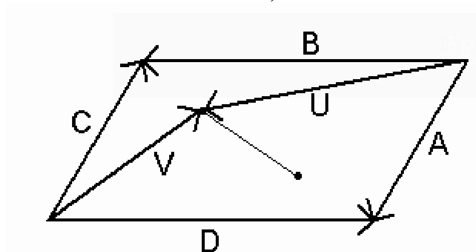
Otrs daudzstūris nekustas, tātad, tās puses paliek kā nogriezni.

Lai noteiktu kolīziju starp kustīgo pusi un nekustīgo vajag atrast paralelograma un nogriežņa krustošanos.

Nogriežņa un daudzstūra krustošanās noteikšana

1. Nogrieznis krusto četrstūra malu. Atrodam nogriežņa krustošanās punktus ar visām četrstūra malām.

2. Nogrieznis atrodas iekšpusē un nekrustojas. Pārbaudām vai viens punkts atrodas starp vektoriem A, B un C, D (vektoru sākums atrodas daudzstūra virsotnēs ar nepāra numuriem, tātad ņemam katru otro virsotni).



3. Ja iepriekšējie punkti nav piepildījušies, tad krustošanās nav.

Dotā metode derēs jebkuram izliektam daudzstūrim. Ņemot vērā faktu, ka daudzstūris ir slēgts ķermenis varam izlaist otro punktu pārbaudē, par cik, ja viens nogrieznis atradās pilnīgi paralelograma iekšpusē, tad tas nozīmē, ka kāds no nogriežņiem iziet ārpus paralelograma robežām. Krustošanās nebūs divos gadījumos: vai nu daudzstūris atrodas pilnīgi paralelogramā, vai arī ārpus tā. Priekš tā pilnīgi pietiek pārbaudīt, vai atrodas kaut viens daudzstūra punkts paralelogramā vai nē. Šeit var izmantot daudzstūra otro punktu un nogriežņa krustošanās atrašanai algoritmā.

Šī metode derēs arī diviem un priekšu ejošo ķermeņiem kolīzijas atrašanasm, jo divu augšupejošu kustību kombinācija ir arī augšupejoša kustība. Citiem vārdiem sakot, mēs varam radīt tādu inerciālu atskaites sistēmu, kurā viens ķermenis ir miera stāvoklī, bet cits kustas. Uzdevums atkal noved pie kustošam un nekustošam ķermeņa kolīzijas atrašanas.

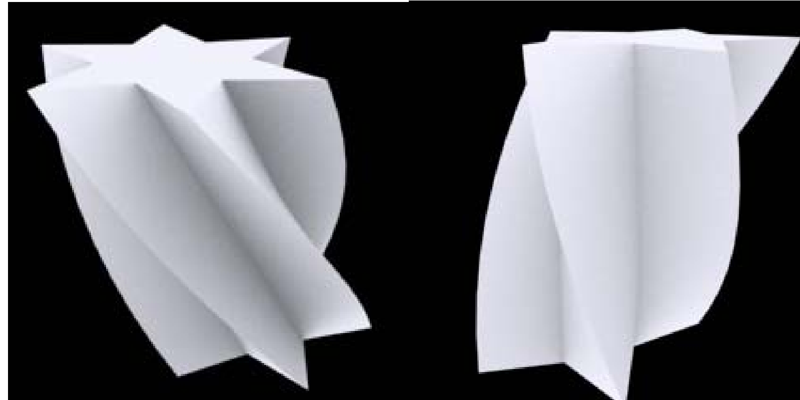
Ko darīt ar rotēšanu? Trase, kuru atstāj daudzstūra viena puse rotācijas rezultātā kļūst par līklīnijas figūru. Bez tam, divu rotējošu kustību kombinācija, nav rotējoša kustība. Dalot laiku kvantos, mēs paredzējām, ka kustība laika kvantā būs vienmērīga. Taču rotējoša kustība izrādās nav vienmērīga kustība. Rotējoša kustība nav iespējama bez centrtieces spēka darbības. Ir spēks – ir paātrinājums. Tātad izmantojot ideju par laika kvantēšanu nevar paļauties uz rotāciju laika kvanta iekšienē, tas nozīmē, situācija 1. zīm. nevar tikt atrisināta precīzi. Šī metode neder rotējošai kustībai.

Cilindru metode – cīņa ar rotāciju

Modelējamā pasaule – divdimensiju. Ņemot vērā laiku, sanāk trīsdimensiju telpa, kuru veido divas telpiskas dimensijas un viena laika.

Iedomāsimies kādu figūru veido ķermenis šādā telpā. Ja izpaliek kustība, tad izveidojas cilindrs, kura pamatā ir mūsu ķermenis. Vertikālā ass (T) – tas ir laiks. Figūra izkārtojas starp pagājušo kadru un tekošo.

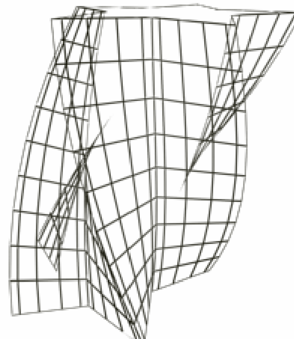
Ja ķermenis kustas, tad cilindrs sanāk slīps. Ja ķermenis rotē, tad cilindrs sanāk vēl arī savīts.



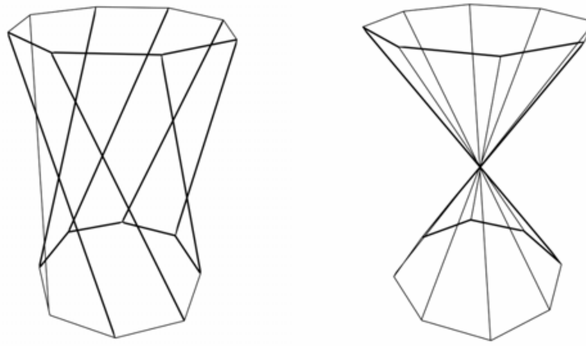
Zīm. 3: tādu cilindru piemērs, kuri izveidojušies no zvaigznes, kura kustas un rotē (modelējamā pasaulē pastāv tikai izliekti daudzstūri, bet labākai uzskatei tiek piedāvāts piemērs ar zvaigzni)

Kolīzijas noteikšana noved pie tādu savītu cilindru krustošanās noteikšanas. Tas nav triviāls uzdevums, bet dod tiešāku rezultātu salīdzinājumā ar triviālo algoritmu. Šī metode palīdz pilnīgi precīzi noteikt bija kolīzija vai nē.

Kā meklēt šādu figūru krustošanos? Tas ir samērā sarežģīts uzdevums. Šo figūru var pietuvināt daudzskaldnim un tad meklēt divu daudzskaldņu krustošanos. Šīs metodes trūkums ir tas, ka lai iegūtu lielu precizitāti, nepieciešams liels daudzums horizontālu šķēlumu. Tas nozīmē, ka algoritma darbības laiks būs atkarīgs no cilindra savīšanas pakāpes. Tā vietā var samazināt kvantu laika izmērus un atbrīvoties no horizontālām šķēlumam. Tomēr, kā parādīts 5. zīm., tas ne vienmēr palīdz. Ja ķermenis veic daudzus apgriezienus ap asi laika kvantā, tad laika kvantu jāņem ļoti mazu, no kā izriet liels kvantu daudzums. Programma nespēs konstatēt kolīzijas reālajā laikā. Bez tam, aproksimācija uz daudzskaldni dod nevis precīzu, bet aptuvenu rezultātu.



Zīm. 4: piemērs par cilindra pietuvināšanu daudzskaldnim



Zīm. 5: piemērs par pietuvināšanu bez horizontālām līnijām uz mazā leņķa pagrieziena un uz 180 grādu pagrieziena.

Cita ideja – mainīt koordinātu sistēmu, tā lai viens no cilindriem būtu taisns. Tad otrs cilindrs būs savīts apkārt pirmajam. Lai sameklētu krustpunktus, var projicēt divas dotās figūras plaknē (atbrīvoties no laika) un meklēt krustpunktus jau plaknē.

Zīm. 6: cilindri sarežģītā koordinātu sistēmā un šo ķermeņu projekcija plaknē.

Divu daudzstūru krustpunktu atrašana plaknē noved vai nu pie divu nogriežņu krustpunktu atrašanas, vai nu atrašanās punkta noteikšanas viena daudzstūra otrā.

Apskatīsim pirmo metodi. Vajag atrast divu nogriežņu krustošanos, kuri pārvietojas laikā. Sastādīsim parametrisko vienādojumu. Ja vienādojumam ir atrisinājums, tad nogriežņi krustojas. Kustības punkta vienādojums (nodaļa Kinemātika – Rotējošā kustība).

$$X_i = x_0 + x_i \cos\Phi - y_i \sin\Phi$$

$$Y_i = y_0 + y_i \cos\Phi - x_i \sin\Phi$$

Lai izskaitļotu punkta koordinātes jebkurā laika momentā kvanta iekšpusē, ieliksim centra un leņķa izskaitļošanas formulu.

$$X_i = x + V_x t + x_i \cos(\Phi + \omega t) - y_i \sin(\Phi + \omega t)$$

$$Y_i = y + V_y t + y_i \cos(\Phi + \omega t) - x_i \sin(\Phi + \omega t)$$

Nogriezni var attēlot kā parametrisko vienādojumu

$$x = x_0 + d_x s$$

$$y = y_0 + d_y s$$

$$0 \le s \le 1$$

Apkopojot visus divu nogriežņu vienādojumus, iegūstam sistēmu no diviem vienādojumiem ar trīs parametriem. Tāda sistēma nav atrisināma konstantā laikā, tas nozīmē, mums šī metode neder tāpēc, ka šīs sistēmas atrisināšanas vietā var vienkārši samazināt laika kvantu līdz tādam stāvoklim, kurā rotācija netiek ņemta vērā.

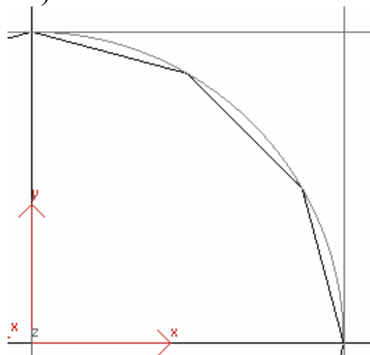
Otrai metodei nepieciešams prast atrast nogriežņa krustošanos ar līkni, kura izveidojusies divu rotējošu kombināciju rezultātā. Tā arī nav atrisināma konstantā laikā.

Pakāpeniskais algoritms +

Atcerēsimies galveno prasību – reālistiskums. Pie ātras kustības (pie kuras ķermenis salīdzinājumā ar saviem izmēriem veic attālumu kvantu laikā) un ātras rotācijas (pusapgrieziens

laika kvantā) cilvēkam būs grūtāk ievērot kolīzijas neesamību, nekā pie viena kustības tipa neesamības. Mēs varam sadalīt kustību divās daļās, un sākumā apstrādāt augšupejošu kustību un pēc tam rotējošu. Augšupejošas kustības modelēšana labi atrisinās ar pakāpenisko metodi.

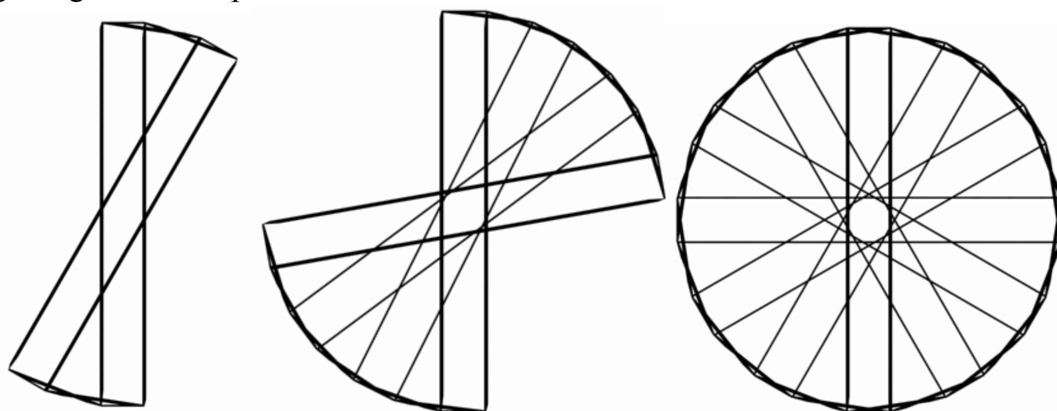
Ko darīt ar rotāciju? Apskatīsim punkta trajektoriju rotējošā kustībā. Pie leņķa pagrieziena, kurš mazāks par 30 grādiem (1/12 apļa) kustības trajektoriju pa līkni var aizstāt ar kustības trajektoriju pa nogriezni (zīm. 7).



Zīm. 7: divpadsmitstūris ne pārāk atšķiras no apļa

Punkta kustību uz leņķim, lielākam par 360 grādiem, var aizstāt ar kustību pa riņķa liniju. Ja leņķa izmēri ir robežās no 30 grādiem – 360 grādiem, tad laika kvantu var samazināt tā, lai leņķa pagrieziens laika kvantā kļūtu mazāks par 30. Kopsummā laika kvantu daudzumam nevajadzētu palielināties vairāk kā 12 reizes, par cik 30 gradi sastāda 1/12 daļu pilna apgrieziena, bet lielā leņķa pilno apgriezienu mēs nesadalām, bet gan salīmējam punkta trajektoriju riņķa līnijā.

Tagad atgriezīsimies pie daudzstūra.

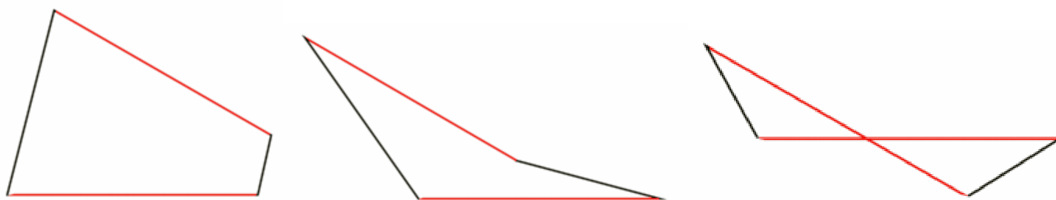


Zīm. 8: daudzstūra pagrieziens par 30,80 un 360 grādiem.

Ideja ir analoga cilindru metodei (zīm.5), tikai lielos pagriezienus mēs nomainām pret dažiem mazākiem, un strādājam 2D, bet nevis 3D. Varam 5. zīm. figūru projicēt plaknē (atbrīvoties no laika). Iegūsim daudzstūri (zīm.8).

Sadursmes noteikšanai varam meklēt tādu neizliektu daudzstūru krustošanos. Ja abi ķermeņi rotē ļoti ātri, tad meklējam divu riņķa līniju krustošanos. Ja viens ķermenis rotē ātri, bet cits lēni, tad meklējam riņķa līnijas krustošanos ar daudzstūri.

Atrast divu neizliektu daudzstūru krustošanos var ar O(nm), bet mums vēl vajag noteikt tiešu kolīzijas notikšanas laiku, šim mērķim sadalīsim doto figūru divos nogriežņos un izsekosim trasei, kuru atstāj nogrieznis kustoties plaknē. Iegūsim četrstūri.

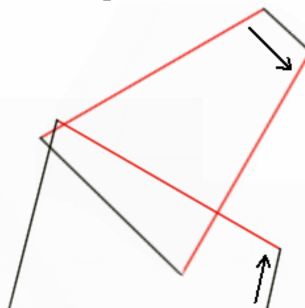


Zīm. 9: četrstūri izveidoti no rotējoša nogriežņa. Ar sarkano norādītas daudzstūri veidojošās malas.

Izmantojot kustības invariantu (ķermeņi nekad nekrustojas), varam apgalvot, ka sadursme var notikt tikai tad, kad krustojas abu ķermeņu kontūras. Lai noteiktu divu projekciju krustošanos pietiek pārbaudīt pa pāriem, vai četrstūri, kuri izveidoti no daudzstūru malām rotācijas rezultātā, nekrustojas.

Kā atrast divu daudzstūru krustošanos? Var pa pāriem pārbaudīt malu krustošanos. Ja krustošanās nepastāv, tad jāpārbauda, vai kāds no pirmā daudzstūra punktiem neatrodas otra iekšpusē, un analoģiski katra otrā daudzstūra punktam.

Kā noteikt sadursmes notikšanas laiku, ja notiek krustošanās? Šo problēmu tik vienkārši atrisināt nesanāks, kā arī, ja ir augšupejoša kustība, par cik divu rotēšanu kombinācija apkārt dažādiem punktiem nav rotēšana apkārt vienam punktam.



Zīm. 10: piemērs, kurā neiespējams noteikt tiešu sadursmes laiku.

Šo problēmu var atrisināt ar binaro meklešanu (scaling) [P1]. Ja krustošanās notika, tad laiku dalām divās daļās un pārbaudām vai bija krustošanās šajos laika intervālos. Sanāk trīs varianti.

1. pirmajā pusē notika krustošanās: atkārtojam algoritmu pirmai pusei, otra laika puse mūs neinteresē, par cik jāatrod pats agrākais laiks.
2. pirmajā pusē nebija krustošanās, bet bija otrajā. Atkārtojam algoritmu otrai pusei.
3. nav krustojuma. Kaut kas tāds ir iespējams riņķa līniju pietuvinot daudzstūrim (zīm.7). tas nozīmē, ka dotajā laika momentā nav sadursmes, bet iespējams tā notiks vēlāk, tajā laika momentā, kuru mēs palaidām garām pirmajā punktā.

Rekursīva algoritma piemērs:

```
func f(start, finish)
{
if (finish - start < Epsilon) return find;
middle = (start + finish)/2;
if (intersect(start, middle))
      if (f(start, middle) == find) return find;
if (intersect(middle, finish))
      if (f(middle, finish) == find) return find;
return error404;
}
```

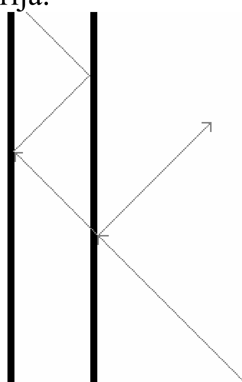
Šī funkcija izpildās, ja ir zināms, ka ir krustošanās un vajag precīzāk aprēķināt sadursmes laiku. Vidēji algoritms nolaidīsies līdz zemākajam līmenim log(precizitāte), kur aprēķinu precizitāte = MaxReal/Epsilon. Log(precizitāte) <64 (bitu daudzums reāla skaitļa mantisā). Var uzskatīt, ka algoritma darbības laiks nav atkarīgs no precizitātes, jo tas ir konstante. Sliktākajā gadījumā trešais punkts piepildīsies iepriekšējā rekursijas dziļumā, kopumā algoritma darbības laiks $2^{\log(\text{precizitāte})}$. Arī konstants laiks, tikai ļoti liels. Vidēji algoritms strādās logaritmiskā laikā.

Var ierobežot rekursīvo funkciju palaišanas laiku, lai palielinātu veiktspēju. Ja maksimālo rekursijas dziļumu apzīmē kā N, tad minimālā algoritma precizitāte vienādosies $\frac{30}{2^N}$ grādiem. Tātad, ja mēs gribam panākt pagrieziena precizitāti 1 grādā, tad tas tiks panākts ar N=5.

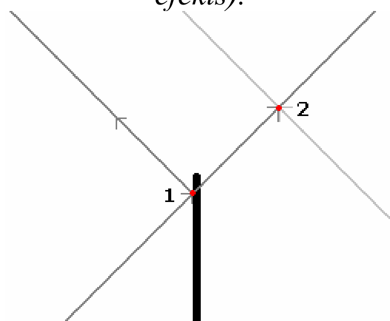
Kolīzijas apstrāde

Ja sadursme tika pamanīta, tad to vajag pareizi apstrādāt**.** Ievadīsim kustības invariantu: ķermeņi jebkurā laika posmā nekrustojas, tie var tikai skart viens otru. Cietie ķermeņi reālajā pasaulē neiet viens otram cauri – tas izskatītos slikti. Sadursmes apstrādes laikā mums jāpanāk invarianta kustības saglabāšanu. Invariants arī ir saistīts ar to apsvērumu, ka būs grūti atdalīt ķermeņus, kad tie krustosies, par cik sadursmes notiks nepārtraukti. Kaut gan atsevišķas metodes [CD1] izmanto to faktu, ka ķermeņi lien viens uz otra.

Ja ar ķermeni notiek vairākas sadursmes laika kvantā, tad no tām jāizvēlas pašu agrāko pēc laika. Ja sadursmes apstrādāt nepareizā secībā, tad no ārpuses tas izskatīsies nereāli, dažreiz iespējams teleportācijas efekts. Ja sadursmes noteikšanas metode var noteikt precīzu sadursmes laiku, tad jāizvēlas visagrāko sadursmi, tā jāapstrādā un pēc tam jāturpina kustība, sākot no dotā laika momenta. Jāizvēlas visagrāko sadursmes laiku nevis vienam ķermenim, bet visiem. Ja to neņem vērā, tad iespējama sadursme ar ķermeni, kuram šajā vietā nav jāatrodas, tā kā sadursmes dēļ otrajam ķermenim jāmaina trajektorija.



Zīm.1: ja sākumā apstrādāt sadursmi ar kreiso ķermeni, tas izskatīsies nereāli (teleportācijas efekts).



Zīm.2: Kolīzijai jānotiek tikai punktā 1, ķermeņi nevar sadurties punktā 2 tāpēc, ka kreisais ķermenis mainīs trajektoriju pēc pirmās sadursmes.

Kā noteikt tiešu sadursmes laiku? Izmantojot pakāpenisko algoritmi (paralelograma krustošanās ar daudzstūri) var atrast punktu, kas ir vistuvākie paralelograma pamatam, kurš atbild par pamatnogriežņa stāvokli laika kvanta sākumā. Zinot šo attālumu un ķermeņu ātrumu, var noteikt tiešu kolīzijas rašanās laiku. Algoritma pseidokods, kas nosaka kolīzijas apstrādes pareizu secību:

```
push(solid_queue, all);
empty(collision_queue);
t1 = 0; t2 = laika kvanta izmērs;
do
{
     for all(solud_queue) move();
     for all
     {
         cleck_collmove(next);
         if collision push(collision_queue,sort by time);
```

```
    }
    if(pop(collision_queue))
    {
        perform(collision);
        remove(collision_queue, collision.solids) // atcelt visas kolīzijas, 
saistītas ar šiem ķermeņiem
        empty(solid_queue);
        push(solid_queue, collision.solids)
        t1 += collision.time;
    }
    else
        break;
}
while(collisions);
```

Tādā veidā mēs pareizā secībā apstrādāsim visas kolīzijas. Bet, ja viens ķermenis saņem daudz sadursmju vienā laika kvantā – būs nepieciešams ļoti daudz izskaitļojošo resursu, lai izpildītu visu algoritmu. No tā var izvairīties, piemēram, sasaldējot ķermeņus pēc četrām kolīzijām. Iznākumā kolīzijas nevarēs būt vairāk kā 2*N(tas ir O(N)), kur N – ķermeņu daudzums. Bez šī ierobežojuma, kolīziju skaits teorētiski nav ierobežots laika kvantā.

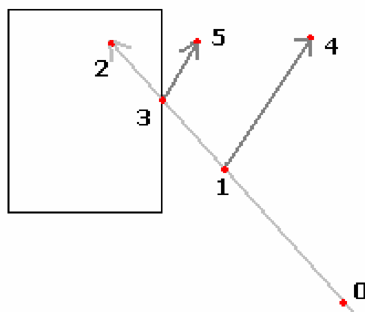
Pēc kolīzijas apstrādes vajag atcelt visas atkarīgās kolīzijas un no jauna pārskaitīt kolīzijas ķermeņiem, uz kuru kustību iespaidu varēja atstāt apstrādājamie ķermeņi. Kādiem ķermeņiem vajag atcelt kolīziju? Sastādīsim grafiku, kura augšpusē būs ķermeņi, bet grafika malā – iespējamās sadursmes. Vajag atrast visas virsotnes, kuras panākamas no diviem ķermeņiem un piedalās apstrādātajā kolīzijā. Vislabāk to darīt meklējot dziļumā, par cik tādu ķermeņu daudzums pie normāliem ātrumiem būs mazākais.

Šai metodei ir viens trūkums – kas jādara, ja viens ķermenis sadursties ar sasaldēto ķermeni? Ievērosim tādu situāciju, kad abas ķermeņos ātrumi ir versti viena virzienā, bet otra ķermeņa ātrums ir lielāks nekā pirmajam, bet tas nekustas, jo ir sasaldots. Ja ķermenis nebūtu sasaldots, tad kolīzija nenotiktu. Arī kolīzijas apstrādes metode, kas ir aprakstīta zemāk, izmanto faktu, ka ķermeni kustas viens pretī otram, no ka tāda pseidokolizija būs nepareizi apstrādāta. Ja neapstrādāt kolīziju, tad var sanākt, ka ķermeni krustosies. Optimālais risinājums šajā gadījumā ir sasaldēt arī pirmo ķermeni.

Atcerēsimies mūsu invariantu: daudzstūriem savajag krustoties. Pastāv divas metodes, lai saglabātu šo invariantu.

Triviāls (*rollback*): atjaunot ķermeņa stāvokli (izvietojums, pagrieziena leņķis), kuri bija iepriekšējā sastāvā, pie tam var rasties problēmas, līdzīgas ar tām, kuras parādītas 1 un 2 zīm.

Otrais: atrast pašu vēlāko ķermeņa stāvokli līdz kolīzijas momentam, pie kura tie vēl nekrustojas (tiešu kolīzijas laiku iekš kvantā).



Zīm. 3: algoritma darbības piemērs.

0 - ķermeņa stāvoklis iepriekšējā laika momentā.

1 – pašreizējā laika momentā.

2 – tas uz kurieni ķermenim bija jāpārvietojas, ja nebūtu kolīzijas.

Triviāla metode pārvietos ķermeni punktā viens, pēc kura nākošajā sastāvā tas pārvietosies punktā 4.

Otrā metode pārvietos ķermeni punktā trīs, kaut gan ideālā variantā ķermenim būtu jānonāk punktā pieci. Savienojot otro metodi un prioritāro konsekvento ķermeņa apstrādi, tad ķermeni var atstāt punktā 3, tikai aizvilkt atpakaļ laikā, tā, lai pie nākamās apstrādes ķermenis nolidotu lielāku attālumu, kompensējot nenolidoto attālumu šajā sastāvā.

Ķermeņu ātruma noteikšana pēc kolīzijas

Reālistiskai kustībai ķermeņu ātrumu pēc sadursmes jāaprēķina pēc fizikas likumiem: impulsu saglabāšanas likums (kustību daudzums) un enerģijas saglabāšanas likums.

Teorētiskais pamatojums

Apskatīsim ķermeņu sadarbību pēc sadursmes [F1]. Ko nozīmē sadarbība? Fizikā sistēmu sadarbība nozīmē kaut kāda lieluma tālāku nodošanu no vienas sistēmas uz otru. Protams, lai lieluma nodošana viennozīmīgi raksturotu sadarbību dotā ķermeņa lielumam nav jāmainās, ja sadarbība izpaliek. Tam jāmainās tikai sadarbības dēļ. Citiem vārdiem sakot, lielumam, kura nodošana raksturo sadarbību, jāpakļaujas saglabāšanas likumam: slēgtā sistēmā šim lielumam jāpaliek konstantei laikā.

Mehānikā sadarbībai ir virzošs raksturs. Ķermenis, piemēram, var saņemt grūdienu tajā vai citā virzienā. Tas nozīmē, ka mehānikā tam lieluma, kura nodošana raksturo ķermeņu sadarbību, jābūt ar vektoru raksturu. To sauc par impulsu.

Tātad, ķermeņa impulss ir lielums, kura izmaiņas dotajam ķermenim ir mēri viņa mehāniskajai sadarbībai ar citiem ķermeņiem. Apskatīsim divu ķermeņu sadarbību. Līdz sadarbībai pirmā ķermeņa impulss bija p1, otrā – p2, pēc sadarbības pirmā ķermeņa impulss būs p1, otrā – p2. impulsa nodošanas princips nozīmē to, ka pirmā ķermeņa impulsa izmaiņas ir saistītas ar otra ķermeņa impulsa izmaiņām, ņemtam ar pievelšanas zīmi, var uzrakstīt vienādojumu:

$$p_1` - p_1 = -(p_2` - p_2) \qquad (1)$$

vai

$$p_1 + p_2 = p_1` + p_2` \qquad (2)$$

Vienādojums (2) arī izsaka impulsa saglabāšanas likumu: divu ķermeņu slēgtās sistēmas kopējais impulss līdz sadarbībai vienāds ar kopējo ķermeņu impulsu pēc sadarbības. Impulss tiek nodots, tas neparādās ne no kā un nepazūd bez pēdām.

Impulsa nodošana no ķermeņa uz ķermeni var kalpot par sadarbības mēru, tomēr pašu par sevi impulsa nodošanu vēl nepilnīgi raksturo sadarbību dotajā laika momentā: tā var attīstīties gan ātri, gan lēni. Vēlams ieviest tādu sadarbības raksturojumu, kurš atspoguļotu impulsa nodošanas ātrumu dotajā laika momentā. Šis tad arī ir spēka raksturojums. Lai pēc laika t viens ķermenis nodotu citam impulsu p. Vidējais impulsa nodošanas ātrums pēc laika t, tiek noteikts ar attiecību

$$\frac{\Delta p}{\Delta t} \qquad (3)$$

Bet ja mūs interesē tālāk nodošanas ātrums, kurš attiecināms uz doto laiku, tad vēlams pāriet līdz robežai pie kuras t tiecas uz nulli:

$$\lim_{\Delta t \longrightarrow 0} \frac{\Delta p}{\Delta t} = \frac{dp}{dt} \qquad (4).$$

Tāda robeža no sevis izsaka vektora funkcijas atvasinājumu laikā, pēc definējuma tas arī ir spēks, ar kuru viens ķermenis iedarbojas uz otru, tātad, par spēku saucams ķermenim piemītošais pirmais atvasinājums no impulsa laikā:

$$F = \frac{dp}{dt} \qquad (5)$$

Atgriezīsimies pie formulas (1), kura izsaka divu ķermeņu sadarbošanos (impulsa nodošanu). Pārrakstīsim to veidā

$$\Delta p_1 = -\Delta p_2 \quad (6)$$

Sadalīsim abas vienādības laika t mijiedarbības daļās un pāriesim pie robežas, pie t, kas tiecas uz nulli. Tad iegūsim sekojošu vienādojumu:

$$F_{12} = -F_{21} \quad (7)$$

Šis vienādojums atspoguļo trešo Ņūtona likumu: darbība ir vienlīdzīga un virzīta pretēji pretdarbībai. Tātad fizikā jebkurai mijiedarbībai ir savstarpējs raksturs: ja ķermenis iedarbojas uz citu ķermeni, tad pēdējais iedarbojas uz citu, mijiedarbības spēki ir vienādi un virzīti pretēji.

Mūsu gadījumā, cieto ķermeņu mijiedarbība notiek acumirklī t laikā, kas tiecas uz nulli.

Ķermeņu mijiedarbība notiek kontakta punktā. Pakāpeniskais algoritms kolīzijas noteikšanas ļauj noteikt punktu, kurā notika sadursme. Tieši šajā vietā tiks izmantots spēks.

Ķermeņu mijiedarbība neaprobežojas tikai ar impulsu nodošanu. Enerģiju sauc par skalāro lielumu, kuras nodošana no ķermeņa uz ķermeni ir universāls to mijiedarbības mērs kā mehānisks, tā arī nemehānisks.

Kā impulss, tā arī enerģija ir zināma ķermeņa stāvokļa parametru funkcija. Tikai impulss ir atkarīgs no viena kustības stāvokļa parametra, no ķermeņa ātruma, enerģija, esot universāls mijiedarbības mērs, var būt atkarīga no daudziem parametriem, kuri raksturo gan ārējo, gan iekšējo ķermeņa stāvokli (kustības ātrumu). Tie ir ātrumi (lineārais un leņķiskais), koordinātes, kas nosaka ķermeņa stāvokli starp citiem ķermeņiem, ar kuriem dotais ķermenis atrodas saistībā (piemēram, ķermeņa augstums virs zemes virsmas), deformācijas parametri, temperatūra un ķermeņa tilpums, parametri, kuri nosaka ķīmisko ķermeņa sastāvu, viņa elektrisko lādiņu, strāvas, kas iziet tam cauri utt.

Šajā gadījumā enerģija ir daudz sarežģītāks jēdziens nekā impulss. Tomēr abi šie lielumi pakļaujas saglabāšanas likumam, t. i. ja mijiedarbība izpaliek, tad arī enerģijai un impulsam būtu jāpaliek pastāvīgam. Daudzos praktiski interesantos gadījumos stāvoklis veidojas tā, ka ķermeņa enerģija sadalās uz saskaitāmiem, kur katrs no tiem ir atkarīgs no viena vai dažiem parametriem. Mūs interesē tikai kinētiskā enerģija, ķermeņa augšupejošā un rotācijas kustības. Visu pārējo ķermeņa enerģiju kustības laikā var ignorēt.

Fakts, ka parasti enerģiju var iedomāties kā dažādu veidu enerģiju summu, ļauj paskatīties uz enerģijas saglabāšanas likumu, kā uz enerģijas saglabāšanas un pārvēršanas likumu: enerģija ne tikai pāriet no ķermeņa uz ķermeni, bet arī pārvēršas no viena veida otrā.

Impulsi var izrēķināt, izmantojot formulu $p = mv$, enerģiju - $E = mv^2$, spēks darbojas uz ķermeni caur formulu $F = m\frac{dv}{dt}$, kur m ir masa un v ir ātrums.

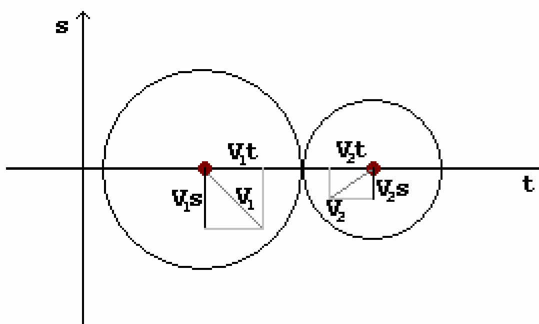
Kolīzijas apstrāde pie augšupejošas kustības

Izanalizēsim kolīzijas apstrādi bez rotācijas vērā ņemšanas. No vienas puses tam jāizskatās nereālistiski, tomēr, pastāv reāli piemēri, kad kolīzija notiek izņemot rotāciju: tās ir absolūti gludas (izņemot berzi) biljarda bumbas, un atomu sadursmes. Grūti iedomāties divu neregulāru formu sadursmi bez rotāciju, bet pastāv vairums spēļu, kurās nav ķermeņu brīvās griešanās, un tas nav īpaši pamanāms.

Ķermenim, kurš nevar rotēt, masas sadalīšana nav svarīga (masu centrs nav svarīgs). Kolīzijas momentā pat nav svarīga ķermeņa forma atšķirībā no kontakta vieta. Tas ir, ķermeņus var aizstāt ar riņķa līnijām, kuras atradīsies kontakta vietās.

Kā notiek divu riņķa līniju kolīzija? Sākumā ir jāatrod ass, garām kurai notiks mijiedarbība. Impulss mainīsies tikai pa šīs ass virzienu. Šī ass (t) iziet cauri pieskaršanās punktam un tās virziens sakrīt ar normāliem pie virsmas saskarsmes punktā. Otra ass (s) ir perpendikulāra t asij. V_1s un V_1t – ātruma projekcijas V_1 pirmajai riņķa uz attiecīgās ass, analoģiski V_2s un V_2t – otrā ķermeņa ātruma projekcija. Kā atrast šīs projekcijas? Projekcija uz t

ass – tas ir ātruma skalārais reizinājums uz t virzošā ass normētu vektoru. Projekcija uz s ass – tas ir pseidoskalārais reizinājums.



Zīm. 4: asis s un t, ātrumu projekcija uz šīm asīm.

Ātrumu projekcijām uz s ass nav jāmainās, par cik nenotiek nekāda mijiedarbība šīs ass virzienā. Asīm t un s visā garumā jādarbojas impulsa saglabāšanās likumam.

$It = const$
$Is = const$
$I_1t + I_2t = const$
$V_1t*m_1 + V_2t*m_2 = const$
$It = V_1t*m_1 + V_2t*m_2 = U_1t*m_1 + U_2t*m_2$ (10),
kur U – ķermeņa ātrumi pēc kolīzijas

Pastāv bezgalīgs risinājumu daudzums vienam vienādojumam ar diviem nezināmajiem, kas nav pretrunā ar fizikas likumiem. Iedomājieties divas bumbiņas, kuras lido viena otrai pretī. Ja tās uztaisītas no tērauda, tad tās aizlidos katra uz savu pusi. Bet, ja no plastilīna – tad salips. Bumbiņas no plastikas atdursies viena no otras, bet ātrums būs mazāks nekā tērauda bumbiņu trieciena laikā. Te spēkā stājas enerģijas saglabāšanas likums. Saduroties divām tērauda bumbiņām kinētiskā enerģija gandrīz nepazūd (notiek absolūti elastīgs trieciens). Plastilīna bumbiņu trieciena laikā, daļa kinētiskās enerģijas pāriet potenciālā (ķermenis deformējas, sasilst – neelastīgs trieciens).

Iztirzāsim elastīgā trieciena gadījumu. Kinētiska enerģija nemainās.

$E = const$
$E1 + E2 = const$
$E = m_1*V_1^2 + m_2*V_2^2 = m_1*U_1^2 + m_2*U_2^2$ (11)

Risinot sistēmu no diviem vienādojumiem mēs atradīsim ātrumu U_1 un U_2. Kā atrast ātrumu, zinot ātruma funkcijas uz perpendikulārās ass? Pitagora teorēma – taisnleņķa trijstūrī katešu kvadrātu summa vienāda ar hipotenūzas kvadrātu. $U^2=U_t^2 + U_s^2$.

$m_1*(V_1t^2 + V_1s^2)+ m_2*(V_2t^2 + V_2s^2) = m_1*(U_1t^2 + U_1s^2) + m_2*(U_2t^2 + U_2s^2)$ (12)
$V_1t*m_1 + V_2t*m_2 = U_1t*m_1 + U_2t*m_2$ (10)
$m_1*U_1s = m_1*V_1s$ (13)
$m_2*U_2s = m_2*V_2s$ (14)

Risināsim vienādojumu sistēmu (10) un(12). Nezināmie – U_1t un U_2t. Vienādojumi (13) un (14) nozīmē impulsu nemainīgumu s ass garumā. Ieliksim atrastās saknes vienādojumā (15) un iegūsim U_1t_1 un U_1t_2.

Izteiksim U_1t no vienādojuma (10).
$U_1t = (It – m_2*U_2t)/m_1$ (15)
Līkam U_1t vienādojumā (12) un iegūstām kvadrātvienādojumu:

$$U_2t^2*(m_2^2/m_1+m_2) + U_2t*(-2*It*m_2/m_1) + It^2 + m_1*V_1s^2 + m_2*V_2s^2 - E = 0 \quad (16)$$

Risinām caur diskriminantu:

$A = m_2^2/m_1+m_2$

$B = -2*It*m_2/m_1$

$C = It^2 + m_1*V_1s^2 + m_2*V_2s^2 - E$

$D = B^2 - 4*A*C$

$$U_2t_1 = \frac{-B+\sqrt{D}}{2A} \quad (17)$$

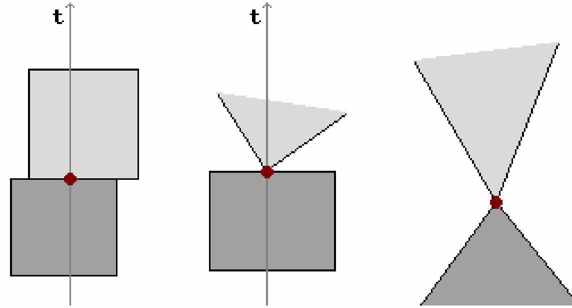
$$U_2t_2 = \frac{-B-\sqrt{D}}{2A} \quad (18)$$

Līkam risinājumos vienādojumā (15), iegūstam U_1t_1 un U_1t_2.

Kuru no risinājumiem – (17) vai (18) izvēlēties – abi saglabāšanas likumi izpildās. Viens risinājums ir pareizs dotā punkta kontaktam, otrs – punktam ar tādu pašu t asi, bet no citas ķermeņa puses. Mēs varam atrast risinājumus, ņemot vērā to, ka ķermeņiem pēc sadursmes jālido katram uz savu pusi. Viens risinājums izmētā ķermeņus uz malām, otrs – otrādi. Līdz triecienam ķermeņi satuvinājās, savādāk tie nevarētu sadurties. Atradīsim pirmā ķermeņa ātrumu attiecībā pret otru. Tā vienādojas $V_1t - V_2t$. Ja šis lielums ir pozitīvs, tad attiecīgajam ātrumam pēc trieciena jābūt negatīvam. Tas ir:

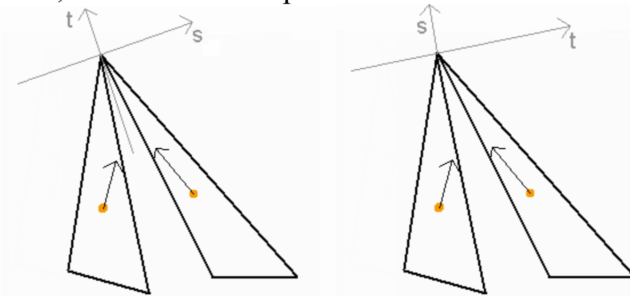
$$\mathrm{sgn}(V_1t - V_2t) = -\mathrm{sgn}(U_1t - U_2t) \quad (19)$$

Tādā veidā, mēs varam izvēlēties pareizo risinājumu no (17) vai (18). Atgriezīsimies pie poligoniem. Kā atrast t asi? Pastāv trīs trieciena varianti: puse ar pusi, stūris ar pusi un stūris ar stūri. Pirmajā un otrajā gadījumā t ass ir normāle plaknes malas sadursmes punktā. Trešais gadījums notiek ļoti reti un ir ļoti grūti noteikt vai ir bijis otrais sadursmes variants, vai pirmais.



Zīm. 5: kolīzijas 3 varianti

Vajag atrast tadu asī, kur projekcijas V_1t un V_2t ai vizēti viens pret otru, tas ir ķermeniem jātuvojas gar asi t, lai algoritms var atrast pareizo risinājumu un izgrūstīt ķermeņus. Nevar gluži vienkārši atrast vidējo normāli, kā zīm. 6 kreisa pusē.



Zīm. 6: nepareizs (pa kreisi) un pareizs (pa labi) asu t un s varianti.

Tā kā daudzstūri ir izliekti un krustojas tikai viena punktā, tad iznāk, ka eksistē sadalīšanas taisne, no vienas puses ir pirmais daudzstūris, no otras – otrais. Tā kā ķermeni tuvīnajās, tad ķermeni cenšas krustos šo taisni. Tad iznāk, ka ķermeņu ātrumu projekcijas uz šo taisni normāli ir versti viens pret otrai. Par s asi var ņemt tādu sadalīšanas taisni. Par sadalīšanas taisni var ņemt vienu no daudzstūru malām.

Ka būt ar nekustīgajiem ķermeņiem, tādiem, ka grīda? Grīda ir nekustīga tāpēc, ka tā ir stingri savienota ar zemi, kura ir nekustīga, jo zeme ir atskaitīšanas punkts. Patiesībā impulss un enerģija nododas zemei, bet zemes masa ir ļoti liela, tāpēc tas ir nemanāmi. Mūsu gadījumā varam nekustīgajam ķermenim piešķirt lielu masu (1/Epsilon), un pēc kolīzijas apstrādes noanulēt ātrumu.

Produktivitātes paaugstināšanas metodes

Premature optmztion is rt of all evl
/programmētāju devīze/

Produktivitātes paaugstināšanas pamatprincips – uzlabot algoritmu. Algoritma laika asimptotika samazināšana var novest pie izmantojamās atmiņas palielināšanas.

Kinemātika

Ķermeņiem ir jākustas tur neko nevar izdarīt. No citas puses, ķermenis, kurš stāv uz grīdas nekustēsies un nerotēs. Tomēr uz to iedarbojas gravitācijas spēks, kura virzīta uz leju. Tātad, ķermenis saņems paātrinājumu, mainīs ātrumu un parādīsies sadursme ar grīdu, laika kvanta sākumā. Par cik tiek izmantota Eilera metode, lai aprēķinātu ķermeņu ātrumu no to paātrinājuma, ātrums šai laika momentā atšķirsies no nulles, no kā ķermenis palēksies uz augšu. Jo lielāks būs laika kvanta izmērs, jo augstāk ķermenis palēksies. Tas izskatīsies nedabiski - bez redzamiem iemesliem trīcošs un ik pa brīdim palecošs ķermenis. Pie tam, uz to tiek tērēts procesora laiks. Izeja no šīs situācijas varētu būt ķermeņu „sasaldēšana”. Ķermeni, kurš nekustas visi spēki, kas uz viņu iedarbojas ir līdzsvaroti, var izslēgt no kinemātikas aprēķiniem līdz tam brīdim, kamēr līdzsvars netiks iztraucēts. Tas var notikt tikai tad, kad ķermenis maina stāvokli, kurā viņš stāv, vai kolīziju ar citu objektu.

Ķermeni, kuru mēs neredzam var apstrādāt retāk. Tas ir palielināt tam laika kvantu. Šī viltība saistīta ar nelielām grūtībām, piemēram, nāksies pārbaudīt kolīziju starp ķermeņiem dažādos laika kvantu momentos. Bet no citas puses, absolūta precizitāte nav vajadzīga, par cik spēlētājam ir grūti pārbaudīt to, ko viņš neredz. Var vispār visus ķermeņus iesaldēt, kuri atrodas diezgan lielā attālumā no redzamās pasaules daļas. Līdzīgu metodi izmanto spēlē GTA, kurā mašīnas pazūd, kuras spēlētājs neredz. GTA šis fakts nedaudz kaitina: aizgriezies un mašīna pazudusi.

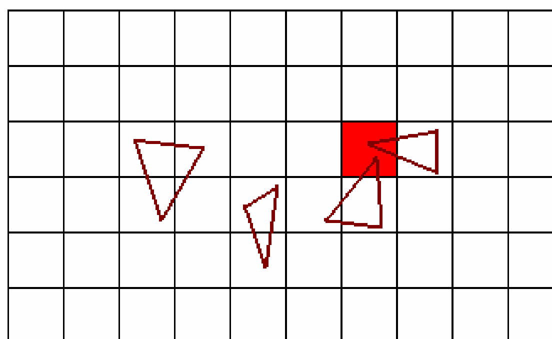
Tīri tehniskā produktivitātes palielināšana – neņemt vērā rotējošā ķermeņa formu, ja tā leņķa ātrums mazāks par Epsilon. Epsilon – tas ir ļoti mazs lielums, piemēram, 10^{-10}. Visi lielumi, kas mazāki par Epsilon skaitās vienādi ar nulli. Tas palīdzēs izvairīties no operāciju cos un sin izskaitļošanas, kurus jāpilda ļoti „ilgi”.

Kolīzijas noteikšana

Visas metodes, kas aprakstītas nodaļā „kolīzijas noteikšana” darbojas par $O(N^2)$ no ķermeņu skaita. Tie pa pāriem pārbauda kolīzijas klātesamību starp katriem diviem daudzstūriem. Acīmredzami ir tas, ka nav iemesla pārbaudīt ķermeņu sadursmes, kuri atrodas tālu viens no otra.

Par acīmredzamu risinājumu kļūst pasaules plaknes sadalīšana klasteros, kuros atrodas daudzstūri. Pēc tam nepieciešams kolīziju pārbaudīt tikai tiem ķermeņiem, kuri atrodas tikai vienā klasterī (dažās realizācijās – arī blakus). Ideālā variantā visos klasteros jābūt vienādam objektu daudzumam. Algoritma darbības laiks labākajā gadījumā – $O(K*(N/K)^2)=O(N^2/K)$, kur N – kopējais ķermeņu daudzums, K – klasteru daudzums. Šeit kā reiz izpaudās izmantojamās atmiņas daudzuma palielināšanas princips (klasteru glabāšanai ir vajadzīga atmiņa) produktivitātes paaugstināšanas dēļ.

Pati vienkāršākā klasterizēšanas metode – tā ir divdimensiju jaukšana (2D-hash) [CD1]. Visa pasaule dalās kvadrātos (heššūniņas, klasteri). Par cik plakne ir bezgalīga, tad kvadrātos sadalīsim ne visu plakni, bet dažus taisnstūrus no mūsu spēļu laukuma, bet visus pārējos attiecināsim uz atsevišķu heššūniņu. Kolīzijas ir jāpārbauda tikai ar ķermeņiem, kuriem ir kaut viena kopīga rūtiņa. Ja ķermenis maina stāvokli, nav jāaizmirst atjaunot informāciju par heššūniņām.



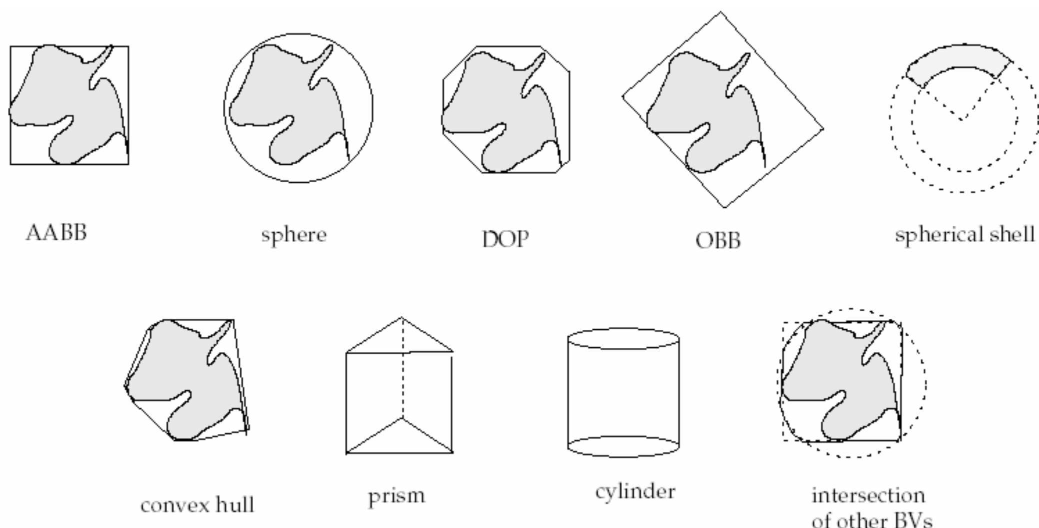
2D-hash

Šīs metodes trūkums ir tas, ka lieliem spēles laukumiem var būt nepieciešams daudz atmiņas, lai glabātu posmu. Toties šī metode realizēšanā ir vienkārša. Var izmantot Varonova diagrammu [AG1], katras diagrammas šūnas centrs kalpos par daudzstūra masas centru. Šī metode rada pašu efektīvāko visuma klasterizāciju, bet šī diagramma nav mūžīga – pie ķermeņa stāvokļa izmaiņām ir nepieciešams no jauna veidot diagrammu. Diagramma veidojas laikā O(N*logN). Šie zaudējumi atmaksājas ar to, ka netiks veiktas liekas kolīzijas pārbaudes.

Nedaudz mazāku efektu dot algoritma optimizācijas pārbaude divu daudzstūru krustošanās laikā (mūsu gadījumā daudzstūris sastāv no vecā un jaunā ķermeņa stāvokļa, kā arī no savienojošiem nogriežņiem). Parastā divu daudzstūru krustošanās metode strādā uz O(N*M), kur N un M – daudzstūra malu daudzums, pati ātrākā metode strādā uz O(N+M), bet šī metode pieprasa abu plakņu izliekumu.

Pirms daudzstūru krustpunktu pārbaudes varam pārbaudīt kādu punktu kopu krustojumu, kuri ietver sevī izejošos daudzstūra punktus. Par piemēru var kalpot AABB (Axis-Aligned Bounding Box). Tas ir taisnleņķa trijstūris, kura malas ir paralēlas koordinātu asīm un kuras ietver visus punktus no izejošās plaknes. Divu AABB krustojumu pārbaude notiek konstantā laikā, ja tās nekrustojas, tad nav jēgas pārbaudīt divu daudzstūru krustošanos – tas nekrustosies.

Bez AABB var izmantot riņķa līniju, divu riņķa līniju krustošanās pārbaude notiek konstantā laikā.



Zīm.: dažu daudzumu piemēri, kuru krustošanās pārbaude notiek ātrāk, nekā daudzstūru krustošanās [CD3].

Var panākt produktivitātes palielinājumu pēc kolīzijas noteikšanas rotācijas kustībā, jo ķermenis veic pilnu apgriezienu laika taktī. Kā jau tika minēts „kolīzijas apstrādes” nodaļā, daudzstūri var nomainīt pret riņķa līniju.

Kolīzijas apstrāde

Kolīzijas apstrāde notiek konstantā laikā. Viens no optimizācijas principiem saka – nepieciešams optimizēt to koda daļu, kas strādā visilgāk un aizņem mazāko koda daļu. Kolīzijas apstrādi optimizēt ir bezjēdzīgi.

Noslēgums

Šajā kursa darbā tika apskatīti dažas kolīzijas noteikšanas un apstrādes metodes. Tiek realizēts: aprakstītas datu struktūras, kinemātika augšupejošai un rotējošai kustībai ar Eilera metodes palīdzības, pakāpeniskais algoritms + kolīzijai noteikšanai, kolīzijas apstrādātais, kurš ievēroja pareizo kartību kolīzijas apstrādes, Kolīzijas apstrāde pie augšupejošas kustības. Viss tiek realizēts C++ vidē ar standartu šablonu bibliotēku STL un grafisku bibliotēku OpenGL palīdzību.

Kopumā modelis parada labus, reālistiskus rezultātus. Ķermeni kustas un sadurstās, izmantojot reālus fiziskus likumus. Bet eksistē arī daži problēmas. Pirmā – ātrdarbības krišana pie daudziem kolīzijām viena laika kvantā. Nodaļā „Produktivitātes paaugstināšanas metodes" tika apskatīti daži paātrināšanas metodi lai izvairīties no tādiem problēmām. Otra problēma – skaitļiem ar peldošo punktu aprēķināšanas kļūdas. Pie maziem ātrumiem un attālumiem starp ķermeņiem ķermeni var sākt iet viens caur otru. Piemērām šāda problēma paradās kad ķermeni stāv viens uz otra un uz abiem ķermeņiem darbojas gravitācijas spēks. Daļēji šādu problēmu var atrisināt ķermeņu sasaldināšana un savstarpēji ķermeņu atgrūšanās. Dažreiz analoģiskas problēmas paradās arī populārās spēles. Realizācijas laikā tika pamanītas dažādas algoritmiskas kļūdas, kuri pēc tam tika aprakstīti kursa darbā.

Kursa darbs var noderēt pamatam vienkāršam datorspēles divdimensiju fiziskam modelim bez rotācijas, ar ko autors gatavojas nodarboties tuvāka laikā.

Kursa darba par loģisku turpinājumu var kļūt kolīzijas apstrāde ar rotāciju. Vēl var ievesta tādus primitīvus, ka virve, ķēde, elastīgi ķermeni, ķermeņu grupas lai formēt ķermeņus ar neizliektiem formām un daļiņu sistēmas.

Par citu uzlabošanas variantu var kalpot pāreja no divdimensiju pasaules un trīsdimensiju pasauli. Nav principiālas atšķirības algoritmos, sarežģītība ir tikai kolīzijas noteikšanas un apstrādes formulās. Trīsdimensiju pasaulē katram ķermenim ir jau seši brīvības pakāpes (trīs no tiem ir rotācijas brīvības pakāpes), bet divdimensiju pasaulē it tikai trīs brīvības pakāpes.

Literatūras saraksts

Fizika:
[F1] Ивановский государственный энергетический институт. Физические Основы Механики (http://www.ispu.ru/library/physics/delete/mehan/ogl.html)

[F2] Таврический Национальный Университет. Общая физика (http://www.ccssu.crimea.ua/tnu/structure/physic_fac/departments/general/common_phys/all/mehan4.htm)

[F3] Элементы большой науки. Принцип неопределенности Гейзенберга (http://www.chemistry.ssu.samara.ru/chem1/22_geiz.htm)

[F4] Information at the foundation of modern science and technology from the Physics Laboratory of NIST. International System of Units (http://physics.nist.gov/cuu/Units/units.html)

Modelēšana:
[M1] Александр Куприн. Моделирование физики в компьютерных играх с помощью частиц (http://gamedev.ru/articles/print.shtml?id=30107)

Collision Detection:
[CD1] Григорий Григоренко. Двумерный движок физики многоугольников и окружностей на основе метода импульсов (http://www.gamedev.ru/articles/print.shtml?id=70124)

[CD2] Родерик Кеннеди. Физика контакта (http://dtf.ru/articles/print.php?id=371)

[CD3] Collision Detection for Deformable Objects / Aut. kol. M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Fuhrmann,
M.-P. Cani, F. Faure, N. Magnenat-Thalmann, W. Strasser, P. Volino. The Eurographics Association and Blackwell Publishing, 2005.

Programmēšana:
[P1] W.H. Press, B.P. Flannery, S.A. Teukolsky, andW.T.Vetterling. *Numerical Recipes*. Cambridge, University Press, 1986.

[P2] Роберт Седжвик, Принстонский Университет. Фундаментальные алгоритмы на C++ - Москва: DiaSoft, 2001. – 688 стр.

[P3] Дональд Э. Кнут. Искусство программирования – Москва, Вильямс, 2004 – 3 тома.

Algoritmiska Ģeometrija:
[AG1] Paulis Ķikusts: Algoritmiskā ģeometrija (Lekcijas) (http://www.ltn.lv/~paulis/ALGEOM/galva.html)

[AG2] П. С. Моденов. Аналитическая Геометрия – Москва: Издательство Московского Университета, 1955. – 564 стр.

[AG3] А. Фокс, М. Пратт. Вычислительная Геометрия – Москва: Мир, 1982 – 304 стр.

OpenGL:
[GL1] Френсис Хилл. «OpenGL. Программирование компьютерной графики. Для профессионалов» - Санкт-Петербург: Питер, 2002 – 1088 стр.

Apliecinājums

Ar šo es apliecinu, ka šodien iesniegto kursa darbu es esmu veicis pašrocīgi un esmu izmantojis tikai tajā norādītos palīglīdzekļus.

Rīgā, Paraksts:

Pielikums (Programmas teksts)

Tika uzrakstīta programma, kas realizēja aprakstītus algoritmus. Programma ir daļa no izstrādātas datorspēles. Spēle sastāv no daudziem daļam: grafika, interfeiss, vadība, tikla, resursi, spēles noteikumi un fizikas emulācija. Visa sistēma uzrakstīta C++ valodā ar STL un OpenGL palīdzību. Fizikas emulācijas kodols sastāv uzkodēts failos *fizika.cpp* un *fizika.h*.

fizika

Šī modulī centra ir divi klasi *Solid* un *World*.

Solid – tas ir viens cietais ķermenis, kas ir attēlots ka daudzstūris. *Prev* un *cur* ir ķermeņa stāvoklis laika kvanta saukuma un beigas. *Solid::moveForward()* un *Solid::moveRotate()* ir ķermeņa kinemātikas apstrāde.

Clase *World* atbild par ķermeņu glabāšanu un kolīzijas apstrādes. *World::move()* atbild par laika kvanta apstrādes (*Move – move forward in time*). Iekš funkcijas ir kinemātikas apstrāde, kolīzijas noteikšana un apstrāde. *World::collisionDetect()* parbauda, vai ir kolīzija starp noradītajam ķermenim un visiem citiem.

geometry

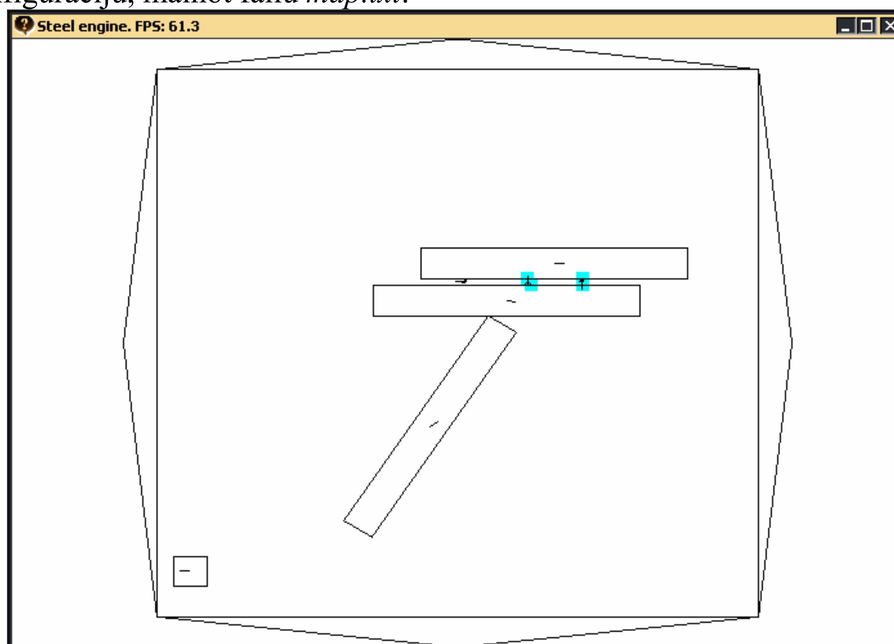
Papildus izmantojas analītiskas ģeometrija, kura ir realizēta modulī *geometry,* kur ir divu nogriežņu krustošanas noteikšana un analoģiski algoritmi.

vector2D

Tas ir modulis, kurā ir aprakstīta klase *Vector2D*, kas realizē divdimentisu vektoru. Klase izmanto *operator overloading*, kas vienkāršo programmas uzrakstīšanu. Piemēram, var uzrakstīt a += b.vectorProduct(c+d*x) , kur a, b, c un d ir vektori.

Programma

Gatava programma demonstrē fizikas emulācijas procesu un izvada to ekrānā. Ķermeni tiek attēloti kā daudzstūri. Kolīzijas vietas apzīmēti ar sarkanam kvadrātam. Ir iespēja uzdot savu ķermeņu konfigurāciju, mainot failu *map.txt*.



Program Screenshot

```
 1: /***************************************************************
 2:   TheGame physics unit (Steel engine)
 3:   (C) KindeX (Andrey Ivanov), kindex@inbox.lv, http://kindex.lv
 4:   Division, 2004-2005.
 5: ***************************************************************/
 6: #ifndef fizikaH
 7: #define fizikaH
 8: 
 9: #include "_cpp.h"
10: #include "geometry.h"
11: 
12: struct World;
13: 
14: struct SolidLocation // положение твёрдого тела в простнанстве
15: {
16:     Vector2 position, velocity;
17:     real w; // uglovaja skorost'
18:     real angle;
19:     real time;
20:     polygon p; // kontur (absolutnie koordinati)
21: };
22: 
23: struct Solid // твёрдое тело
24: {
25:     polygon p; // kontur (otnositelnie koordinati, pri angle=0, position=(0,0))
26:     real mass;
27:     bool movable;
28:     int id;
29:     int collision_count;
30:     bool in_obj, disabled;
31:     SolidLocation prev,cur;
32:     vector<int> collisions; // nomera koolizij, v kotorih zadejstvovano eto telo
33: 
34:     Solid(): id(-1), movable(true)  {}
35:     void addp(const Vector2 & newp) { p.add(newp);}
36: 
37:     void moveForward(real time);
38:     void moveRotate(real time);
39: 
40:     void updatePosition(real t); // calculate new solid position (cur) from prev in t sec
41:     void updatePerimeter(); // calculate kontur
42:     void draw(int method);
43:     void push(const Vector2 &location, real power);
44:     polygon getPerimeterRotate(real time);
45: 
46:     void normalize(); // update mass center
47:     real getw() { if (!disabled) return prev.w; else return 0; }
48:     Vector2 getvelocity() { if (!disabled) return prev.velocity; else return Vector2(0,0);
    }
49: 
50:     void getNearestSide(Vector2 point, real &dest, Vector2 &normal, SolidLocation second);
51: };
52: 
53: struct Collision
54: {
55:     int solids[2];
56:     real time;
57:     Vector2 point;
58:     int id;
59: };
60: 
61: typedef vector<int> ObjList;
62: typedef enum {forward, rotate} MoveType;
63: 
64: struct World
65: {
66:     int turn;
67:     vector<Solid> solid;
68:     vector<Collision> collision, processedCollision; // TODO priority_queue
69:     int collisionId;
70: 
71:     World() {     turn = 0; }
72: 
73:     void move();
74:     void draw();
75:     void drawdebug();
```

```
76:     void addSolid(Solid &S) { S.id = solid.size(); solid.push_back(S); }
77:     bool collisionDetectForward(Solid &first, Solid &second, real &time, Vector2 &Point);
78:     bool collisionDetectRotate(Solid &first, Solid &second, real &time, Vector2 &Point);
79:     bool collisionDetectRotateTime(Solid &first, Solid &second, real start, real finish,
     Vector2 &Point);
80:     void prepareObj(ObjList &obj);
81:     bool addunique(ObjList &obj, int s); // true if added, false if already was added
82:     void collisionDetect(Solid &first, real start, real finish, MoveType t);
83:     void cancelCollision(int collision_id, ObjList &obj, bool disabled);
84:     void load(string filename);
85:     void save(string filename);
86:     void moveTime(real time, MoveType t);
87:     void collisionHandle(Solid &first, Solid &second, Vector2 point);
88: };
89:
90: extern World world;
91:
92:
93: //---------------------------------------------------------------------------
94: #endif
95:
```

```
1: /***************************************************************
2:   TheGame physics unit (Steel engine)
3:   (C) KindeX (Andrey Ivanov), kindex@inbox.lv, http://kindex.lv
4:   Division, 2004-2005.
5: ***************************************************************
6:
7: Main algorithm:
8:
9: push(solid_queue, all);
10: empty(collision_queue);
11: t1 = 0; t2 = turn_duration;
12: do
13: {
14:     for all(solud_queue) move();
15:     for all
16:     {
17:         cleck_collmove(next);
18:         if collision push(collision_queue,sort by time);
19:     }
20:     if(pop(collision_queue))
21:     {
22:         perform(collision);
23:         remove this collisions from collision queue - отменить все коллизии для этих тел
24:         empty(solid_queue);
25:         push(solid_queue, collision.solids)
26:         t1 = collision.time;
27:     }
28:     else
29:         break;
30: }
31: while(collisions);
32: ***************************************************************************/
33:
34: #pragma hdrstop
35:
36: #include "fizika.h"
37: #include "renderer.h"
38:
39: World world;
40:
41: void Solid::draw(int method)
42: {
43:     polygon P;
44:     if (method == 0) P = cur.p; else P = prev.p;
45:     for(int i=0; i<p.n; i++)
46:         glLine(
47:             P.p[i].x,
48:             P.p[i].y,
49:             P.p[(i+1)%p.n].x,
50:             P.p[(i+1)%p.n].y);
51:
52:     Vector2 speed = 0.03*(cur.velocity.GetNormalized());
53:     glLine(
54:         cur.position.x,
55:         cur.position.y,
56:         cur.position.x + speed.x,
57:         cur.position.y + speed.y);
58:
59:     if (method==1)
60:         glPoint(P.p[0].x,P.p[0].y,5);
61: }
62:
63: void Solid::moveForward(real time)
64: {
65:     if (!disabled)
66:     {
67:         if (time<0)    time = 0;
68:         cur.position    = prev.position + prev.velocity*time;
69:         updatePerimeter();
70:     }
71: }
72:
73: void Solid::moveRotate(real time)
74: {
75:     if (!disabled)
76:     {
```

```
 77:         if (time<0)    time = 0;
 78:         cur.angle    = prev.angle + prev.w*time;
 79:         updatePerimeter();
 80:     }
 81: }
 82: 
 83: 
 84: void World::prepareObj(ObjList &obj)
 85: {
 86:     obj.clear();
 87:     for(unsigned int i=0; i<solid.size(); i++)
 88:     {
 89:         obj.push_back(i);
 90:         solid[i].collision_count = 0;
 91:         solid[i].collisions.clear();
 92:         solid[i].in_obj = true;
 93:         solid[i].disabled = !solid[i].movable;
 94:     }
 95: }
 96: 
 97: void World::collisionDetect(Solid &first, real start, real finish, MoveType t)
 98: {
 99:     for(unsigned int j = 0; j<solid.size(); j++)
100:         if ((!solid[j].in_obj ||  j > (unsigned int)first.id)
101:              &&!(first.disabled && solid[j].disabled))
102:         {
103:             real collision_time = finish - start;
104:             Collision c;
105:             if (
106:             t == forward && collisionDetectForward(first,solid[j],collision_time,c.point)
107:             ||
108:             t == rotate && collisionDetectRotate(first,solid[j],collision_time,c.point)
109:             )
110:             { // RollBack
111:                 collisionId++;
112:                 c.solids[0] = first.id;
113:                 c.solids[1] = j;
114:                 c.id = collisionId;
115:                 first.collisions.push_back(collisionId);
116:                 solid[j].collisions.push_back(collisionId);
117:                 switch(t)
118:                 {
119:                 case forward: c.point += solid[j].getvelocity()*collision_time; break;
120:                 }
121:                 c.time = start + collision_time;
122:                 collision.push_back(c);
123:             }
124:         }
125: }
126: 
127: bool World::addunique(ObjList &obj, int s)
128: {
129:     if (solid[s].disabled) return true;
130:     for(int i=0; i<obj.size(); i++)
131:         if (obj[i] == s) return false;
132:     obj.push_back(s);
133:     solid[s].in_obj = true;
134:     return true;
135: }
136: 
137: void World::cancelCollision(int m, ObjList &obj, bool disabled)
138: {
139:     int n = -1;
140:     for(int i=0; i<collision.size(); i++)
141:         if (collision[i].id == m)
142:         {
143:             n = i;
144:             break;
145:         }
146:     if (n<0) return; // collision not found
147: 
148:     int a = collision[n].solids[0],
149:         b = collision[n].solids[1];
150: 
151:     solid[a].disabled |= disabled;
152:     solid[b].disabled |= disabled;
```

```
153:
154:     collision.erase(&collision[n]);
155:
156:     if (addunique(obj, a))
157:     {
158:         while(!solid[a].collisions.empty())
159:         {
160:             int x = solid[a].collisions.back();
161:             solid[a].collisions.pop_back();
162:             cancelCollision(x, obj, disabled);
163:         }
164:     }
165:     a = b;
166:     if (addunique(obj, a))
167:     {
168:         while(!solid[a].collisions.empty())
169:         {
170:             int x = solid[a].collisions.back();
171:             solid[a].collisions.pop_back();
172:             cancelCollision(x, obj, disabled);
173:         }
174:     }
175:
176: }
177:
178: void World::moveTime(real time, MoveType t)
179: {
180:     for(unsigned int i=0; i<solid.size(); i++)
181:     {
182:         Solid &s = solid[i];
183:         s.prev = s.cur;
184:         switch (t)
185:         {
186:         case forward: s.moveForward(time); break;
187:         case rotate: s.moveRotate(time); break;
188:         }
189:     }
190: }
191:
192:
193: void World::move() // F Engine
194: {
195:     turn++;
196:     MoveType t;
197:     ObjList obj;
198:     processedCollision.clear();
199:     collisionId = 0;
200:     int subturn;
201:
202: for(int t = forward; t<rotate; t++) // forward, rotate
203: {
204:     subturn = 0;
205:     collision.clear();
206:     prepareObj(obj);
207:     real start=0, finish = settings.speed;
208:
209:   while (subturn<100)
210:   {
211:     subturn++;
212:     moveTime(finish-start, (MoveType)t);
213:     if(settings.debug) { videoClearAll(); drawdebug(); flush();}
214:
215:     for(unsigned int i = 0; i<obj.size(); i++)
216:         collisionDetect(solid[obj[i]],start, finish, (MoveType)t);
217:
218:     if (collision.empty()) break; // no collisions
219:
220:     int ci = 0;
221:     real mintime = collision[ci].time;
222:     for(unsigned int i=1; i<collision.size(); i++)
223:         if (mintime>collision[i].time)
224:         {
225:             mintime = collision[i].time;
226:             ci = i;
227:         }
228:     if (ci < 0) break; // no collisions left
```

```
229:
230:     Collision c = collision[ci];
231:     obj.clear(); for(unsigned int i=0; i<solid.size(); i++) solid[i].in_obj = false;
232:
233:     bool dis = false;
234:     for(int i = 0; i<2; i++)
235:     {
236:         Solid &s = solid[c.solids[i]];
237:         if(s.disabled && s.movable)
238:             dis = true; // disable all objects, dlja kotorih mogla proizojti kollizija
239:     }
240: //    dis |= mintime < 0.001;
241:     // collision_handling
242:     for(unsigned int i=0; i<solid.size(); i++)
243:     { // all
244:         Solid &s = solid[i];
245:         real time = 0.99*(mintime-start)-0*EPSILON;
246:         if (dis || mintime-start < 1000*EPSILON) time = 0;
247:         switch (t)
248:         {
249:         case forward: s.moveForward(time); break;
250:         case rotate: s.moveRotate(time); break;
251:         }
252:     }
253:     cancelCollision(c.id, obj, false);
254:
255:     if (!dis) // esli eta kollizija mogla proizojti
256:     {
257:         collisionHandle(solid[c.solids[0]], solid[c.solids[1]], c.point);
258:
259:         for(int i = 0; i<2; i++)
260:         {
261:             Solid &s = solid[c.solids[i]];
262:             s.disabled |= dis;
263:             s.collision_count++;
264:             if (s.collision_count>=2)
265:                 s.disabled = true;
266:         }
267:         processedCollision.push_back(c);
268:     }
269:     start = mintime;
270:   }
271: }
272: // TODO - process forces
273:     for(unsigned int i=0; i<solid.size(); i++)
274:     { // all
275:         Solid &s = solid[i];
276:         if (!s.disabled)
277:         {
278:             s.cur.velocity *= 1-settings.resistance;
279:             s.cur.velocity += settings.g*settings.speed;
280:         }
281:     }
282: }
283:
284: void World::collisionHandle(Solid &first, Solid &second, Vector2 point)
285: {
286:     real dest1, dest2;
287:     Vector2 n1,n2,n;
288:     first.getNearestSide(point, dest1, n1, second.cur); // searching for normal for
       razdelajushaja linija
289:     second.getNearestSide(point, dest2, n2, first.cur); // vdol normali budet proishodit'
       vzaimodejstvie
290:     assert(dest1>-EPSILON || dest2>-EPSILON );// хоть одна разделяющая прямая была найдена
291:     if (dest1>dest2) n = n1; else n = n2; // t
292:
293:     Vector2 m = n.rot_90(); // s
294:
295:     real v1t = first.cur.velocity.dotProduct(n),
296:          v2t = second.cur.velocity.dotProduct(n),
297:          v1s = first.cur.velocity.pseudoscalarProduct(n),
298:          v2s = second.cur.velocity.pseudoscalarProduct(n);
299:     real &m1 = first.mass, &m2 = second.mass,
300:          v1sqr = first.cur.velocity.GetSquaredLength(), v2sqr =
       second.cur.velocity.GetSquaredLength();
301:
```

```
302:     real E = m1*v1sqr + m2*v2sqr;
303:     real I = m1*v1t   + m2*v2t;
304: // m1*v1t + m2*v2t = const
305:     real a = m2 + m2*m2/m1;
306:     real b = -2*I*m2/m1;
307:     real c = I*I/m1 + m1*sqr(v1s) + m2*sqr(v2s) - E;
308:     real d = b*b-4*a*c;
309:     if (fabs(d)<EPSILON) d = 0;
310:     d = sqrt(d);
311:     real u2t1 = (-b - d)/a*0.5,
312:          u2t2 = (-b + d)/a*0.5;
313:     real u1t1 = (I - m2*u2t1)/m1,
314:          u1t2 = (I - m2*u2t2)/m1;
315:
316:     real E1 = m1*(sqr(u1t1) + sqr(v1s)) + m2*(sqr(u2t1) + sqr(v2s));
317:     real E2 = m1*(sqr(u1t2) + sqr(v1s)) + m2*(sqr(u2t2) + sqr(v2s));
318:     // 4to vibrat' 1 ili 2 ?
319:
320:     real u1t, u2t;
321:     if (sgn(v1t-v2t) == sgn(u1t1-u2t1))
322:     { // skorosti dolzni izmenitsja na protivopoloznie
323:         u1t = u1t2;
324:         u2t = u2t2;
325:     }
326:     else
327:     {
328:         u1t = u1t1;
329:         u2t = u2t1;
330:     }
331:
332:     if (first.movable)    first.cur.velocity = n*u1t + m*v1s;
333:     if (second.movable)   second.cur.velocity = n*u2t + m*v2s;
334:
335:     if (!first.movable) // TODO ???
336:         second.cur.velocity -= n*u1t + m*v1s;
337:     if (!second.movable)
338:         first.cur.velocity -= n*u2t + m*v2s;
339: }
340:
341: void Solid::getNearestSide(Vector2 point, real &dest, Vector2 &normal, SolidLocation
      second)
342: {
343:     Vector2 cp;
344:     for(int i=0; i<p.n; i++)
345:         if (intersect(
346:                 cur.p.p[i],
347:                 cur.p.p[(i+1)%p.n] - cur.p.p[i],
348:                 cur.position,
349:                 (point - cur.position)*1000,
350:                 cp
351:                 ))
352:         {
353:             real dest1 = (cp-cur.p.p[i]).GetSquaredLength();
354:             real dest2 = (cp-cur.p.p[(i+1)%p.n]).GetSquaredLength();
355:             int v;
356:             if (dest1<dest2)
357:             {
358:                 dest = dest1;
359:                 v = i;
360:             }
361:             else
362:             {
363:                 dest = dest2;
364:                 v = (i+1)%p.n;
365:             }
366:
367:             if (dest>EPSILON)
368:                 normal = (cur.p.p[(i+1)%p.n] - cur.p.p[i]).rot_90().GetNormalized();
369:             else
370:             // nahodim razdeljajushuju prjamuju
371:             {
372:                 Line r;
373:                 r.a = cur.p.p[v];
374:                 r.b = (cur.p.p[(v+1)%p.n]-cur.p.p[v]).GetNormalized();
375:                 int s1 = oneside(r, second.p);
376:                 int s2 = oneside(r, cur.p);
```

```
377:                if (s1 != 0 && s1 == -s2)
378:                    normal = r.b.rot_90();
379:                else
380:                {
381:                    r.a = cur.p.p[(v+1)%p.n];
382:                    r.b = (cur.p.p[(v+2)%p.n]-cur.p.p[(v+1)%p.n]).GetNormalized();
383:                    int s1 = oneside(r, second.p);
384:                    int s2 = oneside(r, cur.p);
385:                    if (s1 != 0 && s1 == -s2)
386:                        normal = r.b.rot_90();
387:                    else
388:                        dest = -1; // net razdeljajushej prjamoj, zna4it est' vo vtorom
     poligone
389:                }
390:            }
391:            return;
392:        }
393:    assert(false);
394: }
395:
396: void Solid::updatePerimeter() // calculate new solid position (cur) from prev in t sec
397: {
398:     real sina = sin(cur.angle);
399:     real cosa = cos(cur.angle);
400:     cur.p = p;
401:     for (int i=0; i<p.n; i++)
402:     {
403:         cur.p.p[i].rotate(sina,cosa);
404:         cur.p.p[i] += cur.position;
405:     }
406: }
407:
408: real getangle(Vector2 pos, Vector2 point, Vector2 orig)
409: {
410:     real a = atan2(point.x - pos.x, point.y - pos.y);
411:     real angle = atan2(point.x - orig.x, point.y - orig.y);
412:     if (a-angle>M_PI) a -= 2*M_PI;
413:     if (a-angle<-M_PI) a += 2*M_PI;
414:     return angle - a;
415: }
416:
417: polygon Solid::getPerimeterRotate(real time)
418: {
419:     cur.angle = prev.angle + getw()*time;
420:     updatePerimeter();
421:     return cur.p;
422: }
423:
424: bool World::collisionDetectRotateTime(Solid &first, Solid &second, real start, real
     finish, Vector2 &cpoint)
425: {
426:     polygon A = first.getPerimeterRotate(start);
427:     polygon B = first.getPerimeterRotate(finish);
428:     polygon C = second.getPerimeterRotate(start);
429:     polygon D = second.getPerimeterRotate(finish);
430:
431:     polygon P,Q;
432:     P.p.resize(4);
433:     Q.p.resize(4);
434:     P.n = 4;
435:     Q.n = 4;
436:     for(int i=0; i<first.p.n; i++)
437:     {
438:         P.p[0] = A.p[i];
439:         P.p[1] = A.p[(i+1)%first.p.n];
440:         P.p[2] = B.p[(i+1)%first.p.n];
441:         P.p[3] = B.p[i];
442:
443:         for(int j=0; j<second.p.n; j++)
444:         {
445:             Q.p[0] = C.p[j];
446:             Q.p[1] = C.p[(j+1)%second.p.n];
447:             Q.p[2] = D.p[(j+1)%second.p.n];
448:             Q.p[3] = D.p[j];
449:
450:             for (int a=0; a<4; a++)
```

```
451:                  for (int b=0; b<4; b++)
452:                      if (intersect(
453:                          P.p[a], P.p[(a+1)%4]-P.p[a],
454:                          Q.p[b], Q.p[(b+1)%4]-Q.p[b],
455:                          cpoint))
456:                          return true;
457:
458:              for (int b=0; b<4; b++)
459:                  if (P.isIn2(Q.p[b]))
460:                  {
461:                      if (P.isIn2(Q.p[b]))
462:                      cpoint = Q.p[b];
463:                      return true;
464:                  }
465:
466:              for (int a=0; a<4; a++)
467:                  if (Q.isIn2(P.p[a]))
468:                  {
469:                      cpoint = P.p[a];
470:                      return true;
471:                  }
472:          }  // for j
473:      } // fro i
474:      return false;
475: }
476:
477: bool World::collisionDetectRotate(Solid &first, Solid &second, real &time, Vector2 &Point)
478: {
479:     Vector2 cpoint;
480:     real t = time;
481:     real rot1 = fabs(first.getw()*t);
482:     real rot2 = fabs(second.getw()*t);
483:     real rot = rot1;
484:     if (rot2>rot) rot = rot2;
485:
486:    if (rot<EPSILON) return false;
487:     // MAX rota angle = 30 grad = Pi/6
488:     int m = (int)(rot/(M_PI/6)+0.9);
489:     if (m <= 0) m = 1;
490:     real delta = rot/m;
491:
492:     polygon A = first.prev.p,
493:             B = second.prev.p;
494:
495:     SolidLocation F = first.cur, S = second.cur; // BACKUP
496:
497:     bool collision = false;
498:     bool a1,a2;
499:     time = 0;
500:     real start, finish, middle;
501:     Vector2 cpoint1,cpoint2;
502:
503:     for (int k=0; k<m && !collision; k++) // scaling
504:     {
505:         start = (real)t*k/m;
506:         finish = (real)t*(k+1)/m;
507:         if (collisionDetectRotateTime(first, second, start, finish, cpoint))
508:         {
509:             do{
510:                 middle = (start+finish)*0.5;
511:                 a1 = collisionDetectRotateTime(first, second, start, middle, cpoint1);
512:                 if (a1)
513:                     finish = middle;
514:                 else
515:                 {
516:                     a2 = collisionDetectRotateTime(first, second, middle, finish,
    cpoint2);
517:                     if (a2)
518:                         start = middle; // a2 = true
519:                     else
520:                         break;   // ne bilo kollizii, lozhnaja trevoga, no mogla bit'
    pozhe. Nuzhen rekursibnij algoritm
521:                 }
522:             } while ((finish-start)*rot/t>M_PI/180/2 /* 0.5 gradus*/);
523:
524:             {
```

```
525:                 if (a1)
526:                 {
527:                     Point = cpoint1;
528:                     time = start;
529:                 }
530:                 else if (a2)
531:                 {
532:                     Point = cpoint2;
533:                     time = middle;
534:                 }
535:                 else
536:                 {
537:                     Point = cpoint;
538:                     time = start;
539:                 }
540:                 collision = true;
541:                 break;
542:             }
543:         }
544:     }// for k
545:
546:     first.cur = F;
547:     second.cur = S;
548:     return collision;
549: }
550:
551: bool World::collisionDetectForward(Solid &first, Solid &second, real &time, Vector2
      &Point)
552: {// delaem takuju sistemi ots4jeta otnosilelno second (eto skorost'=0)
553: /*    Vector2 rel =
554:       cur.position + second.prev.position - second.cur.position;*/
555:
556: //    Vector2 rel = first.getvelocity()*time + second.prev.position - second.cur.position;
557:     Vector2 rel = (first.getvelocity() - second.getvelocity())*time;
558:     if (rel.GetSquaredLength()<EPSILON) return false;
559:     float t = time; // >1
560:     time = t + 1;
561: // cross prev,rel; second.prev,second.prev
562:     polygon P;
563:     P.p.resize(4);
564:     P.n = 4;
565:     bool collision = false;
566:     for(int i=0; i<first.p.n; i++)
567:     {
568:         P.p[0] = first.prev.p.p[(i+1)%first.prev.p.n];
569:         P.p[1] = first.prev.p.p[i];
570:         // zadnie storoni ne obrabativaem
571:         Vector2 b = P.p[1]-P.p[0];
572:         if (rel.pseudoscalarProduct(b)>-EPSILON)
573:             continue; // TODO
574:
575:         P.p[2] = rel + first.prev.p.p[i]; // cur -> prev
576:         P.p[3] = rel + first.prev.p.p[(i+1)%first.prev.p.n];
577:         Line base;
578:         base.a = P.p[0];
579:         base.b = P.p[1] - P.p[0];
580:         real total_h = distance(base, P.p[2]);
581: //        if (total_h<-EPSILON) continue;
582:         real minh = total_h + 1;
583:
584:         Vector2 point, cpoint;
585:         for(int j=0; j<second.prev.p.n; j++)
586:         { // Polygon intersection detection O(N*M)
587:             LineSegment Line;
588:             Line.a = second.prev.p.p[j];
589:             Line.b = second.prev.p.p[(j+1)%second.prev.p.n];
590: /*            if ((Line.b - Line.a).pseudoscalarProduct(base.b-base.a)<0)
591:                 continue;
592:  */
593:             // line vs line intersection
594:             for(int k=1; k<4; k++) // P.p TODO k=0 ?
595:             if (intersect(
596:                 Line.a, Line.b - Line.a,
597:                 P.p[k], P.p[(k+1)%P.p.size()] - P.p[k], point))
598:             {
599:                 real h = distance(base, point);
```

```
600:                 if (minh > h && h > EPSILON) // TODO
601:                 {
602:                     minh = h;
603:                     cpoint = point;
604:                 }
605:             }
606:             // polygon vs any point
607:             if (P.isIn(second.prev.p.p[j]))
608:             {
609:                 real h = distance(base, second.prev.p.p[j]);
610:                 if (minh > h
611: //                   && h > EPSILON
612:                 ) // TODO
613:                 {
614:                     minh = h;
615:                     cpoint = second.prev.p.p[j];
616:                 }
617:             }
618:         }
619:         if (minh <= total_h + EPSILON)
620:         {
621:             real ctime = minh / total_h * t;
622:             if (!collision)
623:             {
624:                 time = ctime;
625:                 collision = true;
626:                 Point = cpoint;
627:             }
628:             else
629:                 if (time > ctime)
630:                 {
631:                     time = ctime;
632:                     Point = cpoint;
633:                 }
634:         }
635:     } // for first.p
636:     return collision;
637: }
638:
639: void Solid::push(const Vector2 &location, real power)
640: {
641: // f=ma (Newton 2) TODO
642: }
643:
644: void Solid::normalize()
645: {
646:     Vector2 center(0.0,0.0);
647:     mass = 0.0;
648:     for(int i=0; i<p.n; i++)
649:     {
650:         float c = p.p[i].pseudoscalarProduct(p.p[(i+1)%p.n]);
651:         mass += c;
652:         center += (p.p[i]+p.p[(i+1)%p.n])/3*c;
653:     }
654:     if (fabs(mass) < EPSILON)
655:     {
656:         center.LoadZero();
657:         for(int i=0; i<p.n; i++)
658:             center += p.p[i];
659:         center /= p.n;
660:     }
661:     else
662:     {
663:         center /= mass;
664:     }
665:     if (!movable) mass = 1.0E+7; else mass *= 1000;
666:
667:     cur.position = center;
668:     for(int i=0; i<p.n; i++)
669:         p.p[i] -= center;
670:
671:     prev.p = p;
672:     cur.p = p;
673: }
674:
675: void World::draw()
```

```
676: {
677:     glColor(1,0,0);
678:     for(unsigned int i=0; i<processedCollision.size(); i++)
679:         glPoint(processedCollision[i].point.x, processedCollision[i].point.y,10);
680:
681:     glColor(1,1,1);
682:     for(unsigned int i=0; i<solid.size(); i++)
683:         solid[i].draw(0);
684: }
685:
686: void World::drawdebug()
687: {
688:     videoClearAll();
689:
690:     glColor(1,1,1);
691:     for(unsigned int i=0; i<solid.size(); i++)
692:         solid[i].draw(1);
693:
694:     glColor(0,1,1);
695:     for(unsigned int i=0; i<solid.size(); i++)
696:         solid[i].draw(0);
697:
698:     flush();
699: }
700:
701: void World::save(string filename)
702: {
703:     FILE *f = fopen(("res/" + filename).c_str(),"wt");
704:     fprintf(f,"%d\n", solid.size());
705:     for(int i=0; i<solid.size(); i++)
706:     {
707:         fprintf(f,"%d %d %lf %lf %lf %lf %lf %lf", solid[i].cur.p.n, solid[i].movable,
708:             solid[i].cur.position.x, solid[i].cur.position.y,
709:             solid[i].cur.velocity.x, solid[i].cur.velocity.y,
710:             solid[i].cur.angle,
711:             solid[i].cur.w);
712:         for(int j=0; j<solid[i].p.n; j++)
713:             fprintf(f," %lf %lf",solid[i].p.p[j].x,solid[i].p.p[j].y);
714:         fprintf(f,"\n");
715:     }
716:     fclose(f);
717: }
718:
719:
720: void World::load(string filename)
721: {
722:     solid.clear();
723:     FILE *f = fopen(("res/" + filename).c_str(),"rt");
724:
725:     int k;
726:     fscanf(f,"%d",&k);
727:     while (k-->0)
728:     {
729:         Solid S;
730:         int n,m;
731:         double dx,dy;
732:         fscanf(f,"%d %d %lf %lf %lf %lf %lf %lf",&n,&m,&dx,&dy,
733:             &S.cur.velocity.x,&S.cur.velocity.y,
734:             &S.cur.angle, &S.cur.w);
735:
736:         S.movable = m;
737:         while(n-->0)
738:         {
739:             real x,y;
740:             fscanf(f,"%lf %lf",&x,&y);
741:             S.addp(Vector2(x,y));
742:         }
743:         S.cur.position = Vector2(0.0,0.0);
744:         S.normalize();
745:         S.cur.position += Vector2(dx,dy);
746:         S.updatePerimeter();
747:         addSolid(S);
748:     }
749:     fclose(f);
750: }
751:
```

```
752: //---------------------------------------------------------------------------
753: #pragma package(smart_init)
754:
755:
```

```
 1: /**************************************************************
 2:   TheGame 2D & 3D geometry unit (Steel engine)
 3:   (C) KindeX (Andrey Ivanov), kindex@inbox.lv, http://kindex.lv
 4:   Division, 2004-2005.
 5:
 6:   Vse razmeti v igre v sisteme CI:
 7:   kilogramm, metr, sekunda, m/s, m/ss, dzhoul', Njuton i t.d.
 8:   ili v % (real 0..1)
 9: **************************************************************/
10:
11: #ifndef __types_h
12: #define __types_h
13:
14: #define NDEBUG
15: #include <assert.h>
16: #include <math.h>
17:
18: #include "math/maths.h"
19: #include "_header.h"
20:
21: #define Vector3 VECTOR3D
22: #define  Vector2 VECTOR2D
23:
24: // cos(a+b) = cos(a)*cos(b) - sin(a)*sin(b)
25: // sin(a+b) = sin(a)*cos(b) + cos(a)*sin(b)
26:
27: // otrezok
28: struct LineSegment
29: {
30:     VECTOR2D a,b; // a - koordinata odnogo konca, b - koordinata vtorogo
31:
32:     LineSegment() {}
33:     LineSegment(VECTOR2D const &A, VECTOR2D  const &B): a(A), b(B) {}
34:     LineSegment(LineSegment const &A) { a = A.a; b = A.b; }
35: };
36:
37: //prjamaja
38: struct Line
39: {
40:     VECTOR2D a,b; // a - koordinata to4ki prjamoj, b - napravljajushaja
41:     Line() {}
42:     Line(VECTOR2D  const &A, VECTOR2D  const &B): a(A), b(B) {}
43:     Line(Line const &A) { a = A.a; b = A.b; }
44: };
45:
46: struct DirectedLineSegment
47: {
48:     VECTOR2D s,t; // s - koordinata na4ala konca,
49:     // t - vektor napravlenija
50: };
51:
52: real sqr(real x);
53: // to4ka perese4enija otrezkov (a; a+b) i (c;c+d) v to4ke point
54: //bool cross(vector2 &a, vector2 &b,vector2 &c, vector2 &d, vector2 &point)
55:
56: // to4ka perese4enija otrezkov (a; a+b) i (c;c+d) v to4ke point
57: bool intersect(Vector2  const &a, Vector2  const &b,Vector2  const &c, Vector2  const &d,
     Vector2 &point);  // to4ka perese4enija otrezkov (a; a+b) i (c;c+d) v to4ke point
58: Vector2 mirror(Vector2 &v, Vector2 &a);
59: Vector3 vectmul(Vector3 &a, Vector3 &b);
60: int sgn(real A);
61:
62: struct polygon // Mnogougolnik
63: { // koordinati vershin mnogougolnika
64:     std::vector<Vector2> p;
65:     int n; // p.size()
66:
67:     polygon(): n(0) {}
68:     void add(const Vector2 &P) { p.push_back(P); n++;}
69:     bool isIn(const Vector2 A); // nahoditsja li to4ka vnutri vipuklog poligona?
70:     bool isIn2(const Vector2 A); // nahoditsja li to4ka vnutri 4etirehugolnika s
     samoperese4eniem
71: //    Vector2& operator [] (const int n)        { return p[n]; }
72: };
73:
74: bool intersection_LineSegment_polygon(LineSegment &Line, polygon &P);
```

```
75:
76: // rasstojanie ot to4ki do pramoj Line
77: real distance(Line &line, Vector2 &point);
78: real lineSecment_PointDistance(LineSegment line, Vector2 p);
79:
80: // lezhit li prjamaja po odnu storonu ot poligona
81: // -1 - left, +1 Right, 0 - net razdelenija
82: int oneside(Line line, polygon p);
83:
84: #endif
85:
```

```
 1: /****************************************************************
 2:   TheGame 2D & 3D geometry unit (Steel engine)
 3:   (C) KindeX (Andrey Ivanov), kindex@inbox.lv, http://kindex.lv
 4:   Division, 2004-2005.
 5:
 6:   Vse razmeti v igre v sisteme CI:
 7:   kilogramm, metr, sekunda, m/s, m/ss, dzhoul', Njuton i t.d.
 8:   ili v % (real 0..1)
 9: ****************************************************************/
10: #pragma hdrstop
11: #include "geometry.h"
12:
13: // prinadlezhit li to4ka c otrezku (a,a+b)
14: // esli izvestno 4to oni lezhat na odnoj prjamoj
15: bool intersect(Vector2  const &a, Vector2 const &bb,Vector2 const &c)
16: {
17:     Vector2 b = a + bb;
18:     return
19:       ((a.x<=c.x+EPSILON)&&(c.x<=b.x+EPSILON)
20:       ||
21:       (a.x+EPSILON>=c.x)&&(c.x+EPSILON>=b.x))
22:       &&
23:       ((a.y<=c.y+EPSILON)&&(c.y<=b.y+EPSILON)
24:       ||
25:       (a.y+EPSILON>=c.y)&&(c.y+EPSILON>=b.y));
26: }
27:
28: // to4ka perese4enija otrezkov (a; a+b) i (c;c+d) v to4ke point
29: bool intersect(Vector2  const &a, Vector2  const &b,Vector2  const &c, Vector2  const &d,
     Vector2 &point)  // to4ka perese4enija otrezkov (a; a+b) i (c;c+d) v to4ke point
30: {
31:     real db = d.pseudoscalarProduct(b);
32:     if (db!=0)
33:     {
34:         real s = (a-c).pseudoscalarProduct(b)/db;
35:         if (s<-EPSILON || s>1+EPSILON) return false;
36:         real t = (c-a+d*s).dotProduct(b)/*/b.GetSquaredLength()*/;
37:         if (t<-EPSILON || t>b.GetSquaredLength()+EPSILON) return false;
38:         point = c+d*s;
39:         return true;
40:     }
41:   else // parallelno
42:   {
43:     if (fabs(b.pseudoscalarProduct(c - a)) < EPSILON) // na odnoj linii
44:     {
45:         if (intersect(a,b,c)) { point = c; return true; }
46:         if (intersect(a,b,c+d)) { point = c+d; return true; }
47:         if (intersect(c,d,a)) { point = a; return true; }
48:         if (intersect(c,d,a+b)) { point = a+b; return true; }
49:     }
50:   }
51:   return false;
52: }
53:
54: real sqr(real x) {return x*x;}
55:
56: // otra#ennij vector v ot a
57: Vector2 mirror(Vector2 &v, Vector2 &a)
58: { // V - na6 vektor, A - ot 4ego otra#aem
59:   real d;
60: //  vt:=(V*a/a^2)*;
61:   d=(v.x*a.x+v.y*a.y)/(sqr(a.x)+sqr(a.y));
62:   return Vector2(v.x-2*a.x*d , v.y-2*a.y*d);
63: };
64:
65: //  | A B |
66: //  | C D |
67: real det2(real a, real  b, real c, real d)
68: {
69:   return (a*d - b*c); // Determinant
70: }
71:
72: Vector3 vectmul(Vector3 &a, Vector3 &b)
73: {
74: // | A.Y  A.Z |   | A.Z  A.X |    |  A.X  A.Y  |
75: // | B.Y  B.Z | , | B.Z  B.X | ,  |  B.X  B.Y  |
```

```
 76:    return Vector3(
 77:       det2(a.y, a.z, b.y, b.z),
 78:       det2(a.z, a.x, b.z, b.x),
 79:       det2(a.x, a.y, b.x, b.y)
 80:    );
 81: }
 82:
 83: real abs(real a) { return (a>=0)?a:-a; }
 84: int sgn(real A) { return (A>EPSILON)-(A<-EPSILON);}
 85:
 86: bool intersection_LineSegment_polygon(LineSegment &Line, polygon &P)
 87: {
 88:     if (P.isIn(Line.a) || P.isIn(Line.b)) return true;
 89:     Vector2 point;
 90:     //bool intersect(Vector2 &a, Vector2 &b,Vector2 &c, Vector2 &d, Vector2 &point)
 91:     // to4ka perese4enija otrezkov (a; a+b) i (c;c+d) v to4ke point
 92:     for (unsigned int i=0; i<P.p.size(); i++)
 93:         if (intersect(
 94:             Line.a, Line.b-Line.a,
 95:             P.p[i], P.p[(i+1)%P.p.size()]-P.p[i], point)) return true;
 96:
 97:     return false;
 98: }
 99:
100: bool polygon::isIn(Vector2 A)// ona vnutri vseh uglov
101: {
102:     int all = 0;
103:     for(int i=0; i<n; i += 1)
104:     {
105:         if (p[i] == A ||
106:             fabs((p[(i+1)%n]-p[i]).pseudoscalarProduct(A-p[i]))<EPSILON &&
107:             intersect(p[i],p[(i+1)%n]-p[i], A)) return true;
108:         Vector2 first = p[i] - p[(i+1)%n];
109:         Vector2 second = A - p[i];
110:         int z = sgn(first.pseudoscalarProduct(second));
111:         if (all==0)
112:             all = z;
113:         else
114:             if (all != z) return false;
115:     }
116:     return true;
117: }
118:
119: bool polygon::isIn2(const Vector2 A) // nahoditsja li to4ka vnutri 4etirehugolnika s
      samoperese4eniem
120: {
121:     bool left[4], Aleft[4], all = true;
122:     for(int i=0; i<4; i++)
123:     {
124:         Vector2 base = p[i] - p[(i-1+n)%n];
125:         left[i] = base.pseudoscalarProduct(p[(i+1)%n]-p[(i)%n])>0;
126:         Aleft[i] = (p[(i+1)%n]-p[(i)%n]).pseudoscalarProduct(A-p[i])>0;
127:         all &= left[i] == left[0];
128:         if (p[i] == A ||
129:             fabs((p[(i+1)%n]-p[i]).pseudoscalarProduct(A-p[i]))<EPSILON &&
130:             intersect(p[i],p[(i+1)%n]-p[i], A)) return true;
131:     }
132:     if (all) // vipuklij mnogoug
133:         return isIn(A);
134:     else
135:     {
136:         for(int i=0; i<4; i++)
137:         {
138:             if (left[i] == left[(i+1)%n])
139:             {
140:                 if (left[i] == Aleft[(i+n-1)%n] && left[i] == Aleft[i] &&
141:                 left[i] == Aleft[(i+1)%n])
142:                     return true;
143:
144:             }
145:         }
146:     }
147:     return false;
148: }
149:
150: real distance(Line &line, Vector2 &point)
```

```
151: {
152:     Vector2 diag = point - line.a;
153:     Vector2 LineDirection = line.b.GetNormalized();
154:     real proj = LineDirection.dotProduct(diag);
155:     real s = sqr(diag.GetLength())-sqr(proj);
156:     if (s<0 && s>-EPSILON)
157:       s = 0;
158:     if (s<0)
159:       assert(s >= 0);
160:     real h = sqrt(s);
161:     return h;
162: }
163:
164: real lineSecment_PointDistance(LineSegment line, Vector2 p)
165: {
166: // libo proekcija, libo konec otrezka
167:     Vector2 linedirection = (line.b-line.a).GetNormalized();
168:     Line l;
169:     l.a = line.a;
170:     l.b = linedirection;
171:     real proj = (line.a-p).dotProduct(l.b);
172:     if (intersect(line.a, line.b, l.a + l.b*proj))
173:     // popadaem v otrezok - minimalnie rasstojanie - projekcia
174:         return distance(l, p);
175:     else
176:     {
177:         real d1 = (line.a-p).GetLength();
178:         real d2 = (line.b-p).GetLength();
179:         if (d1<d2) return d1; else return d2;
180:     }
181: }
182:
183: int oneside(Line line, polygon p)
184: {
185:     int res = 0;
186:     for(int i=0; i<p.n; i++)
187:     {
188:         int m = sgn(line.b.pseudoscalarProduct(p.p[i]-line.a));
189:         if (res == 0)
190:             res = m;
191:         else
192:             if (m !=0 && res != m) return 0;
193:     }
194:     return res;
195: }
196:
```

```
  1: /*****************************************************************
  2:   TheGame 2D geometry unit (Steel engine)
  3:   (C) KindeX (Andrey Ivanov), kindex@inbox.lv, http://kindex.lv
  4:   Division, 2004-2005.
  5: ******************************************************************/
  6: //////////////////////////////////////////////////////////////////////////////////////
  7: //  based on VECTOR2D.h
  8: //  Class declaration for a 2d vector
  9: //  You may use this code however you wish, but if you do, please credit me and
 10: //  provide a link to my website in a readme file or similar
 11: //  Downloaded from: www.paulsprojects.net
 12: //  Created:    6th November 2002
 13: //  Modified:   7th January 2003    -   Added QuadraticInterpolate
 14: //////////////////////////////////////////////////////////////////////////////////////
 15:
 16: #ifndef VECTOR2D_H
 17: #define VECTOR2D_H
 18:
 19: #include "../_header.h"
 20:
 21: class VECTOR2D
 22: {
 23: public:
 24:     //constructors
 25:     VECTOR2D(void)  :   x(0.0f), y(0.0f){}
 26:
 27:     VECTOR2D(real newX, real newY): x(newX), y(newY)    {}
 28:
 29: //  VECTOR2D(const real * rhs)       :   x(*rhs), y((*rhs)+1)    {}
 30:
 31:     VECTOR2D(const VECTOR2D & rhs)  :   x(rhs.x), y(rhs.y)  {}
 32:
 33:     ~VECTOR2D() {}  //empty
 34:
 35:     void Set(real newX, real newY){ x=newX; y=newY; }
 36:
 37:     //Accessors kept for compatibility
 38:     void SetX(real newX) {x = newX;}
 39:     void SetY(real newY) {y = newY;}
 40:
 41:     real GetX() const {return x;}   //public accessor functions
 42:     real GetY() const {return y;}   //inline, const
 43:
 44:     void LoadZero(void) { x=0.0; y=0.0; }
 45:     void LoadOne(void);                     //fill with (1, 1)
 46:
 47:     void Normalize(); // set length to 1
 48:     VECTOR2D GetNormalized() const;
 49:
 50:     real GetLength() const
 51:     {   return (real)sqrt((x*x)+(y*y)); }
 52:
 53:     real GetSquaredLength() const
 54:     {   return (x*x)+(y*y); }
 55:
 56:     //linear interpolate
 57:     VECTOR2D lerp(const VECTOR2D & v2, real factor) const
 58:     {   return (*this)*(1.0f-factor) + v2*factor;   }
 59:
 60:     VECTOR2D QuadraticInterpolate(const VECTOR2D & v2, const VECTOR2D & v3, real factor)
      const
 61:     {   return (*this)*(1.0f-factor)*(1.0f-factor) + 2*v2*factor*(1.0f-factor) +
      v3*factor*factor;}
 62:
 63:     //overloaded operators
 64:     //binary operators
 65:     VECTOR2D operator+(const VECTOR2D & rhs) const  {   return VECTOR2D(x + rhs.x, y +
      rhs.y);  }
 66:     VECTOR2D operator-(const VECTOR2D & rhs) const  {   return VECTOR2D(x - rhs.x, y -
      rhs.y);  }
 67:     VECTOR2D operator*(const real rhs) const    {   return VECTOR2D(x*rhs, y*rhs);  }
 68:
 69:     real pseudoscalarProduct(VECTOR2D const &A) { return x*A.y-y*A.x; } // Psevdoskalarnoe
      proizvedenie vektorov
 70:     real dotProduct(const VECTOR2D &A) { return x*A.x+y*A.y; } // skalarnoe proizvedenie
      vektorov
```

```
 71:
 72:     friend bool operator==(const VECTOR2D &L, const VECTOR2D &R) { return (L.x==R.x) &&
       (L.y==R.y); }
 73:
 74:     VECTOR2D operator/(const real rhs) const
 75:     {   return (rhs==0) ? VECTOR2D(0.0f, 0.0f) : VECTOR2D(x / rhs, y / rhs);    }
 76:
 77:     //multiply by a real, eg 3*v
 78:     friend VECTOR2D operator*(real scaleFactor, const VECTOR2D & rhs) {return
       VECTOR2D(rhs.x*scaleFactor, rhs.y*scaleFactor);}
 79:
 80: //  bool operator==(const VECTOR2D & rhs) const;
 81:     bool operator!=(const VECTOR2D & rhs) const {   return !((*this)==rhs); }
 82:
 83:     //self-add etc
 84:     void operator+=(const VECTOR2D & rhs)   {   x+=rhs.x;   y+=rhs.y;}
 85:     void operator-=(const VECTOR2D & rhs)   {   x-=rhs.x;   y-=rhs.y;}
 86:     void operator*=(const real rhs)         {   x*=rhs; y*=rhs; }
 87:     void operator/=(const real rhs)
 88:     {
 89:         if(rhs==0.0f)
 90:             return;
 91:         else
 92:         {   x/=rhs; y/=rhs; }
 93:     }
 94:
 95:     //unary operators
 96:     VECTOR2D operator-(void) const {return VECTOR2D(-x, -y);}
 97:     VECTOR2D operator+(void) const {return *this;}
 98:
 99:     //cast to pointer to a (real *) for glVertex3fv etc
100:     operator real* () const {return (real*) this;}
101:     operator const real* () const {return (const real*) this;}
102:
103:     void rotate(double sinA, double cosA)
104:     {
105:         double xx = x * cosA - y * sinA;
106:         y = y * cosA + x * sinA;
107:         x = xx;
108:     }
109:     void rotate(double A)    { rotate(sin(A),cos(A));    }
110:     VECTOR2D rot90() { return VECTOR2D(-y, x);}
111:     VECTOR2D rot_90() { return VECTOR2D(y, -x);}
112:
113:     //member variables
114:     real x;
115:     real y;
116: };
117:
118: #endif  //VECTOR2D_H
```

```
 1: /***************************************************************
 2:   TheGame 2D geometry unit (Steel engine)
 3:   (C) KindeX (Andrey Ivanov), kindex@inbox.lv, http://kindex.lv
 4:   Division, 2004-2005.
 5: ***************************************************************/
 6: //////////////////////////////////////////////////////////////////////////////////////////
 7: //  based on VECTOR2D.cpp
 8: //  Function definitions for 2d vector class
 9: //  You may use this code however you wish, but if you do, please credit me and
10: //  provide a link to my website in a readme file or similar
11: //  Downloaded from: www.paulsprojects.net
12: //  Created:    8th November 2002
13: //////////////////////////////////////////////////////////////////////////////////////////
14: 
15: #include "Maths.h"
16: 
17: void VECTOR2D::Normalize()
18: {
19:     real length;
20:     real scalefactor;
21:     length=GetLength();
22: 
23:     if(length==1 || length==0)          //return if length is 1 or 0
24:         return;
25: 
26:     scalefactor = 1.0f/length;
27:     x *= scalefactor;
28:     y *= scalefactor;
29: }
30: 
31: VECTOR2D VECTOR2D::GetNormalized() const
32: {
33:     VECTOR2D result(*this);
34: 
35:     result.Normalize();
36: 
37:     return result;
38: }
39: 
40: VECTOR2D operator*(float scaleFactor, const VECTOR2D & rhs)
41: {
42:     return rhs*scaleFactor;
43: }
44: 
```

Kursa darbs izstrādāts

LU Datorikas nodaļā

Autors:
Fizikas un matemātikas . . . .
fakultātes students Andrejs Ivanovs
St. apl. Nr. DatZ020002 2005. g. 17. jūnijā.

Darba vadītājs
LU profesors
Paulis Ķikusts . . . .

Darbs iesniegts Datorikas nodaļā 2005. g. . . . . . . jūnijā.

Pieņēma sekretāre . . . .

Aizstāvēts datorzinātņu kursa pārbaudījumu komisijas sēdē

2005.g. . . . . .ar atzīmi. . . . .

Kursa pārbaudījumu komisija . . . .