SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
Preddiplomski studij računarstva

Završni rad

Implementacija algoritama za detekciju sudara

Rijeka, rujan 2017.

Marin Jurjević
0069068944

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
Preddiplomski studij računarstva

Završni rad

Implementacija algoritama za detekciju sudara

Mentor: izv.prof.dr.sc. Jerko Škifić

Rijeka, rujan 2017.

Marin Jurjević
0069068944

Umjesto ove stranice umetnuti zadatak za završni ili diplomski rad

Izjava o samostalnoj izradi rada

Sukladno članku 9. Pravilnika o završnom radu i završnom ispitu na preddiplomskim sveučilišnim studijima i stručnim studijima Tehničkog fakulteta Sveučilišta u Rijeci, izjavljujem da sam samostalno izradio završni rad na temu: Implementacija algoritama za detekciju sudara prema zadatku: Klasa: 602-04/17-04/14 Ur.br.: 2170-15-11-17-2, zadanom u Rijeci, 20. 3. 2017.

Rijeka, rujan 2017.

_____

Marin Jurjevi?

Zahvala

Zahvaljujem, prije svega, mentoru izv. prof. dr. sc. Jerku Škifiću na odličnom mentoriranju, susretljivosti i detaljnim i jasnim uputama za izradu radu. Nadalje se zahvaljujem kolegi Tomislavu Milanoviću za korisne savjete koji su mi pomogli pri izradi i uređivanju ovoga dokumenta i samoga rada. Na kraju se zahvaljujem i svim kolegama i profesorima koji su mi s razgovorima i diskusijama pomogli pri rješavanju problema na koje sam naišao prilikom izrađivanja rada.

Sadržaj

Popis slika

Popis kodova

Popis psuedokodova

Poglavlje 1

Uvod

Detekcija sudara temelji se u na preklapanju primitiva(trokut, kvadrat, poligon,...)[5], tj. točaka primitiva objekata. U praksi danas postoji cijeli niz algoritama koji omogućuju brzu detekciju sudara između objekata [5]. Oni se mogu podijeliti na 2 dijela:

- Algoritme za podjelu objekata
- Algoritme za podjelu prostora

Postoje i još neki drugi algoritmi, no u ovom radu usredotočiti ćemo se na ova 2 najčešća pristupa. Ovisno o našim potrebama, odabiremo algoritam koji nam najviše odgovara. U ovom radu bit će opisana oba pristupa i prednosti i mane svakoga.

Prilikom sudara, objekti imaju i neku fizikalnu reakciju na sudar. Ovisno o tome, događa se i gubitak energije na objektima. U ovome radu biti će posvećena posebna pažnja u prijenosu energije gibanja prilikom sudara neovisno o tome događa li se sudar od zida, ili od drugog objekta. U početku je zamišljeno da objekti u ovome radu budu kuglice s obzirom na jednostavnu implementaciju i vjerni prikaz detekcije sudara. Na objekte također mora djelovati neka vanjska sila, objekti se ne mogu kretati u prostoru sami od sebe. Odlučeno je da će na cijeloj sceni djelovati gravitacijska sila koja vuče objekte prema "podlozi" od koje se zatim objekti odbijaju. Kako je i već spomenuto, objekti prilikom odbijanja prenose energiju.

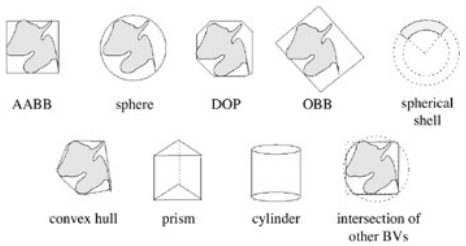
Smisao ovog rada bio je vjerno prikazati i usporediti efikasnost algoritma u odnosu

na primitivnu detekciju sudara gdje se provjerava sudar jednog objekta u odnosu na sve ostale. U konačnosti, kada se efikasnost algoritma svede do zadovoljavajuće razine, objekti dobivaju neka svojstva da animacije dobiju smisao i vizualni izgled (npr. sudari kuglica na biljarskom stolu, kapljice vode, ...).

Poglavlje 2

Ograničavajući volumeni (eng. Bounding volumes)

Ograničavajući volumeni su jednostavni volumeni kojima ograničavamo objekte da bi olakšali i ubrzali detekciju sudara. Ponekad, u praksi, imamo objekte i poligone između kojih je teško detektirati sudar. U tom slučaju, primitivi kojima ograničimo objekt, mogu nam znatno olakšati detektiranje sudara [5]. Umjesto da provjeravamo sudar između objekata, mi provjeravamo sudar između volumena koji ograničavaju objekt. Prednost toga je, kao što je već rečeno, znatno lakše detektiranje sudara, a mana je u tome što nikada ne možemo točno ograničiti objekt pa ćemo i ponekad detektirati sudar koji se nije dogodio.



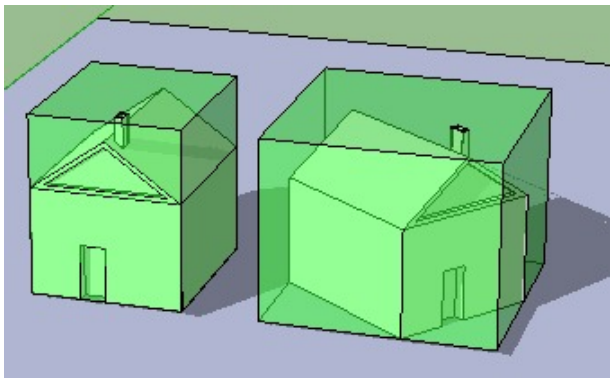
Slika 2.1 Primjeri ograničavajućih volumena

Kako je prikazano na slici 2.1 postoji puno ograničavajućih volumena[5]. U ovom radu biti će objašnjen AABB tj. Axis Aligned Bounding Box. Razlog tomu je AABB

stablo koje se koristilo kasnije kao Bouning Volume Hierarchy, no o tome će biti riječi kasnije.

2.1 AABB - Axis Aligned Bounding Box

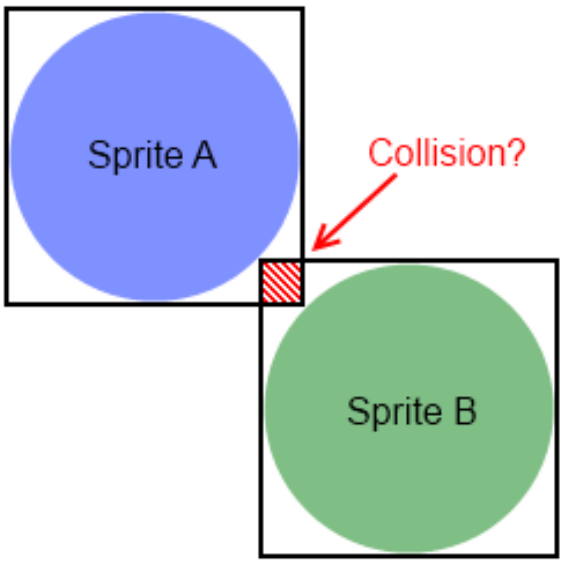
Kao što i samo ime govori, AABB je ograničavajući kvadrat koji je poravnat sa koordinatnim osima. Ovo je i ujedno najjednostavniji, ali i najbrži ograničavajući volumen[5].U 3D geometriji, kvadrat ima 6 stranica, a u 2D geometriji to je pravokutnik (sa 2 strane) [5].



Slika 2.2 Axis Aligned Bounding Box prikazan zelenom bojom

S obzirom da je AABB najjednostavniji oblik volumena, on nije uvijek točan. Ograničeni smo na korištenje pravokutnika pa iz toga slijedi da nećemo moći ograničiti objekt na onaj način na koji bismo željeli tj. postojati će neki "offset". U tom slučaju može se dogoditi da detektiramo sudar između objekata iako se on nije dogodio kao što je prikazano na slici.

AABB u kodu možemo prikazati na nekoliko načina. Postoje 3 različite reprezentacije[5] te ćemo u nastavku proći kroz sve, te pobliže opisati opisati onu koja se koristila u ovome radu.



Slika 2.3 Detektirani sudar između AABB-a koji se nije dogodio između objekata

2.1.1 Reprezentacije AABB-a

Prva i ona najčešća reprezentacija je reprezentacija koja koristi minimalnu i maksimalnu točku objekta. Ukratko, pronađemo minimalnu i maksimalnu točku x,y i z koordinate te od tih točki napravimo kvadrat ako smo u 3D geometriji, tj. pravokutnik ako smo u 2D[5].

```
1 struct AABB{
2         float min[3];
3         float max[3];
4 }
```

Kod 2.1 Struktura AABB-a koji koristi min i max točke

Slika 2.4 AABB min-max reprezentacija u 2D geometriji

Detektiranje sudara za ovu reprezentaciju je trivijalna. Provjerava se je li jedna koordinata minimalne točke jednog AABB-a veća od maksimalne koordinate točke drugog AABB-a ili obrnuto. Ukoliko to nije slučaj u sve 3 dimenzije, događa se sudar[5].

```
1 bool Collision(AABB a, AABB b)
2 {
3         if (a.max[0] < b.min[0] || a.min[0] > b.max[0]) return 0;
4         if (a.max[1] < b.min[1] || a.min[1] > b.max[1]) return 0;
5         if (a.max[2] < b.min[2] || a.min[2] > b.max[2]) return 0;
6         return 1;
7 }
```

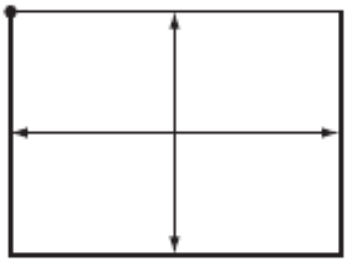
Kod 2.2 Provjeravanje sudara za min-max reprezentaciju AABB-a[5]

Druga reprezentacija je korištenje minimalne točke i dijametra tj. udaljenosti do ostalih vrhova. Određivanje ovakvog volumena malo je kompliciranije nego u prethodnom slučaju upravo zbog dijametra koji moramo odrediti da bi ograničili objekt.

```
1 struct AABB{
2         float min[3];
3         float dx,dy,dz; //udaljenost od minimalne tocke
4 }
```

Kod 2.3 Struktura AABB-a koji koristi min točku i dijametar

Detekcija sudara je za ovu reprezentaciju također malo kompliciranija nego u prethodnom slučaju i najmanje je "privlačna" od sve 3 reprezentacije [5].



Slika 2.5 AABB min-dijametar reprezentacija u 2D geometriji

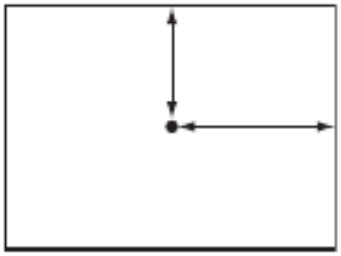
```
1  bool Collision(AABB a, AABB b)
2  {
3          float t;
4          if ((t = a.min[0]- b.min[0]) > b.d[0] || -t > a.d[0]) return 0;
5          if ((t = a.min[1]- b.min[1]) > b.d[1] || -t > a.d[1]) return 0;
6          if ((t = a.min[2]- b.min[2]) > b.d[2] || -t > a.d[2]) return 0;
7          return 1;
8  }
```

Kod 2.4 Provjeravanje sudara za min-width reprezentaciju AABB-a[5]

Posljednja reprezentacija, i ona koju smo koristili u radu, jest centar i radijus. Odredimo središnju točku našeg AABB-a i radijus u svim osima kojima izradimo AABB. Ova reprezentacija je najučinkovitija u terminima uštede memorije[5].

```
1  struct AABB{
2  float center[3];
3  float rad[3]; //radijus
4  }
```

Kod 2.5 Struktura AABB-a koji koristi centar i radijus



Slika 2.6 AABB centar-radijus reprezentacija u 2D geometriji

Detektiranje sudara, za ovu reprezentaciju, je trivijalno. Za svaku dimenziju provjeravamo odnose centara i radijusa u svakoj dimenziji.

```
1 bool Collision(AABB a, AABB b)
2 {
3           if (Abs(a.c[0] - b.c[0]) > (a.r[0] + b.r[0])) return 0;
4           if (Abs(a.c[1] - b.c[1]) > (a.r[1] + b.r[1])) return 0;
5           if (Abs(a.c[2] - b.c[2]) > (a.r[2] + b.r[2])) return 0;
6           return 1;
7 }
```

Kod 2.6 Provjeravanje sudara za centar-radijus reprezentaciju AABB-a[5]

Razlog biranja ove reprezentacije AABB-a leži u tome što smo odabrali kuglice kao objekte između kojih ćemo provjeravati sudar. S obzirom da je sfera, kao objekta, definirana centralnom točkom i radijusom, ova reprezentacija je odabrana kao najpogodnija.

2.2 Kuglice

Kao što je već ranije spomenuto, objekti koji će se koristiti za detektiranje sudara u prostoru su kuglice tj. sfere. Kuglice su odabrane iz razloga što se vrlo jednostavno može odrediti fizikalna reakcija na sudar i što se na njih mogu lijepo primijeniti vanjske sile kao gravitacija i slično. Samim korištenjem kuglica i prijašnjim odabirom AABB-a kao ograničavajućih volumena javljaju se pojedini problemi.

Problem u ovome slučaju je već spomenut. Naime, kako je to i prikazano na slici

2.3, AABB ograniči prostor koji sfera ne pokriva tj. prostor koji je izvan sfere. To se može objasniti vrlo jednostavnom matematikom. Ako prikažemo stranicu kvadrata kao:

$$a = 2r \tag{2.1}$$

gdje nam je r radijus sfere tj. kuglice. Dalje, dijagonala kvadrata iznosi:

$$diag = a\sqrt{2} \tag{2.2}$$

Može se zaključiti da je udaljenost od sjecišta dijagonala tj. centra kvadrata veća od radijusa sfere za $\sqrt{2}$. Što je veća sfera to je pogreška u ograničavajućem volumenu također veća. Zbog toga će se sigurno događati da ćemo otkrivati sudare i onda, kada se oni ne događaju, a pri vrlo velikim sferama to može stvarati puno nelogičnosti i čudnih fizikalnih reakcija na sudar.Kada su sfere vrlo malog radijusa, ta pogreška se u principu i ne primjeti, ali je ipak prisutna.

Ovome problemu uskače se na vrlo dobar način. Kako je prikazano na slici 2.1, sfera sama po sebi može biti ograničavajući volumen [5]. To proizlazi iz činjenice da se na vrlo jednostavan i elegantan način može detektirati sudar između 2 sfere.

```
1 bool SphereCollision(Sphere a, Sphere b)
2 {
3 // Calculate squared distance between centers
4 Vector d = a.c - b.c;
5 float dist2 = Dot(d, d);
6 // Spheres intersect if squared distance is less than squared sum of
      radii
7 float radiusSum = a.r + b.r;
8 return dist2 <= radiusSum * radiusSum;
9 }
```

Kod 2.7 Detekcija sudara između 2 sfere[5]

U stvarnom kodu, implementacija ove funkcije izgleda drugačije nego što je opisana ovdje i u literaturi[5], ali u principu algoritam i pristup su jednaki.

2.2.1 Implementacija klase za kuglice

Implementacija klase za kuglice u principu je jednostavna. Prikazana je sljedećim kodom:

```
1  public:
2          Vector3D vecDir;
3          AABB_box bBox;
4
5          Ball() = default;
6          Ball(const Point center, const float r, const float vecX, const
               float vecY,
7                  const float vecZ, const float m, const uint i);
8          Ball(const float x, const float y, const float z, const float r
               ,
9                  const float vecX, const float vecY, const float vecZ,
                       const float m,
10                 const uint i);
11         void drawSphere();;
12         void updatePosition(const float dt);
13         bool collision(Ball &collider);
14         template <typename T> bool collision(T &collider) {
15                 return isOverlap(this->bBox, collider.bBox);
16         }
17         void resolveCollision(Ball &collider);
18         void resolveCollision(Wall &wall);
19         Point &getCenter() { return this->center; }
20         const float &x() const { return this->center.x; }
21         const float &y() const { return this->center.y; }
22         const float &z() const { return this->center.z; }
23         const float &rad() const { return this->r; }
24         const float &getMass() const { return this->mass; }
25         const uint &getI() const { return this->i; }
26 private:
27         Point center;
28         float r;
29         float mass;
30         uint i;
31 };
```

Kod 2.8 Implementacija klase za kuglice

Da se primijetiti da se u samoj deklaraciji klase uglavnom nalaze konstruktori i

get metode za dobivanje privatnih varijabli. *Get* i *template* metode su definirane u samoj deklaraciji klase zbog svoje jednostavnosti i veće brzine pristupanja.

U privatnim varijablama klase definirali smo najvažnije informacije o kuglici, a to su:

- *Center* - točka centra za svaku kuglicu
- r - radijus sfere
- *mass* - masa sfere koja se kasnije koristi za količinu gibanja
- i - pozicija kuglice u listi kojom se osigurava da kuglica ne provjerava sudar sa samom sobom

U javnim varijablama odredili smo:

- *bBox* - odgovarajući AABB koji je opisan prethodno
  U ovom slučaju *bBox* dobiva vrijednosti radijusa i centra od sfere kroz konstruktor
- *vecDir* - vektor definiran u klasi *Vector3D* koja sadrži osnovne operacije nad vektorima (skalarni produkt, oduzimanje, normaliziranje i sl.)
  Ovaj objekt nam omogućava kretanje kuglice i olakšava izračunavanje sudara od zida i druge kuglice. Implementacija same klase biti će pokazana u kasnijim poglavljima.

Metode koje su definirane, u konačnosti govore same za sebe:

- *collision* - detektira sudar kuglice ovisno o tipu s kojim se sudara.
  Ukoliko se događa sudar sa drugom kuglicom, sudar se detektira na ranije opisani način. Sudar sa zidom i bilo kojim drugim objektom se detektira preko pridodanih AABB-a.
- *resolveCollision* - u metodama ovog imena definiramo fizikalnu reakciju kuglice na sudar. Ovisno o objektu s kojim se kuglica sudara, tako se i poziva određena *resolveCollision* funkcija
- *drawSphere* - funkcija za crtanje kuglica
  Kuglice u prvom trenutku crtamo sa *glutSolidSphere* i pomoću *glTranslatef* ih

pomičemo po prostoru. Kasnije, pri korištenju Opengl 3.3 kuglice će se crtati na drugačiji način.

- *updatePosition* - metoda kojom definiramo kretanje kuglica po prostoru uz pridodanu gravitaciju.
  Detaljnija implementacija će biti kasnije objašnjena

U glavnom programu kuglice smo spremali u *std::vector*. Na taj način smo jednostavno iterativnim prolaženjem kroz listu iscrtavali kuglice, detektirali sudar između kuglica i zidova i u konačnosti pomicali kuglice po prostoru.

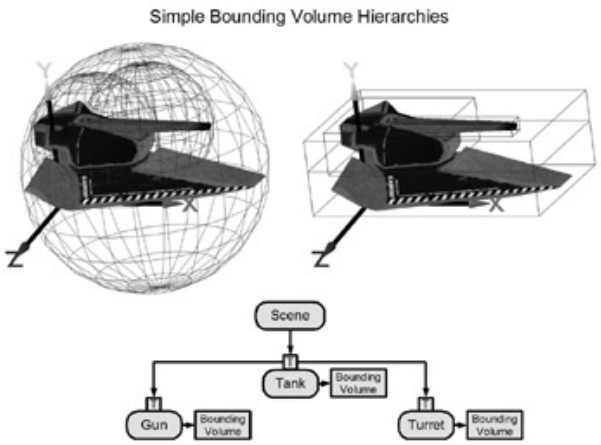
Poglavlje 3

Hijerarhija ograničavajućih volumena (eng. Bounding volume hierarchy - BVH)

S obzirom da su objekti u praksi vrlo kompliciran te sastavljeni od puno različitih primitiva, potrebna nam je struktura podataka koja će na vjeran način ograničiti naš objekt. Ukoliko objekt ograničimo manjim volumenima i te volumene posložimo u stablo, dobili smo određenu hijerarhiju volumena ili BVH. Stablo prikazujemo kao:

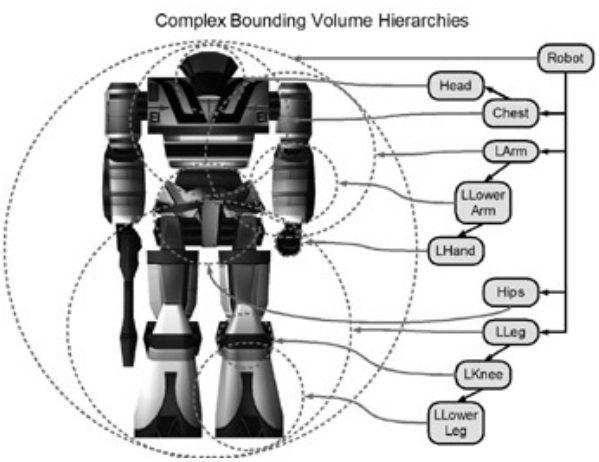
- Root element stabla ograniči cijeli objekt
- Njegova djeca ograniče lijevu i desnu polovicu objekta
- Djeca od ta 2 elementa podijele taj volumen na 2 dijela itd.
- Listovi sadrže ograničavajući volumen najmanjeg primitiva ili sami primitiv

To prikazujemo ovako:



Slika 3.1 Primjer BVH[1]

Konceptualno, objekt bi prikazujemo na takav način. Ograničavajući volumen koji je roditelj u stablu, ograničuje druga 2 volumena i tako sve do listova koji su primitivi. Nešto kompleksnija struktura bila prikazuje se ovako: BVH možemo



Slika 3.2 Prikaz kompliciranije BVH [1]

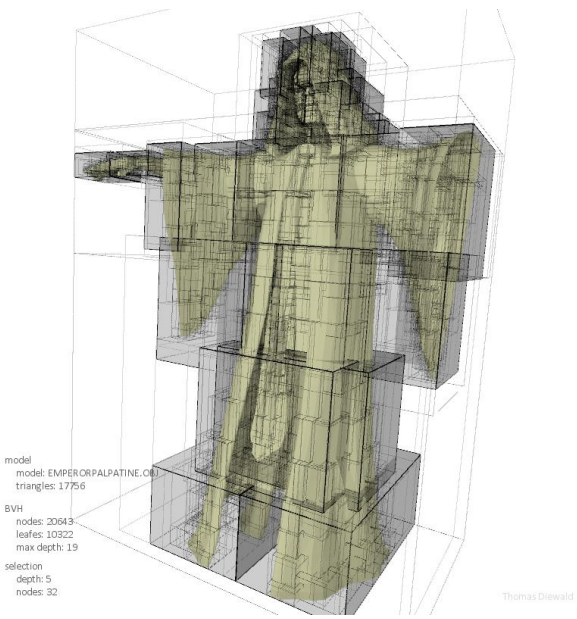
izgraditi od bilo kojih ograničavajućih volumena. U konačnici izgradnja samog stabla je uvijek proporcionalna broju elemenata od kojih je objekt sastavljen (složenost O(n))[5]. Ono što je u ovom slučaju najbitnije je to da stablo ne moramo graditi

u svakom frameu što prilično ubrzava posao. Također, važno je da stablo bude balansirano. Ukoliko je stablo dobro balansirano, objekt koji ograničavamo će biti točnije ograničen, nego što bi to bilo u suprotnom slučaju. Za detektiranje sudara moramo samo provjeriti sudaraju li se u početku root elementi stabala. Ukoliko ne, odmah možemo prekinuti daljnju pretragu sudara, no sam algoritam i implementacija biti će prikazana kasnije.

Za potrebe ovog objekta odabrali smo AABB stablo zbog njegove efikasnosti i jednostavnosti prilikom implementacije. Kako je već i spomenuto, sudar između 2 AABB-a vrlo je jednostavno detektirati i to je glavna prednost ovoga stabla.

3.1 AABB stablo

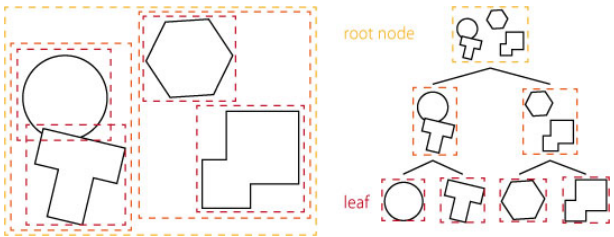
AABB stablo možemo prikazati kao: Stablo sa slike 3.3 izgleda kompliciranо. Ali,



Slika 3.3 Prikaz kompliciranog AABB stabla [2]

već i iz kompliciranog objekta možemo vidjeti da se događaju pogreške. Te pogreške s jedno strane ipak nisu toliko bitne jer kada dođemo do lista u stablu, nećemo

detektirati sudar. Jednostavniji primjer AABB stabla izgledao bi ovako:

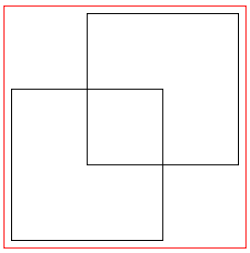


Slika 3.4 Prikaz jednostavnog AABB stabla

Ovdje smo cijelu scenu podijelili AABB stablom, ali pristup je isti dijelimo li cijelu scenu ili jedan objekt. Scenu ćemo podijeliti ukoliko želimo provjeriti sudar zrake sa određenim objektom, a objekte ćemo podijeliti ukoliko želimo provjeriti sudar između samih objekata [5].

3.1.1 Ograničavanje 2 AABB minimalnim volumenom

Prije nego izgradnje AABB stabla, potrebno je pronaći način kako 2 AABB ograničiti jednim većim volumenom. Uvjet je da je taj ograničavajući volumen minimalan za 2 AABB čiji ograničavajući volumen tražimo.



Slika 3.5 Minimalni ograničavajući volumen označen crvenom bojom

Kako je to spomenuto ranije u poglavlju 2.1.1, koristili smo reprezentaciju AABB-a sa točkom centra i radijusom. Pronalaženje ograničavajućeg volumena za ovakvu reprezentaciju AABB-a malo je zahtjevnije nego u slučaju sa min-max reprezentacijom.

Pseudokod 1 Algoritam za izračunavanje minimalnog ograničavajućeg volumena za 2 AABB-a

function GETFATTENAABB($AABB\ a$, $AABB\ b$)
  $AABBFattenAABB$
  $FattenAABB.r = \frac{distance(a.center,b.center)}{2} + Max(a.r, b.r)$
  $FattenAABB.center = median(a.center, b.center)$
  return $FattenAABB$
end function

Jednostavnom matematikom izračunamo udaljenost između 2 centra i pribrojimo joj veći od 2 radijusa (u našem slučaju radijusi su jednaki, ali zbog svih slučajeva stavljen je odabir maksimuma) da dobijemo radijus za naš AABB. Centar je jednostavno određen srednjom točkom između 2 centra od ulaznih AABB-a. Veći radijus odabran je iz razloga da s manjim nećemo moći ograničiti oba AABB-a i to zapravo neće biti ograničavajući volumen za 2 željena AABB-a.

3.1.2 Izgradnja AABB stabla

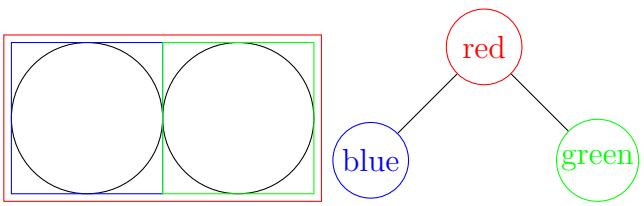
Za izgradnju AABB stabla postoje 3 načina[7]:

- *Top down* konstrukcija
- *Bottom up* konstrukcija
- Dodavanje elemenata u stablo

Za sva 3 načina, potrebno je odabrati strategiju koja će napraviti što manje stablo. Ukoliko nam je stablo veliko pretraga elemenata u kojima se događa sudar će duže trajati. Zbog toga, potrebno je odabrati strategiju koja će stvoriti dobro balansirano ili približno dobro balansirano stablo[5]. Ukoliko i ne stvorimo dobru strategiju za izgradnju stabla, uvijek možemo balansirati stablo "ručno" tj. putem funkcije. Ovo nikako nije poželjan slučaj. Time se gubi dodatno vrijeme na procesoru i funkcija za balansiranje je sama po sebi kompliciranа i zahtjeva O(n) operacija. Za *Top down* i *Bottom up* konstrukciju potrebno je odabrati ravninu presijecanja (eng. Partitioning plane). Izgradnja ovoga stabla je u principu kao izgradnja svakog drugog stabla, samo

je odabir ravnine presijecanja posao koji zahtjeva veliku količinu znanja i testiranja da u konačnici naše stablo bude minimalno. Ravnina presijecanja, kao što joj i samo ime govori, presjeći će listu naših objekata i pripadajućih AABB-a na onaj način kojim ćemo dobiti što manje stablo.Najjednostavniji model ravnine presijecanja je taj da se u listi elemenata odabere median element i da s obzirom na njegov AABB, izrađuje lijeva i desna grana stabla. Kao što je već rečeno, root element stabla će biti cijeli objekt te je potrebno samo odabrati pripadajuću točko centra i radijus (na ranije spomenuti način opisan u poglavlju 3.1.1 samo sa listom elemenata).

U ovom radu, odabrali smo treću strategiju za izgradnju stabla. Ubacivanje elemenata u prvom trenutku činilo se kao najlogičniji izbor jer je glavna ideja bila da se stablo elemenata izgrada prilikom sudara kuglica.



Slika 3.6 Prikaz sudara 2 kuglice i pripadajućeg AABB stabla

Prema slici 3.6 stablo bi izgradili vrlo jednostavno. Nakon detekcije sudara, stvorili bi novi element, koji bi bio root element stabla (na slici 3.6 crvenom bojom). Njegov lijevi element pokazivao bi na lijevi AABB (plavom bojom), a desno dijete bi pokazivalo na desni AABB (zelenom bojom).

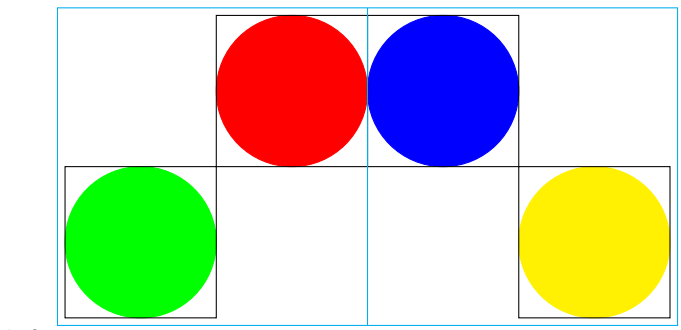
Iako je primjer dosta trivijalan, još uvijek ne znamo kako nam klasa za AABB stablo izgleda, stoga još uvijek je sve prilično apstraktno. Klasa će sadržavati pokazivač na lijevi i desni element stabla. To smo riješili pomoću "pametnih" pokazivača tj. u C++-u, `unique_ptr`. Ovakav tip pokazivača nas rasterećuje brige o memoriji i njezinom brisanju. Osim toga, sadržimo "običan" pokazivač na element roditelja. Važan podatak koji nam također treba je i dubina stabla. Osim svega navedenog, ovakav tip stabla gradili smo u konstrukturu. Nema smisla stvarati AABB stablo od jedne kuglice, stoga je ovo bio dobar način da se olakša sama implementacija algoritma za dodavanje dodatnih elemenata. Jednom kada bi izgradili stablo od 2

elementa, mogli smo dodati u njega bilo koji drugi element ubacivanjem na najbolju poziciju, tj. na poziciju gdje ćemo imati minimalno stablo.

```
1  class Node {
2          typedef std::unique_ptr<Node> ptr;
3          public:
4                  Node() = default;
5                  Node(const AABB_box bbox);
6                  Node(ptr &left, ptr &right);
7          private:
8                  AABB_box box;
9                  ptr left;
10                 ptr right;
11                 Node *parent;
12                 int height;
13 };
```

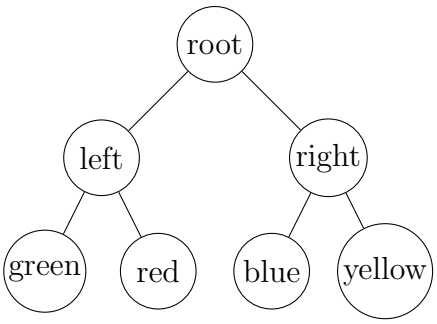
Kod 3.1 Implementacija klase za AABB stablo

Razmotrimo sljedeću situaciju:



Slika 3.7 Stablo od 4 kuglice

Prvo se sudare plava i crvena kuglica te se na taj način preko konstruktora izradi stablo sa 2 elementa. Nakon toga prvo udara žuta, a nakon toga zelena kuglica. Stablo koje ćemo dobiti izgledati će:



Slika 3.8 Prikaz AABB stabla za sliku 3.7

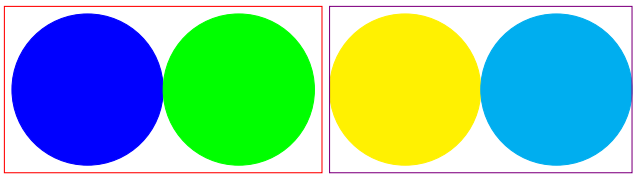
Nakon udara svake kuglice, ubacujemo novi AABB u naše stablo. Za to je korišten vrlo jednostavan algoritam. Primjećujemo da svaki element koji ubacujemo u stablo će biti list, stoga možemo slobodno reći da ova funkcija realizira ubacivanje lista u stablo[8]. Prije samog algoritma, treba naglasiti da napisani algoritam možda nije najtočniji i možda ne radi za sve slučajeve. Algoritam je u svakom slučaju nedovoljno testiran, ali za ovako mali primjer poslužio nam je da vrlo dobro prikažemo strukturu AABB stabla. Algoritam radi na jednostavan način. Iteriramo po kreiranom stablu i izračunavamo površinu AABB-a od elementa koji želimo ubaciti i lijevog i desnog djeteta. Ukoliko je površina AABB-a od lijevog djeteta i elementa manja od površine AABB-a i desnog djeteta, iteriramo u lijevu stranu stabla. Ako je obrnuto, iteriramo na desnu stranu i tako sve dok ne dođemo do posljednjeg elementa stabla[8]. S obzirom jesmo li išli lijevo ili desno, tu ubacujemo naš novokreirani element u stablo. U zadnjem koraku, vraćamo se od lista prema root elementu i radimo "update" AABB-a nad svakim elementom stabla.

Pseudokod 2 Algoritam za ubacivanje lista u AABB stablo

```
function insertLeaf(AABBtree node, AABB box)
    while node is not leaf do
        if Surface(node- > left- > box, box) > Surface(node- > right- > box, box) then
            isLeft = false
            node = node- > right
        else
            isLeft = true
            node = node- > left
        end if
    end while
    if isLeftistrue then
        NewNode(box)
        FatAABB(node- > parent- > left, box)
        NewNode- > parent = node- > parent
        node- > parent- > left = NewNode
    else
        NewNode(box)
        FatAABB(node- > parent- > right, box)
        NewNode- > parent = node- > parent
        node- > parent- > right = NewNode
    end if
    Update tree until root with new leaf
end function
```

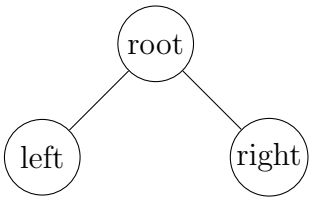
3.1.3 Pronalaženje sudara između dva stabla

Nakon što smo izgradili stabla detektirati sudar između više stabala trivijalna je stvar. S obzirom da su stabla sagrađena od AABB-a potrebno je provjeriti samo *root* elemente, i ako se oni sudaraju idemo u daljnju elemenata stabala. Ukoliko sudara između *root* elemenata nije detektiran, sudar se nije ni dogodio. Ovakav način zahtjeva O(logN) operacija [5]. Za jednostavan primjer možemo promotriti situaciju sa 2 jednostavna stabla koja imaju po 2 elementa.



Slika 3.9 Sudar između 2 jednostavna stabla

Za jedno i drugo stablo vrijedi struktura: U nekom trenutku, mi želimo točno



Slika 3.10 Struktura stabla za sliku 3.9

znati da se dogodio sudar između zelene i žute kuglice (kuglice su prikazane drugom bojom samo radi razumjevanja). Na već opisani način detektiramo na kojem mjestu će se dogoditi sudar. Kada otkrijemo u kojem se listu dogodio sudar, taj sudar ćemo na neki određeni način riješiti i napraviti update na cijelom stablu. U našem slučaju kuglice su se odbile bez gubitka energije.

Pseudokod 3 Algoritam za detektciju sudara između 2 AABB stabla

```
function searchCollision(AABBtree a, AABBtree b)
    if isCollision(a, b) is false then return
    end if
    while a is not leaf do
        if isCollision(a->left, b) then
            a = a->left
        else
            a = a->right
        end if
    end while
    while b is not leaf do
        if isCollision(b->left, a) then
            b = b->left
        else
            b = b->right
        end if
    end while
    ResolveCollision(a, b)
end function
```

Kao i u algoritmu 2, ovaj algoritam također nije istestiran do kraja i vjerovatno u sebi ima neke pogreške, no za ovako jednostavne primjere će raditi. Ovime smo zapravo pokazali da nakon potpunog razumijevanja AABB stabla i strukture možemo vrlo jednostavno napisati kod za izgradnju stabla i detektiranje sudara između 2 ili više stabala.

3.2 Prednosti i mane BVH-a

Glavna prednost BVH-a je brzina izgradnje stabala. Ovakvim jednostavnim algoritmima možemo izgraditi stablo i u njemu vrlo brzo provjeriti sudare. Ukoliko imamo puno manjih objekata koji se kreću po sceni, pretraga sudara je vrlo brza. Stablo ne moramo raditi u svakoj sekundi, nego ga je dovoljno izgraditi na početku i zatim samo tokom vremena raditi "update". Nažalost, u prvoj fazi izrade projekta, nije se smislila adekvatna strategija za detektiranje sudara pa je i samo korištenja BVH-a palo u vodu.

Mana BVH-a je što se vrlo dobro mora razumjeti struktura s kojom se barata. Za razliku od algoritama koji particioniraju prostor i koji su intuitivniji za shvatiti, BVH zahtjeva od nas da dobro razumijemo algoritam i da na pametan način odaberemo os po kojoj ćemo podijeliti objekt ili scenu.

Zaključno, BVH je struktura podataka kojom ćemo na vrlo jednostavan i jeftin način detektirati sudare na sceni ili među objektima. Uz malo više uloženog vremena, možemo napisati vrlo dobar algoritam koji će vrlo brzo detektirati sve sudare na sceni. Kako je i rečeno, od ovoga se odustalo. S vremenom nakon što je i sama struktura postala jasnija, donijele su se neke druge odluke vezane za sam rad i više nije imalo smisla koristiti i dalje implementirati BVH.

Poglavlje 4

Sudari, fizika i vanjske sile

U ovom poglavlju, prije implementacije algoritma za podjelu prostora, prvo je potrebno opisati kakve će se uopće reakcije događati među kuglicama prilikom sudara. Potrebno je također i ograničiti našu scenu svojevrsnim zidovima. Vanjska sila je također važan dio ovoga projekta. Bilo je potrebno dodati gravitaciju na cijeloj sceni da se dobije dojam padanja kuglica i odbijanja od podloge.

Potrebno je realizirati kretanje kuglica po sceni. Kretanja su definirana u pomoćnoj klasi *Vector3D*. Pomoću ove klase također su definirane normale zidova. Ovakav pristup znatno je olakšao samo rješavanje sudara jer se izbjeglo korištenje kompliciranih trigonometrijskih jednadžbi i funkcija.

4.1 Implementacija klase Vector 3D

```
1  class Vector3D {
2  public:
3          Vector3D() = default;
4          Vector3D(float x, float y, float z) {
5                  this->vec.emplace_back(x);
6                  this->vec.emplace_back(y);
7                  this->vec.emplace_back(z);
8          }
9          Vector3D(Point point) {
10                 this->vec.emplace_back(point.x);
11                 this->vec.emplace_back(point.y);
12                 this->vec.emplace_back(point.z);
13         }
14         Vector3D operator*(float const &scalar) {
15                 return Vector3D(this->x() * scalar, this->y() * scalar,
                       this->z() * scalar);
16         }
17         Vector3D operator-(Vector3D &rhs) {
18                 return Vector3D(this->x() - rhs.x(), this->y() - rhs.y
                       (),
19                 this->z() - rhs.z());
20         }
21         Vector3D normal() {
22                 float mag = std::sqrt(std::pow(this->x(), 2) + std::pow
                       (this->y(), 2) +
23                 std::pow(this->z(), 2));
24                 if (mag == 0) {
25                         return Vector3D(1, 0, 0);
26                 }
27                 return Vector3D(this->x() / mag, this->y() / mag, this
                       ->z() / mag);
28         }
29         const float &x() const { return vec[0]; }
30         const float &y() const { return vec[1]; }
31         const float &z() const { return vec[2]; }
32         void setX(const float &x) { vec[0] = x;
33         void setY(const float &y) { vec[1] = y; }
34         void setZ(const float &z) { vec[2] = z; }
35
36 private:
37         std::vector<float> vec;
```

```
38 };
39
40 inline float operator*(const Vector3D &lhs, const Vector3D &rhs) {
41         return lhs.x() * rhs.x() + lhs.y() * rhs.y() + lhs.z() * rhs.z
               ();
42 }
```

Kod 4.1 Implementacija klase Vector3D

Ova klasa kako sadrži implementaciju najvažnijih vektorskih operacija:

- Operator * će nam izračunati skalarni produkt između 2 vektora
- Funkcija *normalize* će normalizirati vektor

Naravno, za same vektore postoji još niz operacija, no ovdje su nam za sada bile potrebne samo ove dvije. Postavljene su i *Set* metode za postavljanje vrijednosti u koordinate vektora. Ova klasa je *headeronly* i napisana je prema literaturi [9].

4.2 Kretanje kuglica

U ovom kratkom paragrafu objasniti ćemo kako se realiziralo kretanje kuglica po sceni. Kako je to objašnjeno u poglavlju 2.2, za kretanje smo koristili funkciju pod imenom *updatePosition* koja je za argument primala broj *dt* što u prijevodu znači *deltatime* ili razlika u vremenu između 2 framea. Brzinu kretanja kuglice definirali smo kao:

- Brzina u X smjeru
- Brzina u Y smjeru
- Brzina u Z smjeru

Informacije o brzinama bile su spremljene u strukturi podataka *Vector3D*, koju smo opisali u prethodnom paragrafu 4.1. Kretanje kuglica definira se prema klasičnoj formuli iz mehanike:

$$v = \frac{s}{t} \tag{4.1}$$

gdje je:

- v - brzina
- s - prijeđeni put
- t - vrijeme

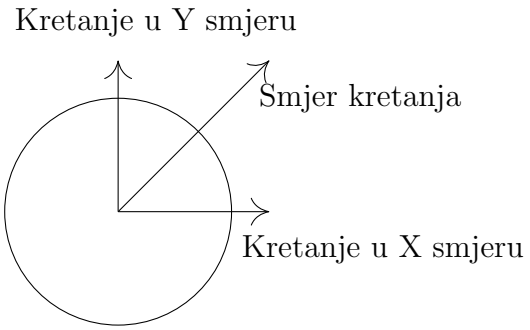
S obzirom da nama ne treba izračun brzine, nego nam je potreban put koji će kuglica prijeći u svakom trenutku, malom izmjenom i prilagodbom jednadžbe dobije se:

$$\begin{aligned} s_x &= vector.x\ dt \\ s_y &= vector.y\ dt \\ s_z &= vector.z\ dt \end{aligned} \tag{4.2}$$

gdje nam je:

- $vector.(x, y, z)$ - brzina u danoj koordinati
- $s(x, y, z)$ - prijeđeni put u danoj koordinati
- dt - razlika vremena između 2 framea

Na ovakav jednostavan način dobili smo kretanje kuglice u bilo kojem smjeru, za sad bez ikakve vanjske sile tj. gravitacije.



Slika 4.1 Kretanje kuglica realizirano jednostavnim zbrajanjem vektora

Sada kada nam je jasno definirano kretanje kuglica, potrebno je definirati kako će se kuglice ponašati prilikom sudara. S obzirom da su nam sve informacije spremljene u vektoru, to ne bi smjelo predstavljati problem. Korištenjem jednostavnih formula iz mehanike može se vrlo jednostavno dobiti reakcija kuglice na sudar sa drugom kuglicom ili zidom.

4.3 Sudari kuglica

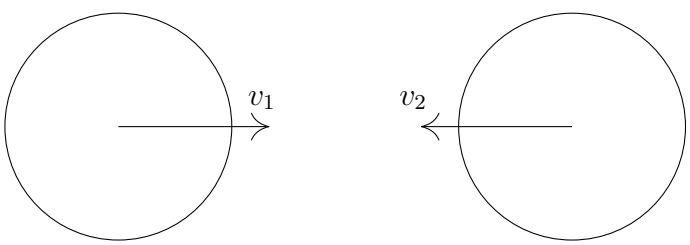
Sudari su općenito mogu podijeliti u 2 skupine:

- Neelastične sudare
- Elastične sudare

Neelastični sudari su oni sudari kod kojih se energija sustava gubi zbog različitih razloga (deformacija tijela u sudaru, toplina i sl.). Budući da u realnom sudaru najčešće dolazi djelomičnog gubitka energije, ima smisla ukazati i na tu problematiku[10]. Analiza takvog sudara i dalje se temelji na zakonima očuvanja. Kod elastičnih sudara u zatvorenom sustavu nalaze se sva tijela masa m_1 i m_2 koja se gibaju brzinama v_1 i v_2 prije sudara, odnosno brzinama v_1' i v_2' poslije sudara[10].

4.3.1 Sudari bez gubitka energije

U prvom i najosnovnijem slučaju, razmotrili smo situaciju gdje nema nikakvog gubitka energije i gdje izlazne brzine nakon sudara ostaju jednake. Prva situacija je bila sljedeća.

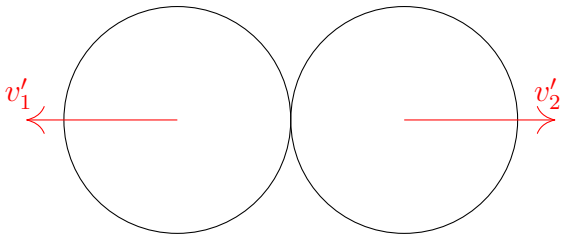


Slika 4.2 Primjer sudara kuglica

Kuglice se kreću jedna prema drugoj, jedan se kreće brzinom v_1, a druga brzinom v_2. U trenutku sudara, kuglice će se odbiti jedna od druge i nastaviti se kretati brzinom v_1' tj. v_2'.

Nakon sudara vrijediti će sljedeće:

- $v_1' = -v_1$

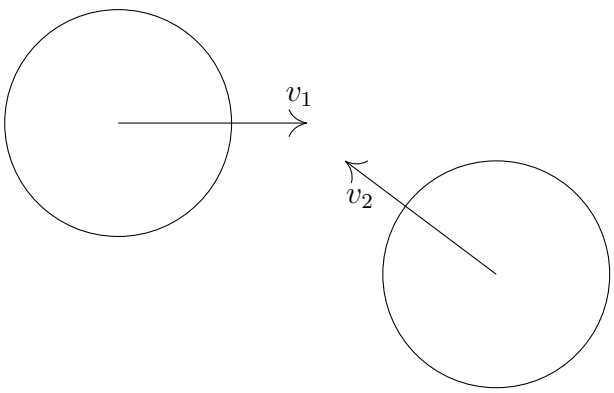


Slika 4.3 Smjer gibanja kuglica nakon sudara

- $v_2' = -v_2$

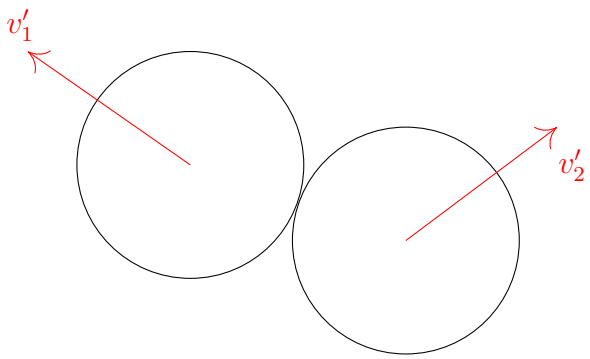
Ovakav slučaj je trivijalan i nije problem odrediti smjer kretanja kuglica nakon sudara. Kretanje kuglica je samo u jednoj dimenziji te ukoliko pomnožimo brzinu kretanja kuglice sa (-1) dobiti ćemo rezultatnu brzinu nakon sudara.

Sljedeća situacija je kompliciranija. Kretanje jedne kuglice se događa u 2 dimenzije i to sada dodatno komplicira stvari. Situacija izgleda ovako: Primjetimo kako se



Slika 4.4 Primjer sudara kuglica u 2 dimenzije

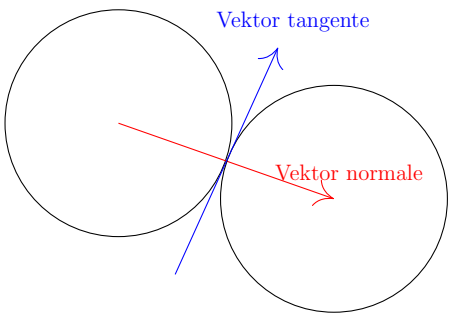
druga kuglica, koja se giba brzinom v_2, giba u 2 dimenzije (ima komponentu brzine u X smjeru i komponentu brzine u Y smjeru). Pretpostavimo da su mase kuglica jednake i da se pri njihovom sudare neće izgubiti nikakva energija. Rezultirajući vektori brzina više neće biti samo invertirani. Kuglica koja se giba samo po X osi (brzinom v_1) nakon sudara dobiti će i neku svoju Y komponentu brzine. Pretpostavimo da su i brzine kuglica u X smjeru jednake, pa će rezultat sudara približno jednak kao na slici



Slika 4.5 Smjer gibanja kuglica nakon sudara u 2 dimenzije

4.2. Ovakav rezultat sudara možemo prikazati u nekoliko matematičkih koraka[9].

Prvo pronađemo vektor normale. To je vektor čije su komponente razlika između koordinata centara kuglica. Sukladno tome odmah pronađemo vektor tangente te oba vektora pomoću klase *Vector3D* normaliziramo (kako je već i spomenuto pomoću ove klase puno nam je lakše pronaći vektor normale i tangente)[9].



Slika 4.6 Vektor normale i tangente

Nakon toga, inicijalne brzine v_1 i v_2 pomnožimo sa jediničnim vektorima normale i tangente[9]. Pomnožiti u ovome slučaju znači izračunati skalarni produkt između 2 vektora.

$$\begin{aligned} v_{1,2}'n &= un\ v_{1,2} \\ v_{1,2}'t &= ut\ v_{1,2} \end{aligned} \tag{4.3}$$

gdje nam je:

- un - jedinični vektor vektora normale
- ut - jedinični vektor vektora tangente
- $v_{1,2}$ - vektori brzine kuglica
- $v'n$ - rezultirajuća brzina u smjeru normale
- $v't$ - rezultirajuća brzina u smjeru tangente

Tangencijalne brzine nakon sudara ostaju jednake što znači da smo u smjeru tangente izračunali brzinu[9]. Nakon što smo odradili sve ove korake, pomoću jednostavne formule izračunamo rezultatne brzine za obje kuglice prema formuli:

$$\begin{aligned} v_1' &= v_2'n\ un + v_1't\ ut \\ v_2' &= v_1'n\ un + v'_2t\ ut \end{aligned} \tag{4.4}$$

S ovim jednostavnim matematičkim koracima dobiti ćemo točno odbijanje kuglica nakon sudara kako je i prikazano na slici 4.5. Ovo se može opisati i jednostavnim algoritmom koji je prikazan na idućoj strani.

Pseudokod 4 Algoritam za izračunavanje smjera brzina nakon sudara između 2 kuglice

function RESOLVECOLLISON(*Ball a*, *Ball b*)
  if Collision(a, b) is false **then return**
  end if
  $normal = (a.center - b.center)$
  $un = unitVector(normal)$
  $ut(-un.y, un.x)$
  $v_1n = a.v\ un$
  $v_1t = a.v\ ut$
  $v_2n = b.v\ un$
  $v_2t = b.v\ ut$
  $a.v = v_1n\ un + v_1t\ ut$
  $b.v = v_2\ un + v_2t\ ut$
end function

Nakon pokretanja animacije primjećuje se jedan problem. Kuglice se slobodno gibaju po sceni i te u vrlo kratkom periodu izađu iz našeg viewporta. Našu scenu sada moramo ograničiti i to sa 4 strane s obzirom da još nismo implementirali gravitaciju. Za to će nam poslužiti klasa *Wall* koju ćemo opisati u sljedećem poglavlju.

4.3.2 Zidovi i klasa Wall

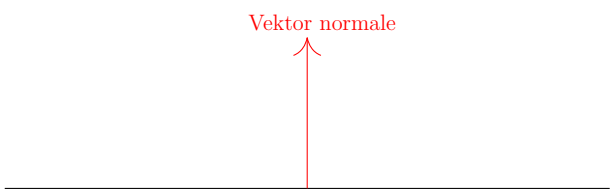
Ova klasa služi za implementaciju granica tj. koji će ograničiti kretanje kuglica. Ove zidove nećemo vidjeti. Oni su nevidljivi i nalaze se na samim rubovima prozora i ovise o *lookAt* vektoru. *LookAt* vektor je vektor koji služi za pozicioniranje kamere na sceni. Na primjer:

```
1  glm::mat4 View = glm::lookAt(
2          glm::vec3(0, 0, 50), // Camera is at (0,0,50), in World Space
3          glm::vec3(0, 0, -1), // and looks at the origin
4          glm::vec3(0, 1, 0)   // Head is up (set to 0,-1,0 to look
               upside-down)
5          );
6  }
```

Kod 4.2 Primjer lookAt vektora iz glm knjižnice

Prva linija govori gdje smo na sceni postavili kameru. To znači da se kameri nalazi na koordinatama (0,0,50) (udaljeni smo samo u Z smjeru). X koordinata, dakle je u rasponu od [-25, 25], a Y u rasponu od [-20,20] (razlog ovome je *Aspectratio* koji je 4:3, no u ovome poglavlju to nije bitno).

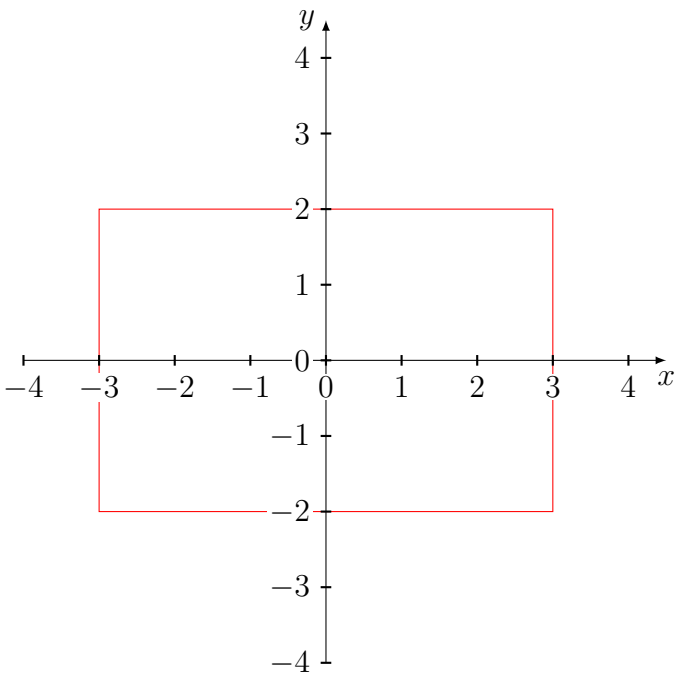
Zidovi su zamišljeni kao obični AABB-i s kojima kuglice provjeravaju sudar. Kako je već spomenuto u poglavlju 2.2 kuglice imaju pridodijeljen odgovarajući AABB, pa te objekte možemo iskoristiti za detekciju sudara između zida i kuglice. Ovdje će se također događati neke pogreške prilikom detekcije no to u principu nije bitno. Ovdje pogreška neće toliko utjecati na kretanje kuglice koliko bi u slučaju sudara između 2 kuglice. Zidovi također imaju pridodijeljen objekt klase *Vector3D*, a u njemu nam je zapisana normala zida. Normala zida je vrlo bitna za izračun sudara kuglice i zida. Važno je naglasiti da je normala jedinični vektor.



Slika 4.7 Prikaz jednog zida i normale na zid

Zid ima AABB iste reprezentacije kao što imaju i kuglice, dakle radijus i centar. Vertikalni zidovi imali su Y koordinatu centra 0, a horizontalni zidovi su imali X koordinatu 0. Zatim je trebalo je odrediti radijus. Zidovi koji su vertikalni, imali su samo Y komponentu radijusa, dok su horizontalni imali samo X koordinatu radijusa.

Primjer je u 2D, no logika i analogija je ista u sve 3 dimenzije. Samo bi dodali još 2 zida koji nam ograničavaju Z os.



Slika 4.8 Prikaz zidova u koordinatnom sustavu

S obzirom na prethodno rečeno, implementacija same klase je trivijalna. Ona je prikazana na idućoj strani.

```
1  class Wall {
2  public:
3          AABB_box bBox;
4          Vector3D vecDir;
5
6          Wall() = default;
7          Wall(const Point center, const float rX, const float rY, const
               float rZ,
8                    const float vecX, const float vecY, const float vecZ,
                         const float m);
9          Wall(const float x, const float y, const float z, const float
               rX,
10                   const float rY, const float rZ, const float vecX, const
                          float vecY,
11                   const float vecZ, const float m);
12         const float &x() const { return this->center.x; }
13         const float &y() const { return this->center.y; }
14
15         const float &z() const { return this->center.z; }
16         const float &getMass() const { return this->mass; }
17         Point &getCenter() { return this->center; }
18
19 private:
20         Point center;
21         float r[3];
22         float mass;
23         unsigned int i;
24 };
```

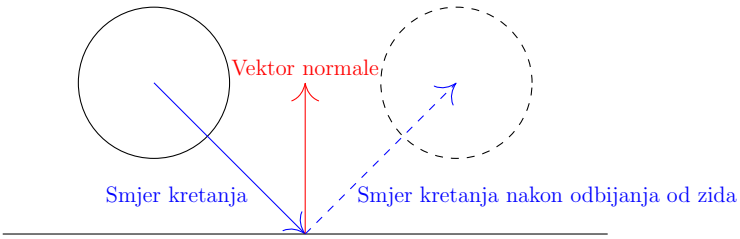
Kod 4.3 Implementacija klase Wall

U klasi nema metoda, osim *Get*, koje nam služe za dohvaćanje pojedinih privatnih varijabli. Među varijablama nalazi se i *mass* što definira masu zida. Ovoj varijabli pridodali smo vrlo veliku vrijednost, i nju ćemo kasnije koristiti za elastične sudare između zida i kuglica.

4.3.3 Sudari kuglica i zida

Nakon implementacije klase koje nam definiraju zidove, same sudare nije problem izračunati. U počeku nismo htjeli da se gubi energija kuglica. Sada će do izražaja

dolazi važnost normale zida. Htjeli smo da se kuglice odbije od zida pod istim kutom pod kojim se i sudarila. Ovakvim definiranjem normala, izbjegli smo korištenje trigonometrije i izračune upadnih i izlaznih kuteva. Ovakav tip sudara možemo



Slika 4.9 Reakcija kuglice na sudar sa zidom

prikazati formulom:

$$vector = vector - 2Dot(vector, normal)normal; \tag{4.5}$$

gdje su:

- *vector* - brzina i smjer kretanja kuglice
- *normal* - normala zida
- *Dot* - skalarni produkt

Ovo se također može prikazati jednostavni algoritmom koji je prikazan na sljedećoj stranici.

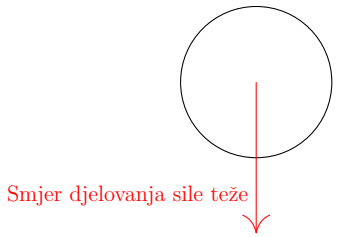
Pseudokod 5 Algoritam za izračunavanje smjera kretanja kuglice nakon sudara sa zidom

function RESOLVECOLLISON($Ball\ ball, Wall\ wall$)
  if AABBCollision($ball, wall$) is false **then return**
  end if
  $DoubleDot = 2(ball.vector * wall.normal)$
  $collisionNormal = DoubleDot * wall.normal$
  $ball.vector = ball.vector - collisionNormal$
end function

Animacija sada izgleda smisleno. Možemo dodati puno kuglica i testirati sudare između njih i sudare između kuglica i zidova. Prostor je ograničen i fizika odbijanja je smislena. Idući korak je dodavanje vanjske sile i gravitacije. Želimo da kuglice padaju, kao u stvarnome svijetu te realizirati padanje i odbijanje od podloge. U sljedećem poglavlju razmotrit ćemo dodavanje gravitacije na kretanje kuglica i njihovo odbijanje od podloge. U konačnici, odbijanje od podloge će biti isto. No, nakon odbijanja s kuglice trebaju imati fizikalnu reakciju na sudar. Kuglice se u stvarnom svijetu ne mogu stalno odbijati od podloge. Mora se dogoditi prijenos (u ovome slučaju gubitak) kinetičke energije što će u nekom trenutku rezultirati da se kuglice prestanu gibati. Sve ove slučajeve razmotriti ćemo u narednim poglavljima.

4.4 Sila teža

Sila teže je sila kojom Zemlja privlači neko tijelo mase m, označavamo ju najčešće slovom G. Ta sila privlači sva tijela prema središtu Zemlje pa je stoga tu i usmjerena. Sila teže razlog je zbog kojeg sva bića i predmeti ne odlete u Svemir, već stoje na površini Zemlje. Kada pustimo tijelo da slobodno pada s određene visine, ono se giba ubrzanjem g koje iznosi 9.81 $\frac{m}{s^2}$ neovisno o njegovoj težini što znači da će istovremeno pasti i iznimno teško i iznimno lagano tijelo ukoliko ih bacimo s iste visine. To je sila kojom predmet djeluje na vodoravnu podlogu na koju je položen, npr. knjiga na stol, ili uteg na nit na koju je obješen. Sila teža i težina jednake su po iznosu i smjeru djelovanja, ali nisu identični pojmovi.



Slika 4.10 Djelovanje sile teže

Programski, ovo je jednostavno realizirati. S obzirom da je sila teža vertikalna sila, ona će djelovati samo na Y komponentu brzine kuglice. Opća formula sile teže glasi:

$$G = m\ g \tag{4.6}$$

gdje je:

- m - masa objekta
- g - akceleracija sile teže koja iznosi 9.81 $\frac{m}{s^2}$

Ovo je samo formula sile i ona nam u ovom slučaju nije korisna. S obzirom da baratamo samo sa silama moramo iskoristiti drugu fizikalnu veličinu kojom ćemo regulirati brzinu da simuliramo silu težu.

Fizikalna veličina koju trebamo je slobodni pad. Brzina slobodnog pada iznosi:

$$v = g\ t \tag{4.7}$$

Slobodni pad je gibanje tijela isključivo pod utjecajem sile teže Slobodnoi pad je prijeđeni put s proporcionalan kvadratu protekloga vremena t, a brzina v jednoliko raste s proteklim vremenom, te da gibanje ne ovisi o masi tijela koje pada [11].Uobičajeno je da se slobodni pad uzima kao primjer jednolikog ubrzanog gibanja (gibanja sa stalnim ubrzanjem). Pritom se pretpostavlja da nema otpora zraka ili trenja [11].

Sama implementacija slobodnog pada je trivijalna:

Pseudokod 6 Algoritam za implementaciju slobodnog pada

```
function updateBallPosition(Ball ball, time dt)
    ball.x+ = ball.vector.x * dt
    ball.y+ = ball.vector.y * dt − 9.81 * dt
    ball.z+ = ball.vector.z * dt
end function
```

U algoritmu 6 *dt* (delta time) nam označava vrijeme koje je dobiveno između 2 framea. Na primjer, ukoliko imamo 60 sličica po sekundi, dt nam iznosi 16.6667 ms. Dobivanje razlike vremena između 2 framea je vrlo jednostavno. U nastavku ćemo prikazati implementaciju funkcije koja nam omogućuje izračunavanje proteklog vremena i kretanje kuglica po prostoru.

```
1
2  void Ball::updatePosition(float dt) {
3  double acc = -9.81;
4  this->vecDir.setY(this->vecDir.y() + acc * dt);
5  this->center.x += this->vecDir.x() * dt;
6  this->center.y += this->vecDir.y() * dt;
7  this->center.z += this->vecDir.z() * dt;
8  }
9
10 void update(int) {
11 nt = glutGet(GLUT_ELAPSED_TIME) / 1000.0f; //new time
12 dt = nt - ot; //delta time = new time - old time
13 ot = nt; //old time = new time
14
15 for (uint i = 0; i < balls.size(); i++) {
16 balls[i].updatePosition(dt);
17 }
18
19 glutPostRedisplay();
20 glutTimerFunc(16, update, 0);
21 }
```

Kod 4.4 Implementacije update funkcija

Funkcija *glutGet(GLUT_ELAPSED_TIME)* će nam vratit broj milisekudni koje su prošle od *glutInit* poziva (koji se stalno poziva na početku main petlje). Funkcija *glutTimerFunc* će nam odrediti najmanji broj milisekundi koji će proći do idućeg poziva Update funkcije.

Stvari s gravitacijom gledane na ovakav način su zapravo jasne. Na ovakav način možemo dodati bilo koju vanjsku silu koja će djelovati na kuglice. Iduće što želimo je da kuglice prilikom sudara prenose energiju. To će biti dodatno objašnjeno u idućem poglavlju.

4.5 Elastični sudari

U svim poglavljima do sada stalno se spominjala nekakva fizikalna reakcija kuglice na sudar. Svaka kuglica prilikom svog kretanja ima nekakvu kinetičku energiju. Opća formula kinetičke energije glasi:

$$E_k = \frac{m\ v^2}{2} \tag{4.8}$$

gdje je:

- m - masa kuglice
- v - brzina kuglice

Prilikom sudara između 2 kuglice, dio kinetičke energije će se izgubiti, tj. prijeći će u iz jedne kuglice na drugu i obrnuto. S obzirom da vrijedi ZAKON OČUVANJA ENERGIJE, zbroj energije 2 kuglice se neće promijeniti. Zakon očuvanja energije nam govori da u u zatvorenom sustavu zbroj svih oblika energije (mehaničke, toplinske, električne, magneske,...) konstantan. Drugim riječima, u zatvorenom sustavu jedan oblik energije može prelaziti u druge oblike, a da se pri tom energija niti stvara niti poništava[12].

Iduća fizikalna pojava koju ćemo analizirati je količina gibanja. Količina gibanja ili zalet (oznaka p) je vektorska fizikalna veličina u klasičnoj mehanici koja opisuje gibanje čestice ili sustava čestica[13]:

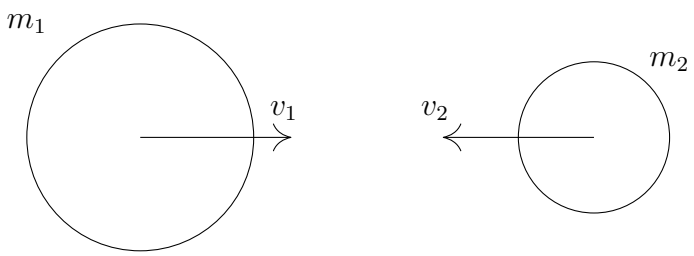
$$p = m\ v \tag{4.9}$$

gdje je:

- m - masa čestice
- v - brzina čestice

Svaka kuglica koja se na sceni kreće, imati će količinu gibanja. Prilikom sudara količina gibanja količina gibanja jedne kuglice prenijeti će se na drugu kuglicu i obrnuto. Slično, kao i kod zakona o očuvanju energije, zakon očuvanja količine gibanja nam govori da količina gibanja izoliranog sustava je konstantna, odnosno, ukupna promjena količine gibanja u vremenu unutar izoliranog sustava jednaka je nuli[13].

Sada kada smo objasnili 2 važna fizikalna zakona, možemo definirati što je to elastični sudar zapravo. Kako smo i ranije rekli, elastični sudar je sudar tijela ulaze nekom brzinom v_1 i v_2, a izlazna brzina iz sudara im je v_1' i v_2'. Kombinirajući zakon o očuvanju energije i zakon o očuvanju količine gibanja možemo reći slijedeće. Ukupna energija i ukupna količina gibanja prilikom sudara se ne mijenjaju, ali energija i količina gibanja određene kuglice se mijenja[14]. Pokažimo to na jednom jednostavnom primjeru.



Slika 4.11 Primjer sudara kuglica različite mase i brzine

gdje je:

- $m_1 > m_1$
- $v_1 > v_2$

S obzirom na mase i brzine možemo reći sljedeće. Kuglice će se nastaviti gibati u suprotnim smjerovima. Brzina v_1 će se smanjiti s obzirom da je masa te kuglice veća. Brzina v_2 će se povećati i kuglica će se kretati u suprotnom smjeru s većom brzinom.

Slika 4.12 Primjer sudara kuglica različite mase i brzine

Iako su se brzine promijenile, ukupna količina gibanja ostala je jednaka. Sukladno tome vrijedi:

$$m_1 v_1 + m_2 v_2 = m_1 v_1' + m_2 v_2' \tag{4.10}$$

Ukupna količina energije ostaje jednaka pa vrijedi i:

$$\frac{1}{2}(m_1 v_1^2 + m_2 v_2^2) = \frac{1}{2}(m_1 v_1'^2 + m_2 v_2'^2) \tag{4.11}$$

gdje su u obje jednadžbe:

- m_1, m_2 - mase kuglica
- v_1, v_2 - brzine kuglica prije sudara
- v_1', v_2' - brzine kuglica nakon sudara

Nakon niza matematičkih operacija i skraćivanja izraza možemo donijeti konačne izraze koji će nam izračunati brzine kuglica nakon sudara. Oni glase[14]:

$$\begin{aligned} v_1' &= \frac{v_1(m_1 - m_2) + 2m_2 v_2}{m_1 + m_2} \\ v_2' &= \frac{v_2(m_2 - m_1) + 2m_1 v_1}{m_1 + m_2} \end{aligned} \tag{4.12}$$

Ovi sudari vrijede samo za 1 dimenziju, ali to je u redu[14]. Prema onome što je ranije navedeno u poglavlju 4.3.1 nama i trebaju samo 1 dimenzionalni sudari. Ranije u poglavlju 4.3.1 opisane su jednadžbe kojima dobivamo smjer kretanja kuglica nakon sudara bez prijenosa količine energije i količine gibanja. U ovome poglavlju, objašnjeno kako će se ponašati samo vrijednosti brzina nakon sudara. Sada samo treba izvesti, ove 2 jednadžbe iz jednadžbe 4.12 dodati u kod za rješavanje sudara

kuglica. Prema algoritmu 4 možemo iznijeti slijedeći algoritam za to:

Pseudokod 7 Algoritam za izračunavanje smjera i iznosa brzina sudara između 2 kuglice uz promjenu količine gibanja jedne kuglice

function RESOLVECOLLISON(*Ball a*, *Ball b*)
  if Collision(a, b) is false **then return**
  end if
  $normal = (a.center - b.center)$
  $un = unitVector(normal)$
  $ut(-un.y, un.x)$
  $v_1n = a.v\ un$
  $v_1t = a.v\ ut$
  $v_2n = b.v\ un$
  $v_2t = b.v\ ut$
  $v_1(x, y, z) = \frac{v_1n(m_1-m_2)+2m_2v_2n}{m_1+m_2}$
  $v_2(x, y, z) = \frac{v_2n(m_2-m_1)+2m_1v_1n}{m_1+m_2}$
  $a.v = v_1\ un + v_1t\ ut$
  $b.v = v_2\ un + v_1t\ ut$
end function

Ovime smo zaokružili cjelinu o kretanjima i sudaranjima kuglica. Ipak, problem je to što nam je sama detekcija sudara između svih kuglica prespora. Kako smo opisali ranije u poglavlju 2, postoje algoritmi za ubrzavanja same detekcije sudara. U narednim poglavljima cilj je opisati algoritam s kojim smo na zadovoljavajući način detektirali sve sudare na sceni i prema gore navedenim jednadžbama, iste sudare izračunali.

Poglavlje 5

Particioniranje prostora

Kako je spomenuto u Uvodu, postoje razni algoritmi za detekciju sudara. Najvažniji od njih su oni kojima dijelimo objekte (BVH), i one kojima dijelimo prostor. Algoritam s kojim dijelimo objekt smo već prošlu u poglavlju 3. Kako smo u nekom trenutku odustali od BVH-a bilo je potrebno pronaći dobar i brz način za detektiranje sudara na cijeloj sceni. Detekcija sudara sama po sebi, komplicirana je operacija i potrebno je da bude manje složena od $O(n^2)$ operacija. Kako smo to opisali u poglavlju 3, BVH-u je ptorebno OlogN operacija sa detekciju sudara na sceni, iako je nekada prije potrebno napraviti dodatne kalkulacije i ažuriranja strukture da bi ona bila funkcionalna. To dodatno komplicira stvari, ali je vrlo efikasno. U ovom poglavlju, pokazati ćemo da je particioniranje prostora također jedna efikasna i lako razumljiva strategija za detekciju sudara.

5.1 Primitivna detekcija sudara

U prošlom poglavlju objašnjeno je kako sudare, nakon što ih detektiramo, kalkulirati. Nigdje nismo spomenuli kako smo i te sudare detektirali. Koristili smo za to vrlo primitivan i jednostavan algoritam. Za svaki objekt na sceni, iterativno bi tražili s kojim se objektom sudara. Ovo u principu nije problem kada je situacija sa zidovima jer u konačnici imamo 4 zida pa je sama složenost te operacije O(4 * N). Problem se javlja kada imamo kuglica. Do 50 kuglica ovaj algoritam će raditi, no nakon

prelaska na veći broj kuglica javlja se problem. Za 500 kuglica potrebno je napraviti isto toliko provjera. To nas dovodi do brojke od 250 000 tisuća provjera i složenosti $O(n^2)$. Kako smo ranije opisali ovo smo željeli izbjeći. Takav algoritam bi glasio ovako: gdje je:

Pseudokod 8 Algoritam za primitivnu detekciju sudara između kuglica

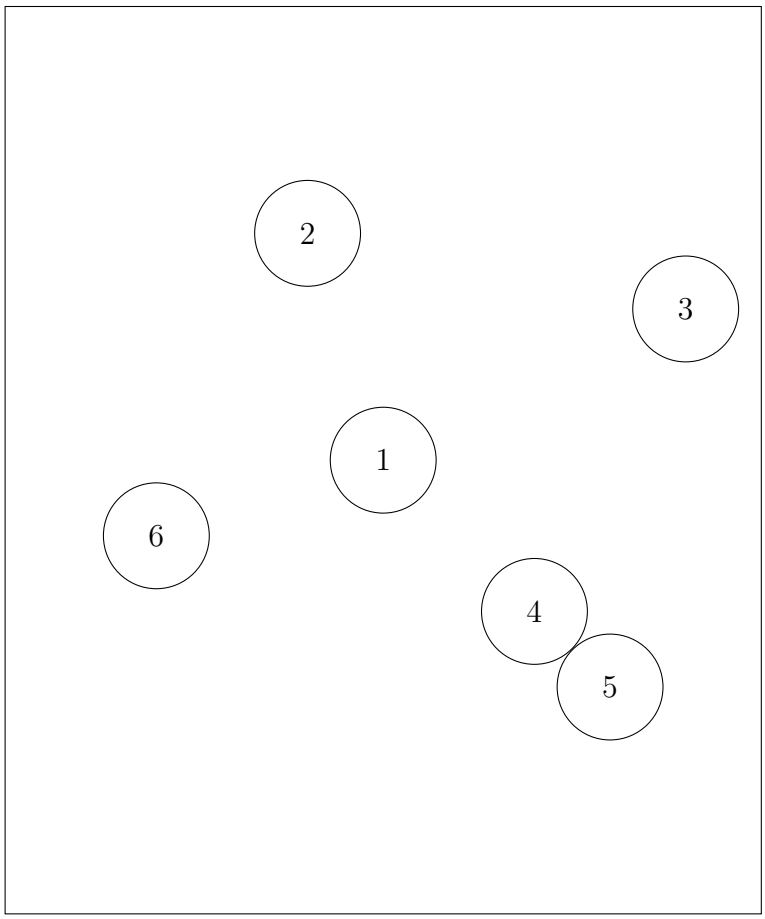
```
function searchCollisions(Ball list[N])
    i = 1
    for ball in list do
        while i < N do
            if ifCollision(ball, list[i]) then
                resolveCollision(ball, list[i])
            end if
            i++
        end while
    end for
end function
```

- list - lista kuglica
- N - broj kuglica

Kako je već spomenuto, za listu smo koristili *std::vector* u C++. Ovakav pristup smo koristili samo pri početku projekta. Za mali broj kuglica i za provjeru ispravnosti proračuna, ovakav pristup je bio dovoljan.

5.2 k-dimenzionalno stablo (K-dimensional tree - k-d)

Složenost koju želimo postići pri detektiranju sudara najmanje O(NlogN) (za N objekata na sceni). Da bi to postigli prostor se mora podijeliti u smislene regije u kojima ćemo provjeravati sudar. Promotrimo to na jednom primjeru:



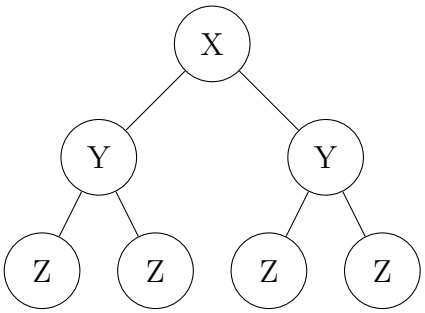
Slika 5.1 Primjer kuglica na sceni

gdje su koordinate centara:

- 1 - (5,4)
- 2 - (4,7)
- 3 - (9,6)
- 4 - (7,2)
- 5 - (8,1)
- 6 - (2,3)

Na slici 5.2 vidimo da ne treba u svim situacijama provjeravati sudar. Na primjer za kuglicu 2 i 6 znamo da se sudar neće dogoditi te tu provjeri ne moramo uzimati u obzir dok između kuglica 5 i 4 te 1 i 2 znamo da bi se mogao dogoditi sudar u skorom vremenu, te, te sudare moramo provjeriti.

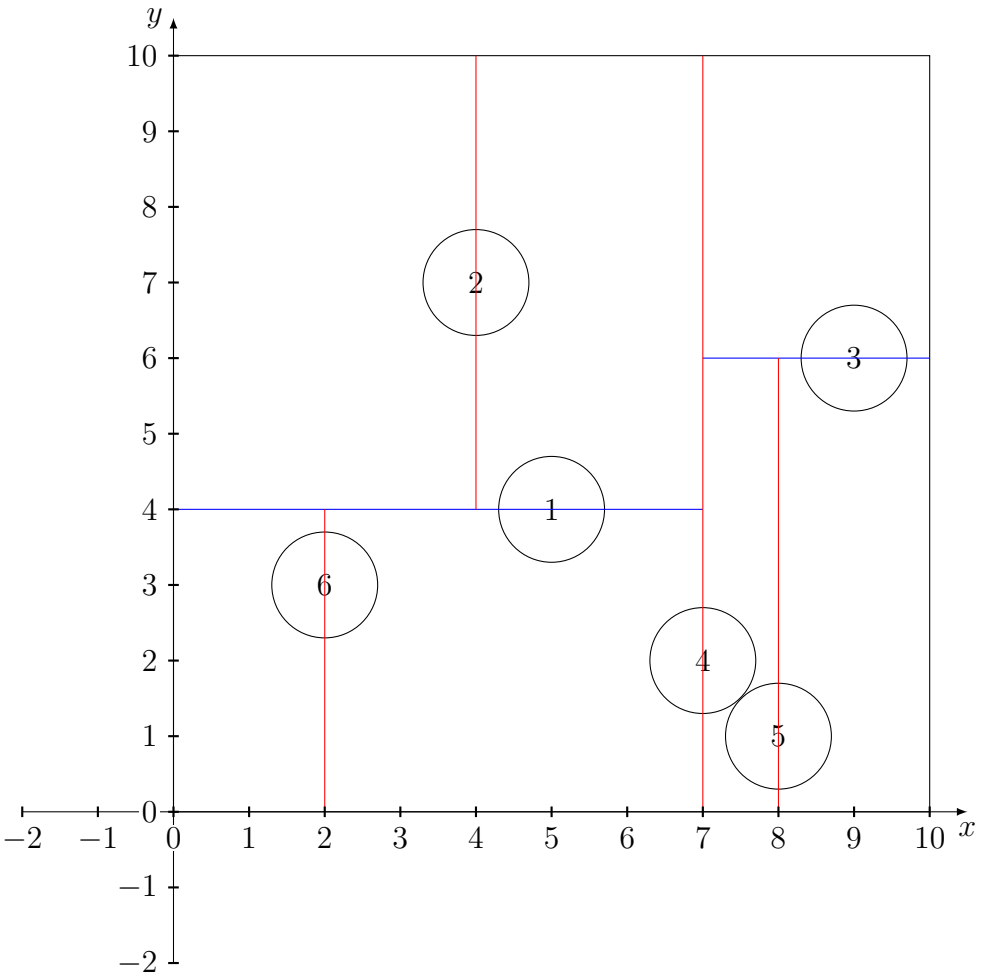
Postoje mnoge tehnike za podijeliti prostor, no ona koju smo mi koristili je k-d stablo. K-d stablo je binarno stablo gdje je svaki čvor k-dimenzionalna točka[6]. Svaki čvor koji nije list, može se shvatit kao ravnina koja dijeli prostor. Točke koje se nalaze lijevo od ravnine predstavlja lijeva strana stabla, dok desne točke predstavlja desna strana stabla[6]. Konkretno, ukoliko odaberemo X os za os podijele, točke manje od vrijednosti X bit će nam u lijevoj grani stabla, dok točke veće od X vrijednosti će biti na desnoj strani stabla[6]. U 3D prostoru intuitivno je da ćemo imat 3 dimenzije. Dakle, root čvor stabla će podijeliti prostor prema X koordinati, zatim njegova djeca prema Y koordinati, djeca djece po Z koordinati itd. .



Slika 5.2 Prikaz k-d stabla za 3 dimenzije

Nakon što smo došli do Z dimenzije, ponovno dijelimo prostor po x koordinati, i tako

se vrtimo dok ne podijelimo cijeli prostor. Sada se možemo vratiti na situaciju iz slike 5.1. k-d stablo će nam onakav prostor podijeliti na sljedeći način[6]:



Slika 5.3 Podjela prostora k-d stablom

Primjetimo da smo prostor podijelili po koordinatama centra svake kuglice.Crvene linije prikazuju podjelu prostora po X koordinati, dok plave ukazuju na podjelu prostora po Y koordinati. Da smo imali još Z koordinatu, analogija je ista, imali bi dodatnu dimenziju, no zbog jednostavnosti dovoljno je ovo prikazati u 2D prostoru.

5.2.1 Izgradnja k-d stabla

U prijašnjem poglavlju podijelili smo prostor bez da znamo zapravo kako je to napravljeno, niti da znamo kako će struktura stabla izgledati za takvu situaciju. Općenito, znamo kako struktura stabla izgleda. Svaka razina stabla particionirala je prostor po nekoj koordinati. Jedino pitanje koje se postavlja je, kako izabrati točke na način da dobijemo kvalitetno podijeljen prostor i balansirano stablo. S obzirom da postoji puno načina na koji možemo odabrati osi presijecenja koje su poravnate s koordinatnim osima, postoji i puno načina da izgradimo k-d stablo. Kanonski način izgradnje je onaj koji smo već opisali[6]:

- Kako se krećemo po stablu, kružno biramo osi presijecanja
- Točke odabiremo tako da odaberemo median element s obzirom na os presijecanja

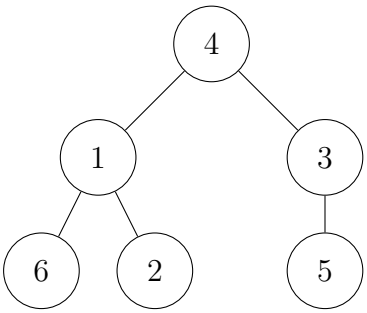
Ovakva metoda će nas dovesti do balansiranog stabla, iako za sve aplikacije to nije nužan uvjet [6].

S obzirom na sve ovo možemo napisati jednostavan algoritam za izgradnju samog k-d stabla.

Pseudokod 9 Algoritam za izgradnju k-d stabla[6]

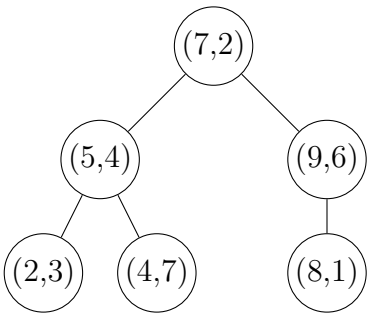
function K-D TREE($Point\ pointList[N]$, $depth$)
  $axis = depth \mod k$
  $Sort(list)$
  Select $median$ by $axis$ from $pointList$
  $Node.point = median$
  $Node.left(pointList[0...Median - 1], depth + 1)$
  $Node.right(pointList[Median + 1...pointList.size], depth + 1)$
end function

Sada kada znamo izgraditi stablo možemo prikazati kako bi stablo konkretno izgledalo za situaciju na slici 5.3 it prošlog poglavlja: Prikažimo ovo isto stablo sa



Slika 5.4 Prikaz strukture k-d stabla za sliku 5.3

koordinatama centara[6]:



Slika 5.5 Prikaz strukture k-d stabla za sliku 5.3

Složenost ovog algoritma ovisi o složenosti operacije sortiranja i pretrage za median elementom. Pozicija median elementa se pronalazi kao:

$$Median = \frac{N}{2} \tag{5.1}$$

gdje je:

- N - broj elemenata

Algoritmi koji se uglavnom koriste za sortiranje su *Mergesort* (implementiran u C++ STL-u) i *Heapsort*. Složenost ovih algoritama u općem slučaju je O(NlogN). Sukladno

tome to je i najbolja složenost koja se može postići s ovim algoritmima sortiranja. Složenost izgradnje stabla u najgorem slučaju može biti O(kNlogN) operacija što ćemo mi i imati s obzirom da u svakoj od k dimenzija moramo sortirati točke i odabrati median element. S obzirom da se kuglice na sceni kreću, stablo moramo graditi u svakom frameu pa je i ovakav način detekcije sudara sam po sebi dosta složen, no još uvijek je brži od onog opisanog u algoritmu 8.

Sada je potrebno pretražiti sve potencijalne sudare na sceni. Dobra osobina k-d stabla je ta što će nam podijeliti prostor po gustoći točaka. Tako možemo vrlo jednostavno detektirati sudare na sceni i izračunati ih ukoliko se sudar dogodio.

5.2.2 Pretraga sudara u k-d stablu

Pretraga sudara u k-d stablu nakon izgradnje je trivijalna stvar. Sve se svodi na jednostavno "šetanje" po stablu. Da dodatno ne povećavamo složenost, šetanje će uključiti samo onu granu stabla koja je bliže u prostoru i/ili onu s kojom se kuglica sudara. Da se izbjegne korištenje funkcije korijena (*sqrt*) koristiti će se kvadrirana udaljenost (eng. *squared distance*). Ukoliko detektiramo sudar, u istom trenutku ga i računamo. Algoritam je rekurzivan i glasi:

Pseudokod 10 Algoritam za pretragu sudara u k-d stablu

```
function searchCollisions(Ballball, K − dTreeNodeNode)
    if Node is leaf then
        if isCollision(ball, node) then
            resolveCollision(ball, node)
        end ifreturn
    end if
    if isCollision(ball, node) then resolveCollision(ball, node)
    end if
    if distance(Node.left, ball) > distance(Node.right, ball) or
        isCollision(ball, node.right) then
        searchCollisions(ball, node.right)
    else if distance(Node.right, ball) > distance(Node.left, ball) or
        isCollision(ball, node.left) then
        searchCollisions(ball, node.left)
    end if
end function
function checkAllBalls(BalllistBall[N], K − dTreeNodeNode)
    for ball in listBall do
        searchCollisions(ball, Node)
    end for
end function
```

Na ovakav ćemo način jednostavno na cijeloj sceni detektirati sudare i izračunati. S obzirom da je kuglica uvijek u sudaru sama sa sobom, potrebno je izbjeći računanje takvog sudara. Prilikom implementacije klase za kuglice, u kodu 2.8, spomenuli smo atribut i tipa integer. Ovaj atribut označava poziciju kuglice u listi i na taj način ćemo vrlo jednostavno izbjeći računanje sudara kuglice same sa sobom. Testiranjem je utvrđeno da se prilikom takvog računanja mogu pojaviti anomalije i ponašanje kuglica koje ne želimo.

5.2.3 Implementacija klase za k-d stablo

U ovome poglavlju nećemo prikazivati implementaciju svih funkcija. Opisati ćemo samo odluke i strukturu same klase.

```
1
2  template <class T> class KDtreeNode {
3  static bool sortbyX(T lhs, T rhs)
4  static bool sortbyY(T lhs, T rhs)
5  static bool sortbyZ(T lhs, T rhs)
6
7  public:
8          KDtreeNode *left()
9          KDtreeNode *right()
10         unsigned long childSize()
11         unsigned int getPlane()
12         void build_tree(std::vector<T> &v, int depth)
13         void treeTraverse()
14         template <class T1> void searchCollisions(T1 &ball)
15 private:
16         std::vector<KDtreeNode> child;
17         T object;
18         unsigned int plane;
19 };
```

Kod 5.1 Implementacija klase za k-d stablo

U početku bila je dilema koristiti li normalnu klasu koja bi radila samo na temelju kuglica, ili *template* klasu koja bi onda radila za sve tipove objekata koje bi stavili scenu. Odabrana je *template* klasa upravo iz tog razloga, da prilikom dodavanja nekih drugih objekata koji nisu kuglice, ne moramo raditi novu strukturu za samo taj specifični objekt. Točnost ove tvrdnje pokazana je kada se ista klasa za k-d stablo koristila i za particioniranje zidova. Jednom strukturom smo odradili posao za sve moguće objekte na sceni. Postoji i bolji pristup da se koriste samo točke objekata, ali za nas je ovakav bio dovoljan. Klase *sortby(x,y,z)* poslužile su za sortiranje cijele liste prema određenoj osi. Ostale funkcije koje smo koristili su ili već opisane (*build_tree*, *searchCollisions*) ili su samo *Get* funkcije za dohvat privatnih varijabli.

5.3 Prednosti i mane korištenja algoritama za particioniranje prostora (k-d stabla)

K-d stablo svakako pronalazi svoju uporabu u stvarnome svijetu. Vrlo jednostavna implementacija i razumijevanje same strukture omogućuje široku primjenu u svijetu igara. Skladno tome, jasno je da je k-d stablo dobar način kojim ćemo pretraživati sudare na sceni. Vrlo je efikasno iako ima svoje mane.

Prva od njih je ta da u svakom koraku izgradnje stabla moramo sortirati cijelu listu točaka i to nam uvijek zahtjeva najmanje O(NlogN) operacija. Na taj način smo ograničeni i tu ne možemo napraviti dodatne optimizacije iako one postoje. Postoji bolji način odabira ravnine presjecanja i točke koja će biti čvor (npr. metoda fiksne točke ili odabir slučajne točke)[6]. Ove metode nam ipak neće dati balansirano stablo, ali to s druge strane ovisi o našim potrebama. Balansirano stablo u praksi nije uvijek prijeko potrebno pa se možemo spasiti od silnog sortiranja u svakom koraku izgradnje stabla.

S druge strane, ukoliko nam treba balansirano stablo, sortiranje ne možemo izbjeći. Zato se k-d stablo u praksi često koristi za statične objekte koji se ne gibaju[6]. Tako možemo izvršiti izgradnju stabla prije pokretanja animacije. Dobar primjer iskorišten je i kod nas gdje smo izgradili stablo od zidova. Iako zida postoje samo 4 na sceni, zgodno je bilo pokazati da je k-d stablo zapravo puno efikasnije kod statičnih objekata.

Kod kuglica se događao problem s velikim brojem kuglica (500+), jer unutar 30-60 FPS-a nismo uspjeli izvršiti sve provjere sudara i izračunati iste. Ovdje se može uvesti jedna optimizacija da se u svakoj pretragi sudara računa i vrijeme do idućeg sudara, no mi je nismo koristili.

Poglavlje 6

Sjenčanje

U fazi sjenčanja, ujedno i posljednjoj fazi ovoga rada, cilj je bio dodati atribute kuglicama kako bi rad izgledao impresivnije. Prije samog korištenja alata za sjenčanje (tzv. Shader) bilo je potrebno promijeniti okolinu u kojoj se rad izvršavao. Do sada, koristili smo OpenGL 1.x, koji je eksperimentalan i ne koristi Shadere za iscrtavanje poligona. S OpenGL-a 1.x, smo iz tog razloga prešli na noviju verziju, OpenGL 3.3 koji nam omogućava korištenje Shadera i nekih naprednijih opcija koje nismo do sada imali.

Iako je noviji, OpenGL 3.3 koristi neke nove principe i modele s kojima se do sada nismo susreli. Zbog toga, morali smo odustati od *GLUT-a*, koji nam je služio za komunikaciju sa operacijskim sustavom (prikaz prozora, korištenje miša i tipkovnice, terminiranje prozora,...). Prešli smo na noviji *GLFW* koji se koristi kod modernijih verzija OpenGL-a. U principu *GLFW* i *GLUT* služe za iste stvari, no *GLFW* je moderniji i pruža nam bolju optimizaciju programa nego stariji *GLUT*. Usprkos tome, *GLFW* nema biblioteke za crtanje sfera, pravokutnika i sl. pa smo morali sami definirati sferu kojom ćemo opisati kuglice. To će biti objašnjeno u narednom poglavlju.

6.1 Sfera

Kako smo spomenuli, *GLFW* nema biblioteke kojima možemo nacrtati jednostavne poligone kao što ima *GLUT*. Dakle, do sada korištena funkcija *glutSolidSphere* za crtanje kuglica, više nam nije bilo od koristi i morali smo definirati sami točke kojima ćemo crtati sferu.

Za crtanje sfere postoje generalno 2 načina:

- Programsko generiranje točaka, indeksa i normala
- Prijenos objekta iz odgovarajućeg formata generiranog u programu za crtanje modela (npr. Blender)

U početku je bila velika dilema koji od ova 2 načina odabrati, s obzirom da je svaki od njih imao neke svoje prednosti i mane, no u konačnosti izabrali smo programsko generiranje zbog relativno lakšeg implementiranja od prijenosa objekata. Inače, prijenos objekata iz nekog drugog alata nije sam po sebi težak, no ili zahtjeva dodatne biblioteke (npr. Assimp) ili zahtjeva pisanje programa koji će preuzeti podatke iz formata i postaviti ih u polja koja iscrtavamo.

Kako smo rekli, OpenGL 3.3 za crtanje objekata koristi Shadere. Bez Shadera na sceni nećemo vidjeti ništa, te su oni ti koje objekte prikazuju. Shadere je potrebno posebno kompilirati i to za sada nećemo ovdje prikazivati, nego u kasnijim poglavljima. Za početak koristili smo jednostavne Shadere. Shadere dijelimo na Shadere točaka i Shadere fragmenata[3]. Trenutno nećemo ulaziti u analizu kodova nego ćemo samo prikazati Shadere. Shader točaka definiran je:

```
1 #version 330 core
2 layout(location = 0) in vec3 vertexPosition;
3 // Values that stay constant for the whole mesh.
4 out vec3 fragmentColor;
5 void main(){
6 // Output position of the vertex, in clip space : MVP * position
7         gl_Position.xyz =  vertexPosition_modelspace;
8         fragmentColor = vertexColor;
9 }
```

Kod 6.1 Jednostavni Shader točaka

Shader fragmenata definiran je kao:

```
1 #version 330 core
2 // Ouput data
3 in vec3 fragmentColor;
4 out vec3 color;
5
6 void main()
7 {
8         color = vec3(0.4,0.2,0.7);
9 }
```

Kod 6.2 Jednostavni Shader fragmenata

U početku, bilo je vrlo teško krenuti od crtanja same sfere. Trebalo je prvo razumjeti sve strukture koje se koriste u OpenGL-u 3.3 pa smo krenuli prvo od crtanja običnog trokuta korištenjem spomenutih Shadera.

6.1.1 Crtanje trokuta

Crtanje trokuta je prvi korak pri učenju OpenGL-a 3.3 [3][4]. Na ovom jednostavnom primjeru, mogu se naučiti sve strukture i tipovi podataka koji se koriste prilikom korištenja OpenGL-a 3.3.

Trokut možemo definirati sa 3 točke. Neka to budu točke:

- (-0.5f, -0.5f, 0.0f)
- (0.5f, -0.5f, 0.0f)
- (0.0f, 0.5f, 0.0f)

Vrlo je važno da točke budu definirane točno ovim redom inače ih OpenGL 3.3 neće nacrtati kako smo to htjeli. Ove točke važno je spremiti u strukturu zvanu VBO tj. Vertex Buffer Object. Ova struktura nam omogućava crtanje objekata na grafičkoj kartici, a ne na procesoru[4]. Prije VBO-a definirali smo i Vertex Array Object, no ovdje on nije posebno bitan pa ga nećemo posebno ni spominjati. U nastavku slijedi kod kojime smo generirali VBO i u njega poslali podatke o točkama trokuta[3].

```
1  static const float vertices[] = {-0.5f, -0.5f, 0.0f, 0.5f, -0.5f, 0.0f,
       0.0f, 0.5f, 0.0f,}; //vertices of triangle
2
3  unsigned int VBO;
4  glGenBuffers(1, &VBO);
5  glBindBuffer(GL_ARRAY_BUFFER, VBO);
6  glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
       GL_STATIC_DRAW);
7
8  glBindBuffer(GL_ARRAY_BUFFER, VBO);
9  glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (
       void *)0);
10 glEnableVertexAttribArray(0);
11 glBindBuffer(GL_ARRAY_BUFFER, 0);
```

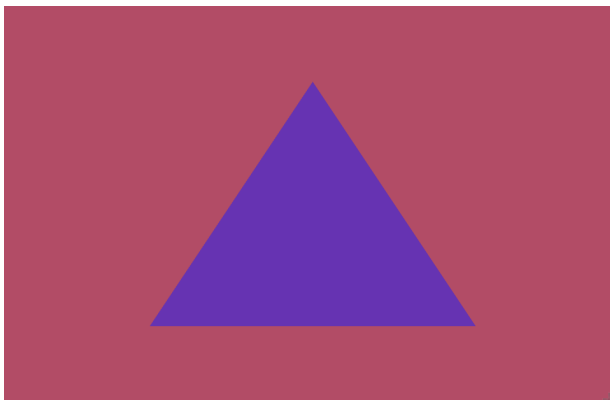
Kod 6.3 Generiranje VBO[4]

Samo crtanje trokuta izvodi se na:

```
1  do {
2          glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
3          glUseProgram(Shader);
4          glEnableVertexAttribArray(0);
5          glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
6          glDrawArrays(GL_TRIANGLES, 0, 3); // 3 indices starting at 0 ->
               1
7          glfwSwapBuffers(window);
8          glfwPollEvents();
9  }
```

Kod 6.4 Crtanje trokuta[3]

Objašnjenje ovoga koda nećemo raditi, s obzirom da su to funkcije i strukture korištene isključivo sa strane OpenGL-a. Nastavak objašnjenja ovih struktura može se pronaći u [3] i [4]. Na ovakav način dobili smo trokut koji izgleda ovako:



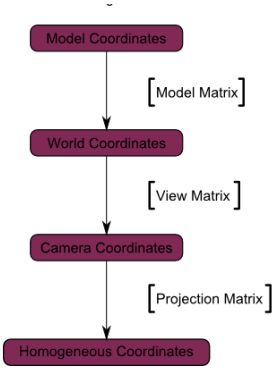
Slika 6.1 Trokut u OpenGL-u 3.3

Idući zadatak bio je ovaj trokut pomicati po prostoru. Ovdje smo umjesto funkcija unutar OpenGL-a koristili *glm* biblioteku matematičkih funkcija koja nam omogućava operacija sa matricama i transformacije u prostoru.

6.1.2 Transformacije

Prije korištenja OpenGL-a 3.3 transformacije smo definirali funkcijama unutar OpenGL-a. S obzirom da smo tada samo crtanje objekata izvršavali na procesoru, ovo je bilo moguće bez problema. Prelaskom na OpenGL 3.3 transformacije smo morali poslati u Shader da bi znali gdje nacrtati objekt.

Svaki objekt ima svoj definirani prostor koji je definiran lokalnim koordinatama[4]. Lokalne koordinate svakog objekta definirane su matricom modela. Transformacijom matrice modela definiramo poziciju svakog objekta unutar cijele scene tj. svijeta. Konačno, matricom pogleda definiramo pogled na svijet tj. koordinate iz kojih imamo pogled na cijelu scenu. U konačnosti definiranjem projekcije definiramo kako ćemo vidjeti stvari na sceni (”Clip space”). Ovo možemo prikazati jednostavnim dijagramom:



Slika 6.2 Transformacije matrica i koordinata u OpenGL-u[3]

Umnoškom:

$$MVP = ProjectionMatrix * ViewMatrix * ModelMatrix \tag{6.1}$$

dobili smo transformaciju točaka u prostor koji mi vidimo (ili ne vidimo)[4]. Transformacijom matrice modela (Model Matrix), možemo obavljati transformacije na objektima (translacija, skaliranje, rotiranje).Jednostavnim umnoškom matrica definiramo transformacije objekta. Na primjer:

```
glm::mat4 projection = glm::perspective(glm::radians(45.0f),(float)4 /
    (float)3, 0.1f, 1000.0f);

glm::mat4 view = glm::lookAt(
  glm::vec3(0, 0, 50), // Camera is at (0,0,50), in World Space
  glm::vec3(0, 0, -1), // and looks at the origin
  glm::vec3(0, 1, 0)   // Head is up (set to 0,-1,0 to look upside-down
      )
);
glm::mat4 Model = glm::mat4(1.0f);
glm::vec3 position = glm::vec3(5.0, 4.0, 0.0);
glm::vec3 radius = glm::vec3(1.0f, 1.0f, 1.0f) * 3.0f;
Model = glm::scale(Model, radius);
Model = glm::translate(Model, position);
glm::mat4 mvp = projection * view * Model;
```

Kod 6.5 Primjer transformacije objekata

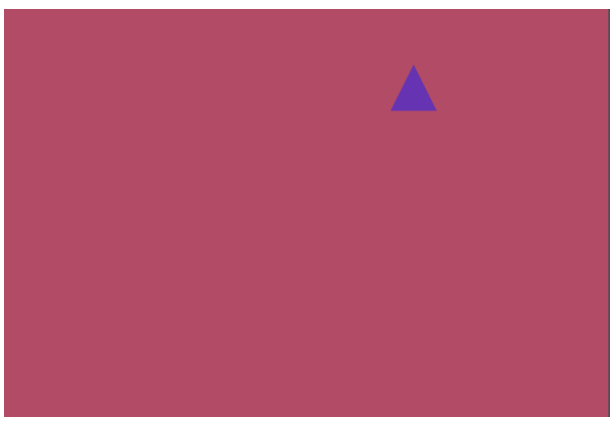
Transformacije objekata moramo primjenjivati prema pravilu, prvo skaliranje, zatim rotacija i u konačnosti translacija[3].

Kada smo na procesoru izračunali transformacijske matrice, iste je potrebno poslati u Shadere. U pravilu, sva množenja matrica potrebno je izvršavati na procesoru, no nekada se množenja matrice mogu izvršavati i na grafičkoj kartici. Ovo će biti kod nas slučaj u kasnijem dijelu crtanja same sfere radi boljeg dizajna koda. Izračunate transformacijske matrice šaljemo u Shadere pomoću ključne riječi *uniform*. Unutar Shadera točaka je potrebno pomnožiti transformacijsku matricu sa svim točkama objekta, da bi dobili točno mjesto gdje ćemo nacrtati naš objekt. To se izvodi na sljedeći način:

```
1 layout(location = 0) in vec3 vertexPosition_modelspace;
2
3 // Values that stay constant for the whole mesh.
4 uniform mat4 MVP;
5
6 void main(){
7 // Output position of the vertex, in clip space : MVP * position
8 gl_Position =  MVP * vec4(vertexPosition_modelspace,1);
9 }
```

Kod 6.6 Primjer transformacije objekata

Trokut sa slike 6.1 sada izgleda ovako: Iako smo trokut skaliranjem povećali, on



Slika 6.3 Trokut sa apliciranim transformacijama

izgleda manje iz razloga što je kamera udaljena u Z smjeru. Rotaciju ovdje nismo primjenjivali jer nam kasnije nije bila potrebna.

Razumijevanje transformacija i matrica bila je jedna od ključnih koraka prije crtanja same sfere. Sada smo mogli definirati veliki broj trokuta, koje ćemo onda crtati na sceni, a sve u nekoliko linija unutar Shadera i C++. U ovom koraku već smo mogli primijetiti da algoritam koji smo prije implementirali radi na trokutima (koji su opisani sferom). U daljnjim koracima potrebno je bilo nacrtati same sfere i definirati Shadere koji će dati sferama određena svojstva.

6.1.3 Crtanje sfere

Za crtanje sfere potrebno je bilo osmisliti matematički model kojim ćemo generirati točke sfere. Sferu crtamo u središtu koordinatnog sustava, sa radijusom veličine 1, pa ju zatim translatiramo i skaliramo po potrebi.

Koordinate sfere u 3D prostoru možemo opisati kao:

$$\begin{aligned} x &= cos\theta\ sin\pi \\ y &= cos\pi \\ z &= sin\theta\ cos\pi \end{aligned} \tag{6.2}$$

gdje je:

- $\theta \in [0, 2\pi]$
- $\pi \in [0, \pi]$

Svaka od ovih jednadžbi još uključuje radijus, no s obzirom da želimo sferu veličine radijusa 1, to nam nije potrebno. Uz točke, za sferu je još potrebno izračunati indekse. Indekse spremamo u Element buffer object i njime omogućavamo korištenje više istih točaka, više puta[3]. Prije smo unutar poziva funkcije za crtanje pozivali Vertex Buffer Object, no sada kada koristimo indekse pozivamo Element Buffer Object.Više informacija o ovoj strukturi podataka može se pronaći u literaturi [3][4].

U konačnosti, klasa za generiranje sfere izgleda ovako:

```
1  Sphere(uint Shader) {
2  class Sphere {
3  public:
4    Sphere(uint Shader);
5    void drawSphere(float r, float x, float y, float z);
6
7  private:
8    uint vao;
9    uint ebo;
10   uint vbo;
11   uint normalbuffer;
12   uint colorbuffer;
13   std::vector<glm::vec3> vertices;
14   std::vector<glm::vec3> normals;
15   std::vector<glm::vec3> colors;
16   std::vector<int> indices;
17   uint shader;
18   int slices, stacks;
19   int ModelMatrixID;
20 };
```

Kod 6.7 Kod za generiranje sfere

U konstrukturu smo definirali matematički model sfere i inicijalizirali sve potrebne strukture koje nam trebaju pri crtanju sfera. Ovdje nećemo prikazivati inicijalizcaiju svih polja i struktura koji su nam potrebni za crtanje, nego ćemo samo pokazati kako smo iz matematičkog modela napisali kod za generiranje točaka i indeksa sfere. Ovdje smo još izračunali i normale te definirali boju svake od sfera (iako boja nije nužno potrebna). Normalu je vrlo jednostavno izračunati s obzirom da se radi o sferi kojoj je centar u ishodištu koordinatnog sustava:

$$\begin{aligned} Normal.x &= x \\ Normal.y &= y \\ Normal.z &= z \end{aligned} \tag{6.3}$$

Kod kojim generiramo sfere izgleda ovako:

```
1  Sphere(uint Shader) {
2    for (int i = 0; i <= this->stacks; ++i) {
3          GLfloat V = i / (float)this->stacks;
4          GLfloat phi = V * glm::pi<float>();
5
6          // Loop Through Slices
7          for (int j = 0; j <= this->slices; ++j) {
8                  GLfloat U = j / (float)this->slices;
9                  GLfloat theta = U * (glm::pi<float>() * 2);
10
11                 // Calc The Vertex Positions
12                 GLfloat x = cosf(theta) * sinf(phi);
13                 GLfloat y = cosf(phi);
14                 GLfloat z = sinf(theta) * sinf(phi);
15
16                 // Push Back Vertex Data
17                 vertices.push_back(glm::vec3(x, y, z));
18                 normals.push_back(glm::vec3(x, y, z));
19                 colors.push_back(glm::vec3(1, 0, 0));
20         }
21   }
22
23   // Calc The Index Positions
24   for (int i = 0; i < this->slices * this->stacks + this->slices; ++i)
         {
25         indices.push_back(i);
26         indices.push_back(i + this->slices + 1);
27                 indices.push_back(i + this->slices);
28
29         indices.push_back(i + this->slices + 1);
30         indices.push_back(i);
31         indices.push_back(i + 1);
32   }
33 }
```

Kod 6.8 Kod za generiranje točaka i indeksa sfere

Sukladno tome, kod za crtanje sfere izgleda ovako:

```
1  void drawSphere(float r, float x, float y, float z) {
2    glm::mat4 Model = glm::mat4(1.0);
3    glm::vec3 rad = glm::vec3(1.0f, 1.0f, 1.0f) * r;
4    Model = glm::scale(Model, rad);
5    glm::vec3 pos = glm::vec3(x, y, z);
6    Model = glm::translate(Model, pos);
7
8    glUniformMatrix4fv(ModelMatrixID, 1, GL_FALSE, &Model[0][0]);
9
10   glUseProgram(this->shader);
11   glBindVertexArray(vao);
12   glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);
13   glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, NULL);
14 }
```

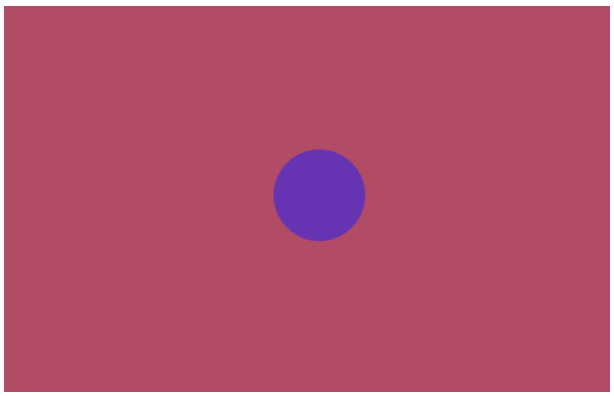
Kod 6.9 Kod za crtanje sfere

U detaljnu analizu ovih kodova nećemo ulaziti. Može se da neposredno prije crtanja (kod 6.8) izvršavamo translaciju i skaliranje sfere. Ovdje ne množimo transformacijske matrice, već to radimo na grafičkoj kartici. Kada bi to izvršavali u ovoj funkciji, iz glavnoga programa bi morali poslati matrice pogleda i projekcije što narušava dizajn koda. Dakle, u glavnom programu izračunamo umnožak između matrice projekcije i matrice pogleda, dok u Shaderu tek računamo potpunu transformacijsku matricu gdje ovaj produkt množimo sa matricom modela. Shader u ovom slučaju izgleda:

```
1  #version 330 core
2
3  // Input vertex data, different for all executions of this shader.
4  layout(location = 0) in vec3 vertexPosition_modelspace;
5  // Values that stay constant for the whole mesh.
6  uniform mat4 VP;
7  uniform mat4 M;
8  void main(){
9  // Output position of the vertex, in clip space : MVP * position
10 gl_Position =  VP * M * vec4(vertexPosition_modelspace,1);
11
12 }
```

Kod 6.10 Množenje matrica u Shaderu

Sfera koju smo nacrtali izgleda trenutno ovako:



Slika 6.4 Sfera

Nakon što smo nacrtali sferu na nju možemo primijeniti algoritam za detekciju sudara koji smo definirali u prijašnjem poglavlju. Ovo sve do sada smo mogli i jednostavnije izvesti u OpenGL-u 1.x. Nakon što smo definirali sferu možemo konačno i definirati prave Shadere koji će tim sferama dati određena svojstva (svjetla, refleksija, prozirnost, ...).

6.2 Shaderi

U prethodnim poglavljima stalno smo spominjali Shadere bez da smo ih uopće definirali. Shaderi su programi koji počivaju na grafičkoj kartici i pokreću se za svaki dio grafičkog cjevovoda (eng. *pipeline*). U suštini, oni nisu ništa drugo nego programi koji pretvaraju određeni ulaz u definirani izlaz[4].

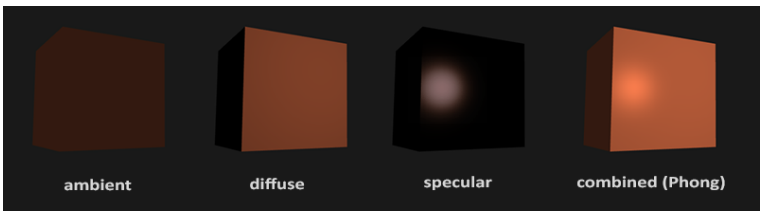
Do sada smo koristili jednostavne Shadere u kojima smo svakoj točki dali točno definiranu boju. Shaderi se u praksi koriste za puno kompleksnije stvari. U njima definiramo rezultirajuće boje za materijale, svjetla, teksture i sl. . U ovome radu definirali smo samo svjetla i refleksiju na kuglicama.

6.2.1 Svjetla

Svjetlo se može podijeliti na 3 komponente[4]:

- Difuzna komponenta
- Ambijentanla komponenta
- Zrcalna komponenta

Difuzna komponenta svjetla definira odbijanje zrake svjetla u svim smjerovima. Ambijentalna komponenta definira izvor svjetlosti na sceni, jer želimo izbjeći izračunavanje svjetlosti iz svake točke (s obzirom da svjetlo dolazi iz svih smjerova). I u konačnosti zrcalna komponenta svjetla definira refleksiju svjetlosti smjeru odbijanja svjetla od objekta[3]. Kombiniranjem ove 3 komponente svjetla dobiti ćemo konačno osvjetljenje na objektu.



Slika 6.5 Komponente svjetla prikazane na objektu[4]

6.3 Sjenčanje kuglica

U ovom, posljednjem poglavlju dodati ćemo Shadere koji će osvijetliti kuglice i kuglice koje će biti prozirne i reflektirati svjetlo. U detaljnu analizu samih Shadera nećemo ulaziti. Kodovi koji su napisani su dobro komentirani pa samo objašnjenje Shadera bi bilo suvišno. Shaderi su napisani po uzoru na literaturu[3].

```
#version 330 core
// Input vertex data, different for all executions of this shader.
layout(location = 0) in vec3 vertexPosition_modelspace;
layout(location = 2) in vec3 vertexNormal_modelspace;
// Output data ; will be interpolated for each fragment.
out vec3 Position_worldspace;
out vec3 Normal_cameraspace;
out vec3 EyeDirection_cameraspace;
out vec3 LightDirection_cameraspace;
// Values that stay constant for the whole mesh.
uniform mat4 VP;
uniform mat4 V;
uniform mat4 M;
uniform vec3 LightPosition_worldspace;
void main(){
  // Output position of the vertex, in clip space : MVP * position
  gl_Position =  VP * M * vec4(vertexPosition_modelspace,1);

  // Position of the vertex, in worldspace : M * position
  Position_worldspace = (M * vec4(vertexPosition_modelspace,1)).xyz;

  // Vector that goes from the vertex to the camera, in camera space.
  // In camera space, the camera is at the origin (0,0,0).
  vec3 vertexPosition_cameraspace = ( V * M * vec4(
      vertexPosition_modelspace,1)).xyz;
  EyeDirection_cameraspace = vec3(0,0,0) - vertexPosition_cameraspace;

  // Vector that goes from the vertex to the light, in camera space. M
      is ommited because it's identity.
  vec3 LightPosition_cameraspace = ( V * vec4(LightPosition_worldspace
      ,1)).xyz;
  LightDirection_cameraspace = LightPosition_cameraspace +
      EyeDirection_cameraspace;

  // Normal of the the vertex, in camera space
  Normal_cameraspace = ( V * M * vec4(vertexNormal_modelspace,0)).xyz;
      // Only correct if ModelMatrix does not scale the model ! Use its
      inverse transpose if not.
  }
}
```

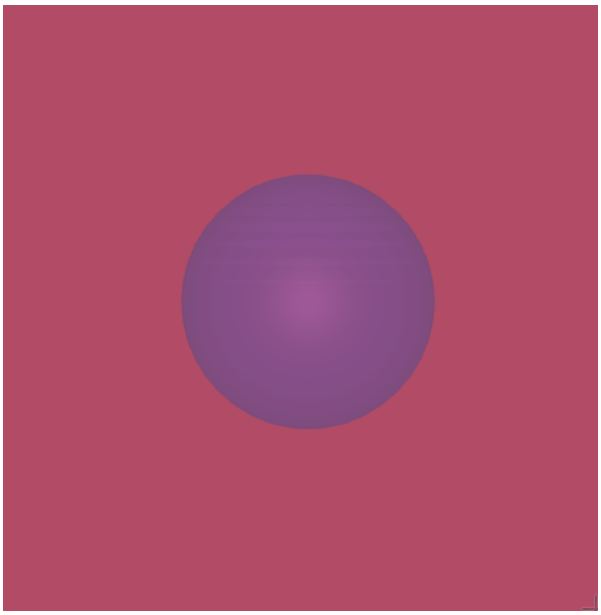
Kod 6.11 Konačni Shader točaka

```
#version 330 core

// Interpolated values from the vertex shaders
in vec3 Position_worldspace;
in vec3 Normal_cameraspace;
in vec3 EyeDirection_cameraspace;
in vec3 LightDirection_cameraspace;

// Ouput data
out vec4 color;

// Values that stay constant for the whole mesh.
uniform sampler2D myTextureSampler;
uniform mat4 MV;
uniform vec3 LightPosition_worldspace;

void main(){

  // Light emission properties
  // You probably want to put them as uniforms
  vec3 LightColor = vec3(0.8,0.4,0.6);
  float LightPower = 300.0f;

  // Material properties
  vec3 MaterialDiffuseColor = vec3(0.5,0.4,1);
  vec3 MaterialAmbientColor = vec3(0.2,0.7,0.6) * MaterialDiffuseColor;
  vec3 MaterialSpecularColor = vec3(0.4,0.3,0.7);

  // Distance to the light
  float distance = length( LightPosition_worldspace -
      Position_worldspace );

  // Normal of the computed fragment, in camera space
  vec3 n = normalize( Normal_cameraspace );
  // Direction of the light (from the fragment to the light)
  vec3 l = normalize( LightDirection_cameraspace );
  // Cosine of the angle between the normal and the light direction,
  // clamped above 0
  //  - light is at the vertical of the triangle -> 1
  //  - light is perpendicular to the triangle -> 0
  //  - light is behind the triangle -> 0
  float cosTheta = clamp( dot( n,l ), 0,1 );
```

```
42
43    // Eye vector (towards the camera)
44    vec3 E = normalize(EyeDirection_cameraspace);
45    // Direction in which the triangle reflects the light
46    vec3 R = reflect(-l,n);
47    // Cosine of the angle between the Eye vector and the Reflect vector,
48    // clamped to 0
49    //  - Looking into the reflection -> 1
50    //  - Looking elsewhere -> < 1
51    float cosAlpha = clamp( dot( E,R ), 0,1 );
52
53    color.rgb =
54    // Ambient : simulates indirect lighting
55    MaterialAmbientColor +
56    // Diffuse : "color" of the object
57    MaterialDiffuseColor * LightColor * LightPower * cosTheta / (distance
          *distance) +
58    // Specular : reflective highlight, like a mirror
59    MaterialSpecularColor * LightColor * LightPower * pow(cosAlpha,5) / (
          distance*distance);
60    //Transparency
61    color.a = 0.2;
62  }
```

Kod 6.12 Konačni Shader fragmenata

Nakon dodavanja ovih Shadera kuglica izgleda ovako:



Slika 6.6 Konačni izgled kuglice sa dodanom prozirnošću

6.4 Zaključno o sjenčanju

Sama svrha ovoga rada nije bilo dodavanje Shadera. Ovo je dodatni dio koji će samo definirati izgled kuglica i nema nikakve ovisnosti o samom algoritmu za detekciju sudara. Primijetimo da kuglica6.6 nije savršena, ali za naše potrebe je bila dovoljna. Sama implementacija OpenGL-a 3.3 oduzela je dosta vremena, no na kraju dobili smo rad koji ima nekakav izgled, a ne samo bijele kuglice na crnoj podlozi. Shaderi su moćan alat kojime se može ubrzati i broj sličica po sekundi ako su dobro definirani[3]. U našem slučaju samo crtanje nam nije ubrzalo algoritam pa je još uvijek crtanje sa velikim brojem kuglica bilo problematično zbog složenosti algoritma za detekciju sudara.

Poglavlje 7

Zaključak

Zaključno, možemo reći da postoje brojne metode za detekciju sudara. U ovom radu obradili smo tek 2 algoritma koji su zapravo vrlo jednostavni. Svaki od ovih algoritama ima svoje prednosti i nedostatke te je bilo zanimljivo usporediti brzine i točnosti detekcije sudara između algoritama. Možemo zaključiti da je K-d stablo, algoritam koji ćemo koristiti onda kada na sceni postoji puno statičnih objekata i kada stablo možemo izgraditi unaprijed, tj. prije samo iscrtavanja objekata. AABB stablo ćemo, za razliku od K-d stabla, primijeniti onda kada imamo puno dinamičnih objekata jer u svakom koraku ne moramo graditi stablo. AABB stablo se pokazalo vrlo jednostavnim za implementirati i pokazuje vrlo visoku efikasnost pri samoj detekciji sudara. Iz tog razloga sve je popularnije u modernim aplikacijama u kojima je potrebna detekcija sudara. Ipak, K-d stablo je popularno u igrama jer pokazuju vrlo visoke perfomanse ako nemamo potrebu graditi stablo u svakom koraku.

U konačnosti, dodavanje Shadera na ova 2 algoritma pokazuje kako je detekcija sudara zapravo vrlo dobar paralelni algoritam gdje samu detekciju možemo izvršavati na procesoru, dok crtanje objekata možemo dati grafičkoj kartici na obradu.

I konačno, ne možemo se procijeniti koji je algoritam za detekciju sudara najbolji. U praksi ćemo analizom algoritama odabrati onaj, koji najbolje odgovara zahtjevima naše aplikacije i sredstvima koja je potrebno uložiti u taj algoritam.

Bibliografija

[1] D. Clingman, *Practical Java Game Programming (Charles River Media Game Development)*. Jenifer Niles, 2004.

[2] T. Diewald. Thomas diewald blog. , s Interneta, http://thomasdiewald.com/blog/?p=1488 19.8.2017.

[3] Opengl tutorial. , s Interneta, http://www.opengl-tutorial.org/ 9.9.2017.

[4] Learn opengl. , s Interneta, https://learnopengl.com/ 9.9.2017.

[5] C. Ericson, *Real-Time Collision Detection*, T. Cox, Ed. CRC Press, 2004.

[6] Kd tree, wikipedia. , s Interneta, https://en.wikipedia.org/wiki/K-d_tree 27.8.2017.

[7] G. V. D. Bergen, "Efficient collision detection of complex deformable models using aabb trees," 1998.

[8] Introductory guide to aabb tree collision detection. , s Interneta, http://www.azurefromthetrenches.com/introductory-guide-to-aabb-tree-collision-detection/ 21.8.2017.

[9] C. Berchek, "2-dimensional elastic collisions without trigonometry," 2009.

[10] M. Martinčić. Elastični i neelastični sudari. , s Interneta, http://dominis.phy.hr/~mmarko/SAMP/Seminar2/Seminar2.html 22.8.2017.

[11] Slobodni pad, wikipedia. , s Interneta, https://hr.wikipedia.org/wiki/Slobodni_pad 24.8.2017.

[12] Zakon očuvanja energije, wikipedia. , s Interneta, https://hr.wikipedia.org/wiki/Zakon_o%C4%8Duvanja_energije 24.8.2017.

[13] Količina gibanja, wikipedia. , s Interneta, https://hr.wikipedia.org/wiki/Koli%C4%8Dina_gibanja 24.8.2017.

[14] Elastični sudari, wikipedia. , s Interneta, https://en.wikipedia.org/wiki/Elastic_collision 24.8.2017.

Sažetak

U ovome radu obrađuju se algoritmi za detekciju sudara, njihova implementacija u C++ programskom jeziku i usporedba istih na temelju jednostavnih kuglica. Cilj algoritama je da budu što efikasniji i da detektiraju sve sudare na sceni. Algoritme možemo podijeliti u nekoliko skupina, no obrađeni algoritmi su: algoritam za podjelu objekta, algoritam za podjelu prostor. Sudari između kuglica definirani su kao elastični sudari u kojima nema gubitka energije, ali se energija prenosi s jedne kuglice na drugu. Dodana je gravitacija kao vanjska sila koja djeluje na cijeloj sceni, pa kuglice cijelo vrijeme padaju i sudaraju se s nevidljivim zidom. U konačnosti, dodano je sjenčanje kuglica kojime se postigao bolji vizualni dojam samog rada.

U početku, praktični dio rada izvršen je u alatu OpenGL 1.x, dok je za sjenčanje bilo potrebno prijeći na OpenGL 3.3 .

Ključne riječi **— detekcija sudara, particioniranje prostora, Hijerarhija ograničavajućih volumena, elastični sudari, sjenčanje**

Abstract

The paper deals with collision detection algorithms, their implementation in the C ++ programming language, and their comparison in simple balls. The goal of algorithms is to be as effective as possible and to detect all collisions on the scene. Algorithms can be divided into several groups, without the processed algorithms are: bouning volume hierarchy algorithms, space partitioning algorithms. Colliding between the balls is defined as elastic collisions in which there is no loss of energy, but the energy is transmitted from one ball another. Gravity was added as an external force acting on the whole scene, so the balls are falling all the time and colliding with the invisible wall. In conclusion, ball shading was added to achieve a better visual appearance of the animation itself.

Initially, the practical part of the paper was done in OpenGL 1.x, while for shading it was necessary to switch to OpenGL 3.3.

Keywords **— collision detection, space partitioning, bounding volume hierarchy, elastic collisions, shading**