Detekcija kolizija

Parać, Ana

Master's thesis / Diplomski rad

2020

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:217:322782

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Ana Parać

DETEKCIJA KOLIZIJA

Diplomski rad

Voditelj rada:
prof. dr. sc. Mladen Jurak

Zagreb, velječa 2020.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Zahvaljujem mojim roditeljima i sestri na bezuvjetnoj potpori i strpljenju tijekom studija te mentoru prof. dr. sc. Mladenu Juraku na velikoj pomoći i strpljenju pri izradi diplomskog rada.

Sadržaj

Uvod

Detekcija kolizija izuzetno je važna tema prisutna u različitim područjima kao što su računalne igre, robotika, grafičke aplikacije, planiranje puta i kretanja vozila odnosno letećih objekata, simulacija virtualne stvarnosti itd. Zbog svoje iznimne važnosti i široke primjene, još uvijek je predmet istraživanja. Za primjenu u praksi razvijeni su algoritmi veće ili manje složenosti odnosno efikasnosti. Koji algoritmi će biti primijenjeni u praksi ovisi o tome kakvu vrstu aplikacije razvijamo tj. u kolikoj mjeri je važna preciznost detekcije kolizije u odnosu na brzinu. Ukoliko se radi o računalnoj igri u kojoj nije presudna preciznost utvrđivanja kolizije dok je izuzetno važan brz odgovor na koliziju cilj je izbjeći izvođenja sporih i zahtjevnijih testova koji uključuju ispitivanje kolizije na samoj geometriji objekata. Objekti se obuhvaćaju jednostavnijim geometrijskim likovima odnosno tijelima, konveksnom ljuskom i slično. Što je obuhvatni lik složeniji, procjena kolizije je točnija, međutim raste i složenost testiranja. U ovom radu opisat ćemo osnovne algoritme za detekciju kolizija polazeći od onih danih u [7] u 2D, te ćemo proučiti njihovo proširenje na 3D. Uz to, razmotrit ćemo metode i algoritme koji se primjenjuju za optimizaciju testa na objektima velike složenosti. Uz detekciju kolizija vežemo i pridružene probleme kao što su računanje kontaktnog skupa, udaljenosti, vremena prvog kontakta i slično. U daljnjem razmatranju bazirat ćemo se na konveksne objekte. Problem detekcije kolizija za nekonveksne objekte može se riješiti razlaganjem objekta na manje konveksne objekte te primjenom algoritma na parove dobivenih manjih objekata. U daljnjem tekstu ćemo proučiti algoritme za navedene probleme te ćemo demonstrirati njihov rad koristeći biblioteke Box2D i Sflm.

Poglavlje 1

Osnovna ideja i vrste algoritama

U današnje vrijeme razvijen je velik broj aplikacija koje simuliraju realni svijet sa svim njegovim zakonitostima. Glavni aspekt realnog svijeta na kojem se temelje aplikacije virtualne stvarnosti je taj da u nijednom trenutku dva čvrsta objekta ne mogu zauzeti istu točku prostora. Ovu pojavu nazivamo *kolizija*. G. van der Bergen u [14] definira koliziju na sljedeći način: *"Kolizija je konfiguracija dvaju objekata koji zauzimaju istu točku u prostoru."* Ako želimo da se simulacija ponaša prema zakonima stvarnog svijeta, potrebno je kreirati mehanizam koji će osiguravati da se ta pojava ne događa. Takav mehanizam nazivamo sustav za detekciju kolizija. Primarni cilj sustava za detekciju kolizija je utvrditi postojanje kolizije između dva ili više objekata te izračunati odgovor na koliziju.

Algoritmi detekcije kolizija moraju biti dizajnirani tako da poštuju sljedeća ograničenja:

- **brzina odgovora na upit** - odgovor na upit o koliziji mora se dati u zadanom vremenskom okviru
- **floating-point format** - kod aritmetičkih operacija moguće je dobiti greške zaokruživanja te time i pogrešno detektirati odnosno ispustiti pojavu kolizije
- **ograničenje dostupnosti memorije** - neke metode mogu zahtijevati značajno više memorije od ostalih funkcija aplikacije te je njihovo korištenje potrebno posebno razmotriti.

1.1 Vrste algoritama za detekciju kolizija

Algoritme za detekciju kolizija dijelimo na diskretne i kontinuirane. Diskretni algoritmi rade na principu diskretnih vremenskih intervala. Algoritam provjerava postojanje kolizije u malim vremenskim intervalima na stacionarnim objektima te ako utvrdi da se objekti

preklapaju ili je udaljenost između njih unutar zadane vrijednosti tolerancije prijavljuje koliziju. Algoritmu diskretne detekcije kolizije dovoljno je proslijediti listu objekata te on ne koristi varijblu vremena na direktan način već promatra odnose objekata na rubovima zadanih intervala. Nedostatak je to što je teško utvrditi optimalnu duljinu vremenskih intervala te nije moguće izračunati točno vrijeme kolizije pošto algoritam detektira koliziju kada se ona već dogodi. Za razliku od diskretnih algoritama, kontinuirani algoritmi računaju mogućnost postojanja kolizije prije iscrtavanja nove scene tj. predviđaju vrijeme kolizije bazirano na udaljenosti objekata i predviđenoj putanji. Iako su kontinuirani algoritmi pouzdaniji od diskretnih, vremenski i prostorno su zahtjevniji. O kontinuiranoj detekciji kolizija govorimo u poglavlju 4.

Osim utvrđivanja preklapanja odnosno dodira dvaju ili više objekata, detekcija kolizija može davati informacije o točnom vremenu kolizije te preklapajućem skupu odnosno dijelovima objekata koji su u koliziji. Uz to razlikujemo algoritme koji se primjenjuju na stacionarnim i na objektima u pokretu. Eberly u [7] kategorizira algoritme za detekciju kolizija s obzirom na tip objekata i informacije koje želimo na:

- algoritme za stacionarne objekte:
  - za testiranje kolizije
  - za pronalaženje kontaktnog skupa
- algoritme za objekte u pokretu:
  - za testiranje kolizije
  - za pronalaženje kontaktnog skupa

Metode kojima se koristimo pri utvrđivanju kolizije između objekata dijelimo na one bazirane na udaljenosti i one bazirane na presjeku. Oba skupa metoda imaju prednosti i mane. Zbog toga se kao alternativa koristi funkcija pseudoudaljenosti. Ona kombinira prednosti obje metode te ju je uz to lakše implementirati od dviju prethodnih metoda. Funkcija pseudoudaljenosti je pozitivna kada su objekti razdvojeni, 0 kada se objekti dodiruju odnosno negativna kada se objekti preklapaju. Popularna metoda bazirana na funkciji pseudoudaljenosti je tzv. metoda razdvajajućih osi o kojoj govorimo u 2.2.

Poglavlje 2

Detekcija kolizija jednostavnih konveksnih tijela

Na početku ćemo proučiti nekoliko osnovnih geometrijskih likova i kako se utvrđuje postojanje kolizije između njih.

2.1 Jednostavni testovi

2.1.1 Točka - krug

Za potrebe ovog algoritma koristit ćemo sljedeće strukture:

Algoritam 2.1: Point

```
1
2 class Point
3 {
4   public:
5     float x = 0.f;
6     float y = 0.f;
7 };
```

Algoritam 2.2: Circle

```
1
2 class Circle
3 {
4   public:
5     Circle() : radius(1.0), center(0.0,0.0) {}
6     Circle(float radius_ , Point center_):
7         radius(radius_), center(center_) {}
```

```
8     protected:
9       float radius;
10      Point center;
11
12  };
```

Neka je K krug i p točka u dvodimenzionalnom Euklidskom prostoru. Tada kažemo da se točka nalazi unutar kruga ako je njezina udaljenost od središta kruga manja ili jednaka radijusu kruga, odnosno ako vrijedi uvjet:

$$||p - K.center|| \leq K.radius \tag{2.1}$$

2.1.2 Krug - krug

Ispitivanje kolizije između dva kruga također je jednostavan test koji primjenjuje logiku sličnu prethodnom primjeru. Dva kruga K_1 i K_2 se sijeku ako je udaljenost između njihovih središta manja ili jednaka zbroju njihovih radijusa tj. ako vrijedi:

$$||K_1.center - K_2.center|| \leq K_1.radius + K_2.radius \tag{2.2}$$

Algoritam 2.3: CircleCircle

```
1  bool CircleCircle(Circle C1, Circle C2)
2  {
3      float sumRadius = (C1.radius + C2.radus)^2;
4      float distCenter = (C1.center.x - C2.center.x)^2 +
5                         (C1.center.y - C2.center.y)^2;
6      return distCenter <= sumRadius ? true : false;
7  }
```

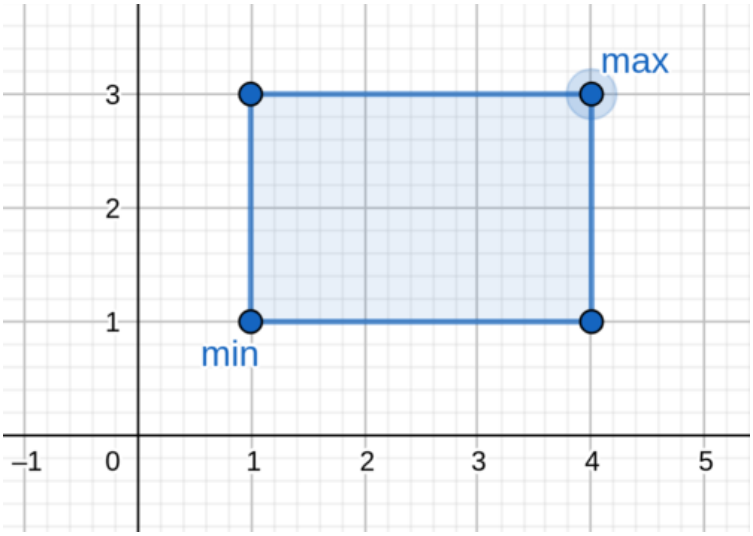
U prethodnom algoritmu koristimo kvadrat zbroja radijusa kako bismo izbjegli vremenski skuplju operaciju pronalaženja drugog korijena.

2.1.3 Točka - pravokutnik

Prvo konstruiramo klasu *Rectangle* koja predstavlja pravokutnik čije su stranice paralelne s koordinatnim osima. Takve pravokutnike zadajemo dvjema točkama od kojih je jedna predstavlja njegovu minimalnu točku, a druga je maksimalna točka. Pravokutnike čije stranice nisu paralelne s koordinatnim osima možemo predstaviti pomoću četiri točke koje predstavljaju njihove vrhove.

Algoritam 2.4: Rectangle

```
1
2  class Rectangle
3  {
4    public:
5      Rectangle() : min(0.0, 0.0), max(1.0,1.0) {}
6      Rectangle(Point min_, Point max_) : min(min_), max(max_){}
7    protected:
8      Point min;
9      Point max;
10
11 }
```



Slika 2.1: Pravokutnik određen minimalnom i maksimalnom točkom

Točka p se nalazi unutar pravokutnika R ako su zadovoljeni sljedeći uvjeti:

$$R.min.x \leq p.x \leq R.max.x \tag{2.3}$$

$$R.min.y \leq p.y \leq R.max.y \tag{2.4}$$

2.1.4 Pravokutnik - pravokutnik

Promotrimo slučaj u kojem su bridovi oba pravokutnika paralelni s koordinatnim osima. Neka su A i B dva takva pravokutnika. Pravokutnici su određeni svojom minimalnom

i maksimalnom točkom odnosno vrhom. Slijedi da se dva pravokutnika preklapaju ako vrijede uvjeti:

$$B.min \leq A.max \tag{2.5}$$

$$A.min \leq B.max \tag{2.6}$$

Potrebno je provjeriti preklapanje na obje osi koordinatnog sustava.

Algoritam 2.5: RectangleRectangle

```
1
2  bool RectangleRectangle(Rectangle r1, Rectangle r2)
3  {
4      Point min_r1 = GetMin(r1);
5      Point min_r2 = GetMin(r2);
6      Point max_r1 = GetMax(r1);
7      Point max_r2 = GetMax(r2);
8      return ((min_r1.x <= max_r2.x) && (min_r2.x <= max_r1.x))
9          && ((min_r1.y <= max_r2.y) && (min_r2.y <= max_r1.y));
10 }
```

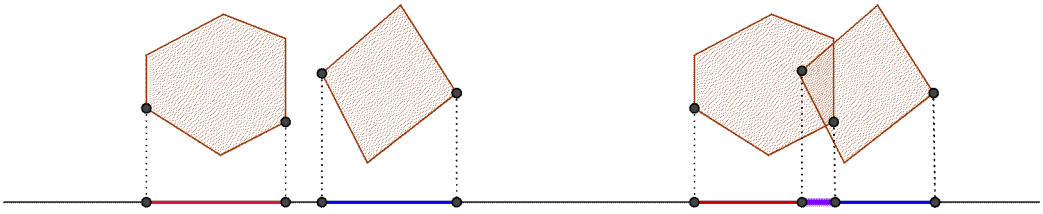
Kod pravokutnika čije stranice nisu paralelne koordinatnim osima ne možemo primijeniti prethodni algoritam. Kod takvih pravokutnika biti će potrebno usporediti položaje svih vrhova. Za sve kompleksnije oblike uvodimo testiranje metodom razdvajajućih osi.

2.2 Metoda razdvajajućih osi

Linija razdvajanja dvaju objekata u 2D je pravac koji dijeli ravninu na dva dijela takav da se jedan objekt nalazi u potpunosti u jednoj otvorenoj poluravnini, a drugi u potpunosti u drugoj otvorenoj poluravnini. Neka je P razdvajajući pravac za objekte A i B i n normala na taj pravac. Tada je n os razdvajanja objekata A i B. Bez smanjenja općenitosti pretpostavimo da je $A \in P^{\oplus}$, a $B \in P^{\ominus}$. Tada vrijedi $n \cdot x > n \cdot y$ za sve $x \in A$ i sve $y \in B$. Na sličan način definiramo ravninu razdvajanja s pripadnom normalom u 3D. Metoda razdvajajuće osi definira se na sljedeći način: ako postoji pravac takav da se projekcije dvaju konveksnih objekata na njega ne sijeku, tada objekti nisu u koliziji.

Bergen u [13] dokazuje da za nekolidirajuće konveksne objekte uvijek postoji os razdvajanja. Potrebno je pokazati da je $a - b$ jedna os razdvajanja za par nekolidirajućih konveksnih objekata A i B gdje su $a \in A$ i $b \in B$ međusobno najbliže točke. Prvo dokazujemo dvije pomoćne leme.

Lema 2.2.1. *Neka su $v \in \mathbb{R}^n$ i $w \in \mathbb{R}^n$ točke. Pretpostavimo da vrijedi $v \cdot w - \|v\|^2 < 0$. Tada dužina koja povezuje v i w sadrži točku u za koju vrijedi $\|u\| < \|v\|$.*



Slika 2.2: Testiranje kolizije metodom razdvajajućih osi - lijevo: poligoni se ne sijeku, dani pravac je jedna razdvajajuća os; desno: objekti su u koliziji

Dokaz. Neka je $u = v + \lambda(w - v)$. Tada se u nalazi na dužini određenoj s v i w za $0 \leq \lambda \leq 1$. Vrijedi $\|u\|^2 - \|v\|^2 = 2\lambda v \cdot (w - v) + \lambda^2 \|w - v\|^2$. Izjednačimo li $\|u\|^2 - \|v\|^2$ s nula, dobivamo korijene $\lambda_1 = 0$ i $\lambda_2 = -2v \cdot (w - v)/\|w - v\|^2$. Iz uvjeta $v \cdot w - \|v\|^2 < 0$ slijedi da je $\lambda_2 > 0$. Pošto je $\|w - v\|^2 > 0$ vrijednost $\|u\|^2 - \|v\|^2$ bit će pozitivna kada λ teži u ∞. Stoga je $\|u\|^2 - \|v\|^2 < 0$ za $\lambda_1 < \lambda < \lambda_2$. □

Lema 2.2.2. *Neka je $C \subset \mathbb{R}^n$ konveksan objekt i $v \in C$ točka najbliža ishodištu. Tada je $v = 0$ ili $v \cdot w > 0$ za sve $w \in C$.*

Dokaz. Promotrimo netrivijalan slučaj, kada $v \neq 0$. Neka je $w \in C$. Pošto je C konveksan slijedi da je svaka točka $(1 - t)v + tw, t \in [0, 1]$ također u C. Kako je v najbliža ishodištu slijedi da je $\|u\| \geq \|v\|$. Pošto za svaki k na dužini određenoj s u i w vrijedi $\|k\| \geq \|v\|$, prema lemi 2.2.1, slijedi $v \cdot w - \|v\|^2 \geq 0$. Stoga imamo $v \cdot w \geq \|v\|^2 > 0$ za sve $w \in C$. □

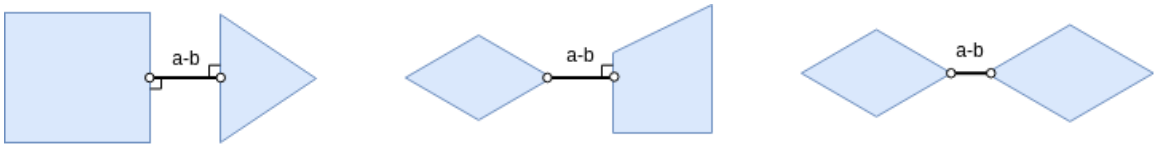
Prije dokaza teorema definirat ćemo udaljenost konveksnih objekata A i B:

$$d(A, B) = min\{\|a - b\| \mid a \in A, b \in B\}.$$

Par najbližih točaka dan je s $\|a - b\| = d(A, B)$ za $a \in A$ i $b \in B$. Uz to očito vrijedi: $d(A, B) = 0$ ako i samo ako $A \cap B \neq \emptyset$.

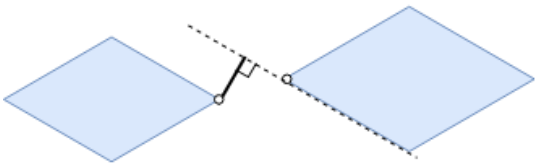
Teorem 2.2.3. *Neka su $A \subset \mathbb{R}^n$ i $B \subset \mathbb{R}^n$ konveksni te $a \in A$ i $b \in B$ par najbližih točaka. Tada je $A \cap B \neq \emptyset$ ili je $a - b$ os razdvajanja objekata A i B.*

Dokaz. Pretpostavimo da je $A \cap B = \emptyset$. Tada $a - b \neq 0$. Pošto su a i b takve da je njihova međusobna udaljenost u C minimalna, slijedi da je $a - b$ točka Minkowskijeve sume $A - B = \{x - y \mid x \in A, y \in B\}$ najbliža ishodištu. Pošto je $A - B$ konveksan, iz leme 2.2.2 slijedi da je $(a - b) \cdot v > 0$ za $\forall v \in A - B$. Neka su $x \in A$ i $y \in B$ proizvoljni. Tada je $(a - b) \cdot (x - y) > 0$ te slijedi $(a - b) \cdot x > (a - b) \cdot y$. Stoga je $a - b$ razdvajajuća os za A i B. □



Slika 2.3: Slučajevi međusobno najbližih točaka

Ako su $a \in A$ i $b \in B$ dvije međusobno najbliže točke poligona A i B u 2D, tada znamo da se one nalaze u vrhu ili na stranici odgovarajućih poligona. Promotrimo slučaj kada se jedna od točaka nalazi na stranici jednog poligona, a druga u vrhu drugog poligona. Točka drugog poligona jedinstveno je određena vrhom dok je njoj najbliža točka prvog poligona dobivena spuštanjem okomice na odgovarajuću stranicu prvog poligona iz vrha drugog poligona. U slučaju da su međusobno najbliže točke obje na stranicama, odgovarajuće stranice su paralelne pa one nisu jedinstveno određene. U takvom slučaju međusobno najbliže točke određuju pravac ortogonalan na obje stranice. Slijedi da će u slučajevima gdje je barem jedna od najbližih točaka na stranici, razdvajajuća os iz teorema 2.2.3 biti ortogonalna na stranicu barem jednog od poligona. Preostali slučaj je kada su obje točke u vrhovima u kojem ortogonalnost ne vrijedi. Tada možemo promatrati vrh jednog od poligona kao dio njegove stranice te pronaći okomicu na pravac određen tom stranicom kroz vrh drugog poligona kao na slici 2.4. Tako dobivena točka neće pripadati poligonu kod kojeg promatramo stranicu, ali dobivena okomica također će razdvajati projekcije objekata. Ova činjenica utjecat će na odabir skupa osi razdvajanja na kojima će se provoditi testiranje.



Slika 2.4: Alternativna os razdvajanja za slučaj dvaju najbližih vrhova

Os $a - b$ iz dokaza teorema 2.2.3 nije jedinstvena pošto točke a i b s međusobno najmanjom udaljenosti ne moraju biti jedinstvene. Pri testiranju kolizije između para objekata metodom razdvajajućih osi postoji beskonačno mnogo osi koje možemo testirati. Zbog toga je ključno utvrditi koje osi je nužno ispitati kako bi algoritam bio efikasan. Zbog svojstva ortogonalnosti vektora $a - b$ slijedi da je u dvodimenzionalnom prostoru kao moguće razdvajajuće osi dovoljno promatrati vektore normala na stranice poligona. Vektor normale na stranice poligona orijentiran je kao na slici 2.6.

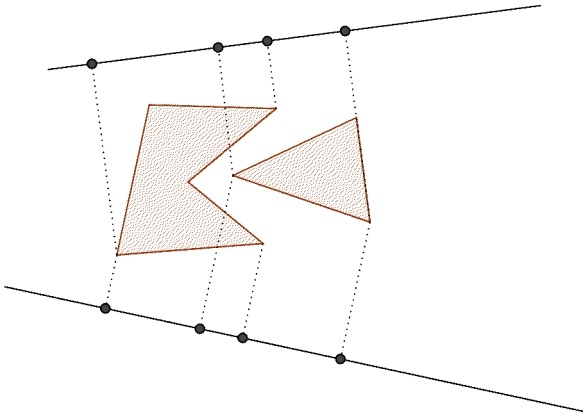
U trodimenzionalnom prostoru kao moguće razdvajajuće osi promatramo vektore normala na stranice poliedra te uz to i vektore dobivene vektorskim produktom dvaju bridova od kojih je svaki od jednog od dvaju poliedara. Razlog zašto koristimo tako odabrane vektore kandidate za moguće razdvajajuće osi je mogućnost postojanja dodira između poliedara bez presjeka. Dva konveksna poliedra mogu doći u kontakt na šest različitih načina. Mogući kontakti su:

- stranica - stranica,
- stranica - brid,
- stranica - vrh,
- brid - brid,
- brid - vrh,
- vrh - vrh.

U [8] Ericson navodi kako je vrhove moguće promatrati kao dijelove bridova te stoga reducira skup kontakata koje analizira na kontakte stranica - stranica, stranica - brid i brid - brid. Slijedi da je za slučajeve stranica - stranica i stranica - brid, kao moguće razdvajajuće osi, dovoljno testirati normale stranica poliedara, dok je za slučaj brid - brid dovoljno testirati vektorske produkte parova bridova. Kada dva brida dođu u kontakt, njihovi vektori smjera definirali bi ravninu čija normala bi određivala ravninu razdvajanja ako razmaknemo poliedre za malu vrijednost ϵ.

Metoda razdvajajućih osi iznimno je korisna pri implementaciji raznih testova kolizije jer pomoću nje možemo utvrditi postojanje kolizije između proizvoljnih objekata na vrlo jednostavan način. Nužno je da su oba objekta konveksna. Ako je neki od objekata nekonveksan, metoda razdvajajućih osi mogla bi dati pogrešan rezultat. Primjer takvog slučaja vidimo na slici 2.5. Metoda razdvajajućih osi zasniva se na pronalaženju jedne osi na kojoj su projekcije objekata razdvojene. Kada smo utvrdili skup potencijalnih osi razdvajanja, pristupamo testiranju svake od njih dok ne pronađemo jednu na kojoj se projekcijski intervali ne sijeku ili testiranjem svih osi utvrdimo da nijedna od njih ne razdvaja projekcije objekata. Tada zaključujemo da su objekti u koliziji. Jedna iteracija algoritma sastoji se od projiciranja objekata na odabranu potencijalnu os, utvrđivanja granica projekcijskih intervala te utvrđivanja sijeku li se dobiveni intervali. Kažemo da se intervali preklapaju ako vrijede sljedeći uvjeti:

$$I_1.min \leq I_0.max$$
$$I_0.min \leq I_1.max.$$



Slika 2.5: Problem nekonveksnih poligona

Prednosti

Detekcija kolizija primjenom teorema razdvajajućih osi je brza i daje visoku točnost. Posebno se ističu prednosti kod primjene na objekte u pokretu. Metoda osigurava jednostavan račun te se, nakon što je pronađena jedna os koja razdvaja projekcijske intervale objekata, može eliminirati daljnje testiranje potencijalnih razdvajajućih osi. Osim toga, moguće je ubrzati uzastopno testiranje kolizije tako da se zapamti razdvajajuća os u trenutnom koraku te se u idućem koraku prvo provodi testiranje te osi. Ovakva optimizacija zaista će biti učinkovita kod objekata u pokretu zahvaljujući vremenskoj i prostornoj koherentnosti objekata između testiranja.

Nedostatci

Ako su u provođenju testa kolizije koji primjenjuje metodu razdvajajućih osi dva brida danih objekata u 3D međusobno paralelna, njihov vektorski produkt bit će nulvektor. Slijedi odbacivanje te osi tj. kolizija će biti potvrđena za tu os iako objekti ne moranju biti u koliziji.

2.3 Utvrđivanje projekcijskih intervala

Označimo s $P_i, i = 1, ..., n$ vrhove poligona u 2D odnosno poliedra u 3D. Najjednostavniji način pronalaženja projekcijskih intervala sastoji se od projiciranja svih vrhova konveksnog poligona odnosno poliedra na os razdvajanja te traženja ekstrema koji će odrediti interval projekcije $[m, M]$. Neka je D vektor smjera razdvajajuće osi. Tada je potrebno

pronaći vrh P_{min} i P_{max} za koje vrijedi

$$m = D \cdot P_{min} = min_{0 \leqslant k \leqslant n-1}\{D \cdot P_k\}$$

$$M = D \cdot P_{max} = max_{0 \leqslant k \leqslant n-1}\{D \cdot P_k\}$$

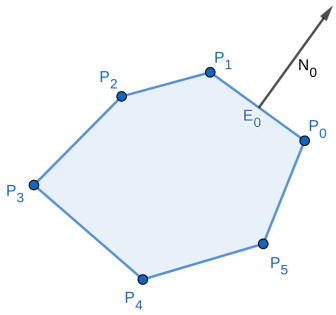
Algoritam dobiven ovom metodom za n vrhova rezultirat će s n projekcija točaka na pravac te će imati linearnu vremensku složenost.
U [7] Eberly navodi postupak kojim je moguće pronaći projekcije u vremenu $O(logn)$ konstrukcijom binarnog stabla i metodom bisekcije. Nakon provedenih mjerenja, Eberly utvrđuje kako je isplativije koristiti navedene optimizacije za poligone koji imaju više od n vrhova gdje je n između 32 i 64. Uz to, pri korištenju optimizacije s binarnim stablom potrebno je paziti da stablo bude balansirano tj. da su sve grane stabla podjednake dubine. Nebalansirano stablo, u najgorem slučaju, ponovno će rezultirati algoritmom linearne vremenske složenosti.

2.4 Ispitivanje postojanja kolizije na statičkim objektima

Prvo ćemo predstaviti algoritme za utvrđivanje postojanja kolizije za proizvoljne konveksne objekte u 2D. Zatim ćemo proširiti ideju na objekte u 3D.

2.4.1 2D

Označimo s $C_i, i = 0, 1$ konveksne poligone na koje primjenjujemo algoritam. Neka su $P^i_j, j = 0, ..., n_i$ vrhovi odgovarajućih poligona te $E^i_j = P^i_{j+1} - P^i_j, j = 0, ..., n_i$ vektori smjera bridova danih poligona. Normale na stranice poligona označimo s $N^i_j, j = 0, ..., n_i$ i odabiremo orijentaciju tako da vrijedi $(N^i_j)^\perp \cdot E^i_j > 0$ gdje je $(x, y)^\perp = (-y, x)$. Odabrana orijentacija prikazana je na slici 2.6.



Slika 2.6: Konveksni poligon

Kao što smo već napomenuli u 2.2 kao vektore D promatramo normale na stranice poligona. Algoritam za odabrani D nalazi projekcije vrhova objekata te određuje ekstremne vrijednosti. Nakon toga uspoređuje dobivene intervale odnosno njihove minimalne i maksimalne vrijednosti. Neka su I_0 i I_1 pronađeni intervali projekcija objekata C_0 i C_1 na os D. Ako su vrijedi $I_0.max < I_1.min$ ili $I_1.max < I_0.min$ tada je D os razdvajanja za C_0 i C_1 te algoritam staje.

Prvo ćemo kreirati potrebnu strukturu podataka koja će sadržavati podatke o poligonima.

Algoritam 2.6: ConvexPolygon

```
1  class ConvexPolygon{
2      protected:
3          int numVertices;
4          Point* Vertices;
5      public:
6          int GetVerticesNum();
7          Point GetVertex(int i);
8          Vector GetEdge(int i);
9          Vector GetNormal(int i);
10 };
```

Slijedi algoritam kreiran naivnim pristupom koji ispituje svaku od normala te projicira sve vrhove svakog od poligona na potencijalnu os razdvajanja. Funkcija `FindInterval` pronalazi projekcijski interval danog poligona te daje njegove granične vrijednosti.

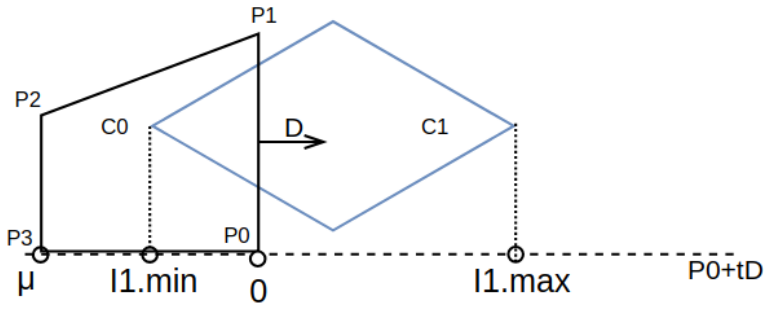
Algoritam 2.7: TestCollision2D - neoptimalni algoritam

```
1  bool TestCollision2D(ConvexPolygon C0, ConvexPolygon C1){
2
3      Point c0Min, c0Max, c1Min, c1Max;
4
5      //testiraj normale C0
6      for(int i = 0; i < C0.GetVerticesNum()-1; i++){
7          Vector normal = C0.GetNormal(i);
8          FindInterval(normal, C0, &c0Min, &c0Max);
9          FindInterval(normal, C1, &c1Min, &c1Max);
10
11         if(c0Max < c1Min || c1Max < c0Min ){
12             //nema kolizije - pronadena je os razdvajanja
13             return false;
14         }
15     }
16
17     //testiraj normale C1
18      for(int i = 0; i < C1.GetVerticesNum()-1; i++){
19         Vector normal = C1.GetNormal(i);
```

```
20            FindInterval(normal, C0, &c0Min, &c0Max);
21            FindInterval(normal, C1, &c1Min, &c1Max);
22
23            if(c0Max < c1Min || c1Max < c0Min ){
24                return false;
25            }
26        }
27        //nije nadena os razdvajanja
28        return true;
29    }
```

Navedeni algoritam u najgorem slučaju morat će provjeriti svih n_0 normala poligona C_0 i svih n_1 normala poligona C_1. Na potencijalnu os projiciramo sve vrhove oba poligona što daje vremensku složenost $O(n)$ za n vrhova poligona. Stoga je ukupna vremenska složenost kvadratna.

Eberly u [7] navodi mogući način optimizacije testa kolizije u 2D u kojem ne pamti sve ekstreme projekcija dvaju objekata koje testiramo, nego samo maksimum prvog i minimum drugog objekta. Naime, projiciranjem C_0 na pravac $P_0 + t \cdot D$, gdje je D normala na stranicu poligona C_0 određenu vrhovima P_0 i P_1, dati će interval $[\mu, 0]$, gdje je $\mu < 0$.



Slika 2.7: Optimizacija detekcije kolizija 2D

Neka je $[I_1.min, I_1.max]$ projekcijski interval koji se dobiva projiciranjem C_1 na isti pravac. Kako bismo testirali razdvojenost uspoređujemo maksimum projekcije C_0 i minimum projekcije C_1. Test provjerava vrijedi li uvjet $I_1.min > 0$ te ako je on ispunjen slijedi da je pronađena os razdvajanja. Vrijednost μ nije potrebna za test te stoga nije potrebno projicirati vrhove objekta C_0 na potencijalnu os razdvajanja. Također, pošto test ne koristi vrijednost maksimuma projekcije objekta C_1, dovoljno je provjeravati daje li projekcija vrha na potencijalnu os vrijednost manju od nule. Ako pri projiciranju dobivena vrijednost bude negativna normala koja određuje potencijalnu os se odbacuje, a ako su sve projekcije bile pozitivne pronađena je os razdvajanja.

Algoritam 2.8: TestCollision2D - optimalnija verzija

```
bool TestCollision2D(ConvexPolygon C0, ConvexPolygon C1){
    //testiramo normale C0
    int numV0 = C0.GetVerticesNum();
    int numV1 = C1.GetVerticesNum();
    for(int i0 = numV0-1, i1 = 0; i1 < numV0; i0 = i1++){
        Point point = C0.GetVertex(i1);
        Point axis = C0.GetNormal(i0);
        //ako je C1 na pozitivnoj strani osi-nema kolizije
        if(WhichSide(C1, point, axis)) return false;
    }

    //testiramo normale C1
     for(int i0 = numV1-1, i1=0;i1 < numV1; i0=i1++){
        Point point = C1.GetVertex(i1);
        Point axis = C1.GetNormal(i0);
        //ako je C0 na pozitivnoj strani osi-nema kolizije
        if(WhichSide(C1,point, axis)) return false;
    }

    return true;
}
```

Algoritam *WitchSide* za zadanu točku P i vektor D provjerava jesu li sve projekcije točaka poligona na os $P + t \cdot D$ na pozitivnoj strani te ako jesu vraća `true`.

Algoritam 2.9: WhichSide

```
bool WhichSide(ConvexPolygon c, Point P, Vector D){
    //proicira vrhove poligona c na pravac P + t*D
    int positive = 0;
    int negative = 0;
    int zero = 0;
    for(int i = 0; i < C.GetVerticesNum(); i++){
        float t = dot(D, C.GetVertex(i) - P);
        if(t > 0) positive++;
        else if(t < 0) negative++;
        else zero++;
    }
    if((positive > 0 && negative > 0) || zero > 0)
        return false;
    return positive ? true : false;
}
```

Složenost ovog algoritma također je kvadratna, no značajno manja nego prethodnog pošto se broj potrebnih projekcija smanjio. Možemo uočiti da složenost testa ovisi i o načinu

na koji pronalazimo projekcijske intervale te je stoga prema [7] moguće dobiti algoritam vremenske složenosti $O(nlogn)$ traženjem projekcija metodom bisekcije i pomoću binarnog stabla.

2.4.2 3D

Analogno kao i za poligone u 2D kreirat ćemo potrebnu strukturu podataka koja će sadržavati podatke o poliedrima.

Algoritam 2.10: ConvexPolyhedra

```
1  class ConvexPolyhedra{
2      protected:
3          int numVertices;
4          Point* Vertices;
5      public:
6          int GetVerticesNum();
7          int GetEdgesNum();
8          int GetFacesNum();
9          Point GetVertex(int i);
10         Vector GetEdge(int i);
11         Vector GetNormal(int i);
12 };
```

Kao potencijalne osi razdvajanja testiramo normale na stranice i vektorske produkte bridova poliedara. Algoritam za detekciju kolizije između poliedara jednostavno je proširenje algoritma 2.7. Ako pri testiranju preklapanja intervala utvrdimo da se oni ne preklapaju pronađena je os razdvajanja te se pretraživanje zaustavlja. Ovdje ćemo opisno navesti korake algoritma.

1. Za sve stranice poliedra C_0:
   - D = $C_0.GetNormal$
   - $FindInterval(C_0, D)$
   - $FindInterval(C_1, D)$
   - testiraj preklapanje intervala
2. Za sve stranice poliedra C_1:
   - D = $C_1.GetNormal$
   - $FindInterval(C_0, D)$
   - $FindInterval(C_1, D)$

- testiraj preklapanje intervala

3. Za sve bridove C_0:

   Za sve bridove C_1:

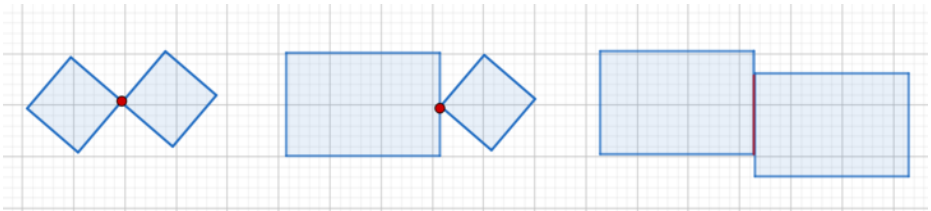
   - D = $C_0.GetEdge \times C_1.GetEdge$
   - $FindInterval(C_0, D)$
   - $FindInterval(C_1, D)$
   - testiraj preklapanje intervala

Korištenjem naivne verzije algoritma *FindInterval* za traženje projekcijskih intervala u najgorem slučaju bit će potrebno testirati sve kombinacije bridova što rezultira kubnom vremenskom složenošću. U najboljem slučaju pronalazimo os među normalama stranica te je stoga složenost algoritma minimalno kvadratna. Ako testiramo poliedre s velikim brojem stranica primjenom algoritma za pronalazak intervala koji koristi binarna stabla moguće je dobiti bolju asimptotsku vremensku složenost.

2.5 Utvrđivanje kontaktnog skupa na statičkim objektima

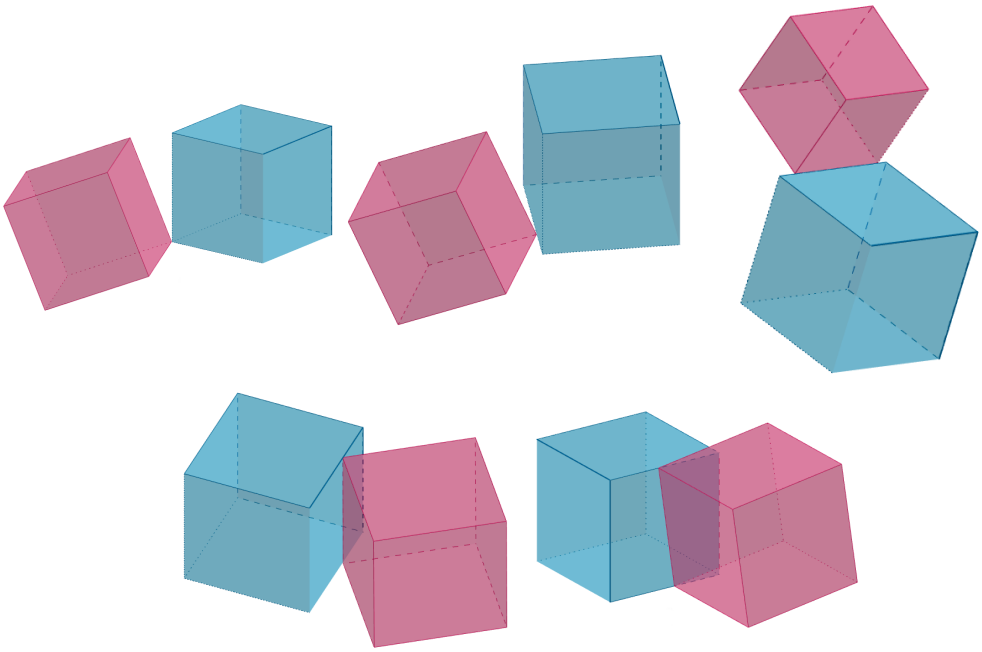
Pri razmatranju kontaktnog skupa kolidirajućih objekata pretpostavit ćemo da su objekti kruti odnosno da ne postoji mogućnost preklapanja.
U dvodimenzionalnom prostoru postoje dvije vrste kontaktnog skupa dvaju objekata koji kolidiraju. To su:

- točka - ako poligoni kolidiraju vrhovima ili vrhom i bridom
- dužina - ako poligoni kolidiraju bridovima.



Slika 2.8: Mogući kontakti u 2D

U trodimenzionalnom prostoru imamo tri vrste kontaktnih skupova:



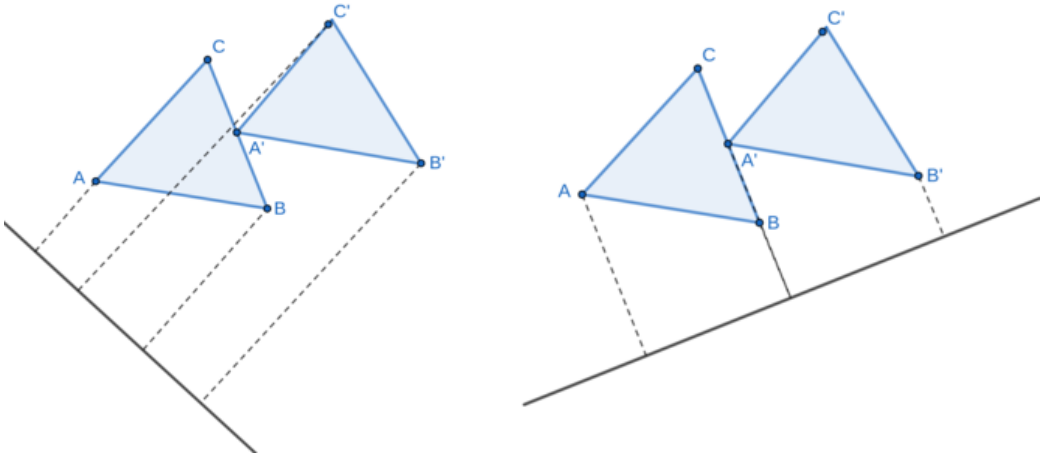
Slika 2.9: Mogući kontakti u 3D

- točka - ako se radi o kontaktima vrh-vrh, vrh-brid, vrh-stranica, brid-brid gdje bridovi nisu paralelni
- dužina - ako se radi o kontaktima brid-brid gdje su bridovi međusobno paralelni i brid-stranica
- poligon - ako se radi o kontaktu među stranicama poliedara

Implementacija izračunavanja kontaktnog skupa generalno je zahtjevnija od implementacije algoritma za testiranje kolizije. Pri implementaciji algoritma za pronalazak kontaktnog skupa koristimo informacije dobivene testiranjem kolizije te stoga proširujemo algoritam za testiranje kolizija iz prethodnog odlomka.

2.5.1 2D

Pri traženju kontaktnog skupa potrebno je primijeniti algoritam na projekcijskim intervalima koji su dali minimalno preklapanje jer će jedino oni dati ispravni kontaktni skup. Ovaj problem vidimo na slici 2.10



Slika 2.10: Lijevo: os koja ne daje projekcije s minimalnim preklapanjem - ekstreme daju vrhovi B i C', desno: intervali s minimalnim preklapanjem - promatramo brid BC i vrh A'

Kako bismo mogli utvrditi koji dijelovi poligona su u međusobnom kontaktu, prema Eberlyju, potrebno je kreirati dodatnu klasu koja će sadržavati informacije o ekstremima projekcijskih intervala.

Algoritam 2.11: Klasa za informacije o projekcijskom intervalu poligona

```
1 class ProjInfo{
2     public:
3         float min, max;
4         int index[2];
5         bool isVertex[2];
6 };
```

Varijable `min` i `max` čuvaju vrijednost minimuma i maksimuma projekcijskog intervala dok varijabla `index` sadrži indekse vrhova ili bridova koji se projiciraju u ekstreme intervala. Posljednja varijabla `isVertex` sadrži `bool` vrijednost postavljenu na `true` ako je element projiciran u odgovarajući ekstrem vrh. Ako se radi o bridu vrijednost `isVertex` je `false`. `index[0]` i `isVertex[0]` odgovaraju minimumu projekcije, a `index[1]` i `isVertex[1]` maksimumu. Uz to, važno je pamtiti međusobni položaj projekcijskih intervala kako bismo prilikom traženja presjeka znali trebamo li promatrati podatke o minimalnim ili maksimalnim vrijednostima za svaki od poligona. Ako se projekcija poligona C_0 nalazi lijevo od projekcije C_1 na osi D, tada znamo da će u kontakt doći maksimum intervala poligona C_0 i minimum intervala poligona C_1 te ćemo tada promatrati vrijednosti varijabli iz odgovarajućih `ProjInfo` instanci s indeksom 1 za prvi poligon te vrijednosti s indeksom 0 za drugi.

Stoga je potrebno prilagoditi funkciju `FindInterval` tako da u varijablu tipa `ProjInfo` sprema vrijednosti ekstrema projekcije, indekse elemenata koji daju ekstreme te oznake jesu li elementi koji su projicirani u ekstreme vrhovi odnosno bridovi u varijabli `isVertex`. Uz to pri svakom testiranju preklapanja intervala potrebno je provjeriti je li dobiven manji presjek od dosadašnjeg te zapamtiti projekcijske intervale koji su dali najmanji presjek i njihove međusobne pozicije.

Kada su utvrđeni dijelovi objekata koji sudjeluju u koliziji s minimalnim preklapanjem projekcijskih intervala prelazimo na traženje njihova presjeka. Kod 2D objekata postoje dvije mogućnosti kontatnog skupa. Jedna je kontaktna točka koja je rezultat toga da je barem jedan od dijelova koji sudjeluju u kontaktu vrh. Ako u kontaktu sudjeluje barem jedan vrh, on će ujedno biti i kontaktni skup te nije potrebno tražiti presjek. Druga mogućnost je kontaktna dužina koja je rezultat kontakta dvaju bridova. Tada je potrebno pronaći njihov međusobni presjek. Detaljnije o presjeku bridova govorimo kod utvrđivanja kontaktnih skupova na dinamičkim objektima. Jedina razlika je u tome što kod statičkih objekata nije potrebno pomicati objekte na pozicije prvog kontakta.

2.5.2 3D

U trodimenzionalnom prostoru primjenjuje se slična ideja. Ponovno imamo strukturu za pohranu informacija o projekcijskim intervalima. Kao i u 2D, potrebno je prilagoditi funkciju za traženje projekcijskih intervala tako da relevantne informacije pohranjuje u varijable tipa `ProjInfo`.

Algoritam 2.12: Klasa za informacije o projekcijskom intervalu poliedara

```
1 class ProjInfo{
2     public:
3         float min, max;
4         int index[2];
5         enum typeOfContact{V, E, F};
6         typeOfContact type[2];
7 };
```

Za pronalaženje kontaktnog skupa dvaju poliedara potrebno je dodatno implementirati funkcije za presjek brida i stranice te dviju stranica. Algoritam će u ovisnosti o međusobnom položaju projekcijskih intervala poliedara i vrsti kontakta pozvati odgovarajuću funkciju za pronalaženje presjeka na odgovarajućim dijelovima poliedara.

Poglavlje 3

Dinamička detekcija kolizija

U ovom poglavlju opisujemo postupak utvrđivanja kolizije te pronalaženjem kontaktnog skupa konveksnih objekata koji se kreću linearno konstantnom brzinom. Algoritam opisan u [7], koji ćemo promatrati, koristi metodu razdvajajućih osi.

3.1 Ispitivanje postojanja kolizije na objektima u pokretu

Korisnik aplikacije doživljava pokret kao neprekidan tok reprezentacija objekata. Algoritam simulira taj protok tako da osvježava konfiguracije objekata u fiksno određenim diskretnim vremenskim intervalima. Ljudsko oko je moguće prevariti tako da vidi glatki pokret objekta ako se prikaže dovoljno velik broj *frame*-ova u sekundi. Stoga se prirodno nameće ideja da se i ispitivanje kolizije objekata u pokretu provodi periodično. U danom vremenskom trenutku promatramo stanje objekata kao da su statični. Iako je prirodno očekivati da će se ovakvim testiranjem lako propustiti neke kolizije, ono se zadržalo zbog svojstva koherentnosti između dva ažuriranja reprezentacija objekata.

- **Prostorna koherentnost:** objekt se obično prostire kroz relativno mali prostor i kolizije su prilično rijetke te se kolizije rješavaju, a ne zadržavaju između ažuriranja
- **Vremenska koherentnost:** pozicije objekata se mijenjaju relativno malo između ažuriranja te se stoga i pokreti grafički prikazuju glatki

Neka su C_0 i C_1 dva konveksna objekta koji se kreću brzinama V_1 odnosno V_2. Kako bismo utvrdili postojanje kolizije, promatramo objekt C_0 kao stacionarni, a objekt C_1 će imati brzinu $V = V_1 - V_0$.

Ako se objekti inicijalno preklapaju, vrijeme prvog kontakta proglašavamo $T = 0$. Ako su objekti inicijalno razdvojeni, koristimo metodu razdvajajućih osi. Projekcija C_1 na os razdvajanja D, koja nije okomita na smjer kretanja V objekta C_1, pomiče se brzinom $V' = (V \cdot D)/|D|^2$. Algoritam se svodi na utvrđivanje odmiče li se projekcija C_1 od projekcije C_0 u kojem slučaju se objekti nikad neće sudariti, ili se projekcija C_1 primiče projekciji C_0 te tada treba utvrditi hoće li se sudar dogoditi u zadanom vremenskom intervalu.

3.1.1 2D

Algoritam 3.1 ispituje normale na bridove C_0 i C_1, traži intervale projekcije, izračuna relativnu brzinu kretanja intervala projekcije C_1 na normali te poziva funkciju `NoIntersection` prikazanu algoritmom 3.2. `NoIntersection` za zadane projekcijske intervale i relativnu brzinu C_1 provjerava mogućnost preklapanja intervala te računa vremena prvog i zadnjeg kontakta intervala. Postojanje kolizije kod objekata u pokretu ispitujemo u vremenskom intervalu $[T_0, T_{max}]$. Razlog ograničavanja vremenskog intervala testiranja nalazi se u tome što nas u primjeni najčešće zanima hoće ni se kolizija dogoditi unutar vremena koje je važno za određeni softverski produkt.

Algoritam 3.1: Test kolizije za poligone u pokretu

```
1  bool TestCollision2D_V(ConvexPolygon C0, ConvexPolygon C1, Vector V0,
2      Vector V1, float maxTime,
3      float &firstTime, float &lastTime){
4
5      V = V1-V0;
6      frstTime = 0;
7      lastTime = inf;
8
9      //testiramo bridove C0
10     for(int i = 0; i < C0.GetVerticesNum(); i++){
11         FindInterval(C0, D, min0, max0);
12         FindInterval(C1, D, min1, max1);
13         speed = dot(D,V);
14         if(NoIntersect(maxTime, speed, min0, max0, min1,
15         max1, firstTime, lastTime))
16             return false;
17     }
18
19      //testiramo bridove C1
20     for(int i = 0; i < C1.GetVerticesNum(); i++){
21         FindInterval(C0, D, min0, max0);
22         FindInterval(C1, D, min1, max1);
23         speed = dot(D,V);
```

```
24          if(NoIntersect(maxTime, speed, min0, max0, min1,
25          max1, firstTime, lastTime))
26              return false;
27      }
28
29      return true;
30  }
```

Algoritam 3.2: Test intervala

```
1   bool NoIntersect(float maxTime,float speed, float min0, float max0,
2       float min1, float max1, float &firstTime, float &lastTime){
3
4       //interval C1 je lijevo od intervala C0
5       if(max1 < min0){
6           if(speed <= 0) return true; //razmicanje
7           t = (min0 - max1)/speed;
8           if(t > firstTime) firstTime = t;
9           if(firstTime > maxTime) return true;
10
11          t = (max0 - min1)/speed;
12          if(t < lastTime) lastTime = t;
13          if(firstTime > lastTime) return true;
14      }
15      //interval C1 je desno od intervala C0
16      else if(max0 < min1){
17          if(speed >= 0) retuen true; //razmicanje
18          t = (max0 - min1)/speed;
19          if(t > firstTime) firstTime = t;
20          if(firstTime > maxTime) return true;
21
22          t = (min0 - max1)/speed;
23          if(t < lastTime) lastTime = t;
24          if(firstTime > lastTime) return true;
25      }
26      //intervali se preklapaju
27      else{
28          if(speed > 0){
29              t = (max0 - min1)/speed;
30              if(t < lastTime) lastTime = t;
31              if(firstTime > lastTime) return true;
32          }
33          else if(speed < 0){
34              t = (min0 - max1)/speed;
35              if(t < lastTime) lastTime = t;
36              if(firstTime > lastTime) return true;
37          }
```

```
38      }
39      return false;
40  }
```

Ovaj algoritam uz korištenje naivnog načina traženja projekcija ponovno je kvadratne vremenske složenosti, moguće je dobiti asimptotsku složenost $O(nlogn)$ korištenjem optimizacije projiciranja navedene u [7].

3.1.2 3D

Analogno kao što smo proširili ideju kod statičkih objekata, tako proširujemo i na dinamičke. Kako bismo dobili test za poliedre potrebno je prilagoditi algoritam 3.1 tako da kao potencijalne osi razdvajanja testira normale na stranice poliedara te vektorske produkte njihovih bridova.

3.2 Utvrđivanje kontaktnog skupa na objektima u pokretu

Kada smo utvrdili da postoji kolizija između dva konveksna objekta u pokretu C_0 i C_1 koji se ne preklapaju inicijalno, prelazimo na utvrđivanje njihovog kontaktnog skupa.

Za pronalaženje kontaktnog skupa koristimo strukture `ProjInfo` koje će pri utvrđivanju kolizije sadržavati informacije o projekcijskim intervalima u trenutku prvog kontakta. Eberly modificira funkciju `NoIntersect` tako da ona, umjesto varijabli za ekstreme projekcija, prima varijable tipa `ProjInfo` te pri detektiranju prvog kontakta intervala u njih sprema informacije o ekstremima i dijelovima objekata projiciranih u njih. Zatim se poziva funkcija koja će izračunati kontaktni skup kojoj moramo osigurati informaciju o poretku projekcijskih intervala. Ideja je u ovisnosti o tipu kontakta pronaći dio poligona koji će tvoriti kontaktni skup te presjek tih dijelova poligona translatiranih na svoju poziciju u vremenu prvog kontakta. Pozicije poligona u vremenu prvog kontakta dane su s:

$$C_0 + TV_0 = \{X + TV_0 : X \in C_0\} \tag{3.1}$$

$$C_1 + TV_1 = \{X + TV_1 : X \in C_1\}. \tag{3.2}$$

Ako je projekcija C_0 lijevo od projekcije C_1:

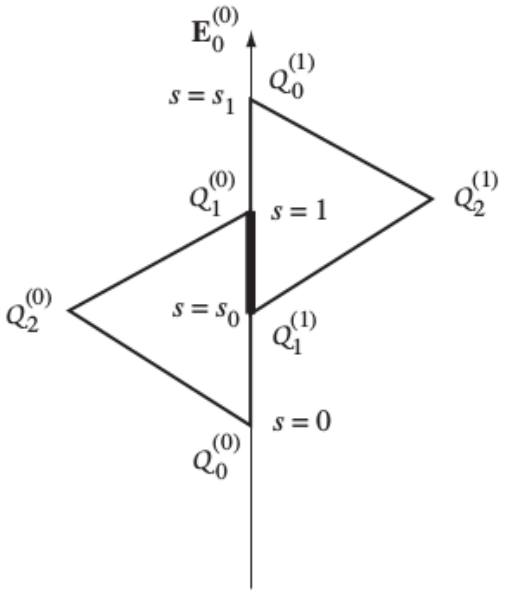
- Provjeri je li maksimum projekcijskog intervala C_0 ili minimum projekcijskog intervala C_1 dao vrh te ako je bar jedan od uvjeta pozitivan pronađi odgovarajuću točku P te je tada kontaktni skup točka dobivena pomicanjem P na njezino mjesto u trenutku prvog kontakta T_{first}

- Preostali slučaj daje kontakt brid-brid. S $Q_i^{(j)} = P_i^{(j)} + T_{first} \cdot V_j$ označimo točke poligona pomaknute na mjesto prvog kontakta. Pretpostavimo da se dogodio kontakt između bridova $E_0^{(0)}$ i $E_0^{(1)}$. Parametriziramo bridove na sljedeći način: $Q_0^{(0)} + s \cdot E_0^{(0)}$ gdje je $s \in [0, 1]$ za brid $E_0^{(0)}$, a za brid $E_0^{(1)}$ $s \in [s_0, s_1]$. Pritom su

$$s_0 = \frac{E_0^{(0)} \cdot (Q_1^{(1)} - Q_0^{(0)})}{|E_0^{(0)}|^2}$$

$$s_1 = \frac{E_0^{(0)} \cdot (Q_0^{(1)} - Q_0^{(0)})}{|E_0^{(0)}|^2}.$$

  Tada je kontaktni skup dobiven kao presjek intervala $[0, 1]$ i $[s_0, s_1]$, a točke poligona u kontaktu su $Q_0^{(0)} + S \cdot E_0^{(0)}$ za vrijednosti S iz presjeka.



Slika 3.1: Prikaz kontakta brid-brid - preuzeto iz [7]

Na sličan način se ispituje kontaktni skup ako je projekcijski interval poligona C_0 desno od projekcijskog intervala C_1. Slučaj u kojem se poligoni inicijalno preklapaju moguće je izvesti kreiranjem algoritma za pronalaženje presjeka konveksnih skupova, no taj slučaj izostavljamo pošto se obično ne primjenjuje u praksi.

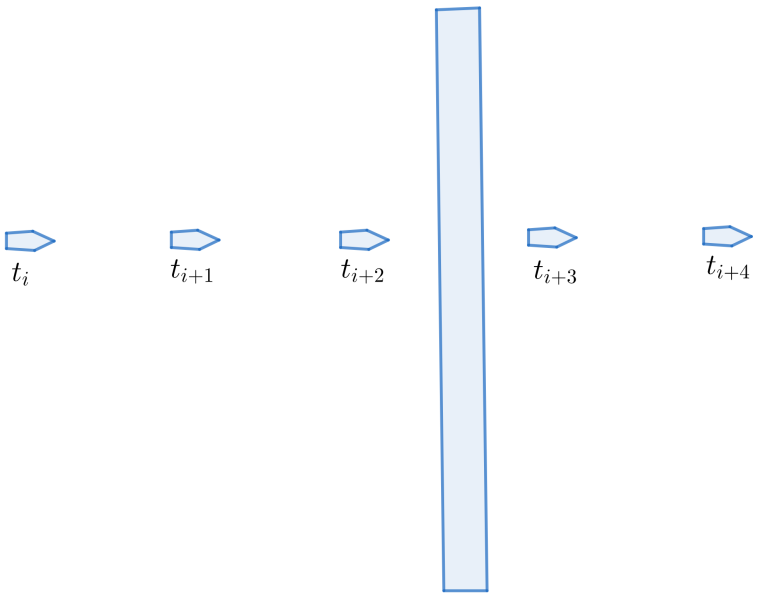
Ideja je slična i u 3D. Ponovno se pohranjuju informacije o projekcijskim intervalima u posebnu strukturu te se ovisno o poziciji i vrsti kontakta kreiraju transformirani vrhovi, bridovi ili stranice i izračunava se njihov presjek.

Mana testiranja kolizije u diskretnim vremenskim koracima je to što se kolizija testira na statičkim objektima u jednom vremenskom trenutku te se stoga utvrdi samo ako se objekti sudare u tom trenutku ili se njihov kontakt u trenutku testa još nije razriješio. Kako je razmak između dva testiranja fiksno određen moguće je da se neke kolizije dogode i razriješe između dva testiranja. Takve kolizije ostaju nedetektirane. Ponekad je neophodno detektirati sve kolizije te se tad pribjegava algoritmima kontinuirane detekcije kolizija (CCD) o kojoj govori sljedeće poglavlje.

Poglavlje 4

Kontinuirana detekcija kolizija

Korištenjem diskretne detekcije kolizija ponekad je moguće ne detektirati neke kolizije, pogotovo ako se radi o malim objektima koji se kreću velikom brzinom. Kod takvih objekata putanja prijeđena između dva testiranja veća je od površine koju oni zauzimaju te ako se na njihovoj putanji nađe neki drugi objekt rezultat može biti neuočena kolizija.



Slika 4.1: Tuneliranje

Moguće je da takvi objekti prođu kroz zid, tlo ili da pogođeni objekti ne reagiraju na koliziju. Na primjer, simulacija ispaljivanja metka iz oružja mogla bi proći neprimijećeno ako se pogodak dogodio unutar jednog vremenska koraka tj. između dva testiranja. Ovaj problem naziva se tuneliranje i moguće ga je riješiti primjenom algoritama za kontinuiranu detekciju kolizija (*eng. Continuous collision detection* ili kraće CCD) koji će predvidjeti koliziju prije nego što se ona dogodi.

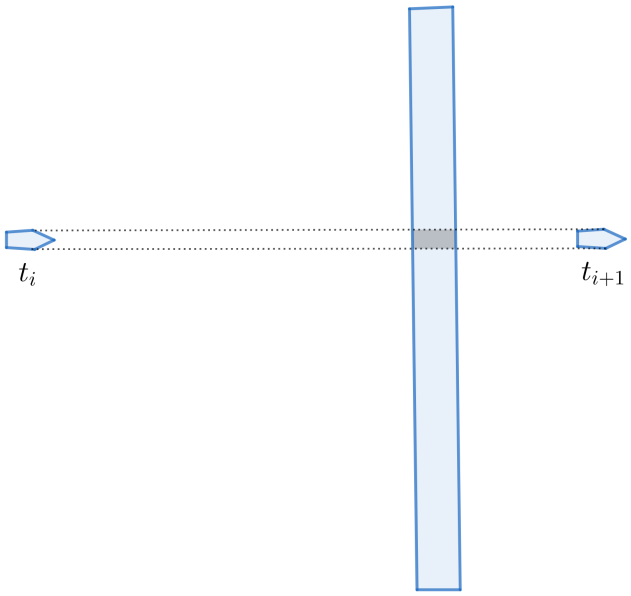
4.1 Razlika između diskretne i kontinuirane detekcije kolizija

Diskretnim metodama testiramo koliziju na statičkim objektima u diskretnim vremenskim trenutcima simulacije. Pri korištenju diskretnih metoda tuneliranje je moguće ublažiti smanjivanjem duljine vremenskog intervala između dva testiranja tako da putanja objekta u jednom koraku bude manja od njegovog promjera. Takve metode dati će dobre rezultate u *offline* aplikacijama dok se njihovo korištenje ne preporučuje kod interaktivnih aplikacija kod kojih je potrebna relativno velika i konstantna brzina renderiranja scene. Štoviše, diskretna detekcija kolizija može utvrditi koliziju kada se objekti već preklapaju. Pošto se najčešće radi o čvrstim tijelima potrebno je pronaći stvarno vrijeme prvog kontakta *backtracking* metodom što je neophodno za rješavanje problema međusobnog prodora i druge odgovore na koliziju.

Za razliku od diskretnih, kontinuirane metode utvrđuju da će se kolizija dogoditi prije nego se zapravo dogodi. One predviđaju buduće putanje objekata te pokušavaju predvidjeti trenutak sudara analizirajući predviđeno kretanje. Kontinuirane metode pronalaze stvarno vrijeme prvog kontakta (*eng. time of inpact)* između dva vremenska koraka simulacije. Vrijeme kontakta (*eng. time of inpact)* je vremenski trenutak u kojem je udaljenost između dva objekta 0 ili unutar zadane vrijednosti tolerancije. Postoji puno različitih algoritama koji na više ili manje robustan način rješavaju ovaj problem.

4.2 Algoritam

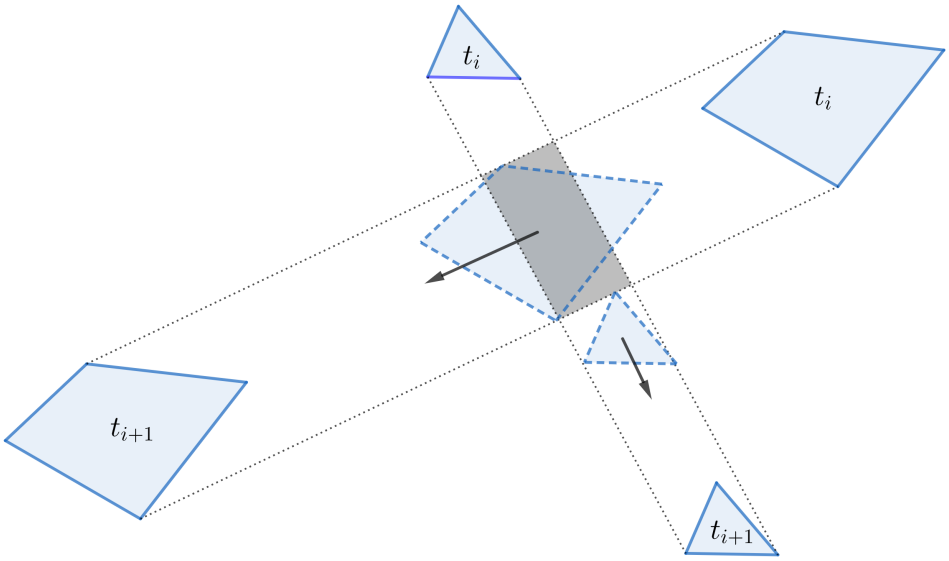
Jedan jednostavan način rješavanja ovog problema je promatrati prošireni objekt (*eng. swept volume*) dobiven kao skup svih točaka koje će objekt pokriti krećući se svojom putanjom u vremenskom intervalu. Tada ako se prošireni objekti sijeku slijedi da će se u danom intervalu dogoditi kolizija. Ovaj način je korektan ako testiramo koliziju sa statičkim objektom (slika 4.2), ali bi mogao dati lažno pozitivne rezultate ako test provodimo na dinamičkim objektima. Ovaj problem prikazuje slika 4.3.



Slika 4.2: Korištenjem *swept* oblika za test sa statičkim objektom kolizija će biti detektirana

Ericson u [8] i Jimenez, Thomas i Torras u [10] navode moguće rješenje bazirano na ideji dijeljenja puta koji će objekt prijeći na poddijelove te provođenju više statičkih podtestova kroz dani vremenski korak. Svaki podtest pozicionira objekt na točku puta koja mu je dodijeljena te provodi standardni statički test kolizije. Previše grubo uzorkovanje odnosno prerijetki podtestovi mogu uzrokovati propust kolizije dok su previše gusti testovi vremenski preskupi. Prema Ericsonu, važno je prilagoditi duljinu putova tako da se objekt na trenutnom položaju djelomično preklapa s onim na prethodnom i sljedećem položaju. Usprkos tomu, još uvijek je moguće da neka kolizija ostane nedetektirana na mjestima između dva podtesta. Ericson predlaže zamjenu originalnog objekta s njegovim proširenim verzijama što daje potpunu pokrivenost kao da se promatra obris tijela po putanji kroz vrijeme.

Jimenez, Thomas i Torras u [10] predlažu adaptivno uzorkovanje prema kojem bi sljedeći uzorak odnosno pozicija trebao biti najranije vrijeme u kojem bi se kolizija mogla zaista pojaviti. Najjednostavnija ideja za estimaciju sljedećeg uzorka bila bi povezati minimalnu udaljenost među objektima s relativnom brzinom. Algoritam promatra stanje objekata tj. njihovu poziciju i orijentaciju između dva vremenska koraka t_i i t_{i+1} i iterativno



Slika 4.3: Problem korištenja *swept* oblika kod dinamičkih objekata - *swept* oblici se sijeku iako kolizija izostaje

pomiče objekte. U svakoj iteraciji izračunava međusobno najbliže točke objekata, kreira vektor određen tim točkama i projicira vektor brzine na njega te određuje gornju granicu mogućeg kretanja objekta tom brzinom prije kolizije. Zatim pomakne objekte do dobivene granice. Korak algoritma ponavlja se sve dok udaljenost objekata ne padne ispod zadane male vrijednosti tolerancije.

Kao ilustraciju rada algoritma kontinuirane detekcije kolizija prihvatljive vremenske složenosti dajemo verziju koju Ericson predlaže u [8]. Radi se o rekurzivnom algoritmu koji prima dva objekta te početnu i krajnju točku vremenskog intervala jednog vremenskog koraka. Algoritam 4.1 radi rekurzivno binarno pretraživanje te uspoređuje međusobnu udaljenost objekata i sumu duljina njihovih maksimalnih prijeđenih putova na početku i na kraju danog vremenskog intervala. Ako je zbroj duljina putanja na početku intervala manji od minimalne udaljenosti objekata znamo da se tada udaljenost među objektima nakon njihova pomicanja na položaje na kraju intervala neće dovoljno smanjiti te će kolizija izostati. Ako je zbroj duljina putanja veći ili jednak minimalnoj udaljenosti objekata na početku intervala tada je potrebno provjeriti odnose na kraju intervala. Slučaj u kojem je udaljenost objekata na kraju intervala veća od zbroja prijeđenih putanja znači da su se objekti udaljili te kolizija izostaje. Ako se u bilo kojem rekurzivnom pozivu dogodi neki

od ovih slučajeva algoritam staje i vraća `false`. Inače se u danom intervalu dogodila kolizija te nastavljamo potragu za vremenom prvog kontakta na polovicama tog intervala. Ako postoji kolizija algoritam vraća `true` kada je pronađen vremenski trenutak u kojem se objekti prvi put sudaraju tj. ako je duljina vremenskog intervala pala ispod zadane vrijednosti tolerancije. U tom slučaju u varijabli `hitTime` će biti spremljeno vrijeme prvog kontakta.

Algoritam 4.1: CCD

```
1  bool CCD(Object a, Object b, float startTime, float endTime,
2      float &hitTime){
3      //maksimalna duljina puta koju objekti prijedu u intervalu
4      float maxMoveA = MaxMovement(a, startTime, endTime);
5      float maxMoveB = MaxMovement(b, startTime, endTime);
6
7      //ako je udaljenost izmedu objekata veca od zbroja
8      //njihovih maksimalnih putanja sudar je nemoguc
9      float minDistStart = MinDistanceAtTime(a, b, startTime);
10     float minDistEnd = MinDistanceAtTime(a, b, endTime);
11
12     if (minDistStart > maxMoveA + maxMoveB) return false;
13     if (minDistEnd > maxMoveA + maxMoveB) return false;
14
15     //povjera je li duljina intervala manja od vrijednosti tolerancije
16     if (endTime-startTime < INTERVAL_EPSILON) {
17         hitTime = startTime; return true;
18     }
19
20     float midTime = (startTime + endTime) * 0.5f;
21
22     //testira prvu polovicu intervala
23     if (CCD(a, b, startTime, midTime, hitTime)) return true;
24     //testira drugu polovicu intervala
25     return CCD(a, b, midTime, endTime, hitTime);
26 }
```

Vremenska složenost ovog algoritma je $O(logn)$.

U Box2D biblioteci, koju koristimo za demonstraciju rada algoritama, inicijalno se koristi CCD za prevenciju tuneliranja dinamičkih objekata kroz statičke i kinematičke objekte.

Poglavlje 5

Detekcija kolizija složenih objekata

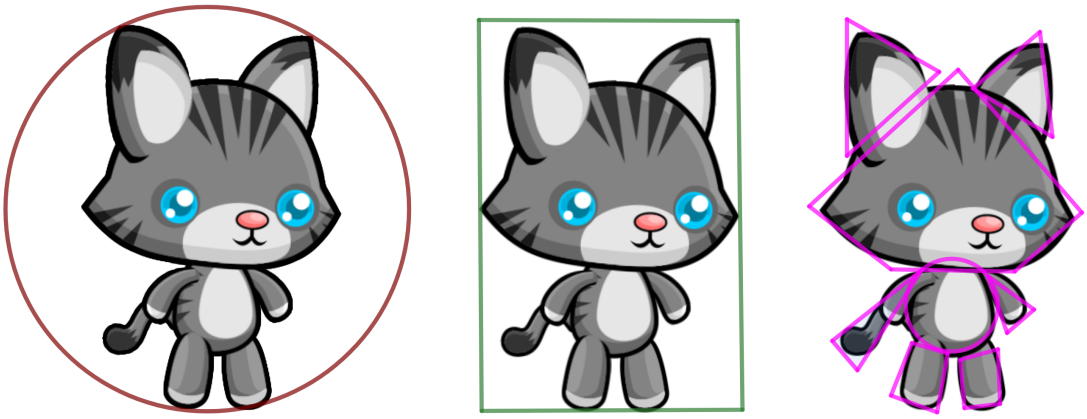
Često je izravno testiranje kolizije geometrije dvaju objekata vrlo skupa operacija, posebno kada se objekti sastoje od stotina ili čak više tisuća stranica ili bridova. Sulaiman i Bale slikovito opisuju ovaj problem u [12]: *"Na primjer, model automobila čija površina se sastoji od 10000 poligona udari u zgradu koja se sastoji od 40 poligona te čeka na signal da se dogodio sudar. Testirat će se svi poligoni, jedan po jedan, kako bi se pronašli dijelovi automobila i zgrade koji su se sudarili."* Ovakav način testiranja odrazio bi se negativno na performanse algoritma. Postoji više strategija za optimizaciju algoritama za testiranje kolizije na složenim objektima. Jedan takav način je pokušati pojednostaviti zadani objekt.

Razlikujemo dvije vrste složenosti objekata. Jednu opisuje gornji primjer, tj. radi se o konveksnim oblicima sastavljenima od velikog broja bridova u 2D ili velikog broja stranica u 3D. Složene konveksne oblike se obično pokušava obuhvatiti jednim jednostavnijim konveksnim objektom. Druga vrsta su nekonveksni oblici. Nekonveksni oblici ne moraju nužno imati velik broj stranica odnosno bridova. Prethodno prikazani algoritmi nisu prilagođeni za nekonveksne oblike te njih obično aproksimiramo s jednim ili više jednostavnih konveksnih oblika. Na taj način je moguće iste algoritme primijeniti i na takve slučajeve. Važno je znati da se porastom broja konveksnih oblika kojima aproksimiramo nekonveksni preciznost kolizije povećava, no istovremeno raste i složenost algoritma pošto je svaki od njih potrebno posebno testirati.

5.1 Granični oblici (*Bounding volumes - BV*)

Granični oblik (BV) je jednostavan konveksni geometrijski lik u 2D, odnosno tijelo u 3D kojim nastojimo što preciznije obuhvatiti granice dijelova nekonveksnog objekta, jednog složenog objekata ili više istovrsnih objekata. Cilj pri kreiranju graničnih oblika je dobiti što preciznije pokrivanje objekta jednostavnim geometrijskim likom odnosno tijelom na

što jeftiniji način čime bi smanjili vremensku složenost testiranja kolizije. Uz samo testiranje, želimo da odgovor na koliziju (translacija i rotacija objekta) bude brz i efikasan. Također, granični oblici zahtijevaju dodatne strukture te je izuzetno važno minimizirati povećanje prostorne složenosti. Ne postoji idealan geometrijski lik ili tijelo koje bi odgovaralo svakom objektu i zadovoljavalo sve ostale zahtjeve. Odabirom oblika koji je složeniji i time točnije aproksimira dani objekt povećavamo njegovu složenost čija posljedica je kompleksniji test i veća vremenska složenost algoritma. Ovisno o potrebama aplikacije kompenziramo preciznost brzinom.



Slika 5.1: Moguća korištenja graničnih oblika

5.1.1 Široka i uska faza

Ispitivanje kolizije svih parova objekata rezultirat će kvadratnom vremenskom složenošću. Ako se radi o složenim objektima reduciranje broja parova koje treba testirati primjetno će smanjiti potrebno vrijeme. Testiranje kolizije izravno na geometriji složenih objekata bez korištenja graničnih oblika zahtijeva ispitivanje svih normala na bridove poligona u 2D, odnosno normala na stranice poliedara te vektorskih produkata bridova poliedara u 3D. Stoga se nameće ideja moguće eliminacije objekata koji nikako ne mogu biti u međusobnoj koliziji. Ideja je testiranje podijeliti na dvije faze: široku i usku.

Široka (*broad*) faza

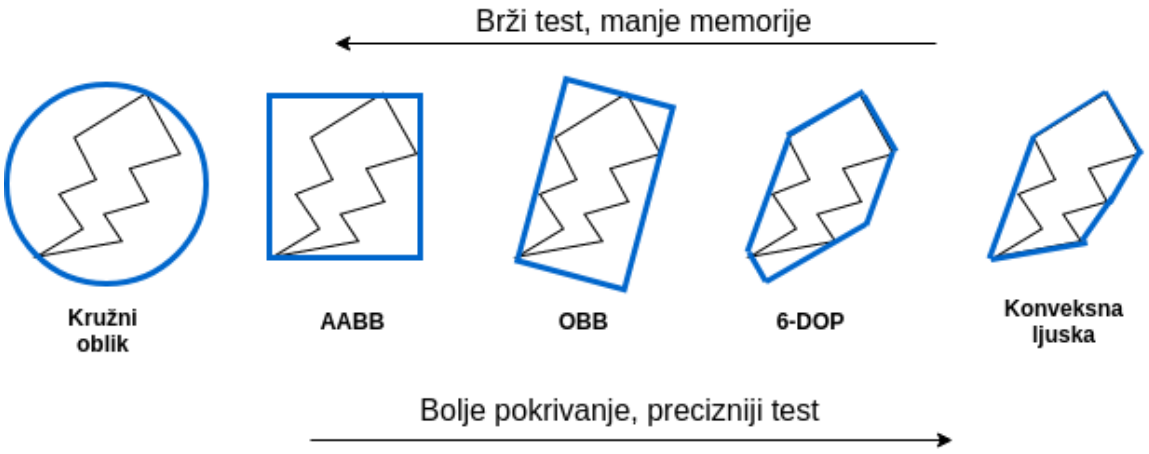
Prije testova široke faze konstruiraju se granični oblici za zadane objekte odnosno dijelove nekonveksnog objekta. Ispitivanjem kolizije na graničnim oblicima prije ispitivanja

na geometriji objekata možemo na vrlo jednostavan i jeftin način utvrditi nepostojanje kolizije među objektima i time zahvaljujući vremenskoj i prostornoj koherentnosti eliminirati većinu potencijalnih parova. Ako su granični oblici dvaju objekata ipak u koliziji, tada je kolizija tih objekata moguća. Neke aplikacije ne zahtijevaju veliku preciznost pa se može stati na ovakvom testu i proglasiti da su objekti u koliziji ako se njihovi granični oblici sijeku.

Uska (*narrow*) faza

U aplikacijama u kojima je važna izuzetna preciznost ispitivanje nastavljamo primjenom algoritma za koliziju geometrije na parovima koje je pronašla široka faza. Prethodnom primjenom široke faze eliminirati ćemo veliki broj nepotrebnih testova koji se mogu izbjeći pa uska faza neće biti vremenski jako zahtjevna.

5.2 Vrste graničnih oblika



Slika 5.2: Vrste graničnih oblika

5.2.1 Sferni i kružni oblik

Sfera u 3D odnosno krug u 2D su jedni od najčešće izabranih graničnih oblika jer je test kolizije prilično jednostavan i nije vremenski i prostorno zahtjevan. Sfere i krugovi su invarijantni na rotaciju te je transformacija stoga trivijalna tj. dovoljno je translatirati ih na

novu poziciju. Uz to, nisu memorijski zahtjevni pošto je potrebno čuvati samo podatke o središtu i radijusu. Testiranje kolizije između dva kruga u 2D prikazana su algoritmom 2.3. Analogno se dobije algoritam za testiranje kolizije između dvije sfere.

Izračunavanje graničnog kruga

Granični krug konstruiramo iz zadanog vektora točaka odnosno vrhova poligona. Prvo pronalazimo središte kruga kao srednju vrijednost svih točaka. Radijus je određen kao udaljenost najudaljenije točke od središta. Pretpostavimo da funkcija *dist()* vraća kvadrat udaljenosti danih točaka kako bismo izbjegli traženje drugog korijena u svakoj iteraciji petlje.

Algoritam 5.1: BoundingCircle

```
1
2  Circle BoundingCircle(Point* points){
3      int pointCount = points.size();
4      Point center = 0.f;
5
6      //srediste
7      for(int i = 0; i < pointCount; ++i)
8          center = center + points[i];
9      center = (float)center / pointCount;
10
11     //radijus
12     float radius = dist(center, point[0]);
13     for(int i = 1; i < pointCount; ++i){
14         float distance = dist(center, points[i]);
15         if(distance > radius) radius = distance;
16     }
17
18     radius = sqrtf(radius);
19     return Circle(center, radius);
20 }
```

Utvrđivanje optimalne sfere je zahtjevan postupak. Neke od metoda izračunavanja optimalnih sfernih volumena su opisane u [8]. Ovdje ukratko opisujemo samo jedan osnovni način.

Izračunavanje optimalne sfere

Sfera je jedinstveno definirana s četiri nekomplanarne točke. Naivni algoritam za pronalaženje optimalne sfere sastoji se od razmatranja svih mogućih kombinacija četvorki točaka odnosno vrhova poliedra. Algoritam iterativno uzima četiri točke, izračunava sferu

koju one određuju te provjerava obuhvaća li sve ostale točke te je li manja od dosadašnjih verzija. Rezultat je minimalna sfera koja obuhvaća sve ostale točke tj. vrhove poliedra. Pretraživanje svih mogućih kombinacija rezultira algoritmom vremenske složenosti $O(n^5)$ te stoga on nije upotrebljiv.

U [15] Welzl predlaže algoritam koji pronalazi minimalnu sferu u linearnom vremenu. Pretpostavimo da je D minimalna granična sfera za skup točaka $P = \{p_0, p_1, ..., p_n\}$. Zatim dodajemo novu točku p_{n+1}. Ako se ona nalazi u sferi, sfera ostaje ista. Ako se točka nalazi izvan sfere proširujemo sferu tako da točka p_{n+1} bude na njezinoj granici. Tako dobivamo minimalnu sferu koja obuhvaća $P \cup \{p_{n+1}\}$. Welzl pokazuje kako odabir nove točke iz skupa neiskorištenih koje se dodaju na slučajan način pridonosi smanjenju vremena potrebnog za pronalazak minimalne sfere te rezultira linearnom vremenskom složenošću. Welzlov rekurzivni algoritam prima listu točaka koje određuju granice sfere te listu onih koje još treba dodati. U svakom rekurzivnom pozivu, ako nismo iskoristili sve točke izabiremo jednu na slučajan način. Napravimo rekurzivni poziv na skupu bez nje. Provjerimo nalazi li se točka u tako dobivenoj sferi te ako da, vratimo tako dobivenu sferu. Ako je točka izvan sfere dodajemo ju u listu potpornih te pozivamo rekurziju. Uz to potrebno je kreirati funkcije koje će izgraditi sferu iz pronađenih potpornih točaka. Algoritam staje kada su potrošene sve točke te tada računa minimalnu sferu zadanu preostalim potpornim točkama. Prema [15] složenost ovog algoritma je $O(dd!n)$ gdje je d dimenzija prostora, a n broj točaka. Stoga je u 3D složenost linearna u ovisnosti o broju točaka.

5.2.2 Orijentirani granični pravokutnici i kvadri (OBB)

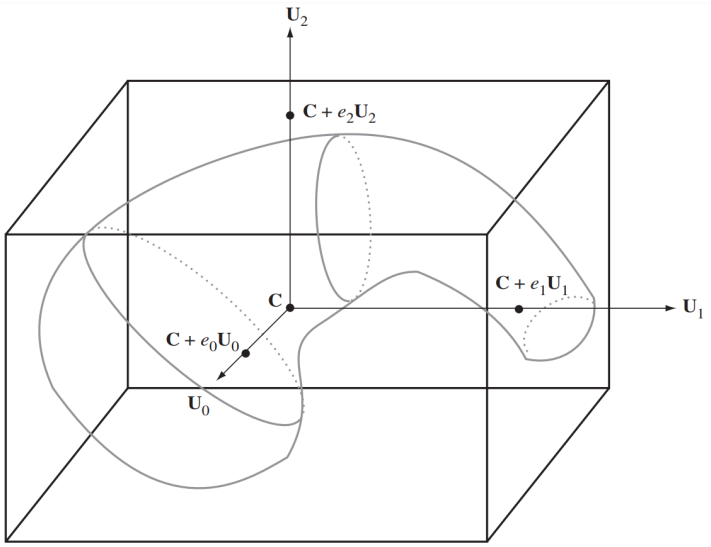
Za razliku od sfernog oblika, OBB je prostorno i konstruktivno zahtjevniji. OBB reprezentiramo pomoću centralne točke, vektora smjera te poluudaljenosti na svakom od vektora smjera. Centralna točka OBB-a predstavlja ishodište njegovog koordinatnog sustava. S U_i označimo njegove koordinatne osi te s e_i odsječke na tim osima, $i = 0, 1, 2$ ako se radi o kvadru odnosno $i = 0, 1$ ako se radi o pravokutniku. Pretpostavimo da su $\sigma_i \in \{-1, 1\}$. Tada, prema Eberlyju vrhove kvadra predstavljamo s

$$P = C + \sigma_0 e_0 U_0 + \sigma_1 e_1 U_1 + \sigma_2 e_2 U_2.$$

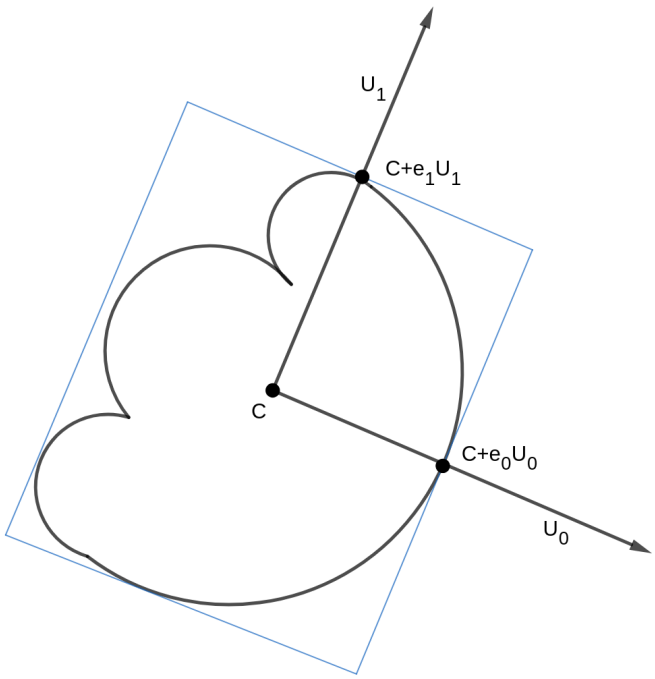
Analogno, vrhove pravokutnika u dvodimenzionalnom prostoru možemo predstaviti kao:

$$P = C + \sigma_0 e_0 U_0 + \sigma_1 e_1 U_1.$$

Za testiranje kolizije u 2D možemo koristiti jednostavni test koji uspoređuje udaljenost centara graničnih pravokutnika sa zbrojem duljina polovica dijagonala. Ovaj test prema Ericsonu može dati lažno pozitivne rezultate ako ga primijenimo na granične kvadre pa stoga se najčešće koristi test baziran na metodi razdvajajućih osi.



Slika 5.3: Orjentirani granični kvadar - preuzeto iz [7]



Slika 5.4: Orjentirani granični pravokutnik

Ericson u [8] opisuje nekoliko načina kako dobiti orijentirani granični pravokutnik. Algoritam najmanje vremenske složenosti $O(nlogn)$ je tzv. *rotating calipers* koji prvo kreira pravokutnik određen ekstremima. Barem jedna od bridova pravokutnika se podudara s bridom poligona. Zatim se u svakoj iteraciji pravci određeni bridovima pravokutnika simultano rotiraju suprotno od smjera kazaljke na satu oko svojih točaka sve dok se ne podudare s bridom poligona. Proces se ponavlja sve dok se pravci nisu zarotirali do 90 stupnjeva. Minimalni OBB je onaj čija je površina bila najmanja tijekom iteracija.

5.2.3 Granični pravokutnici i kvadri poravnati s koordinatnim osima (AABB)

AABB je posebna vrsta OBB-a čije su normale paralelne s osima odabranog koordinatnog sustava. Moguće je promatrati AABB u odnosu na koordinatni sustav "glavnog prostora" tj. okoline odnosno svijeta u računalnim igrama te u odnosu na koordinate jednog od objekata. AABB, kao i sferni oblik, zahtijeva jednostavnu strukturu te je odgovor na kolizije jednostavan. Uz to, izračunavanje optimalnog AABB-a svodi se na pronalazak ekstremnih vrijednosti u skupu vrhova poligona odnosno poliedra. Test preklapanja dvaju AABB-ova sastoji se od usporedbi koordinatnih vrijednosti (vrhova AABB-ova). Algoritam za 2D dan je s 2.5. Jednostavni test kolizije na graničnim kvadrima poravnatim s koordinatnim osima analogan je onome u 2D i svodi se na provjeru nalazi li se vrh jednog kvadra u prostoru koji zauzima drugi.

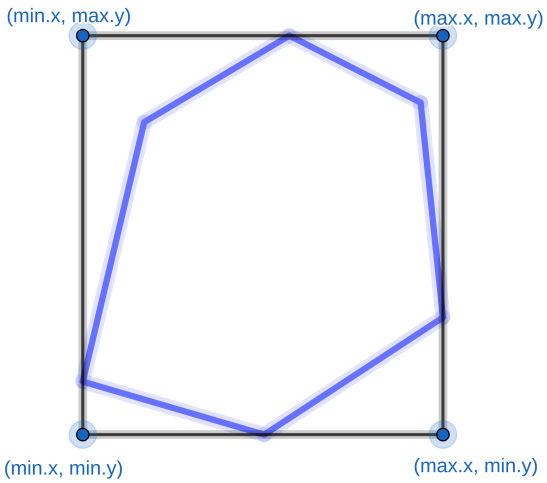
Izračunavanje AABB u 2D

Ulaz algoritma ponovno je vektor vrhova zadanog poligona. Potrebno je pronaći maksimum i minimum x i y koordinata zadanih točaka. Tako dobivene minimalna i maksimalna točka određuju AABB.

Algoritam 5.2: AABB

```
1  Rectangle AABB(Point* points){
2      int pointCount = points.size();
3
4      Point min = point[0];
5      Point max = point[0];
6
7      for(int i = 1; i < pointCount; ++i){
8          min.x = min.x > points[i].x ? points[i].x : min.x;
9          min.y = min.y > points[i].y ? points[i].y : min.y;
10         max.x = max.x < points[i].x ? points[i].x : max.x;
11         max.y = max.y < points[i].y ? points[i].y : max.y;
12     }
13
```

```
14      return Rectangle(min, max);
15  }
```



Slika 5.5: Primjer pronađenog AABB-a za konveksni poligon

Na sličan način dobije se granični kvadar poravnat s koordinatnim osima za zadani trodimenzionalni objekt.

k-DOP

Politopi diskretne orijentacije ili konveksne ljuske diskretnog smjera su generalizacije AABB-a. k-DOP se kao i prethodna dva granična oblika predstavlja skupom ravnina u 3D odnosno pravaca u 2D. Kako bismo kreirali k-DOP u 3D, prema [8] potrebne su nam $k/2$ normale koje su fiksno određene za sve objekte u prostoru. Za svaku od normala pozicioniraju se po dvije ravnine koje ona određuje takve da one obuhvate objekt s obje strane. Pozicije ravnina su minimalna i maksimalna vrijednost dobivene skalarnim produktom vrhova objekta i odabrane normale. Stoga je moguće reprezentirati k-DOP samo pomoću minimalnog i maksimalnog odsječka na svakoj od normala. Primjerice, za 8-DOP, Ericson daje sljedeću strukturu:

```
1  struct DOP8{
2      float min[4]; //minimum odsjecke na normalama s indeksima 0,1,2,3
3      float max[4]; //maksimume odsjecke na normalama s indeksima 0,1,2,3
4  }
```

Korištenjem dane strukture testiranje kolizije postaje vrlo brzo i jednostavno. Potrebno je samo provjeriti preklapaju li se minimalne i maksimalne vrijednosti na odgovarajućim normalama što je slično testu za AABB-ove. Što je vrijednost k veća, prekrivanje je bolje, ali s k raste i kompleksnost testiranja kolizije i usporava se transformacija kao odgovor na koliziju. Iako je k-DOP invarijantan ne translaciju, pri rotaciji, kao i kod AABB-a biti će potrebno ponovno izračunavanje.

Konveksna ljuska

Konveksna ljuska je najmanji konveksni skup koji obuhvaća dani objekt u potpunosti te je ujedno i najprecizniji granični oblik. Primjenjuje se isključivo na nekonveksnim oblicima. Dobivanje konveksne ljuske za zadani nekonveksni poligon slikovito možemo opisati s omotavanjem objekta izvana elastičnom trakom. Jedan od boljih algoritama za izračunavanje konveksne ljuske zadanog skupa točaka prema [11] je tzv. *quickhull*. Vremenska složenost *quickhull* algoritma je $O(nlogn)$. Korištenjem konveksne ljuske na nekonveksnim objektima može rezultirati detekcijom lažnih kolizija. Konveksne ljuske dovoljno su dobra aproksimacija nekonveksnog objekta u aplikacijama u kojima visoka preciznost nije ključna te greške aproksimacije neće uzrokovati štete.

Detaljnije o tome kako konstruirati pojedine oblike graničnih volumena i kako testirati postojanje kolizije među njima može se pronaći u [8] i [3].

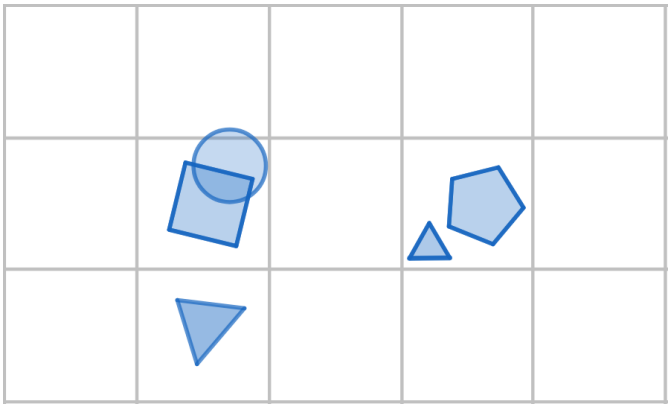
5.2.4 Odabir odgovarajućeg graničnog oblika

Sferni oblici su optimalni u smislu brzine testiranja kolizije i odgovora na koliziju (translacija, rotacija) no imaju velika odstupanja od rubova objekta. AABB su bolje prilagođeni objektu od sfere i kruga iako i dalje neprecizni te još uvijek rezultiraju relativno brzim testiranjem i odgovorom na koliziju. OOB su precizniji kod prekrivanja granica objekta, ali je zato testiranje kolizije i odgovor vremenski zahtjevniji. Kompromis između AABB i OOB predstavlja k-DOP koji kombinira dobru pokrivenost i relativno lako testiranje no sama izgradnja je zahtjevna te se treba zadržati u relativno niskim granicama pri izboru k. Najpreciznije preklapanje dobivamo uporabom konveksne ljuske. Konstrukcija konveksne ljuske i potrebne transformacije vremenski i prostorno su zahtjevne te ih se stoga koristi samo u slučajevima kada je preciznost detekcije kolizije izrazito važna. Pri izradi računalnih igara najčešće se izabiru AABB zbog svoje jednostavnosti implementacije i brzine algoritma. Uz navedene, postoje još brojni drugi oblici kao što su konusi, cilindri elipsoidi itd. Kao što smo već napomenuli, nekonveksne objekte je moguće aproksimirati skupom konveksnih oblika te tako jednom objektu pridružiti više graničnih oblika što će poboljšati točnost testa kolizije. Pridružujući mali broj jednostavnih graničnih oblika jednom objektu, kao što su AABB, OBB ili sferni oblici, neće drastično utjecati na smanjenje brzine algoritma.

Prilikom izrade računalnih igara, scena se često sastoji od velikog broja objekata koji najčešće neće kolidirati. Pošto je u računalnim igrama izrazito važna brzina potrebno je izbjeći nepotrebna testiranja. Ukoliko se scena sastoji od jako velikog broja objekata testiranje kolizija na graničnim oblicima svih objekata u širokoj fazi ponovno bi rezultiralo izrazito sporim algoritmom. Stoga uvodimo algoritme za particioniranje prostora koji će na brz način smanjiti broj potrebnih testova.

5.3 Particioniranje prostora

Particioniranje prostora dijeli Euklidski prostor na više disjunktnih manjih dijelova. Ako utvrdimo da se par objekata ne nalazi u istom dijelu, tada znamo da se objekti sigurno ne mogu sudariti, pa nije potrebno provoditi detaljnije testove. Ovakvim pristupom moguće je eliminirati veliki broj nepotrebnih testova široke faze.



Slika 5.6: Particioniranje prostora

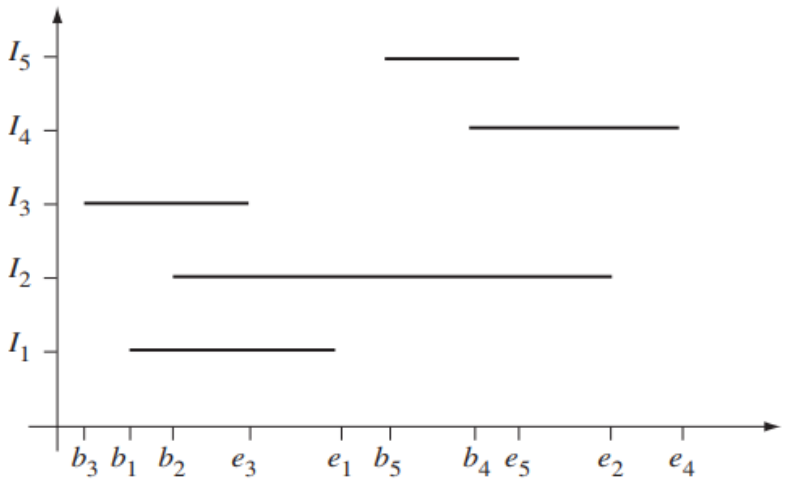
Postoje različiti načini i algoritama za particioniranje prostora, ali mi ćemo promotriti samo dva. Prema [11], neki od najčešće korištenih algoritama za particioniranje prostora su BVH i *sort and sweep* odnosno *sweep and prune*.

5.3.1 *Sweep and prune*

Ideja ovog pristupa je iskoristi svojstvo vremenske koherencije o kojoj smo govorili u trećem poglavlju.

Prvo ćemo prikazati rad algoritma na 1D. S $I_i = [b_i, e_i]$, $i = 1, ..., n$ označimo intervale za koje želimo na što efikasniji način pronaći parove koji su u koliziji. Pritom vrijednosti

b_i određuju donju, a e_i gornju granicu intervala. Svaki interval predstavljen je parom njegovih ekstremnih vrijednosti. *Sweep and prune* algoritam započinje tako što ubacimo sve minimalne vrijednosti b i maksimalne vrijednosti e svih intervala u listu te ih sortiramo.



Slika 5.7: Prikaz sweep faze algoritma - slika preuzeta iz [7]

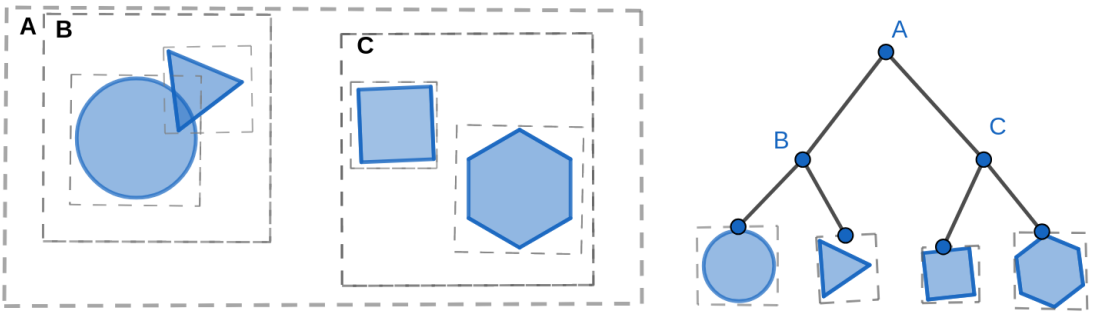
Pri tome pretpostavljamo da vrijedi $b_i < e_i, \forall i$. Zatim prolazimo po listi vrijednosti i kada je pronađena vrijednost b_i ubacujemo interval I_i u listu aktivnih intervala. Kada naiđemo na vrijednost e_i izbacujemo interval I_i iz liste aktivnih intervala. Za svaki interval I_i spremamo sve intervale koji su bili u listi aktivnih intervala u trenutku čitanja b_i iz liste vrijednosti. U svakom trenutku svi intervali koji se nalaze u listi aktivnih intervala međusobno se sijeku. Zahvaljujući vremenskoj i prostornoj koherenciji, između dva vremenska koraka položaji intervala neće se drastično razlikovati te će lista vrijednosti ostati skoro sortirana. Stoga za sortiranje liste izabiremo algoritme koji su brzi pri sortiranju gotovo sortiranih listi. Primjeri takvih algoritama su *insertion sort* i *quicksort*. Faza sortiranja zahtijeva najviše $O(nlogn)$, faza provjere presjeka intervala $O(n)$, što će dati ukupno asimptotsko vrijeme $O(nlogn)$. Bilježenje kolizije, odnosno preklapanja intervala je $O(k)$ vremenske složenosti gdje je k broj preklapanja u danoj iteraciji. Slijedi da je vremenska složenost ovog algoritma $O(nlogn + k)$.

Poopćenje na 2D uključuje dvije liste vrijednosti, jednu za odsječke na x-osi i drugu za odsječke na y-osi. Paralelno se sortiraju obje liste te testira preklapanja. Uvjet za preklapanje u 2D je da se preklapaju i x-intervali i y-intervali AABB-ova. Ako je utvrđeno preklapanje x-intervala, tada se testiraju i odgovarajući y-intervali. Pravokutnici se više ne preklapaju ako je x-intervali ili y-intervali nakon pomaka objekata više ne preklapaju. Sličan pristup primjenjuje se i u 3D.

5.3.2 Hijerarhija graničnih oblika (*Bounding volume hierarchy - BVH*)

Za razliku od prethodnog pristupa hijerarhija graničnih oblika grupira granične oblike pridružene objektima svijeta na temelju udaljenosti. Sličnost s prethodnim pristupom je u tome što se BVH ne bavi cijelim prostorom na svim razinama nego promatra dijelove prostora koji obuhvaćaju objekte. Hijerarhija se reprezentira u obliku k-narnog stabla gdje korijen predstavlja granični oblik kojim se obuhvaćaju svi objekti svijeta obuhvaćen njegovim graničnim oblikom. Unutarnji čvorovi sadrže pokazivače na svoju djecu. Svaki list stabla sadrži po jedan objekt svijeta. Svaki od čvorova, osim listova, ima pridružen granični oblik koji obuhvaća geometrijske oblike svojeg podstabla.

Ako nam je potrebna visoka preciznost testa na nekonveksnim objektima, preporučuje se dijeljenje istog na više preciznijih konveksnih dijelova. Pošto hijerarhija tretira svaki dio nekonveksnog objekta kao zasebni objekt slijedi da će se time povećati broj čvorova stabla. Korištenjem većeg broja čvorova povećava se kompleksnost testa te je on vremenski zahtjevniji. Zato je izrazito važno paziti na balansiranost stabla. Balansirano stablo je stablo čije su sve grane podjednake dubine. Generalno, visina balansiranog stabla je $\lfloor log_k n \rfloor$ gdje je n broj svih čvorova, a k maksimalan broj djece svakog čvora. Gradnjom hijerarhije u obliku dobro balansiranog binarnog stabla smanjujemo vremensku složenost ispitivanja kolizije tako da već u prvom koraku eliminiramo približno polovinu graničnih objekata za koje znamo da nisu u koliziji. Stoga je ovom metodom potrebno napraviti $O(logn)$ testova za n čvorova stabla. U nastavku opisujemo jednostavni primjer korištenja BVH te nekoliko načina kako izgraditi BVH.



Slika 5.8: Hijerarhija graničnih oblika

Ispitivanje postojanja kolizije između dvaju objekata s BVH ilustriramo jednostavnim primjerom na slici 5.8. [3] razlikuje dva načina provođenja testa kolizije na BVH. Prvi je

hijerarhijsko testiranje graničnog oblika svakog objekta sa stablom hijerarhije. Ako želimo utvrditi kolizije kruga potrebno je krenuti od korijena stabla *A*. Pošto je krug nalazi u *A*, testiranje se nastavlja s djecom čvora *A*, tj. graničnim poligonima *B* i *C*. Testiranjem graničnog oblika pridruženog krugu s *C* utvrđuje se da nema kolizije te se podstablo čiji je korijen *C* odbacuje. Testiranje se nastavlja s djecom čvora *B*. Pošto su djeca čvora *B* listovi, važno je paziti da se za granični oblik ćije kolizije tražimo ne prijavi koliziju sa samim sobom. Testiranjem s graničnim oblikom pridruženim trokutu utvrđuje se potencijalna kolizija kruga i trokuta.

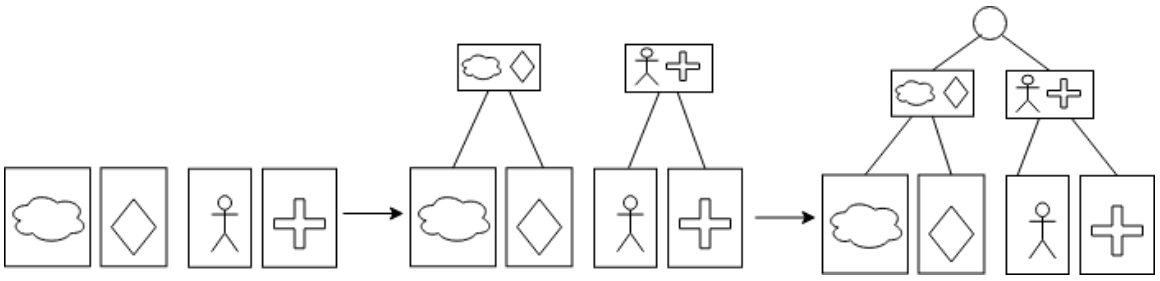
Drugi način je testirati stablo hijerarhije sa samim sobom. Testiranje staje kada se utvrdi da se granični oblici ne preklapaju na obje grane nekog nivoa ili kada se utvrdi da su dva različita lista u koliziji. Testiranje počinje u korijenu i pošto se korijen preklapa sam sa sobom, prelazimo na testiranje njegove djece. Utvrđujemo da se čvor B preklapa sam sa sobom te testiramo odnose njegove djece. Pošto smo došli do listova i jedina kombinacija u kojoj ne testiramo listove same sa sobom daje koliziju, bilježimo da su krug i trokut u potencijalnoj koliziji. Testiranje se nastavlja s čvorovima B i C gdje se utvrđuje da oni nisu u koliziji te se ova mogućnost odbacuje. Preostalo je još testirati preklapanje čvora C sa samim sobom te, nakon potvrdnog odgovora analognim postupkom kao i kod čvorova B, testiramo međusobna preklapanja djece čvora C.

5.3.3 Izgradnja BVH

Struktura koja se najčešće koristi pri izgradnji BVH je *k*-narno stablo tj. stablo čiji čvorovi mogu imati najviše *k* djece. Najčešći izbor parametra *k* je prema [3] 2, 4 i 8. Razlog tomu je što je izgradnja takvih stabala jednostavna. Za vrijednost *k* izabiremo 2 pošto je ona najčešće korištena u praksi. Prema poglavlju *"Collision Detection"* iz [3], postoje četiri glavna pristupa za gradnju BVH, a to su *bottom-up*, *top-down*, inkrementalni pristup te linearni pristup.

Bottom-up

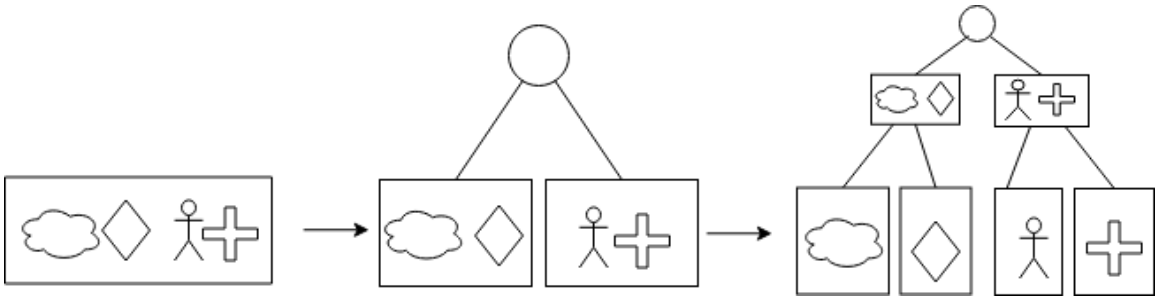
Na početku svakom objektu kreiramo njegov granični oblik. Izgradnja BVH *bottom-up* pristupom započinje grupiranjem nekoliko najbližih objekata i kreiranjem graničnog oblika za njih. Pri pronalaženju najbližih objekata koristimo funkciju udaljenosti između objekata. Zatim grupiramo postojeće granične oblike u novi, veći oblik. Algoritam se nastavlja sve dok ne dobijemo jedan granični oblik koji obuhvaća sve objekte. Tako dobiveni oblik postaje korijen stabla te su zbog korištenja funkcije udaljenosti dobiveni oblici optimalne veličine.



Slika 5.9: Kreiranje BVH bottom-up pristupom

Top-down

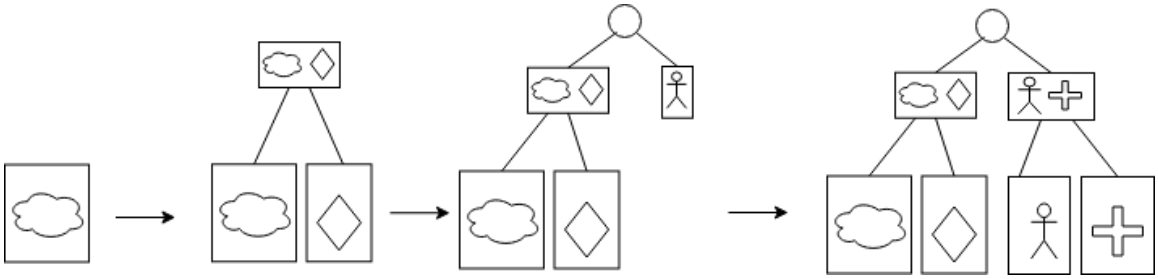
Top-down pristup, nasuprot *Bottom-up* pristupu, počinje kreiranjem jednog graničnog oblika kojim obuhvaćamo sve objekte. Taj oblik predstavlja korijen stabla. Zatim se dobiveni granični oblik dijeli na najviše k dijelova gdje je k unaprijed zadana konstanta te se za svaki dio pronalaze odgovarajući objekti. Postupak se ponavlja rekurzivno sve dok svakom objektu ne bude pridružen njegov granični oblik.



Slika 5.10: Kreiranje BVH top-down pristupom

Inkrementalni pristup

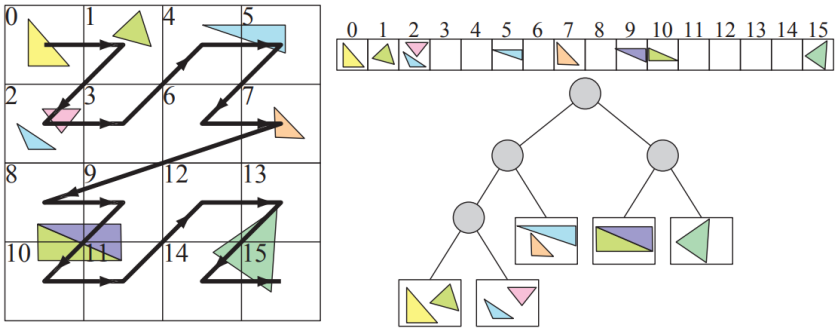
Inkrementalni pristup započinje s praznim stablom. U svakom koraku dodajemo novi čvor odnosno jedan objekt s pridruženim graničnim oblikom. Ključno je naći najbolje mjesto za umetanje novog čvora u stablo kako bi stablo bilo balansirano tj. kako bi zadržalo što manju dubinu. Prvo izračunamo cijene umetanja objekta na različita mjesta u stablu te izaberemo ono kojem je cijena najmanja. Prema [3] ovaj algoritam je $O(nlogn)$ vremenski složen. Uz to korisno je dodavati objekte u stablo slučajnim poretkom.



Slika 5.11: Kreiranje BVH pristupom inkrementalnim umetanjem

Linearni pristup

Prvo se kreira kvadrat oko svih objekata te mreža kvadrata unutar njega tako da je ukupan broj čelija djeljiv s 4. Zatim se rekurzivno kreira krivulja Z oblika kao na slici 5.12. U svakom rekurzivnom pozivu trenutni kvadrat dijeli se na četiri jednaka dijela i to dok se dobiveni dijelovi ne budu sasatojali od po jedne čelije. Svakoj čeliji dodjeljuje se njezin broj prativši dobivenu krivulju s početkom u gornjem lijevom kutu glavnog kvadrata. Objektima se pridružuju kodovi na temelju toga kojoj čeliji pripada njegovo težište. Objekti se zatim spremaju u listu na odgovarajuća mjesta obzirom na dodijeljene kodove. Zatim se kreira binarno stablo tako što se lista rekurzivno dijeli na pola te se u svakom pozivu kreira unutarnji čvor koji povezuje svoje objekte. Proces se zaustavlja kad u listi ostane 1 objekt koji predstavlja list ili kada lista postane prazna.



Slika 5.12: Kreiranje BVH linearnim pristupom - preuzeto iz [3]

Poglavlje 6

Demonstracija rada osnovnih algoritama za detekciju kolizije u 2D

6.1 Potrebno okruženje i biblioteke

Praktični dio rada sastoji se od skupa programa izrađenih u jeziku C++ kojima demonstriramo osnovne algoritme za detekciju kolizija u 2D. Pri demonstraciji osnovnih algoritama za detekciju kolizija koristit ćemo biblioteke Box2D i Sfml. Biblioteka Box2D namijenjena je simuliranju fizike igre. Pošto Box2d ne predviđa metode za grafičko prikazivanje objekata, kombiniramo ga s bibliotekom Sfml koja je namijenjena vizualizaciji objekata. U izradi ovog rada korištene su verzije Box2d (verzija 2.3.2) i Sfml (verzija 2.5.1).

6.1.1 Box2D

Box2D je biblioteka namijenjena za simulaciju fizike krutih tijela te detekciju kolizija između njih. Box2D podržava simulacije u dvodimenzionalnom prostoru na tijelima nepromjenjivog oblika. Izvedena je u programskom jeziku C++ te je portabilna.

Box2D se sastoji od tri modula:

- Common
- Collision
- Dynamics.

Common se brine o alokaciji memorije, matematici i postavkama. Pomoću Collision modula definiramo oblike (*shapes*), on provodi široku fazu te se bavi kolizijskim upitima i funkcijama. Dynamics modul osigurava simulaciju objekata `world, body, fixture` i `joint`.

6.1.2 Kreiranje svijeta i osnovne postavke

Kako bismo mogli kreirati objekte i nihove interakcije primjenom Box2D biblioteke prvo je potrebno kreirati instancu klase `b2World` te joj zadati željenu gravitaciju. Za potrebe programa koristimo standardnu gravitaciju.

```
1     b2Vec2 gravity(0.0f, 9.81f);
2     b2World world(gravity);
```

Uz to, potrebno je postaviti vremenski korak za ažuriranje.

```
1  // vremenski korak, osvjezavanje 60 puta u sekundi
2  #define TIMESTEP 1.0f/60.0f
3  // broj iteracija za racunanje brzine -> standardno 8
4  #define VELITER 8
5   // broj iteracija za racunanje pozicije -> standardno 3
6  #define POSITER 3
7           (...)
8  //osvjezi b2World
9  world.Step(TIMESTEP, VELITER, POSITER);
```

Box2D predstavlja tijela objektima tipa `b2Body`. Svaki objekt tipa `b2Body` je nosioc objekata `b2Shape` koji predstavljaju geometrijska obilježja tijela. `b2Body` objekt ima poziciju i brzinu kretanja te se na njega može primijeniti sila ili impuls tj. promijeniti brzinu i smjer kretanja.

Box2D razlikuje tri vrste tijela:

- `b2_staticBody` - statička tijela koja kolidiraju samo s kinematičkim i dinamičkim tijelima
- `b2_dynamicBody` - dinamička tijela
- `b2_kinematicBody` - kinematička tijela koja se kreću zadanom brzinom no na njih ne djeluju sile.

Prije kreiranja tijela, potrebno je kreirati definiciju tijela `b2BodyDef`. Definiciji pridružujemo informacije potrebne za kreiranje i inicijalizaciju tijela. Pri kreiranju `b2Body` tijela kopiraju se informacije iz `b2BodyDef` definicije. Preko definicije `b2BodyDef` moguće je zadati tip tijela, poziciju, nagib, faktore prilagodbe gravitacije, smanjenja brzine i drugo. Glavni razlog korištenja `b2BodyDef` definicije je to što je tip tijela potrebno postaviti pri kreiranju. Postavljanje tipa nakon kreiranja `b2Body` tijela izrazito je skupa operacija te se ono ne preporučuje. Nadalje, kako bismo `b2Body` tijelu pridružili geometrijski oblik `b2Shape`, koristimo učvršćenje `b2Fixture` te za njega kreiramo definiciju tipa

`b2FixtureDef`. `b2FixtureDef` sadržava geometrijski oblik, informacije za široku fazu, gustoću, trenje i restituciju, zastavice za filtriranje sudara, pokazivač na matično tijelo, korisničke podatke (`UserData`) te zastavicu senzora.

Za potrebe programa kreiramo statička i dinamička tijela. Slijedi primjer izrade jednog dinamičkog `b2Body` tijela s kružnim oblikom. Potrebno je kreirati objekt `b2CircleShape` te mu zadati poziciju i radijus. Preko definicije tijela `b2BodyDef` postavljamo tip tijela na dinamičnički i kreiramo tijelo s tom definicijom. Na kraju na tijelo učvršćujemo geometrijski oblik `b2CircleShape` koristeći `b2FixtureDef` definiciju.

```
1   b2Body *m_body;
2   b2CircleShape circleShape;
3   b2BodyDef bodyDef;
4   b2FixtureDef fixtureDef;
5
6   circleShape.m_p.Set(0, 0);
7   circleShape.m_radius = 1.f;
8   bodyDef.type = b2_dynamicBody;
9   bodyDef.position.Set(10.f,10.f);
10
11  m_body = world.CreateBody(&bodyDef);
12  fixtureDef.shape = &circleShape;
13  m_body->CreateFixture(&fixtureDef);
```

Jednom `b2Body` tijelu, korištenjem učvršćenja, moguće je pridružiti više `b2Shape` objekata. Također, svakom tijelu moguće je postaviti osobine kao što su masa, elastičnost itd. Detaljnije o tome nalazi se u [5].

6.1.3 Široka faza i dinamička stabla

Klasa `b2BroadPhase` provodi široku fazu nad objektima koristeći hijerarhiju graničnih oblika implementiranu klasom `b2DynamicTree`. Široka faza će reducirati broj parova koje treba ispitati uska faza. U uskoj fazi Box2D pronalazi kontaktne skupove među preostalim parovima. Box2D koristi klasu `b2DynamicTree` za uspješno upravljanje velikim brojem `b2Shape` objekata. Klasa radi na AABB-ovima koji su dodijeljeni oblicima te reprezentira hijerarhiju graničnih pravokutnika pomoću binarnog stabla. Sama klasa održava balansiranost stabla korištenjem rotacija. Box2D korištenjem ove klase eliminira veliki broj testova kolizije te ubrzava rad aplikacije pogotovo ako se ona sastoji od velikog broja objekata. Box2D interno provodi široku fazu te ju nije potrebno samostalno kreirati.

Više o biblioteci Box2D može se pronaći u [5].

6.1.4 SFML

Grafiku praktičnog dijela rada izvodimo pomoću biblioteke SFML koja je, kao i Box2D, napisana u programskom jeziku C++. Algoritam 6.1 prikazuje osnovno SFML okruženje.

Algoritam 6.1: Osnovni SFML program

```
#include
int main()
{
    //kreiraj glavni prozor
    sf::RenderWindow window(sf::VideoMode(800, 600), "SFML window");

    //glavna petlja
    while (window.isOpen())
    {   (...)
        //o isti scenu
        window.clear();
        //iscrtaj sprite
        window.draw(sprite);
        //azuriraj scenu
        window.display();
    }
    return EXIT_SUCCESS;
}
```

Više o biblioteci SFML može se pronaći na [9].

6.1.5 Povezivanje biblioteka Box2D i SFML

Box2D sadrži klasu `b2Draw` koja služi za iscrtavanje objekata i svrha joj je olakšati debugiranje. `b2Draw` samo navodi metode za iscrtavanje te ih je potrebno samostalno implmentirati za potrebe vlastitog programa. Implementacija `b2Draw` iz skupa testnih programa pridruženih biblioteci koristi OpenGL za grafiku, no iscrtava samo osnovne oblike i nije pogodna za razvoj igara. Stoga je potrebno kreirati klasu koje nasljeđuje klasu b2Draw i prilagoditi njezine funkcije za iscrtavanje za naše potrebe. Sljedeći algoritam prikazuje definiciju klase `SFMLDebugDraw` koja služi za iscrtavanje objekata `b2Shape` pridruženih objektima tipa `b2Body` koji reprezentiraju objekte simulacije. Funkcije `GLColorToSFML` i `B2VecToSFVec` služe za konverziju `b2Color` boja u `sf::Color` boju odnosno prostornih koordinata Box2D-a u piksele.

Algoritam 6.2: SFML iscrtavanje za debug

```
class SFMLDebugDraw : public b2Draw
{
private:
    sf::RenderWindow* m_window;
public:
    SFMLDebugDraw(sf::RenderWindow &window);

    static sf::Color GLColorToSFML(const b2Color &color,
        sf::Uint8 alpha = 255);

    static sf::Vector2f B2VecToSFVec(const b2Vec2 &vector,
        bool scaleToPixels = true);

    void DrawPolygon(const b2Vec2* vertices, int32 vertexCount,
        const b2Color& color);

    void DrawSolidPolygon(const b2Vec2* vertices,int32 vertexCount,
        const b2Color& color);

    void DrawCircle(const b2Vec2& center,float32
        radius,const b2Color& color);

    void DrawSolidCircle(const b2Vec2& center, float32 radius,
        const b2Vec2& axis,const b2Color& color);

    void DrawSegment(const b2Vec2& p1, const b2Vec2& p2,
        const b2Color& color);

    void DrawTransform(const b2Transform& xf);

    void DrawPoint(const b2Vec2& p, sf::Color color);
};
```

`b2Body` objekt povezujemo s njegovim `sf::Shape` objektom preko pozicije, kuta, te veličine. Najvažnije komponente povezivanja će biti pozicija i rotacija budući da kreiramo dinamičke objekte te želimo da pomak `b2Body` objekta prati pomak njegove vizualizacije.

Kako bismo ilustrirali vizualiziranje Box2D tijela zbog preglednosti prikazujemo samo implementaciju funkcije `DrawSolidPolygon`.

Algoritam 6.3: DrawSolidPolygon

```
void SFMLDebugDraw::DrawSolidPolygon(const b2Vec2* vertices,
int32 vertexCount, const b2Color& color){
    sf::ConvexShape polygon(vertexCount);
```

```
4
5       for(int i = 0; i < vertexCount; i++){
6           polygon.setPoint(i, sf::Vector2f(
7               std::floor(SFMLDebugDraw::B2VecToSFVec(vertices[i]).x),
8               std::floor(SFMLDebugDraw::B2VecToSFVec(vertices[i]).y)));
9       }
10
11      polygon.setOutlineThickness(-1.f);
12      polygon.setFillColor(SFMLDebugDraw::GLColorToSFML(color, 60));
13      polygon.setOutlineColor(SFMLDebugDraw::GLColorToSFML(color));
14
15      m_window->draw(polygon);
16  }
```

Kako bismo mogli iscrtavati objekte potrebno je još dodati sljedeće dvije linije u glavni program:

```
1       SFMLDebugDraw debugDraw(m_window);
2       m_world.SetDebugDraw(&debugDraw);
```

Iscrtavanje se izvršava u glavnoj petlji prilikom ažuriranja cijele scene pomoću `world.DrawDebugData();`. Na ovaj način dobivamo mogućnost iscrtavanja sličnu onoj koju je implementirana kao test za Box2D. Kako bismo iscrtavali teksture i animacije svakom objektu simulacije pridružit ćemo metodu `draw`.

6.2 Vrijeme kontakta i kontaktne točke

Kako bismo na najočitiji način prikazali kolizije i kontaktne točke u programu `simple_shapes` kreiramo nekoliko jednostavnih geometrijskih likova polazeći od resursa [6]. Klasa `Shapes` bazna je klasa iz koje izvodimo sljedeće geometrijske oblike: kvadrat, krug, trokut i romb. U svakoj od izvedenih klasa u konstruktoru kreiramo `b2Body` objekt kojem pridružujemo odgovarajući `b2Shape` koji mu daje geometrijski oblik, poziciju i veličinu. Uz to svaki objekt ima pripadnu metodu `void draw(sf::RenderWindow window)` koja kreira pripadni `sf::Shape` te ga iscrtava.

Box2D implementira bilježenje kontakata i kontaktnih točaka. Stoga je potrebno samo u svakom koraku simulacije proći kroz kolekciju kontakata i njihovih kontaktnih točaka te prilagoditi njihove podatke za vlastite potrebe.

Kontakte objekata prikazujemo promjenom boje u crvenu. Stoga klasi `Shape` dodajemo dvije dodatne metode: `void startContact()` i `void endContact()`. Metoda `void startContact()` inkrementira varijablu članicu `m_contacting` koja predstavlja broj objekata s kojima dani objekt dolazi u dodir. Metoda `void endContact()`

dekrementira `m_contacting` kada kontakt danog objekta s nekim objektom završi. U `draw` metodi objekta, ako je `m_contacting` > 0, postavljamo boju sf::Shape-a na `sf::Color::Red`, a inače je boja postavljena na vrijednost varijable članice `m_color` danog objekta. Posljedica će biti da će naš objekt postati crven kada se dogodi kolizija te će takav ostati sve dok se ne razriješe svi njegovi kontakti s drugim objektima.

Box2D sadrži klasu `b2ContactListener` s metodama:

- `void BeginContact(b2Contact* contact)`
- `void EndContact(b2Contact* contact)`
- `void PreSolve(b2Contact* contact, const b2Manifold* oldManifold)`
- `void PostSolve(b2Contact* contact, const b2ContactImpulse* impulse)`.

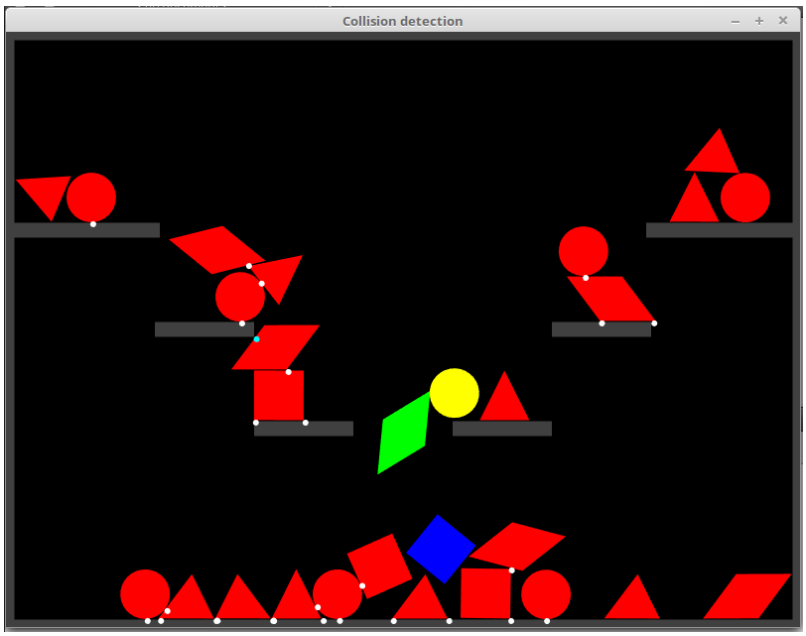
Kako bismo mogli iskoristiti podatke kontakta potrebno je implementirati vlastite verziju klase `b2ContactListener`, kreirati instancu te klase u glavnom programu (`main.cpp`) prije glavne petlje te je postaviti kao slušatelja kontakata korištenjem metode `b2World::SetContactListener`. Za potrebe ovog programa kreiramo klasu `ContactListener` koja nasljeđuje `b2ContactListener` i predefiniramo prve tri metode. `BeginContact` će za svaki kontakt iz `b2Contact` liste pronaći dva odgovarajuća tijela te ako su dinamička pozvati njihovu metodu `startContact` koja će postaviti crvenu boju. Lista koju prima `BeginContact` sadrži kontakte koji su započeli dok `EndContact` prima listu kontakata koji su završili. `EndContact` će analogno proći kroz listu i na dinamičkim tijelima pozvati metodu `endContact`.

Algoritam 6.4: Klasa ContactListener

```
1
2  #ifndef CONTACTLISTENER_H
3  #define CONTACTLISTENER_H
4  #include
5
6  //struktura za info o kontaktnim tockama
7  struct ContactPoint{
8      b2Vec2 position;
9      b2PointState state;
10 };
11
12 class ContactListener : public b2ContactListener {
13 public:
14     int m_pointCount;
15     ContactPoint m_points[300];
```

```
16
17      void BeginContact(b2Contact* contact) {
18        auto tipA = contact->GetFixtureA()->GetBody()->GetType();
19        auto tipB = contact->GetFixtureB()->GetBody()->GetType();
20        if ( tipA == b2_dynamicBody ) {
21            static_cast( contact->GetFixtureA()->
22            GetBody()->GetUserData() )->startContact();
23        }
24
25        if ( tipB == b2_dynamicBody ) {
26           static_cast( contact->GetFixtureB()->
27           GetBody()->GetUserData() )->startContact();
28        }
29      }
30
31      void EndContact(b2Contact* contact) {
32          auto tipA = contact->GetFixtureA()->GetBody()->GetType();
33          auto tipB = contact->GetFixtureB()->GetBody()->GetType();
34          if ( tipA == b2_dynamicBody ) {
35              static_cast( contact->GetFixtureA()->
36              GetBody()->GetUserData() )->endContact();
37          }
38
39          if ( tipB == b2_dynamicBody ) {
40               static_cast( contact->GetFixtureB()->
41               GetBody()->GetUserData() )->endContact();
42          }
43      }
44
45      void PreSolve(b2Contact* contact, const b2Manifold* oldManifold){
46           const b2Manifold* manifold = contact->GetManifold();
47           if(!manifold->pointCount)    return;
48
49           b2PointState state1[b2_maxManifoldPoints],
50           state2[b2_maxManifoldPoints];
51           b2GetPointStates(state1, state2, oldManifold,
52           manifold);
53
54           b2WorldManifold wM;
55           contact->GetWorldManifold(&wM);
56
57           for(int i = 0; i < manifold->pointCount &&
58             m_pointCount < 300; ++i){
59               ContactPoint* cp = m_points + m_pointCount;
60               cp->position = wM.points[i];
61               cp->state = state2[i];
62               ++m_pointCount;
```

```
63              }
64      }
65  };
66
67  #endif
```



Slika 6.1: Odgovor na koliziju i kontaktne točke

Box2D biblioteka sadrži klasu `b2Contact` koja služi za upravljanje kolizijama između dva `b2Fixture` objekta. Postoji više vrsta kontakata izvedenih iz `b2Contact` ovisno o tipu pridruženih `b2Shape` objekata (poligon, krug itd.). Box2D aproksimira kontakte među objektima pomoću točaka te je kontaktna dužina predstavljena pomoću dvije kontaktne točke. `b2Collision::b2Manifold` je struktura koja sadrži informacije o kontaktnim točkama. Svaka instanca klase `b2Contact` sadrži instancu strukture `b2Collision::b2Manifold` te metodu `GetManifold` koja daje informacije o točkama koje sudjeluju u tom kontaktu. Uz to sadrži i metodu `GetWorldManifold` koja pomoću trenutne globalne pozicija tijela računa globalne pozicije kontaktnih točaka. Manifold sadrži informacije o dinamici i geometrijskim obilježjima. Kako bismo prikazali kontaktne točke prvo kreiramo strukturu `ContactPoint` koja sadrži varijablu koja čuva globalnu poziciju točke te varijablu za stanje točke. Box2D razlikujemo četiri stanja kontaktne točke:

- `b2_nullState` - nepostojeća točka
- `b2_addState` - novonastale točke
- `b2_persistState` - točke nerazriješenih kontakata
- `b2_removeState` - točka je uklonjena tijekom osvježenja stanja.

Pri iscrtavanju razlikujemo dvije vrste stanja točaka, `b2_addState` i `b2_persistState` tako što nove točke obojimo plavo-zelenom (Cyan) bojom dok starima dajemo bijelu boju. U metodi `PreSolve` kreiramo niz točaka `ContactPoint` s informacijama iz manifolda svih kontakata u nizu `contact`. U glavnom programu iscrtavamo točke pomoću metode `SFMLDebugDraw::DrawPoint` na instanci klase `SFMLDebugDraw`.

6.3 Kontinuirana detekcija kolizija

U Box2D simulacijama problem tuneliranja može se pojaviti samo između dva ili više dinamičkih tijela. Box2D primjenjuje algoritam CCD na statička i kinematička tijela po *defaultu*, dok je za dinamička tijela potrebno postaviti `body.bullet = true`. CCD se ne primjenjuje na sva dinamička tijela kako bi se zadržale razumne performanse algoritma. Box2D implementira funkciju `b2TimeOfImpact` koja utvrđuje vrijeme prvog kontakta dvaju dinamičkih tijela. Glavna svrha joj je prevenirati tuneliranje. Funkcija utvrđuje os razdvajanja i osigurava da se projekcije objekata ne preklope na toj osi. Funkcija zahtijeva dva dinamička objekta pretvorena u `b2DistanceProxy` objekte te dvije `b2Sweep` strukture za pohranu inicijalne i finalne transformacije objekata.

6.3.1 Algoritam

Algoritam za sprječavanje tuneliranja prvo interpolira kretanja tijela kako bi pronašao vrijeme prvog kontakta. To radi tako da kroz jedan vremenski korak pomiče tijela s njihovih starih pozicija na nove pozicije te tijekom pomicanja bilježi nove kolizije. Zatim pomiče tijela na pozicije prvog kontakta i rješava kolizije te izvrši potkorak (izvodi se sve što će se dogoditi u preostalom vremenu) kako bi završio taj vremenski korak.

6.3.2 `CCD` implementacija

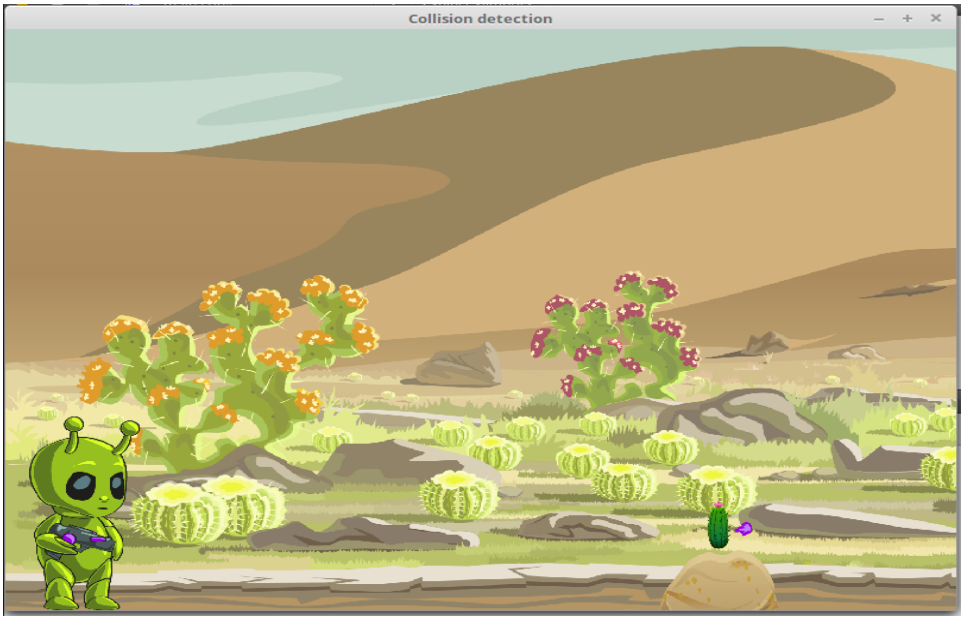
Kao primjer rada algoritma kontinuirane detekcije kolizija razvijamo program *CCD*. Glavna komponenta ovog programa je klasa *Bullet* koja predstavlja metak kojim igrač pokušava pogoditi metu predstavljenu klasom *Target*.

Objekt tipa `Bullet` predstavljamo kao `b2CircleShape` drastično manje veličine od ostalih objekata te mu u konstruktoru dajemo linearno ubrzanje primjenom sile na x koordinatu njegovog centra. To se postiže pozivom funkciju `ApplyForce` na `b2Body` tijelu objekta. Funkcija kao argumente prima `b2Vec2` vektor s iznosom sile na svaku od koordinata te točku tijela na koju će se sila primijeniti.

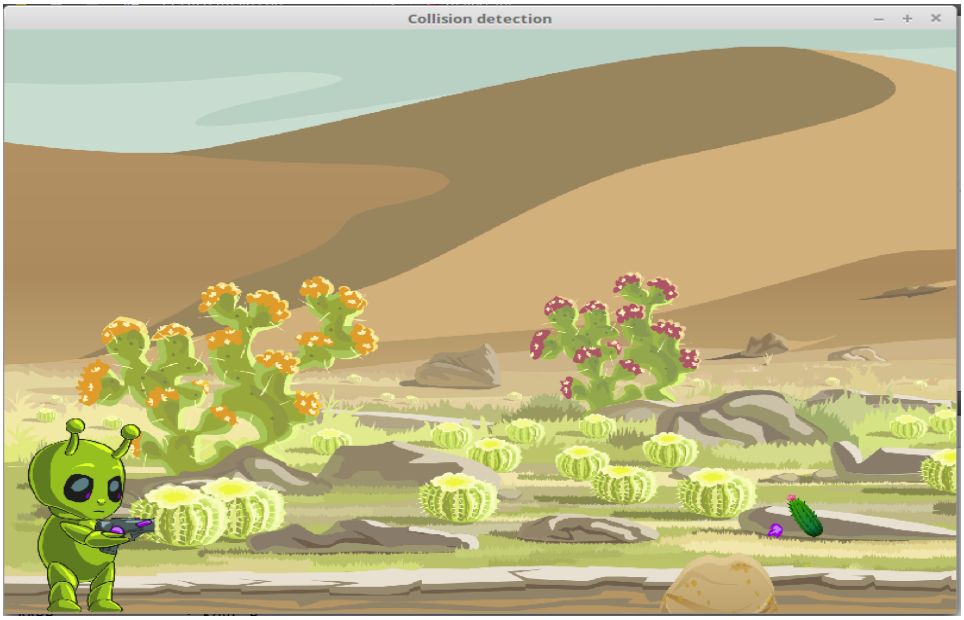
Algoritam 6.5: Kontruktor klase Bullet

```
1  Bullet::Bullet(b2World &world, sf::Vector2f center){
2      m_name = "Bullet";
3      m_size = sf::Vector2f(15,15);
4      m_radius = m_size.x / 2;
5
6      b2CircleShape circleShape;
7      b2BodyDef bodyDef;
8      b2FixtureDef fixtureDef;
9
10     circleShape.m_p.Set(0, 0);
11     circleShape.m_radius = m_radius * MPP;
12     bodyDef.type = b2_dynamicBody;
13     bodyDef.position.Set(center.x * MPP, center.y * MPP);
14
15     ////////////////////////////////
16     ///postavi bullet mod za primjenu CCD
17     //bodyDef.bullet = true;
18     ////////////////////////////////
19
20     m_body = world.CreateBody(&bodyDef);
21     m_body->ApplyForce(b2Vec2(100,0), m_body->GetWorldCenter(), true);
22     fixtureDef.shape = &circleShape;
23     fixtureDef.friction = 1;
24     fixtureDef.restitution = 0;
25     fixtureDef.density = 0.7f;
26     m_body->CreateFixture(&fixtureDef);
27
28     m_texture.loadFromFile("flash1.png");
29     m_body->SetUserData(this);
30
31 }
```

Komentirani dio algoritma 6.5 ključni je dio programa. Ako želimo primijeniti kontinuirani test kolizije na objekt tipa `Bullet` potrebno je otkomentirati liniju 17 u 6.5. Da bismo vidjeli razliku instanciramo dinamički objekt tipa `Target` i pokušamo ga pogoditi sa i bez linije `bodyDef.bullet = true;`. Sljedeće slike prikazuju dobivene efekte.



Slika 6.2: CCD program bez korištenja kontinuirane detekcije kolizija



Slika 6.3: CCD program s korištenjem kontinuirane detekcije kolizija

Pri izradi ovog programa koristimo besplatno dostupne teksture preuzete s [1] i [2]

6.4 Nekonveksni objekti i granični oblici

Programom `nonconvex_shapes` predstavljamo kako tretirati nekonveksne oblike te primjenu graničnih oblika. Pri izradi ovog programa koristimo se teksturama preuzetim s [4].

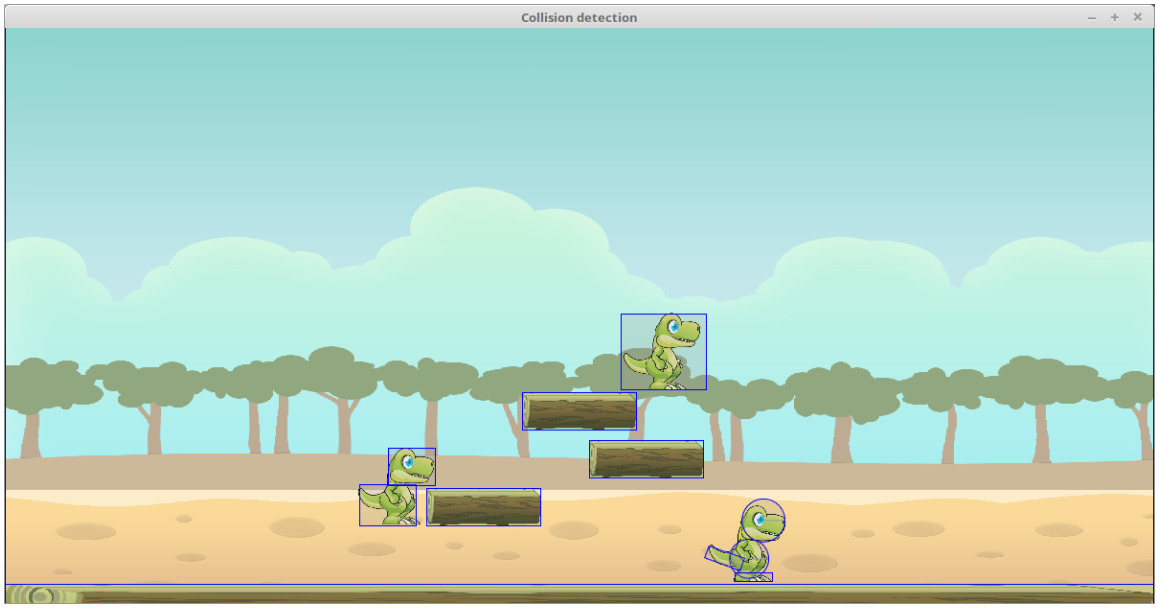
Ideja je prikazati aproksimaciju jednog nekonveksnog objekta na tri načina. Stoga kreiramo tri klase `Character1`, `Character2` i `Character3`. Klasa `Character3` će reprezentirati objekt na najjednostavniji način pridružujući `b2Body` tijelu samo jedan `b2Shape` pravokutnog oblika koji će napraviti precizno pokrivanje nekonveksnog objekta.

Slika 6.4: S lijeva: Character3, Character2, Character1

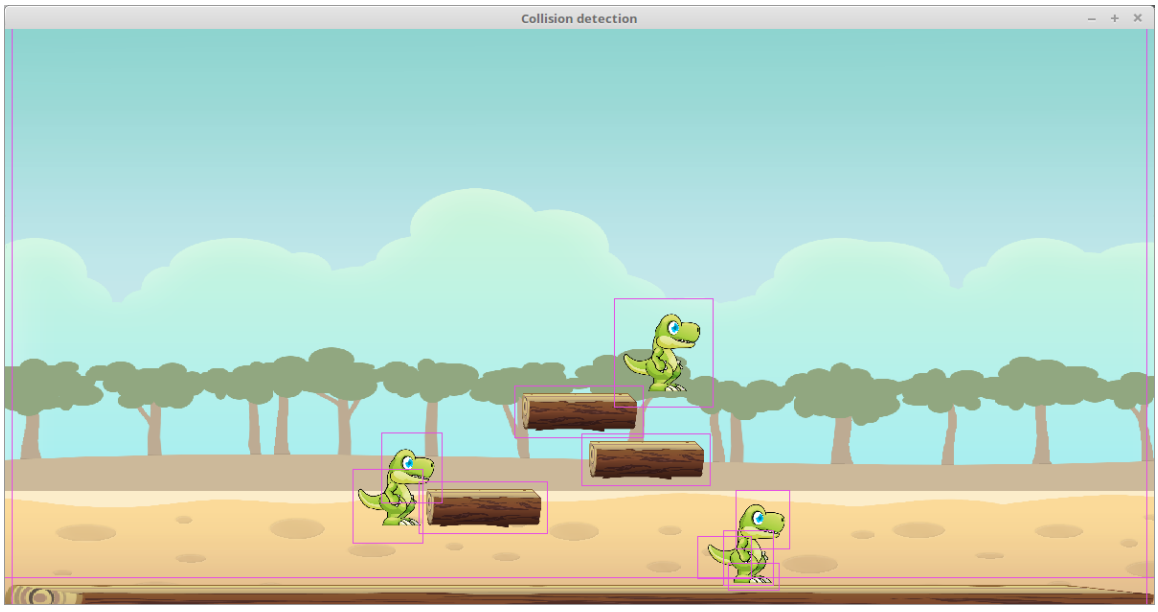
Klasom `Character2` pokušavamo na nešto precizniji način obuhvatiti granice korištenjem dvaju pravokutnika. Box2D dopušta pridruživanje više instanci klase `b2Shape` jednoj instanci `b2Body`. Tako pridružena učvršćenja neće se testirati na međusobne kolizije no svako od njih će se testirati zasebno na kolizije s drugim tijelima. Klasom `Character3` želimo dobit još preciznije prekrivanje, ali i zadržati jednostavnost objekta. Stoga ćemo povećati broj učvršćenja na 4, no zadržat ćemo se na jednostavnim geometrijskim likovima (krug i pravokutnik). Na slici 6.4 prikazano je dobiveno prekrivanje za sve tri klase.

Za testiranje preciznosti kolizije kreiramo jednostavnu platformu pravokutnog oblika. Pri pomicanju igrača tipa `Character3` po platformi primjećujemo da će igrač ostati "u zraku" i neće pasti kada se odmakne previše u lijevo. Ovo je nepoželjno svojstvo u igrama. Krečući se drugim igračem tipa `Character2` po platformi primjetiti ćemo da se greška kolizije smanjila, no igrač ponovno zaostaje ostajući glavom na platformi. Igrač tipa `Character1` aproksimiran je znatno bolje nego prethodna dva te pada s platforme kada bi zapravo i trebao.

Kako bismo bolje vidjeli gdje se događaju kolizije dodajemo iscrtavanje kontaktnih točaka te graničnih pravokutnika. Na ovaj način je vidljivo kako Box2D tretira učvršćenja jednog tijela kao zasebne objkte te svakom od njih pridružuje njegov AABB.



Slika 6.5: Prikaz efekta korištenja više b2Shape objekata



Slika 6.6: Pridruženi granični pravokutnici

Bibliografija

[1] *Web stranica Craftpix.net*, https://craftpix.net/freebies, posjećeno 15.12.2019.

[2] *Web stranica itch.io*, https://itch.io/game-assets/free, posjećeno 15.12.2019.

[3] T. Akenine-Moller, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki i S. Hillaire, *Real-Time Rendering Fourth Edition*, CRC Press, 2018.

[4] Z. Alfitra, *Game art 2D*, https://www.gameart2d.com/freebies.html, posjećeno 30.12.2019.

[5] E. Catto, *Web stranica biblioteke Box2D*, https://box2d.org/documentation/index.html, posjećeno 20.01.2020.

[6] M. Cruz, *Collision Shapes - Box2D SFML*, (2015), https://blog.martincruz.me/2015/05/collision-shapes-box2d-sfml.html.

[7] D. E. Eberly, *3D Game Engine Design*, CRC Press, 2007.

[8] C. Ericson, *Real-Time Collision Detection*, CRC Press, 2005.

[9] L. Gomila, *Web stranica biblioteke SFML*, https://www.sfml-dev.org/index.php, posjećeno 20.01.2020.

[10] P. Jimenez, F. Thomas i C. Torras, *Collision detection algorithms for motion planning*, (2006), https://www.researchgate.net/publication/225117241_Collision_detection_algorithms_for_motion_planningon-shapes-box2d-sfml.html.

[11] N. Souto, *Video Game Physics Tutorial - Part II: Collision Detection for Solid Objects*, (2014), https://www.toptal.com/game/video-game-physics-part-ii-collision-detection-for-solid-objects.

[12] H. A. Sulaiman i A. Bade, *Bounding Volume Hierarchies for Collision Detection*, (2012), https://www.researchgate.net/publication/224829148_Bounding_Volume_Hierarchies_for_Collision_Detection.

[13] G. van den Bergen, *Collision Detection in Interactive 3D Computer Animation*, Eindhoven University of Tachnology, 1999.

[14] _____, *Collision Detection in Interactive 3D Enviroment*, Morgan Kaufman Publisher, 2004.

[15] E. Welzl, *Smallest enclosing disks (balls and ellipsoids)*, (1991.), http://www.stsci.edu/~RAB/Backup%20Oct%2022%202011/f_3_CalculationForWFIRSTML/Bob1.pdf.

Sažetak

U ovom radu proučavamo jedan od najvažnijih problema u računalnim igrama koji pronalazimo i u brojnim drugim domenama kao što su aplikacije virtualne stvarnosti i robotika. Započinjemo s razmatranjem jednostavnih testova. Zbog uočenih ograničenja jednostavnih testova, uvodimo metodu razdvajajućih osi koja rezultira mogućnošću ispitivanja kolizije na svim konveksnim objektima. Opisujemo njezinu primjenu te navodimo njezine prednosti. Zatim izlažemo primjere algoritama za utvrđivanje kolizije bazirane na metodi razdvajajućih osi te uz to proučavamo pridruženi problem utvrđivanja kontaktnih skupova. Prethodno opisane algoritme proširujemo na 3D objekte te na slučajeve kada se objekti kreću konstantnom linearnom brzinom. Nadalje proučavamo problem tuneliranja te njegova potencijalna rješenja. Peto poglavlje posvećujemo promatranju problem testiranja kolizije na nekonveksnim objektima i konveksnim objektima izrazito velike složenosti koji zahtijevaju izgradnju posebnih struktura, tzv. graničnih oblika. Uz to, govorimo o nekoliko mogućih optimizacija preciznosti i brzine algoritama. Naposljetku demonstriramo rad navedenih algoritama izradom programa u C++-u uz korištenje biblioteka Box2D za fiziku igre te Sfml za izradu grafike.

Summary

In this thesis, we study one of the most important issues occurring in computer games, which can also be found in many other domains, such as virtual reality applications and robotics. We begin by reviewing simple tests. Due to the observed limitations of simple tests, the method of separating axes is introduced, which results in the possibility of collision testing on all convex shapes. We explain the application of the method, and list its strengths. Furthermore, we present the appropriate collision detection algorithms based on the separating axis theorem, and we examine the associated problem of contact sets determination. We extend the algorithms to 3D objects and to cases where objects are moving at constant linear velocity. Moreover, we analyse the problem of tunneling and its potential solutions. The fifth chapter deals with the problem of collision testing on non-convex objects and convex objects of extremely high complexity, which require the construction of a special structure, the so-called bounding boxes. In addition, we discuss several possible optimizations in order to improve the precision and speed of the algorithms. Finally, we demonstrate the effectiveness of the algorithms by creating a program in C++, using the Box2D library for the game physics and Sfml for the game graphics.

Životopis

Rođena sam 12. srpnja 1995. u Zagrebu. Osnovnu školu završila sam 2010. godine u Zagrebu te iste godine upisujem X. gimnaziju "Ivan Supek" u Zagrebu. Po završetku srednje škole 2014. godine upisujem preddiplomski studij Matematika, smjer: nastavnički na Prirodoslovno-matematičkom fakultetu u Zagrebu koji sam završila 2017. godine te svoje obrazovanje nastavljam upisujući diplomski studij Računarstvo i matematika na istom fakultetu. 2019. godine sudjelovala sam na studentskom natjecanju Mozgalo. Uz to, iste godine započinjem raditi u na pozciji Junior Machine Learning inženjera.