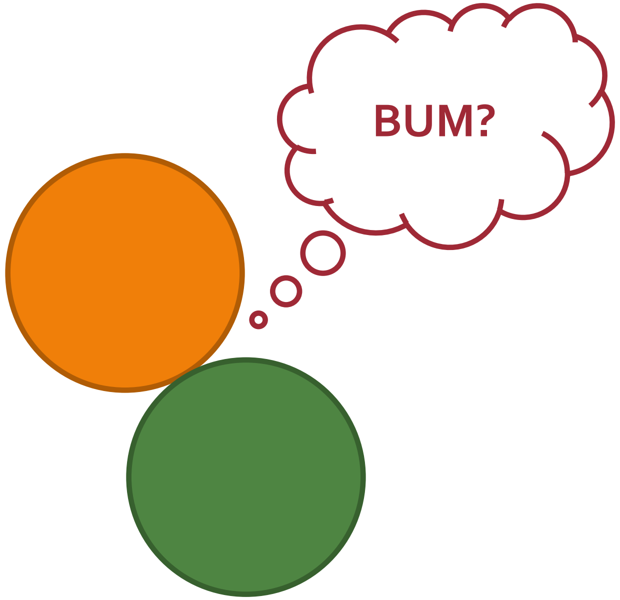
DETEKCIJA TRKOV

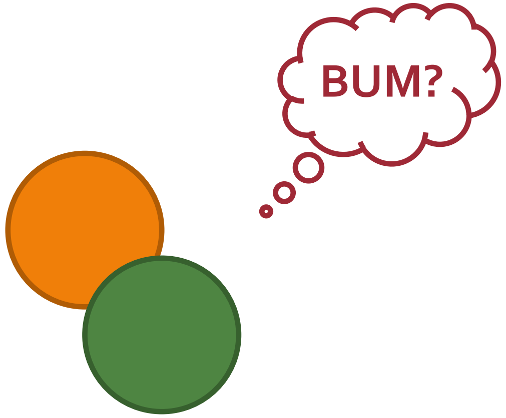
Trki: detekcija in odziv

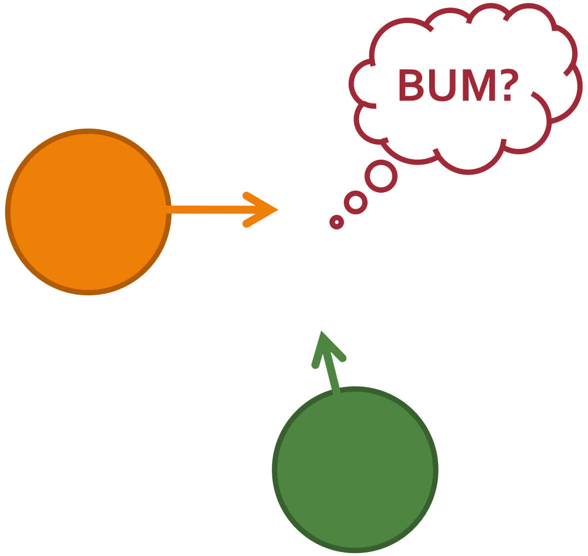
- Kdaj predmeti med seboj trčijo in kako se obnašajo po trku
- Težek problem
  - predmeti so lahko zelo hitri
  - predmeti imajo lahko kompleksno geometrijo
  - preveriti moramo trk vsakega predmeta z vsakim in to v vsakem koraku (ki je lahko krajši od hitrosti izrisa animacije oz. igre)
    - $O(n^2)$ preverjanj, ki so že sama lahko kompleksna



Kdaj izvedemo detekcijo?

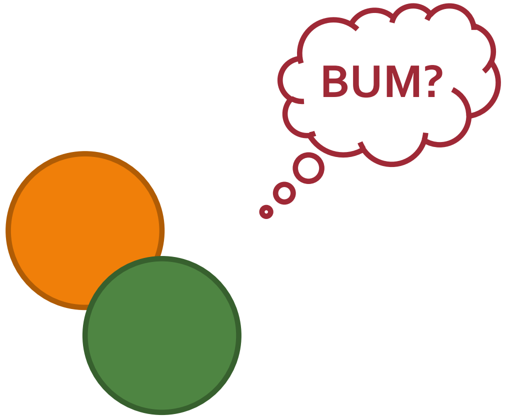
- Glede na trenutek detekcije ločimo dva pristopa:
  - **preverjanje prekrivanj** (*overlap testing*)
    - preveri ali je že prišlo do trka
    - preveri v eni točki
    - veliko se uporablja v igrah
    - računamo na koncu koraka simulacije (po premiku)
  - **preverjanje križanj** (*intersection testing, continuous CD*)
    - preveri ali se bosta poti predmetov križali v naslednjem koraku in bo prišlo do trka
    - preveri na celotni poti, ki jo predmet prepotuje
    - računamo na začetku koraka simulacije (pred premikom)





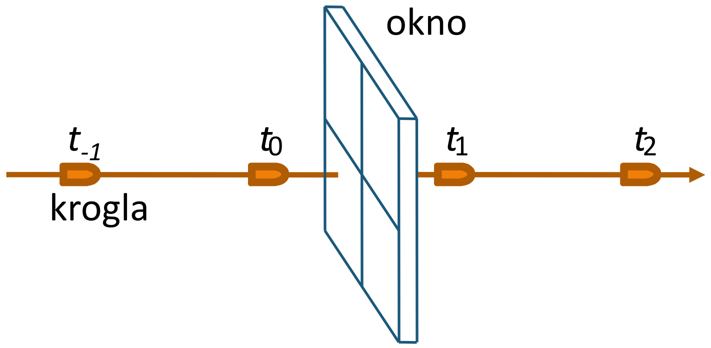
Preverjanje prekrivanj

- Najbolj standardna metoda v igrah
- **Diskretno** preverjanje
- Prekrivanje preverimo **na koncu** koraka simulacije
  - najprej premaknemo predmete
  - potem preverimo prekrivanja
- Unity:
  - *CollisionDetectionMode.Discrete*



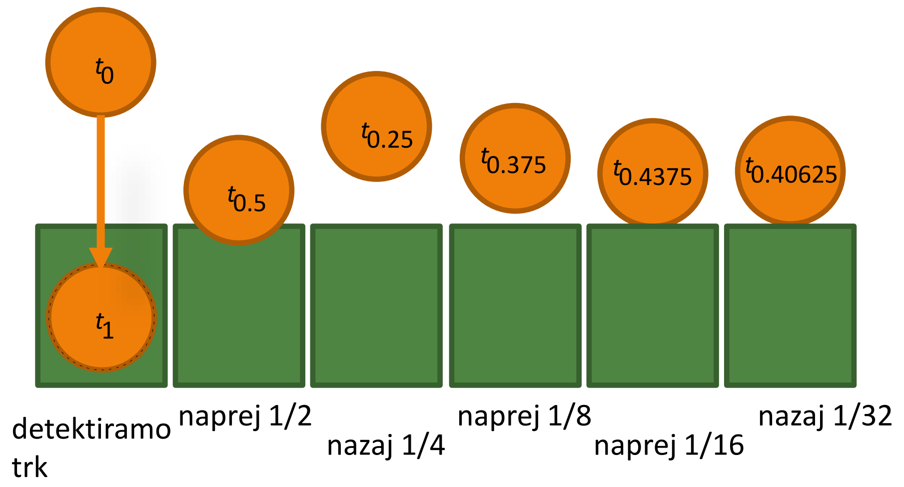
Preverjanje prekrivanj

- Problem, če se predmeti premikajo **prehitro**
  - zgrešimo trk
- Hitrost najhitrejšega predmeta krat časovni korak morata biti manjša kot najtanjši predmet
  - omejitev največje hitrosti
  - majhen časovni korak (veliko računanja, počasi)
  - scena izdelana tako, da so predmeti ustrezno veliki
    - oz. da so obsegajoči volumni s katerimi računamo trk dovolj veliki



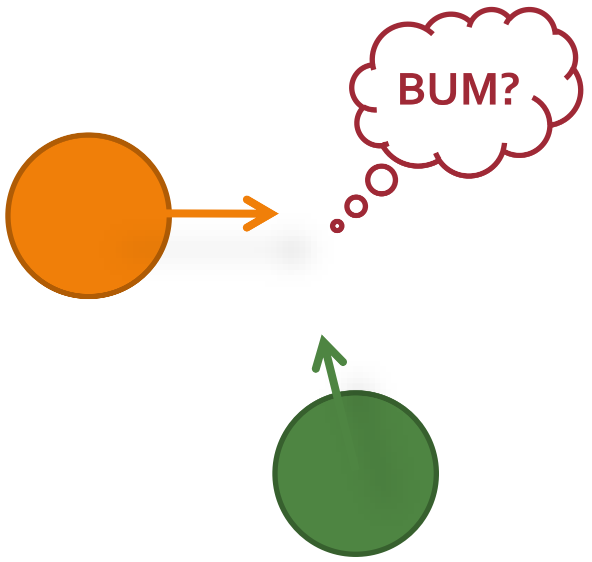
Preverjanje prekrivanj

- Če je do trka prišlo, lahko izvemo bolj natančen **čas trka**
  - predmet premikamo nazaj v čas do tik pred trkom
  - uporabimo bisekcijo
  - ko najdemo čas trka, **premaknemo čas** simulacije **nazaj** v to točko, izračunamo rezultat trka in izvedemo korak simulacije do konca
  - postopek je časovno potraten, izvajamo ga, če res potrebujemo natančnost



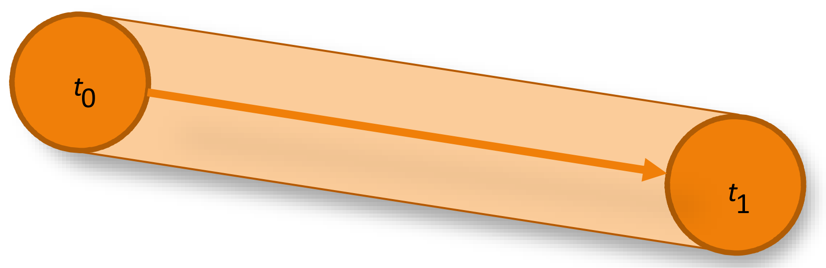
Preverjanje križanj

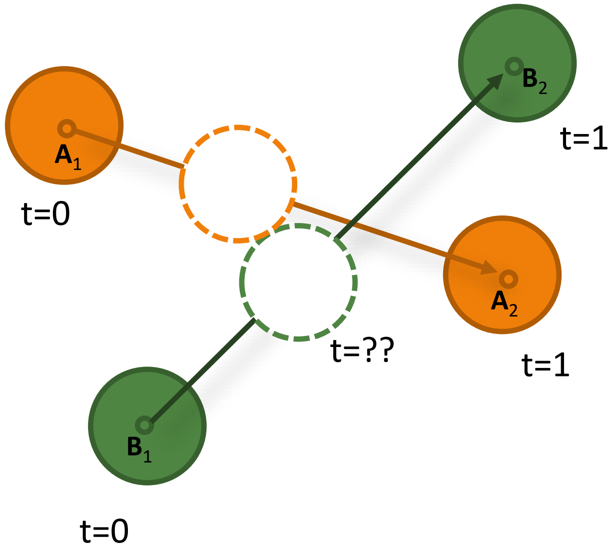
- **Zvezno** preverjanje
- "Napovedovanje" trkov preden se zgodijo
  - napovedovanje **na začetku** koraka simulacije
- Če bo trk, **premaknemo** simulacijo na **čas** trka, izračunamo rezultat in izvedemo simulacijski korak do konca
- Bolj **kompleksno** in natančno kot preverjanje prekrivanj
- Unity:
  - *CollisionDetectionMode. Continuous/ContinousDynamic*



Preverjanje križanj

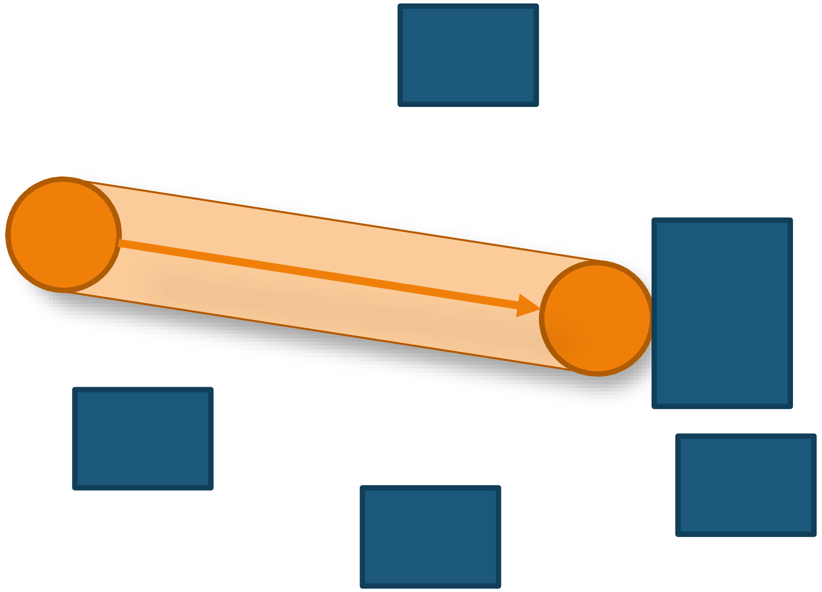
- **Ekstrapoliramo** predmet po poti
  - npr. krogla se “raztegne” po smeri svoje poti
- Izračunamo **križanje** ekstrapoliranih predmetov
  - npr. za krogle
    - $A(t) = A_1 + t(A_2 - A_1)$
    - $B(t) = B_1 + t(B_2 - B_1)$
    - $\left(B(t) - A(t)\right)^2 = (r_A + r_B)^2$
    - rešimo kvadratno enačbo za t





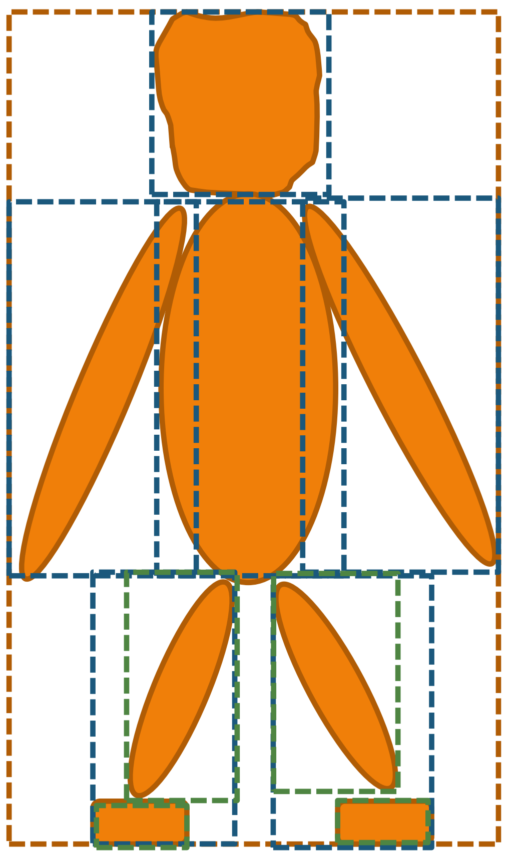
Preverjanje križanj

- Predpostavke:
  - poznamo **natančno stanje** sistema v času *t*
    - pri omrežnih igrah zaradi latence to ni nujno res
  - hitrost je **konstantna** - ni pospeševanja v koraku simulacije
    - vpliva na fizikalni model, ki ga izberemo
- Lahko uporabljamo tudi za preverjenje v **kateri predmet** se bomo zaleteli najprej, če iščemo v neki smeri
  - posplošeno metanje žarka (*RayCast*)
  - npr. *SphereCast* in *CapsuleCast* v Unity



Kako hitro izvedemo detekcijo

- Kako lahko **dovolj hitro** preverimo trke vseh predmetov v vsakem koraku?
- Pohitritve
  - **poenostavimo** kompleksne modele
    - hitreje detektiramo potencialne trke
    - uporabimo lahko posebne algoritme za hitro detekcijo O(n)
  - **zmanjšamo** število primerjav
    - sceno/modele razdelimo s tehnikami delitve prostorov



Večnivojska detekcija

- Za pohitritev detekcije trkov, v interaktivni grafiki detekcijo trkov delimo na dva podproblema
  - **groba** detekcija (*broad phase*)
    - hiter test za ugotavljanje potencialnih trkov
    - testiramo s poenostavljenimi predmeti
  - **fina** detekcija (*narrow phase*)
    - bolj natančna detekcija trkov med omejenim naborom predmetov
    - lahko do nivoja poligonov
- Tako deluje detekcija v večini fizikalnih pogonov
  - primer: PhysX (NVidia, open source, uporablja se tudi v Unity)

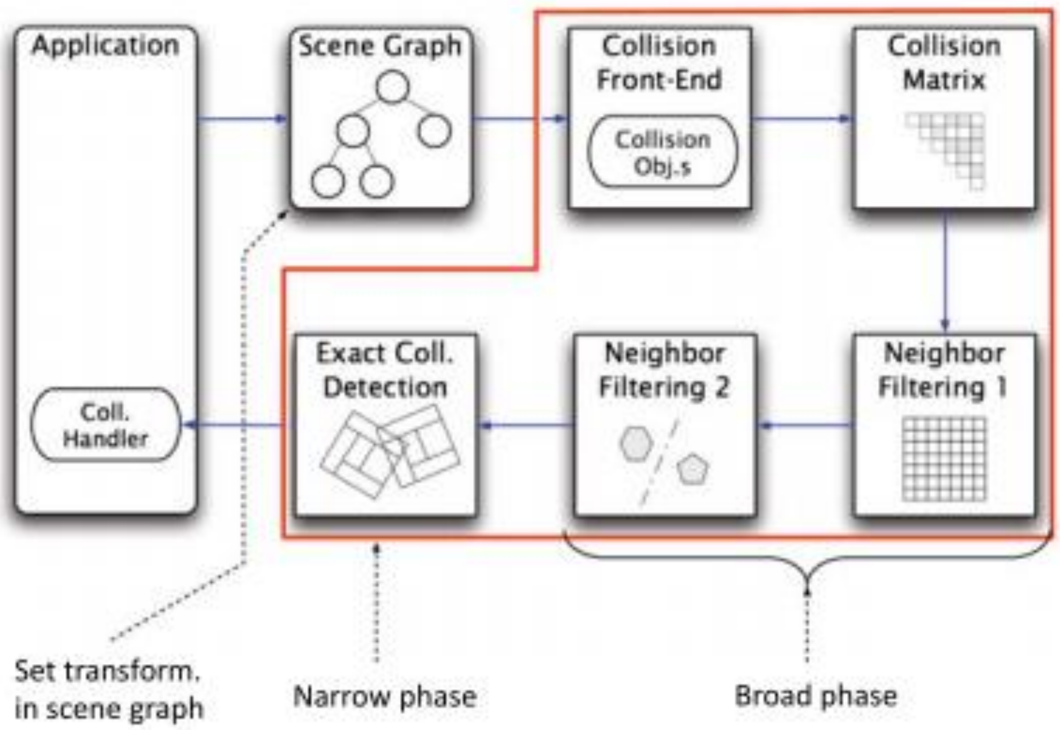
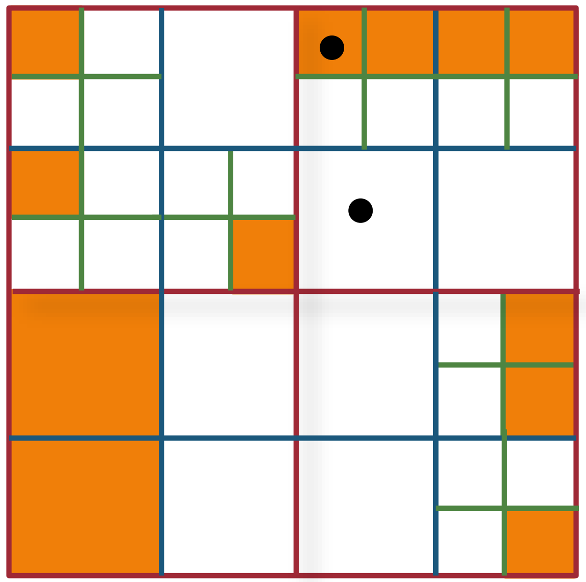


Fig. 2.1 The typical design of a collision detection pipeline

Vir: E. Christer. Real-Time Collision Detection.

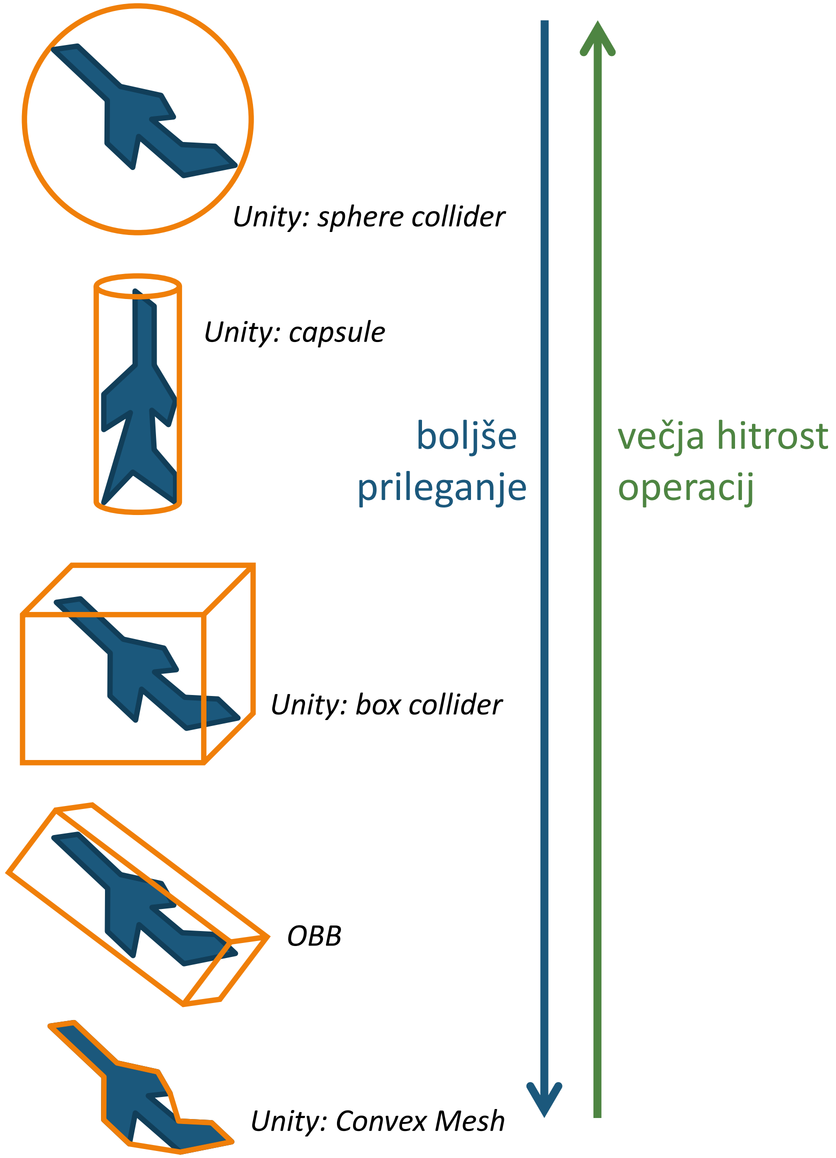
Groba detekcija

- Pohitrimo detekcijo prekrivanj z **očrtanimi telesi**
  - poenostavitev geometrije
  - enostaven izračun presekov
- Uporabimo **delitev prostora**
  - največkrat za statično geometrijo
  - navadna mreža, osmiška drevesa, KD drevesa …



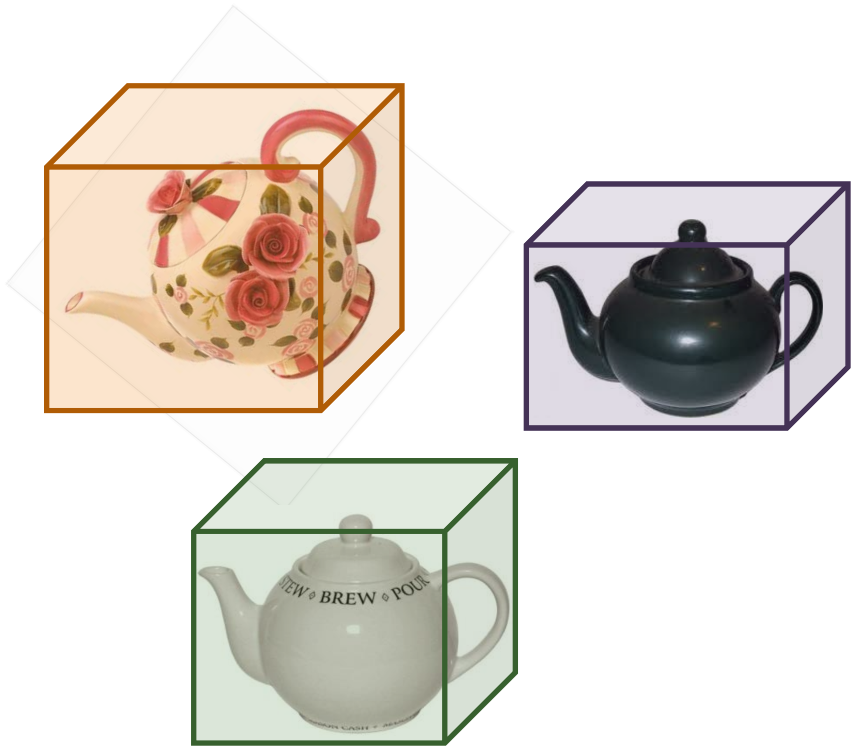
Nekatera očrtana telesa

- **Krogla** (*bounding ball*)
  - v povprečju se slabo prilega
  - hitre operacije, ne potrebuje sprememb ob rotaciji
- **Valj** (*bounding cylinder*)
  - navpičen valj, primeren za predmete, ki se vrtijo le okoli navpične osi – npr. osebki v igrah
  - hitre operacije, ne potrebuje sprememb ob rotaciji
- **Osno poravnani očrtani okvir** (axis aligned *bounding box - AABB*)
  - boljše prileganje od krogle
  - še vedno hitre operacije
- **Usmerjeni očrtani okvir** (orien*ted bounding box - OBB*)
  - dobro prileganje, počasnejši preseki
- **Konveksni politop**
  - najboljše prileganje, počasni preseki



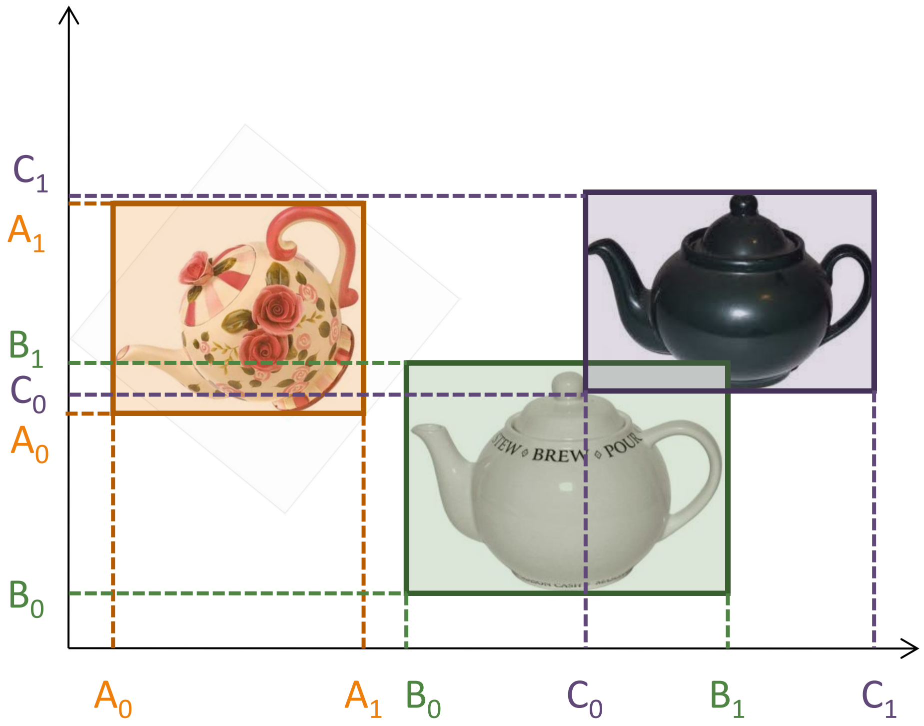
Groba detekcija

- Tudi, če uporabimo očrtana telesa za predstavitev predmetov
  - **kompleksnost** preverjanj vsak z vsakim še vedno $\boldsymbol{O}(\boldsymbol{n}^2)$
- Za nekatera telesa lahko uporabimo **hiter algoritem**, da najdemo pare, ki se prekrivajo
- PhysX (s tem tudi Unity) uporablja:
  - *sweep and prune* oz. *sort and sweep*
    - v povprečju $O(n)$
  - *multi box pruning*
    - nekoliko boljši, ko se večina predmetov premika ali na novo nastaja/izginja



Prekrivanje AABB: algoritem *sweep and prune*

- Okvir predstavimo s tremi intervali
  - [x0, x1], [y0, y1], [z0, z1]
- Dva okvira se prekrivata, če se **prekrivata na vseh treh intervalih**
- **Algoritem**:
  - 1. imamo n okvirov in tri $2n$ dolge urejene sezname začetkov in koncev intervalov – za vsako os enega
  - 2. v vsakem seznamu s sprehodom poiščemo vse hkrati aktivne intervale in jih shranimo v $n \times n$ matriki prekrivanj
  - 3. vsi okviri, ki imajo **vse tri intervale hkrati aktivne**, se prekrivajo
- Urejanje seznama je drago, vendar se med koraki simulacije se položaji le malo spremenijo
  - sortiranja, ki hitro delujejo na skoraj urejenih seznamih (*insertion, bubble*)
  - pričakovana kompleksnost je $O(n)$



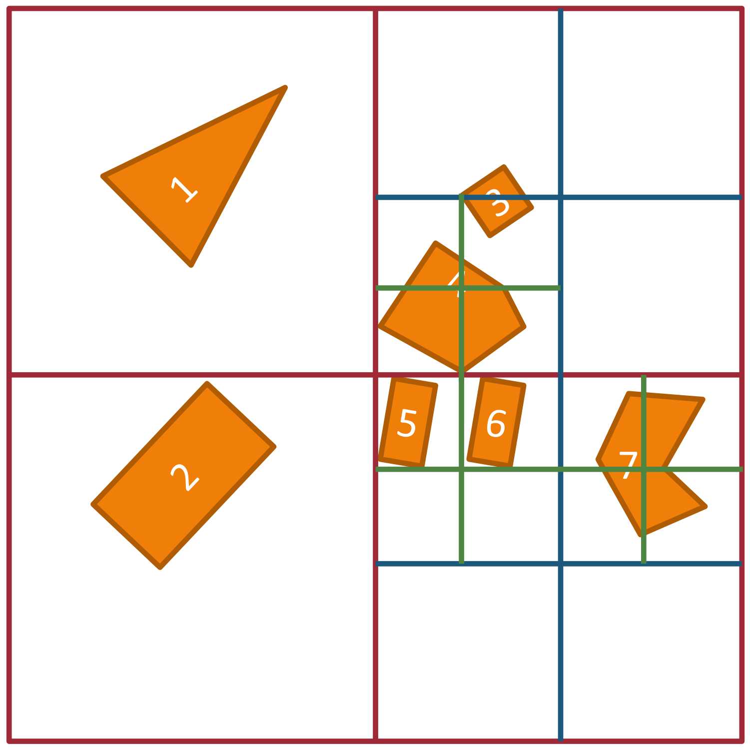
Groba detekcija z delitvijo prostora

- Pohitritev
  - izračunamo **razdelitev prostora** z eno od tehnik (npr. osmiško, KD ali BSP drevo)
- Za **statične** ovire lahko enostavno preverimo
  - ali smo prišli v notranjost (trk)
- Prav tako lahko tehniko uporabimo za **dinamične** predmete
  - omejevanje teles s katerimi računamo trke (sosedi)
  - podatkovno strukturo moramo posodabljati ob premikih

Monopoly Tycoon

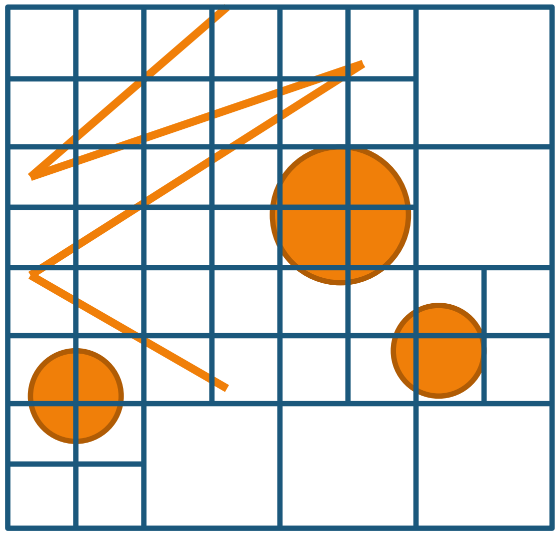
Tehnike delitve prostora

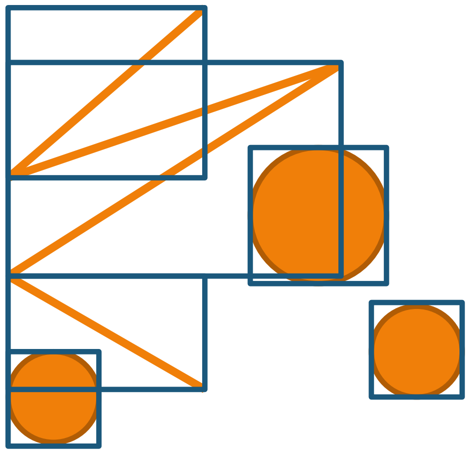
- Tehnike delitve prostora razdelijo 3D prostor (lahko tudi posamezen predmet) na več delov
- So splošno uporabne za bolj zgoščeno predstavitev in hitrejše povpraševanje:
  - detekcija trkov
  - izločanje
  - iskanje najbližjih sosedov
  - sledenje žarka (izračun predmetov, ki jih žarek zadene)
  - predstavitev polnih teles
  - …



Tehnike delitve prostora

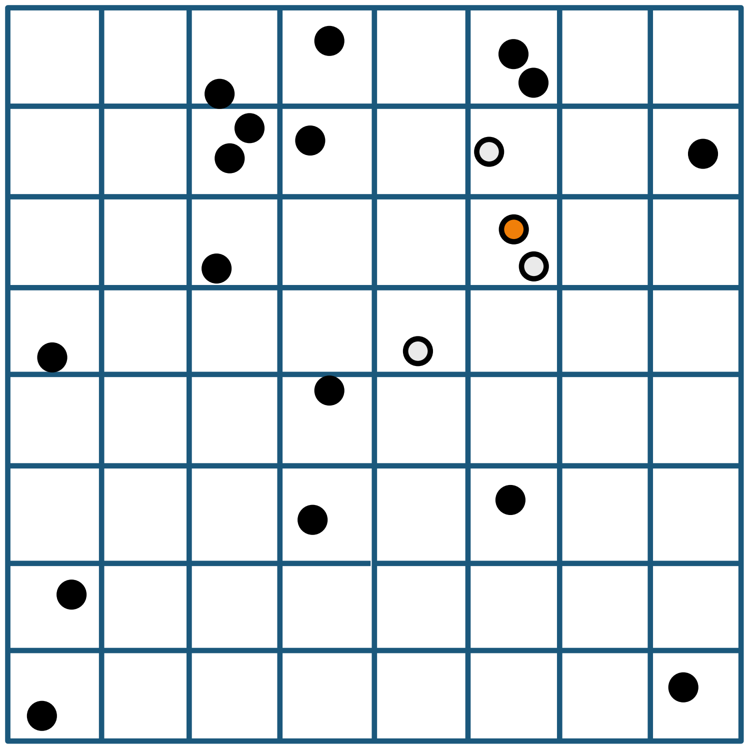
- **Prostorsko** usmerjene
  - delijo prostor
    - predstavljen je tudi prazen prostor
  - točka v prostoru je v eni veji drevesa
  - predmet je v prostoru lahko v več razdelkih oz. vejah drevesa
  - osmiška drevesa, BSP drevesa, KD drevesa
- **Predmetno** usmerjene
  - delijo prostor glede na očrtan obseg predmetov
    - ni praznih prostorov
  - prostor je lahko redundantno predstavljen – točka spada v več vej drevesa
  - predmet v prostoru spada v eno vejo drevesa
  - AABB, OBB drevesa





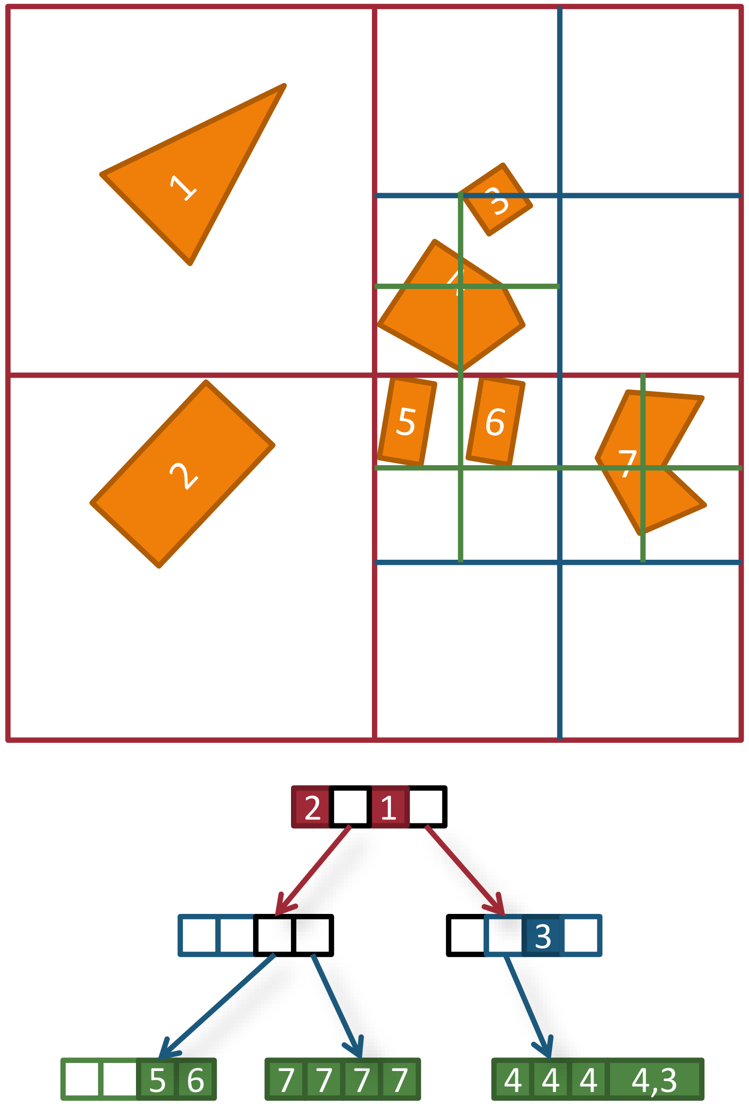
Delitev prostora: navadna mreža

- **Navadna mreža** (*grid*): prostor razdelimo z "mrežo" kock oz. kvadrov
- Pri detekciji trkov lahko **omejimo sosede**, s katerimi računamo trke
  - vemo, da moramo gledati celice, ki jih pokriva trenutni predmet
- Kritična je ločljivost mreže
  - premalo pridobimo, če je mreža pregroba



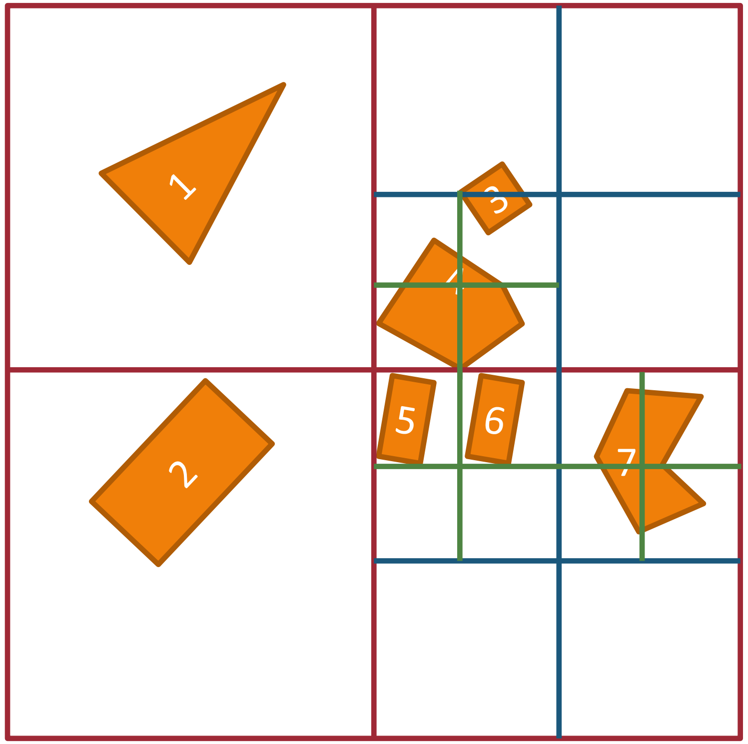
Delitev prostora: štiriška in osmiška drevesa

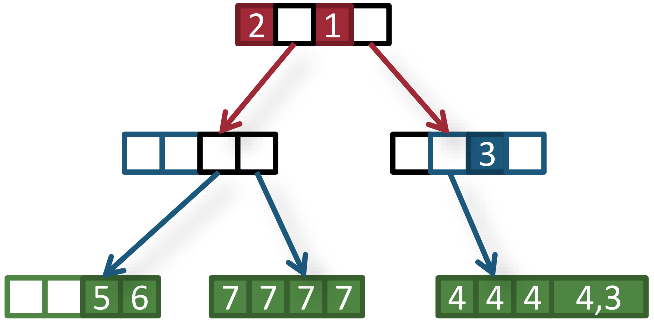
- **Štiriško drevo** - *quadtree* (2D) in **osmiško drevo** - *octree* (3D) sta hierarhični drevesni predstavitvi prostora
- Obe rekurzivno delita ravnino (*quadtree*) oz. prostor (*octree*), do neke globine
  - *quadtree* deli ravnino na štiri dele
  - *octree* prostor na 8 delov
  - kriterij za deljenje je odvisen od aplikacije; npr. max. število poligonov v celici
  - z deljenjem se tvori drevo



Štiriško drevo

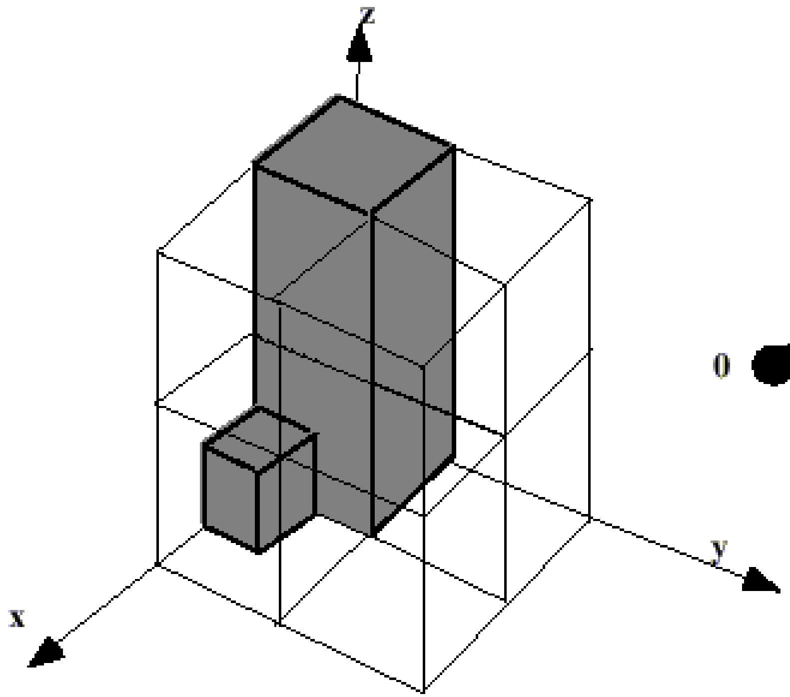
- Primer (štiriškega) drevesa za sceno s poligoni
  - pogoj za ustavitev gradnje je, da v nobenem delu ne ostanejo več kot štirje poligoni.
  - polna polja označujejo vozlišča, ki vsebujejo geometrijo in številko predmeta, katerega (del) vsebujejo

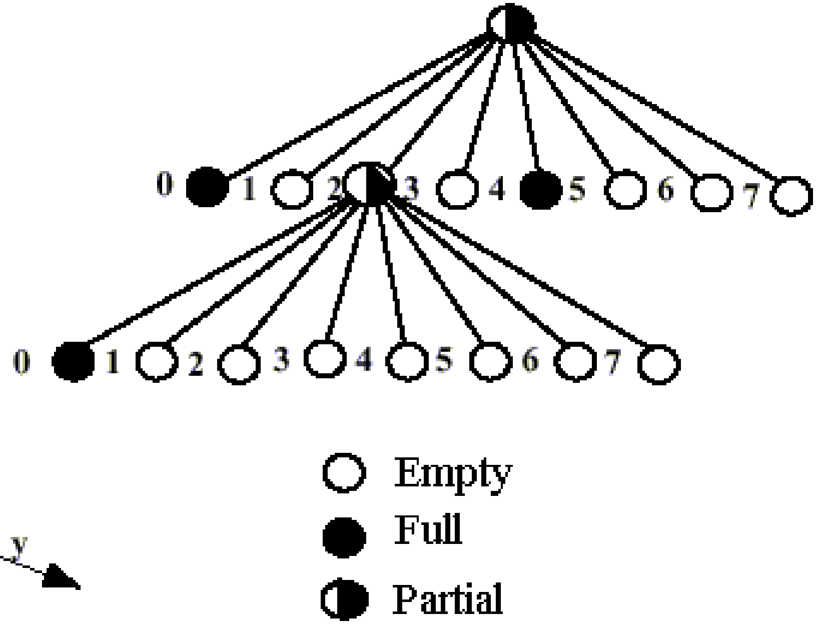




Osmiško drevo

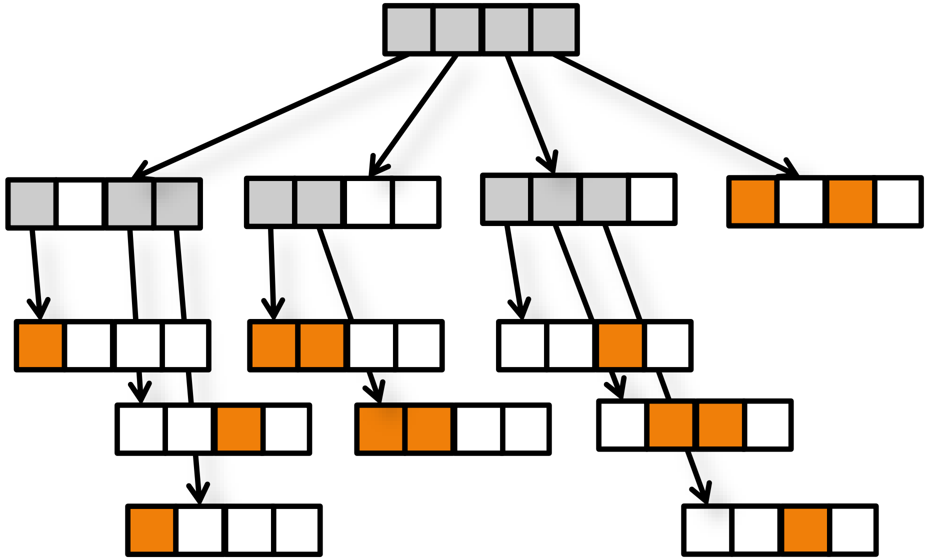
- Primer osmiškega drevesa, ki deli prostor na prazen/poln
  - prostor s tremi ravninami razdeli na 8 podprostorov

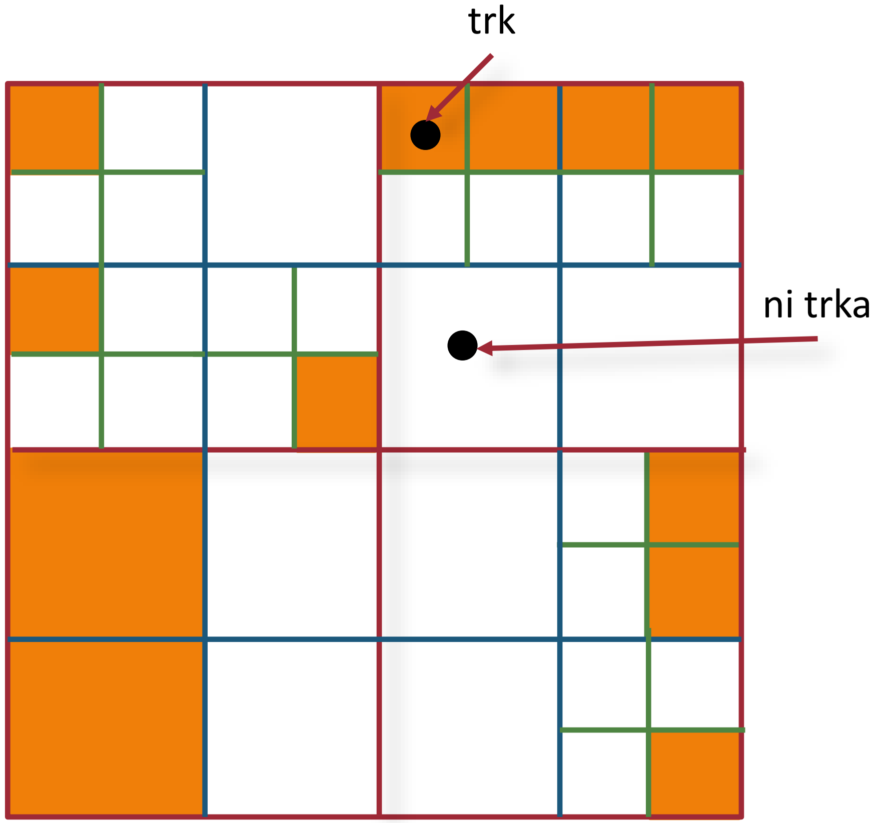




Štiriško/osmiško drevo za grobo detekcijo trkov

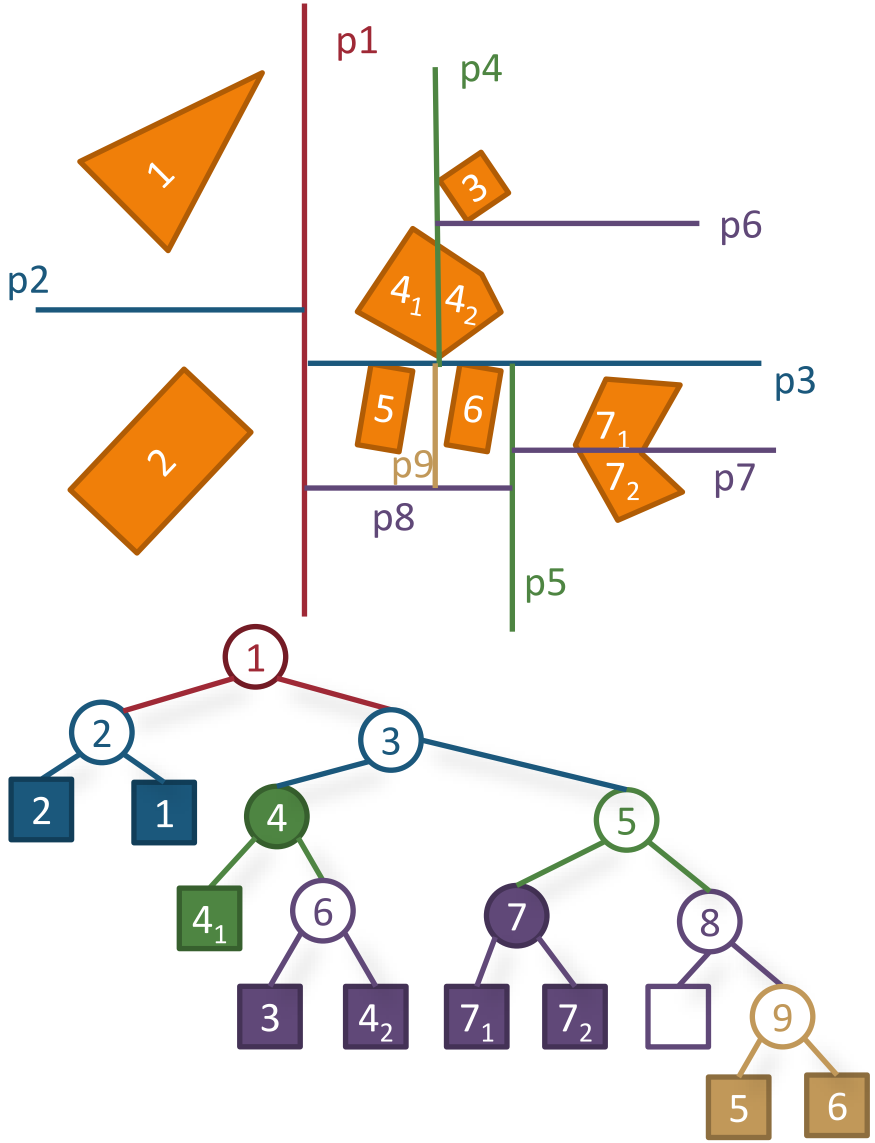
- **Statične** ovire (npr. stavbe) lahko predstavimo kot **polno/prazno** štiriško (osmiško) drevo
  - oranžna pomeni polno (stavba), belo prazno
  - enostavno preverimo ali smo v trku
- Lahko računamo tudi trke s trikotniki v drevesu
  - *Demo three.js*





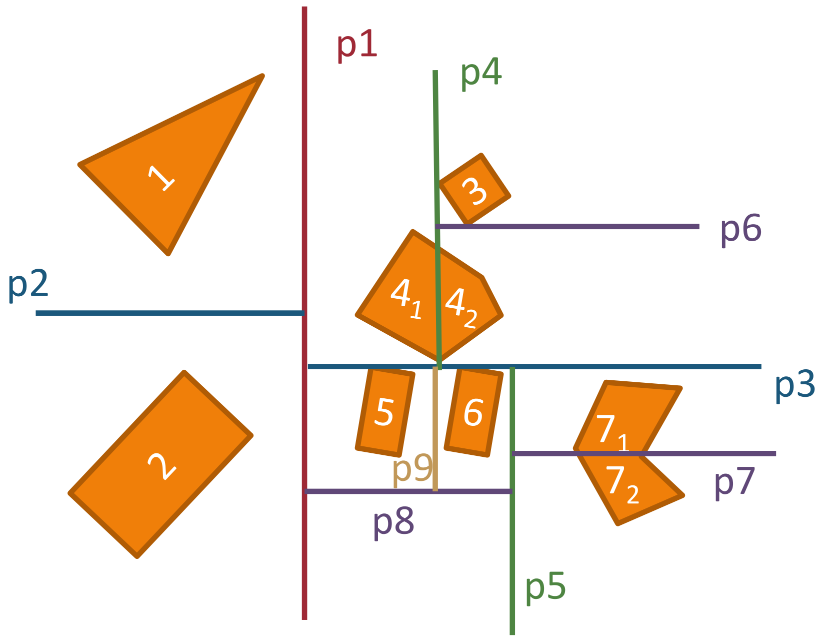
Delitev prostora: KD drevo

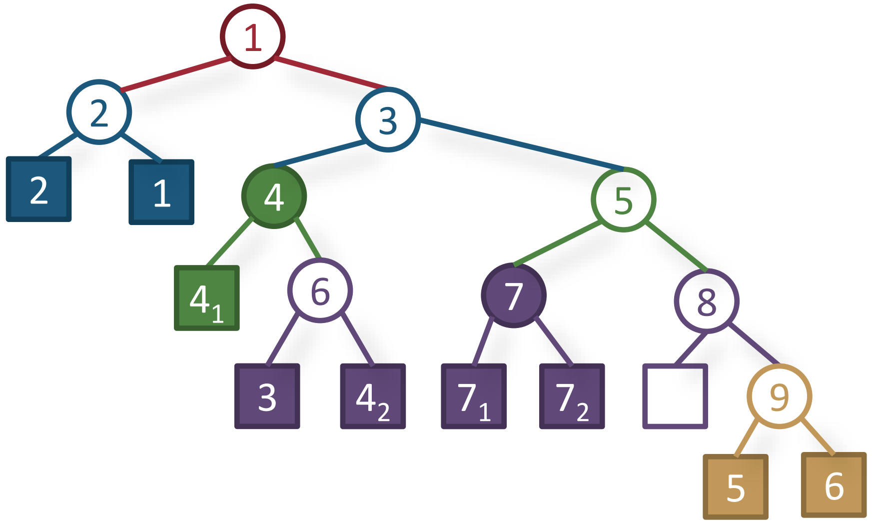
- **KD drevesa** so posplošitev osmiških dreves
  - binarna drevesa
  - vsakič (pod)prostor razdelimo na dva dela glede na ravnino, ki je pravokotna na eno od osi
- Delimo ciklično po vrsti po oseh
  - najprej prvi (npr. x), drugi (npr. y) in tretji (npr. z)
- Razdelitev prostora je torej nekoliko bolj fleksibilna kot pri osmiških, po drugi strani pa je gradnja nekoliko zahtevnejša
  - analiza možnosti izbire ravnine



KD drevo

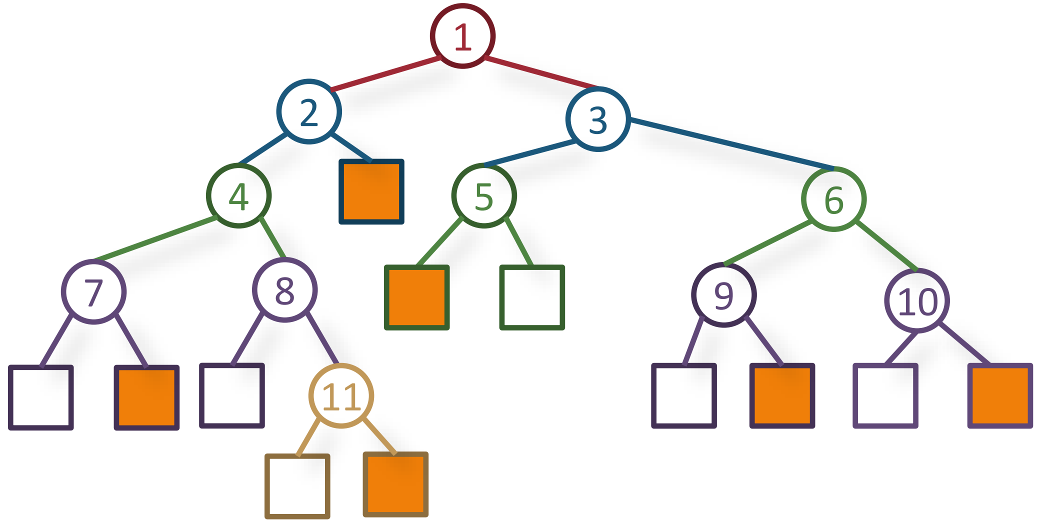
- Primer KD drevesa za sceno s poligoni
  - pogoj za ustavitev gradnje je, da v nobenem delu ne ostanejo več kot štirje poligoni.
  - polna polja označujejo vozlišča, ki vsebujejo geometrijo in številko predmeta, katerega (del) vsebujejo, prazna označujejo ravnine

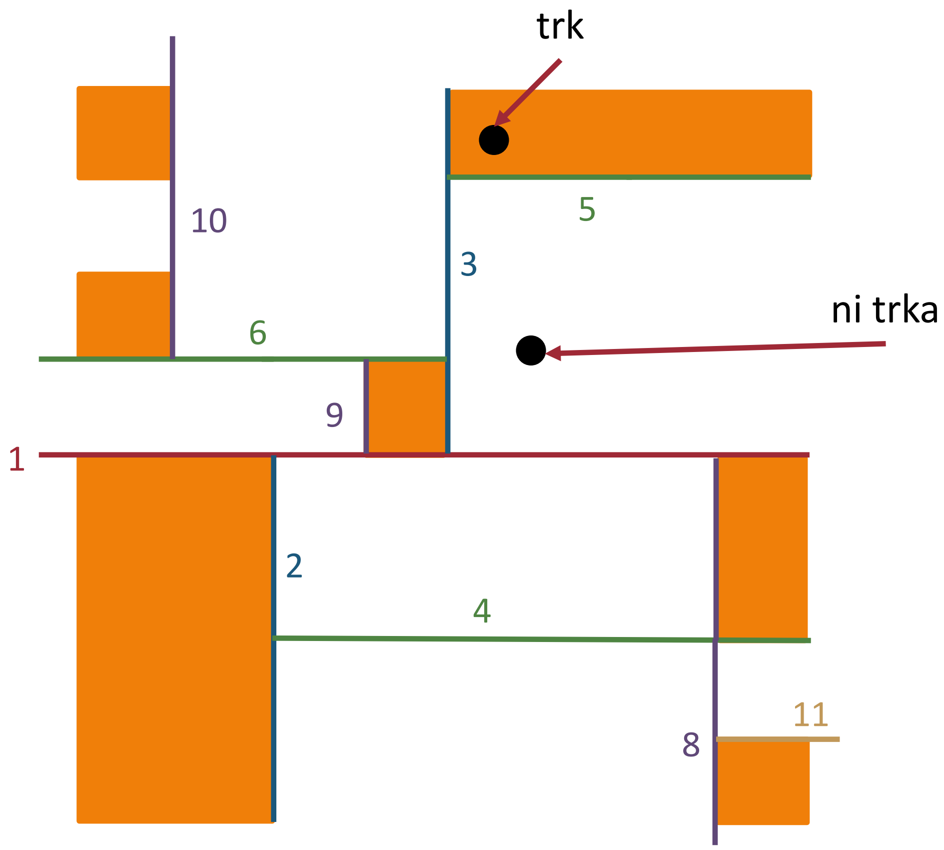




KD drevo za grobo detekcijo trkov

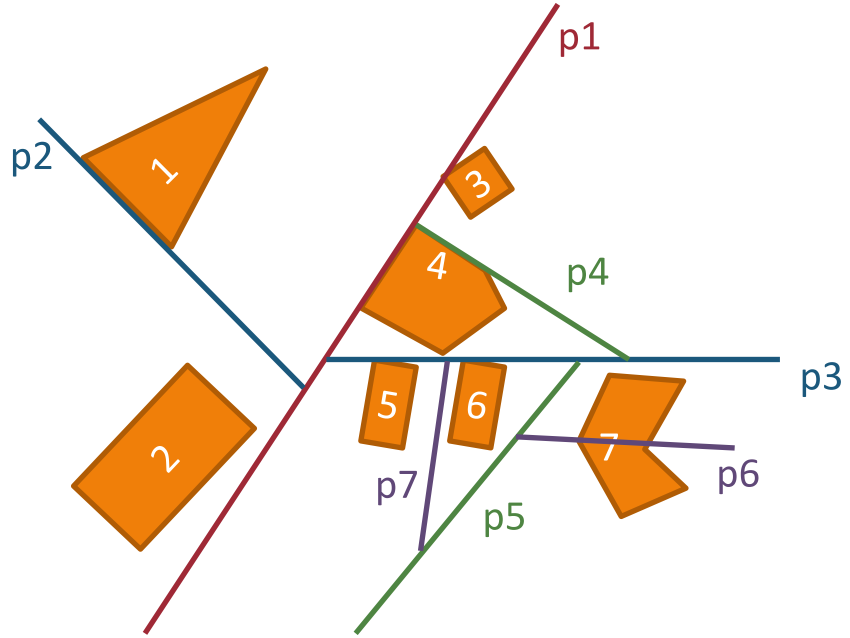
- **Statične** ovire (npr. stavbe) predstavimo kot KD drevo
  - oranžna pomeni polno (stavba), belo prazno
  - enostavno preverimo ali smo v trku





Delitev prostora: Binary Space Partitions (BSP)

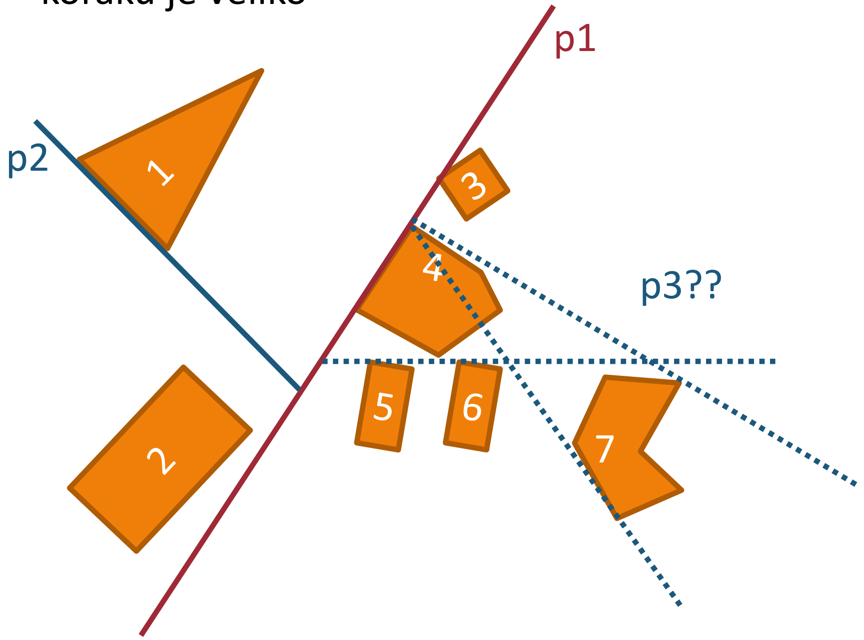
- **BSP drevesa** so bolj splošna od KD dreves
- Prostor delimo s poljubno ravnino vsakič na dva dela
  - ker so ravnine lahko poljubne, BSP drevesa omogočajo dobro delitev prostora za poligonske modele
- Pogoj za ustavitev deljenja je odvisen od namena, npr.:
  - za detekcijo trkov, dokler vsebina v listih ni dovolj enostavna



BSP drevo

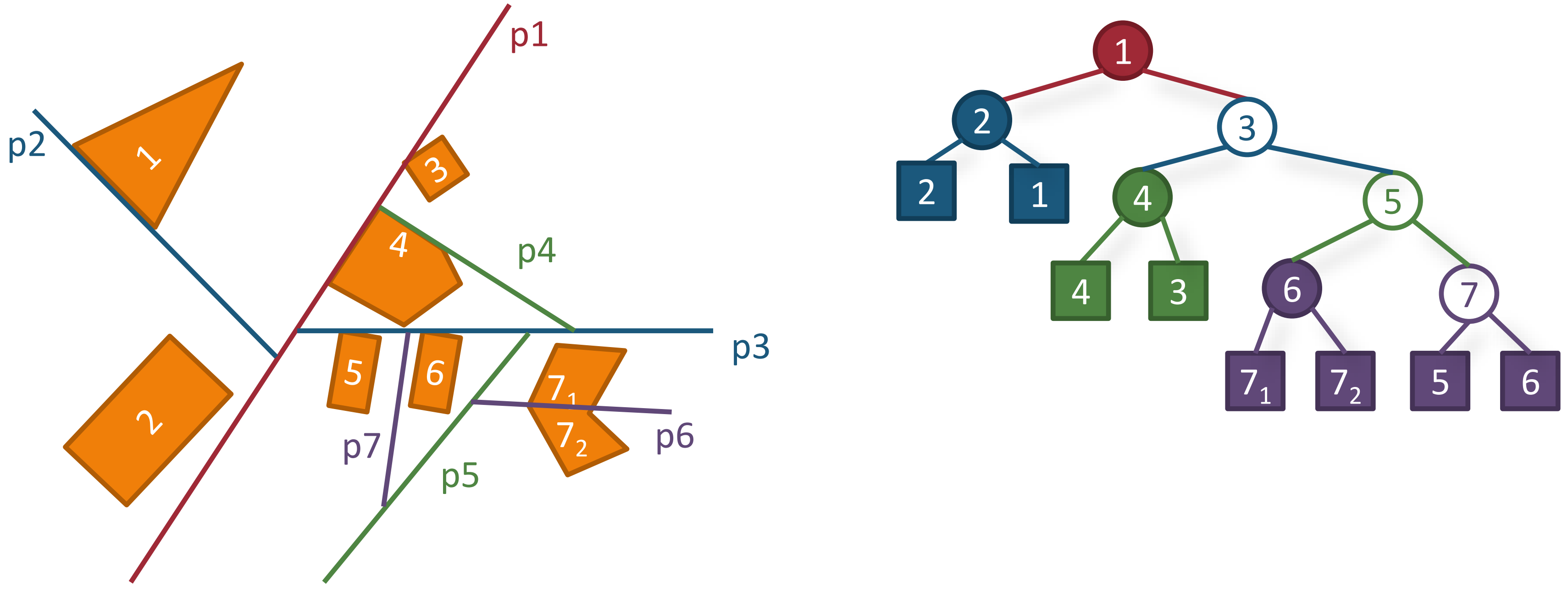
- Gradnja drevesa je **zahtevna**
  - izbira ravnine v vsakem koraku je odvisna od uporabe, možnosti je veliko
    - npr. na vsaki strani naj bo približno enako število poligonov
  - cilj je čimbolj uravnoteženo drevo (logaritmična globina), algoritmi navadno preizkušajo veliko variant, da izberejo ravnino
  - po izbiri ravnine je potrebno ugotoviti kateri predmeti so, levo in desno od ravnine in katere je potrebno razdeliti na pol

možnosti za izbiro ravnine deljenja v vsakem koraku je veliko



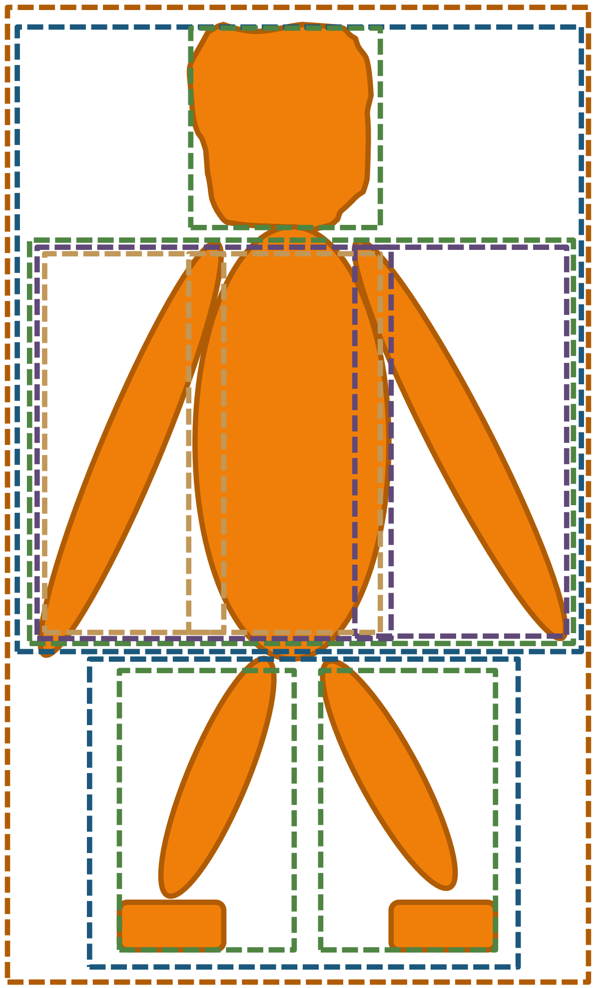
BSP drevo

- Primer drevesa za sceno s poligoni; pogoj za ustavitev gradnje je v tem primeru, da v nobenem delu ne ostanejo več kot štirje poligoni. Polno pobarvana vozišča in listi vsebujejo seznam (delov) predmetov, prazni nimajo predmetov



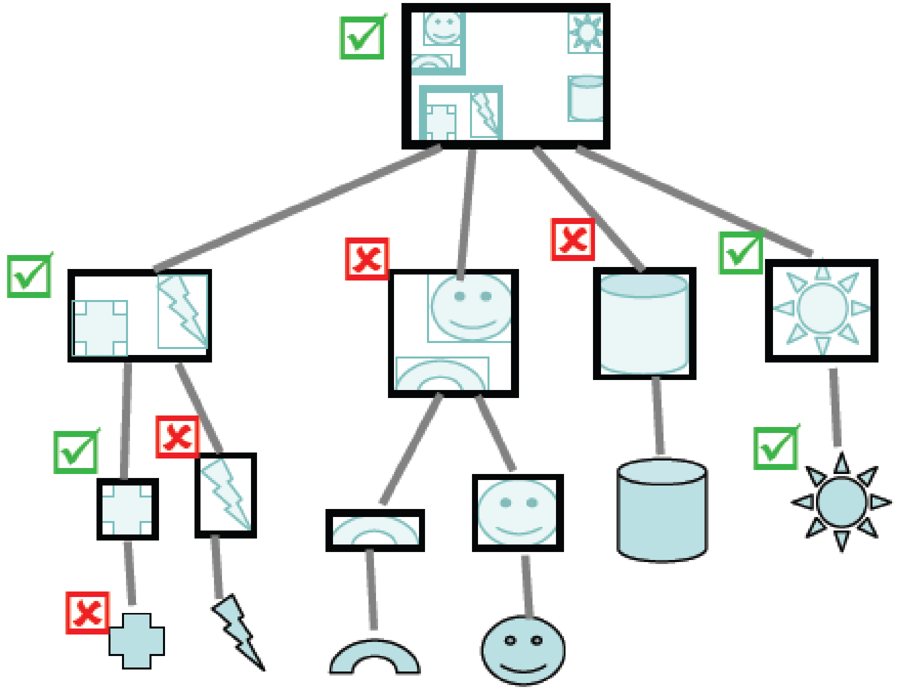
Fina detekcija trkov

- Ko z grobo detekcijo najdemo pare teles, ki se verjetno prekrivajo, je potrebno bolj **natančno** preverjanje
  - Telesa so lahko **kompleksna** (veliko poligonov), presek vsakega poligona z vsakim je $O(n^2)$
- Uporabimo lahko **hierarhije očrtanih prostorov** (*bounding volume hierarchy*)
  - AABB, OBB drevesa …
- Ko pridemo do konca, lahko testiramo še **prekrivanje poligonov**



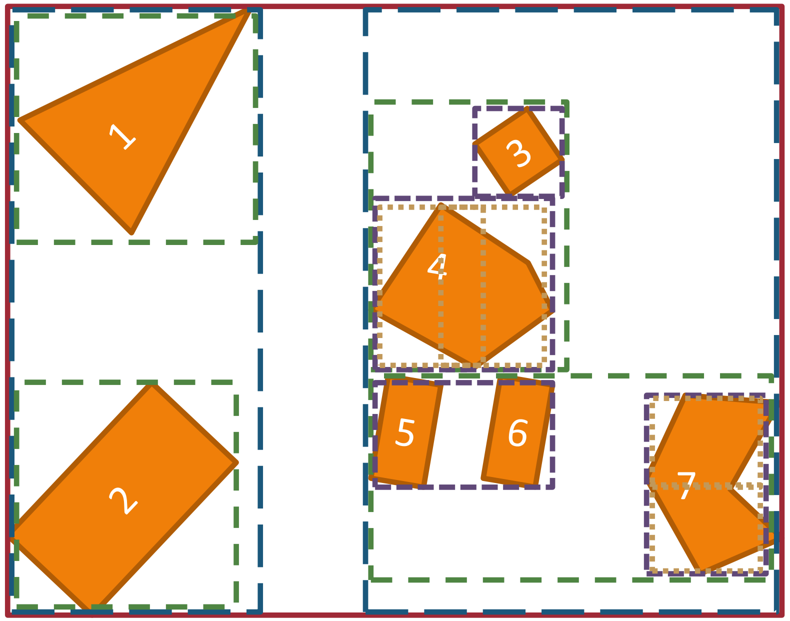
Delitev prostora: Hierarhije očrtanih prostorov (BVH)

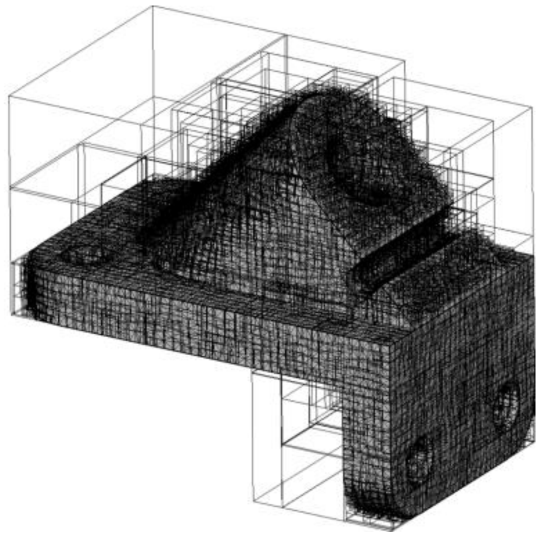
- **Očrtane prostore** (npr. z osno poravnane okvirje) postavimo v **drevo**
  - na vrhu je očrtana celotna scena
  - vsako vozlišče drevesa je očrtano prostor celotnega poddrevesa
  - listi vsebujejo geometrijo
  - Uporabno za računanje presekov - detekcijo trkov, izločanje, sledenje žarkov
- Ko iščemo preseke, začnemo pri korenu drevesa (celotna scena) in nadaljujemo proti predmetom v listih



AABB drevo

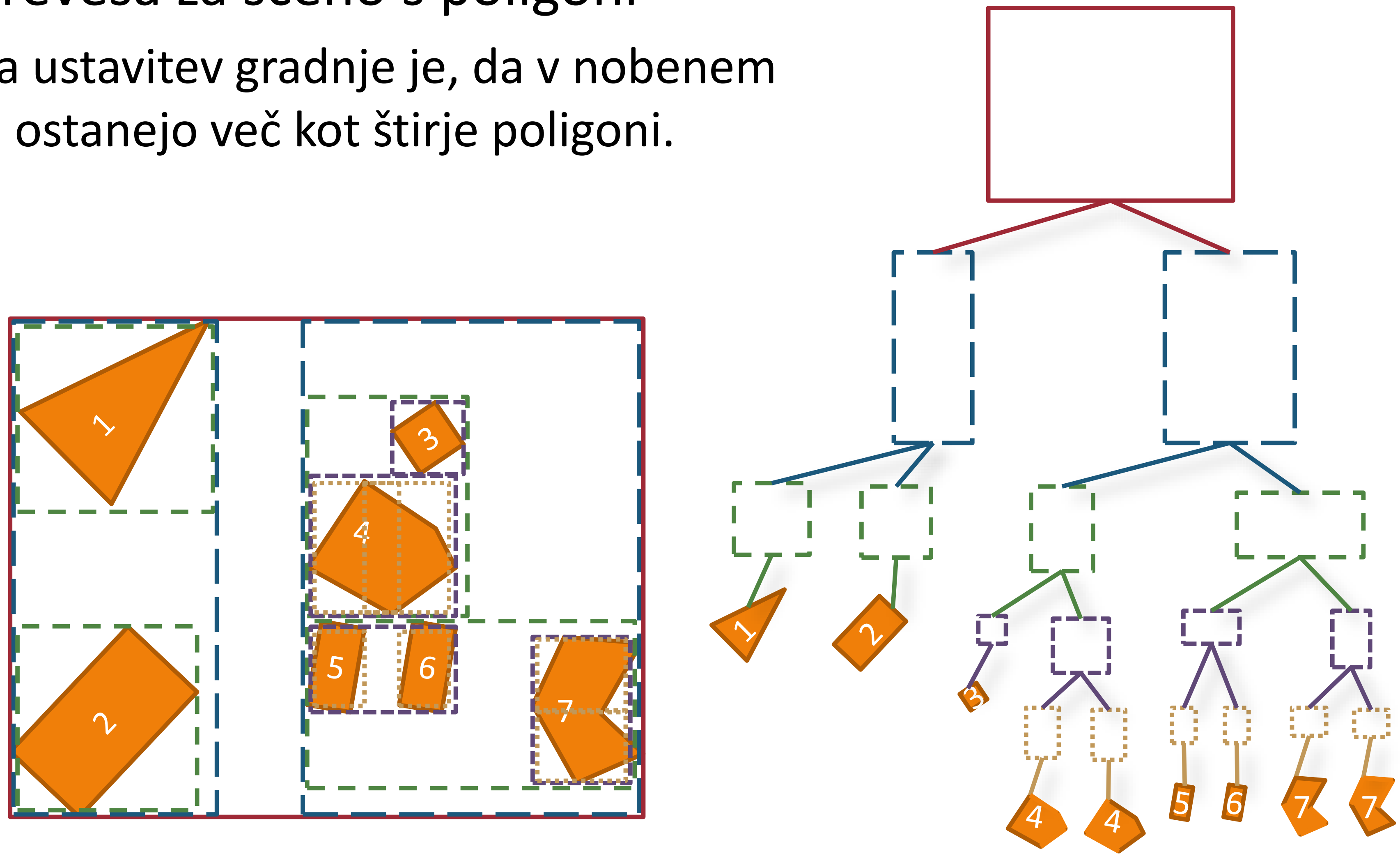
- **Binarno drevo osno poravnanih očrtanih okvirjev**
- Gradimo ga rekurzivno od zgoraj navzdol ali od spodaj navzgor
  - pogoji za ustavljanje so lahko različni npr. maksimalna globina ali maksimalno število poligonov v posameznem okviru
  - očrtani okvirji različnih predmetov naj bi se čim manj prekrivali
- Gradnja od zgoraj navzdol:
  - na vsakem koraku izračunamo najmanjši očrtan okvir vseh predmetov
  - vzdolž najdaljše stranice izberemo ravnino, ki bo okvir razdelila na dva dela
  - predmete/poligone razdelimo v obe polovici okvirja





AABB drevo

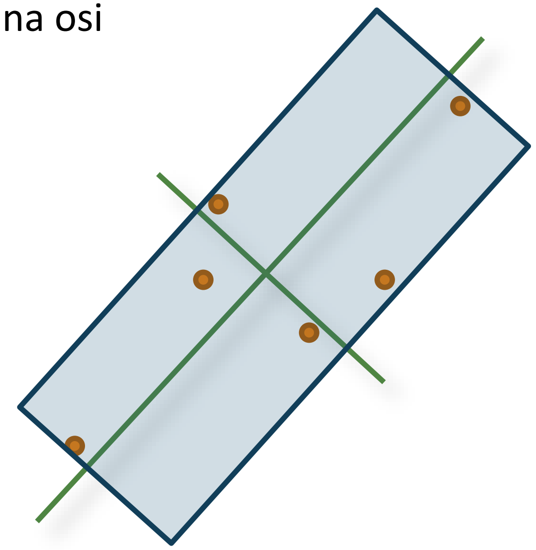
- Primer drevesa za sceno s poligoni
  - pogoj za ustavitev gradnje je, da v nobenem delu ne ostanejo več kot štirje poligoni.



OBB drevo

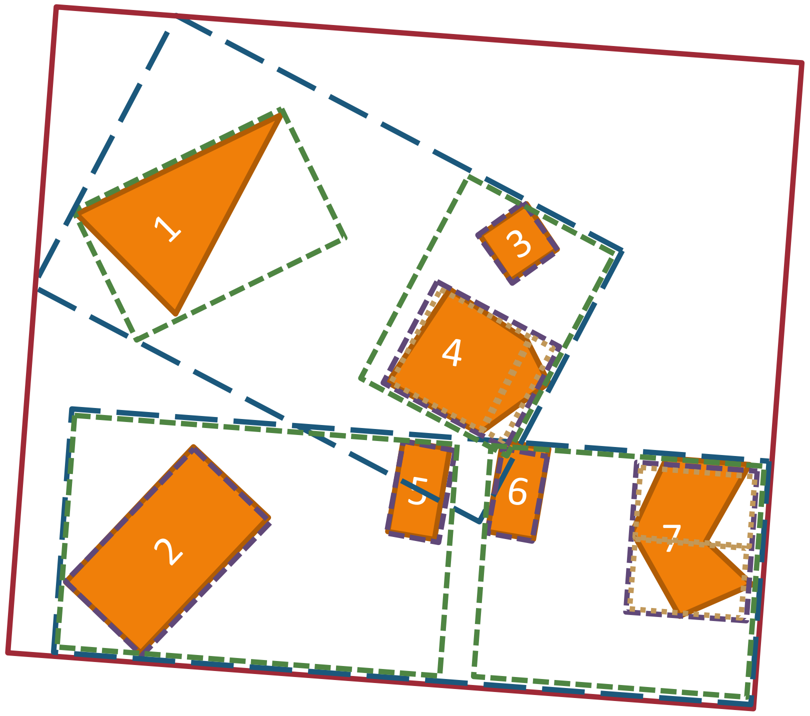
- **Binarno drevo usmerjenih očrtanih okvirjev** (OBB)
- Podobno kot AABB drevo, le da se bolje prilega, gradnja in računanje presekov pa je bolj zapleteno
  - več možnosti kako razdeliti geometrijo
  - navadno izračunamo lastne vektorje kovariančne matrike točk znotraj območja
    - dobimo osi z maksimalno in minimalno varianco položajev točk, na podlagi katerih izračunamo usmerjenost okvira

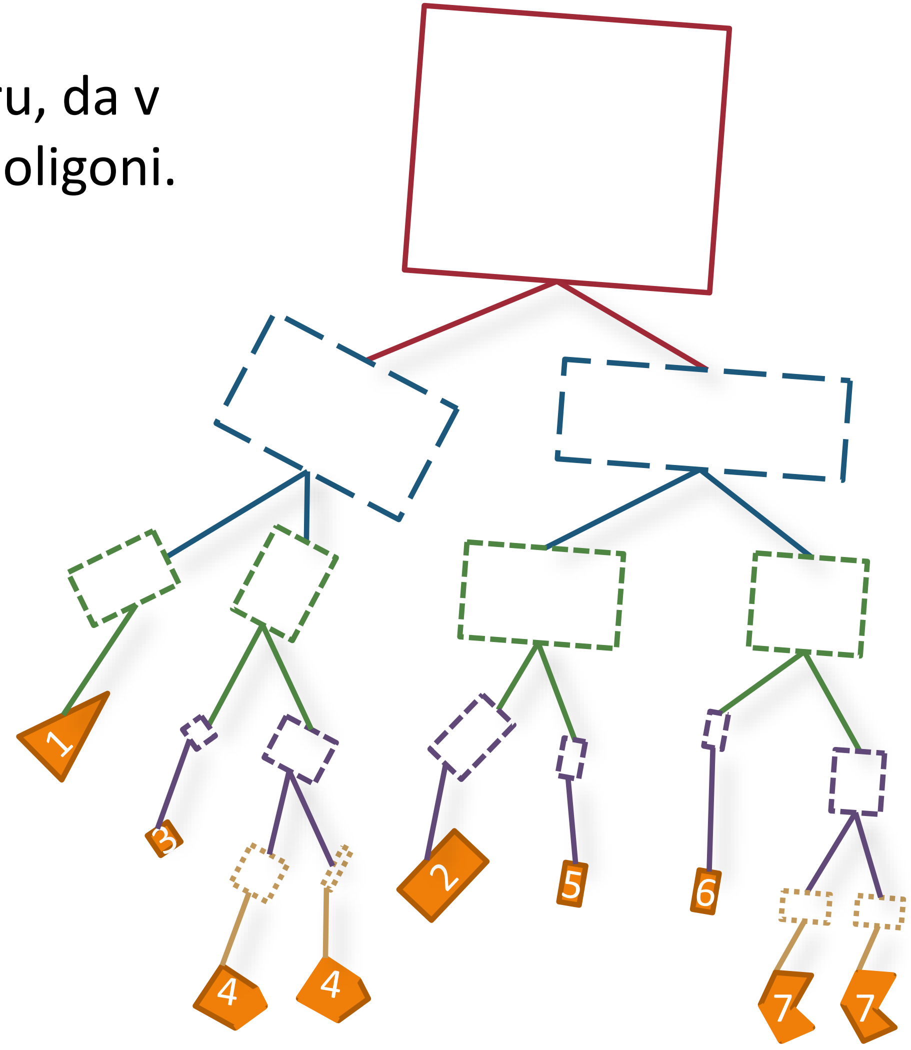
orientacijo okvira dobimo iz lastnih vektorjev kovariančne matrike, okvir usmerimo glede na osi



OBB drevesa

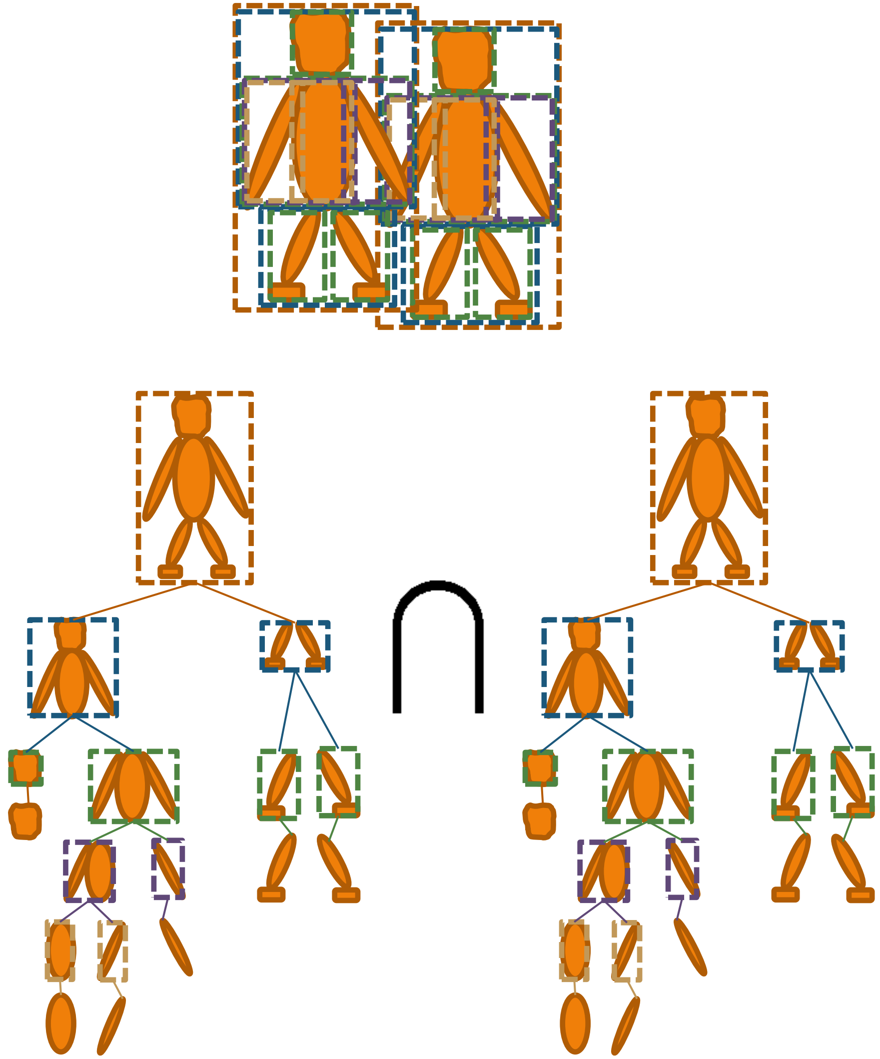
- Primer drevesa za sceno s poligoni
  - pogoj za ustavitev gradnje je v tem primeru, da v nobenem delu ne ostanejo več kot štirje poligoni.





Prekrivanje BVH

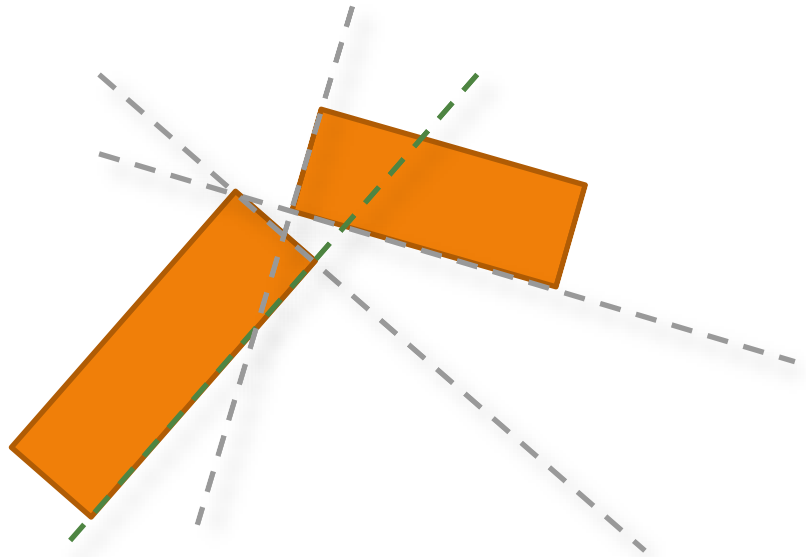
- Iskanje **presekov** dveh BVH dreves poteka rekurzivno, primerjamo pare vozlišč obeh dreves:
    1. če se očrtana prostora obeh vozlišč ne sekata, vrnemo *false*
    2. če sta obe vozlišči lista, izračunamo presek predmetov v listih in vrnemo rezultat
    3. če je eno vozlišče list, drugo pa ne, računamo presek lista z vsemi nasledniki drugega vozlišča
    4. če sta obe vozlišči notranji, računamo presek vozlišča z manjšim volumnom z nasledniki vozlišča z večjim volumnom



Prekrivanje OBB: ločitvena os

- Prekrivanje dveh AABB je enostavno izračunljivo
- Kako preverimo, če se dva **usmerjena (OBB)** okvirja prekrivata?
- **Izrek o ločitveni osi**:
  - če imamo dve **konveksni** telesi in najdemo os, na kateri se projekciji obeh teles ne prekrivata, se telesi ne prekrivata
- Za okvirje v 3D obstaja 15 možnih ločitvenih osi, ki jih vse preverimo (2x3 robovi vsakega okvira, in 3x3 križni produkti med robovi obeh okvirov)
  - če najdemo vsaj eno projekcijo na os brez prekrivanja, se okvirja ne prekrivata

Primer **ločitvene osi** v 2D:
na treh sivih oseh se projekciji prekrivata, na zeleni se ne – torej se pravokotnika ne prekrivata



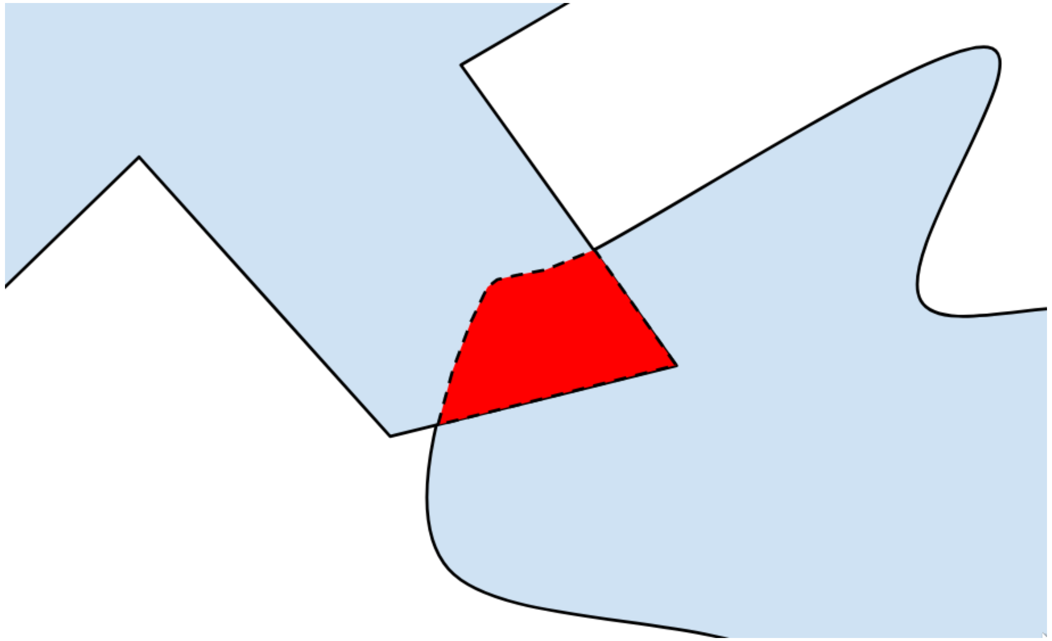
Ločitvena os: različni predmeti

- Tabela navaja osi, na katerih moramo preveriti ali se projekcije predmetov prekrivajo

3D predmeta	Normala (A)	Normala (B)	Robovi (AxB)	Skupaj osi
Črta-trikotnik	0	1	1x3	4
Črta–OBB	0	3	1x3	6
AABB–AABB	3	0(3)	0(3x0)	3
OBB–OBB	3	3	3x3	15
Trikotnik–Trik.	1	1	3x3	11
Trikotnik–OBB	1	3	3x3	13

Prekrivanje / razdalja med predmeti

- Računanje razdalj med konveksnimi predmeti:
  - *algoritem Gilbert-Johnson-Keerthi*
  - iterativna metoda
  - lahko vrne tudi najbližji točki med predmetoma
  - *opis implementacije*
- Če predmeti niso konveksni, jih lahko razdelime na konveksne poddele
  - npr. *V-HACD*



Vir: *the Gilbert–Johnson–Keerthi algorithm explained as simply as possible*

Presek dveh trikotnikov

- Veliko pristopov, npr. *interval overlap method*
  1. preveri prekrivanja na **normalah** – če je nek trikotnik popolnoma na eni strani ravnine drugega, ni preseka
  2. izračunaj premico preseka obeh ravnin
     $$L(t) = P + t(\boldsymbol{n}_1 \times \boldsymbol{n}_2)$$
  3. Projiciramo oba trikotnika na L - če se preseka sekata, se trikotnika prekrivata

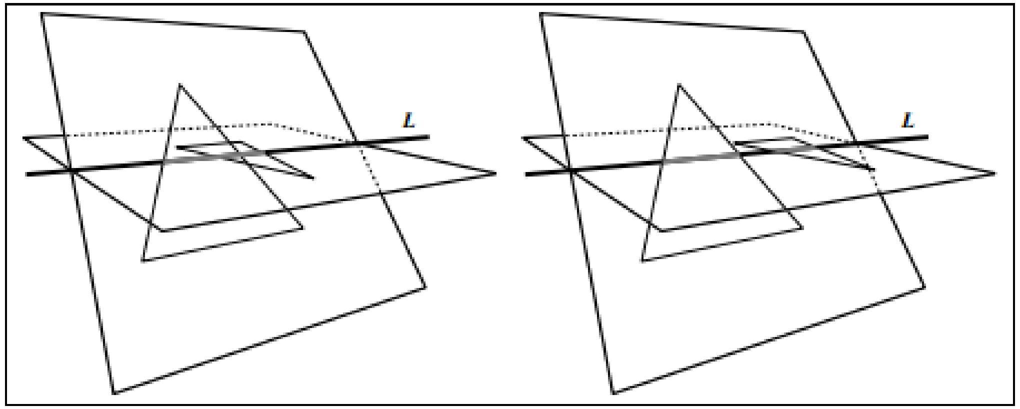


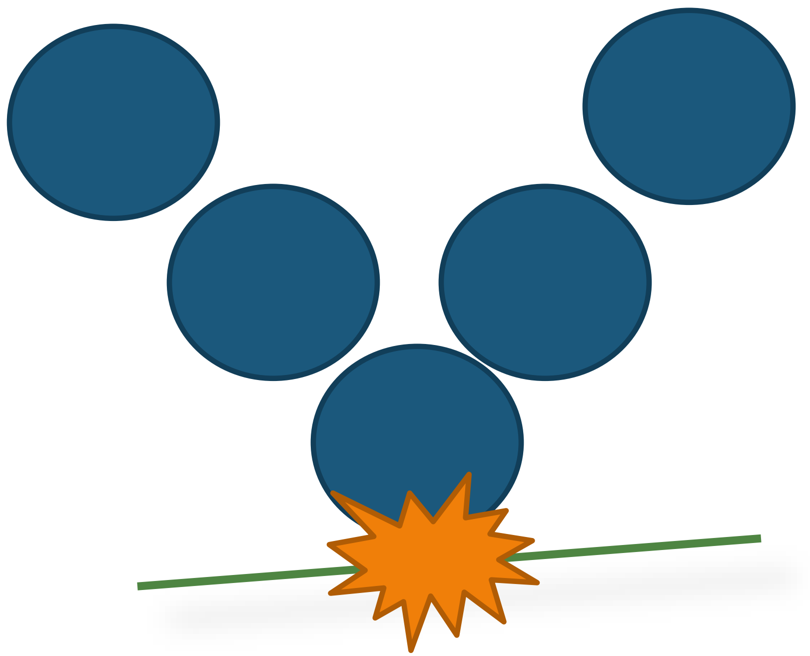
Figure 1: Triangles and the planes in which they lie. Intersection intervals are marked gray in both figures. Left: the intervals along L overlap as well as the triangles. Right: no intersection, the intervals do not overlap.

Vir: Moller, *A Fast Triangle-Triangle Intersection Test*, *1997*

Odziv na trk

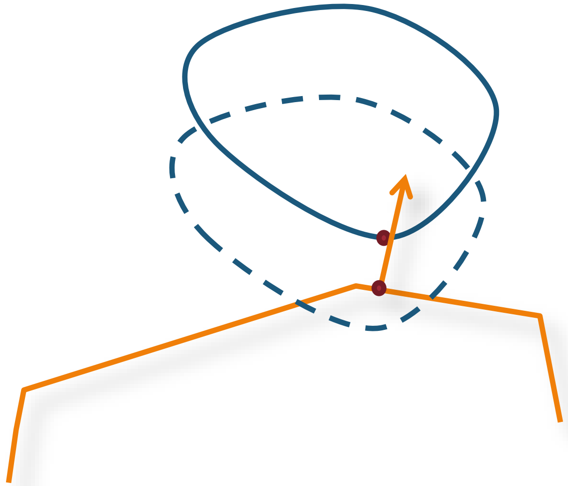
Odziv na trk

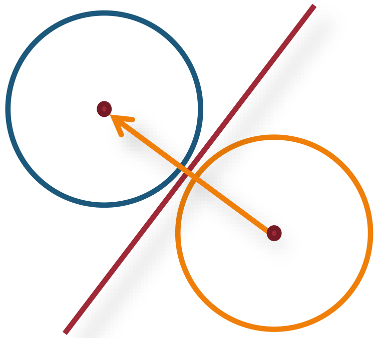
- Odziv na trk tipično pomeni določitev novih položajev/hitrosti predmetov, ki so trčili
  - lahko tudi deformacije ipd., sprožitev novih dogodkov (zvok, eksplozija itn.)
- Odziv pomeni:
  - izračun **normale trka**
  - izračun točnega **časa** trka
    - *če uporabimo preverjanje prekrivanj*
      - izračun globine prekrivanja
      - premik predmetov nazaj na položaj trka
  - izračun **novega gibanja** predmetov



Izračun normale

- Normala trka
  - vpliva na izračun odziva
  - predmeta se odbijeta „okoli“ normale trka
    - generiramo „impulz“ v smeri normale trka
- Za izračun normale lahko uporabimo položaj predmetov tik pred trkom
  - najdemo najbližji točki na obeh predmetih – tam je normala trka
  - pri kroglah je zelo enostavno, tam je normala trka razlika med obema središčema

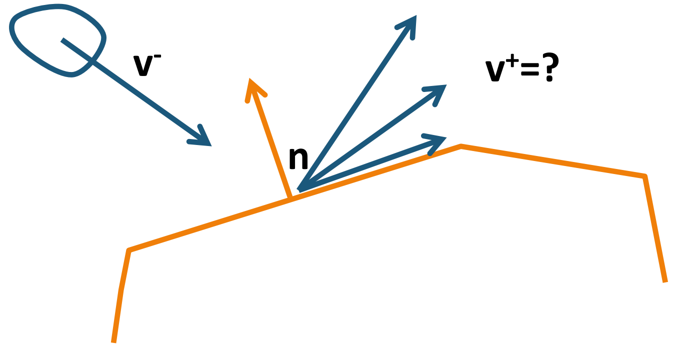




Gibanje po trku

- **v**⁻ hitrost pred trkom
- **v**⁺ po trku bo?

- Hitrost po trku v smeri normale trka se lahko zmanjša
  - **restitucijski** koeficient ε
    - 1 – popolnoma elastičen trk, 0 – se ne odbije
- Gibalna količina se spremeni: ob trku generiramo *impulz* moči *j* v smeri normale trka **n**
  - gibalna količina se spremeni za impulz *j***n**
- Podobna izpeljava tudi za
  - 2 telesi
  - če poleg hitrosti upoštevamo tudi rotacije



$$\boldsymbol{n}\cdot\boldsymbol{v}^{+} = -\varepsilon(\boldsymbol{n}\cdot\boldsymbol{v}^{-})$$

$$m\boldsymbol{v}^{+} = m\boldsymbol{v}^{-} + j\boldsymbol{n}$$
$$\boldsymbol{v}^{+} = \boldsymbol{v}^{-} + \frac{j\boldsymbol{n}}{m}$$

$$j = -\boldsymbol{n}\cdot\boldsymbol{v}^{-}(1+\varepsilon)\mathrm{m}$$
$$\boldsymbol{v}^{+} = \boldsymbol{v}^{-} - (\boldsymbol{n}\cdot\boldsymbol{v}^{-}(1+\varepsilon))\boldsymbol{n}$$

REFERENCE

- Eberly D. Intersection of Convex Objects: The Method of Separating Axes, 2008
- Ericson, Christer. *Real-Time Collision Detection*. Morgan Kaufmann 2005.
- J.V. Verth: Collision Response, Slides
- N. Souto: Video Game Physics Tutorial - Part II: Collision Detection for Solid Objects