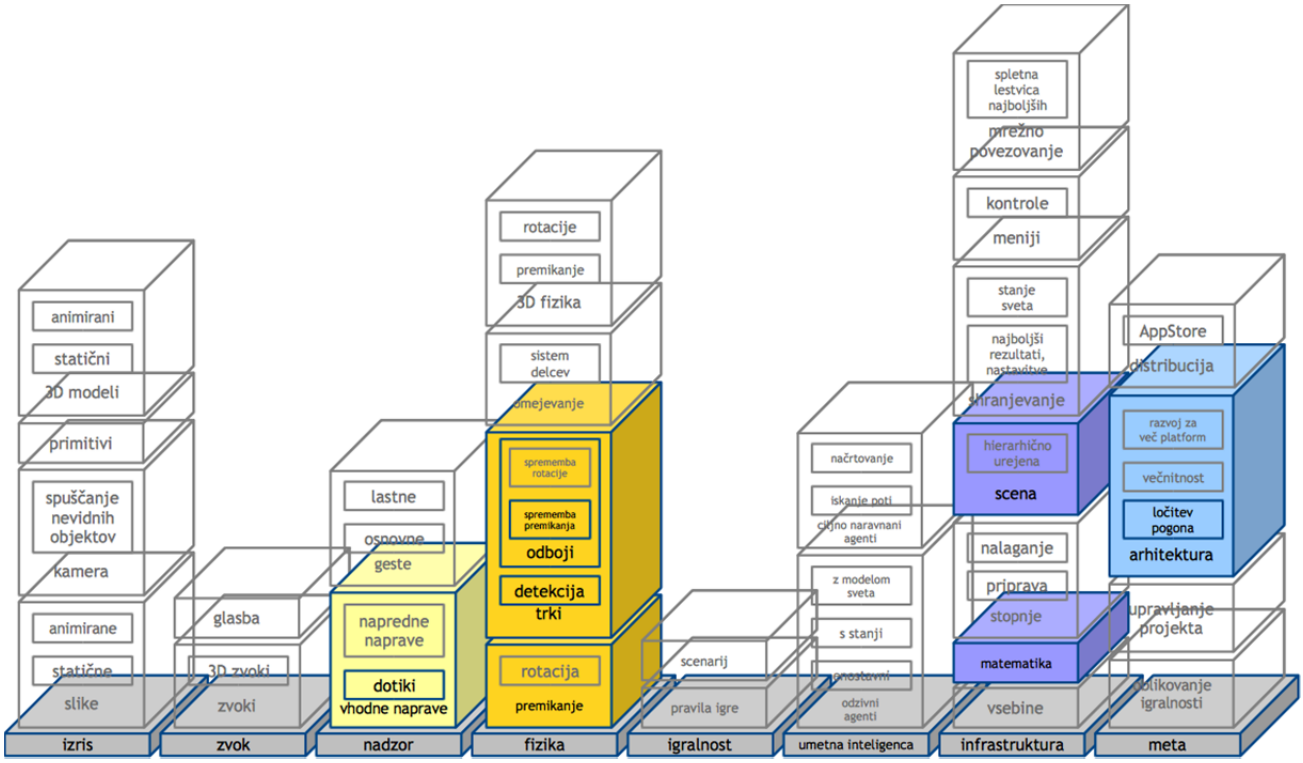
Sklop 3


animirani
statični
3D modeli
primitivi
spuščanje nevidnih objektov
kamera
animirane
statične
slike
izris
glasba
3D zvoki
zvoki
zvok
lastne
osnovne
geste
napredne naprave
dotiki
vhodne naprave
nadzor
rotacije
premikanje
3D fizika
sistem delcev
simejevanje
sprememba rotacije
sprememba premikanja
odboji
detekcija trki
rotacija
premikanje
fizika
scenarij
pravila igre
igralnost
načrtovanje
iskanje poti
cilino naravnani agenti
z modelom sveta
s stanji
odzivni agenti
umetna inteligenca
spletna lestvica najboljših
mrežno povezovanje
kontrole
meniji
stanje sveta
najboljši rezultati, nastavitve
shranjevanje
hierarhično urejena
scena
nalaganje
priprava
stopnje
matematika
vsebine
infrastruktura
AppStore
distribucija
razvoj za več platform
večnitnost
ločitev pogona
arhitektura
upravljanje projekta
oblikovanje igralnosti
meta

Nadzor

Kako s svojimi dejanji vplivamo na svet igre?

Brez interakcije seveda ni igre. Ko imamo svet izrisan na zaslon je zato prvo vprašanje, kako ga spreminjati. Odgovor se skriva v vhodnih napravah.

Običajno bi se na tem mestu pogovarjali o tipkovnici, miški in igralnih ploščkih. Pri igrah za moderne telefone z dotikovnim vmesnikom tega seveda nimamo. Namesto njih je tu večdotikovni zaslon.

Stanje vhodnih naprav

Vhod običajno obravnavamo na samem začetku cikla igrine zanke. To storimo tako, da iz vsake vhodne naprave preberemo trenutno stanje.

V XNAju tako iz razreda *Keyboard* dobimo strukturo *KeyboardState*, iz razreda *Mouse* stanje *MouseState*, za igralni plošek sta tu *GamePad* in *GamePadState* ter za konec, za mobilne telefone tudi razred *TouchPanel*, na katerem s statično metodo *GetState* dobimo seznam trenutnih dotikov v seznamu *TouchCollection*.

V XNIju, ki je namenjen napravam iOS, je od klasičnih vhodnih naprav na voljo le *TouchPanel* (zato pa pozna dodatne naprave kot je *Accelerometer* in *Compass* o katerih kasneje v poglavju o naprednih napravah).

Obstaja še malenkostna razlika: ker v objective-cju ni možno ustvarjati statičnih lastnosti na razredih (le na objektih), je za dostop do lastnosti *TouchPanela* prvo potrebno pridobiti objekt z uporabo statične metode getInstance. Takemu načinu izvedbe se sicer reče "edinec" (Singleton pattern).

Torej, za dostop do trenutnega stanja dotikov:

```
TouchCollection *touches = [TouchPanel getState];
```

Ter za uporabo lastnosti TouchPanela:

```
TouchPanel *touchPanel = [TouchPanel getInstance];

touchPanel.displayWidth – (int) total width of the input device
touchPanel.displayHeight – (int) total height of the input device
touchPanel.displayOrientation – (DisplayOrientation) orientation of the input device
touchPanel.enabledGestures – (GestureType) what gestures should the built in recognition engine report
```

Stanje dotikovnega vmesnika

Z ukazom *getState* dobimo objekt *TouchCollection*, ki je seznam vseh trenutnih dotikov na večdotikovnem vmesniku. Od tu naprej obstaja veliko možnosti, kako dotike obravnavamo.

V marsikaterem primeru igro upravljamo samo z enim dotikom, kot bi na računalnikih to počeli z miško. V tem primeru je najenostavneje, če več dotikov sploh ne obravnavamo in reagiramo zgolj v primeru, ko imamo en sam dotik.

```
if ([touches count] == 1) {
    TouchLocation *touch = [touches objectAtIndex:0];
    // Do something with touch.
}
```

Drugi glavni način je, da enostavno obravnavamo vse dotike in po potrebi filtriramo samo tiste, ki ustrezajo določenemu pogoju, recimo, če se lokacija dotika nahaja na določenem delu zaslona.

```
for (TouchLocation *touch in touches) {
        // Check some condition if this touch applies to us.
    if (touch ... ) {
        // Do something with touch
    }
}
```

Lastnosti dotika

Na tak ali drugačen način smo iz seznama TouchCollection prišli do konkretnega dotika, objekta TouchLocation. Kakšne lastnosti lahko preverjamo na njem? Pozicijo, stanje in identifikacijsko številko.

```
TouchLocation *touch;

touch.position – (Vector2) position of the touch on the input device
touch.state – (TouchLocationState) whether the touch was just pressed or released
touch.id - (int) an identifier that doesn't change in a life of the same touch
```

Ko se dotik prvič pojavi, dobi novo, do zdaj še neuporabljeno vrednost id in stanje TouchLocationStatePressed. V vseh naslednjih ciklih zanke bo stanje TouchLocationStateMoved, tudi če uporabnik dotika zares ne premika (vredonst position je lahko ista kot prej). Življenski cikel dotika se konča s stanjem TouchLocationStateReleased, nakar se v naslednjem ciklu zanke touch s tem id-jem odstrani iz seznama dotikov.

Naloga: dotiki

Zdaj je potrebno v igro dodati interaktivnost, tako da glede na dotike uporabnika spreminjamo neko vrednost objektov scene. Najenostavnejše je, da na lokacijo dotika nastavimo lokacijo obravnavanega objekta na sceni. Ko imamo izdelan tudi osnovni fizikalni pogon, ki premika objekte na sceni glede na hitrost, lahko namesto spreminjanja pozicije objektov glede na dotike spreminjamo hitrost objekta.

Naloga je opravljena, ko se objekti na sceni spreminjajo glede na dotike uporabnika, v skladu z želenim učinkom glede na igralnost vaše igre.

Fizika

Kako simuliramo fizikalne zakone v našem virtualnem svetu?

Za spreminjanje sveta igre v glavnem skrbi fizikalni pogon. Ko v zanki igre poskrbimo za interakcijo in smo glede na stanje vhodnih naprav spremenili svet igre, sceno prevzame fizikalni pogon in na njem izvede simulacijo fizikalnih zakonov. Med njimi si bomo zaenkrat pobliže ogledali premikanje in trke.

Premikanje

Zaradi izrisa smo že spoznali protokol IPosition, ki pove kje na sceni je objekt. Da dosežemo premikanje moramo enostavno spreminjati ta vektor. Če v vsakem obhodu zanke koordinati x prišetejemo 10 (v našem koordinatnem sistemu to ustreza 10 pikslom), se bo objekt že začel premikati in sicer 10 pikslov na obhod zanke. Logično!

```
item.position.x += 10;
```

Problem je, da se pri različno dolgem času zanke (lastnost *targetElapsedTime* na *Game* objektu) oziroma različnem številu obhodov na sekundo (FPS, frame per second), objekt premika različno hitro. Če igra teče na 60 FPS, se bo premaknil 600 pikslov v sekundi, če je FPS 30, bo sprememba za 300 pikslov. Še huje, če izklopimo fiksno dolžino prehodov (*isFixedTimeStep* na *Game* objektu nastavimo na *NO*), bo premikanje precej nekonstantno.

Če si želimo doseči premikanje s konstantno hitrostjo, moramo nekako upoštevati, koliko časa je dejansko preteklo od prejšnjega obhoda zanke in glede na to manj ali več premakniti objekt. Spomnimo se osnovnošolske fizike in enačbe za hitrost

$$v = \frac{\Delta x}{\Delta t}$$

oziroma, ker nas zanima, koliko se predmet premakne

$$\Delta x = v\Delta t$$

Pozicija objekta se mora spremeniti za hitrost pomnoženo s pretečenim časom. Slednjega najdemo v objektu *gameTime*, ki je poslan kot parameter metode *updateWithGameTime* pri igri in vseh komponentah.

Zdaj lahko enačbo za premikanje prepišemo v:

```
item.position.x += 10 * gameTime.elapsedGameTime;
```

Na ta način smo dosegli, da se objekt premika v desno s konstantno hitrostjo 10 pikslov na sekundo.

Hitrost

Običajno hitrosti nimamo podane kot fiksno številko, temveč je to lastnost objekta. Poleg tega nas ne zanima le sama hitrost (speed), temveč tudi smer v katero se objekt premika (direction). Oboje skupaj opisuje vektor hitrosti (velocity) s smerjo in dolžino.

Da bomo lahko iz pogonov preverjali, ali določen objekt podpira hitrost, ustvarimo nov protokol *IVelocity*, ki zahteva vektor hitrosti.

```
@protocol IVelocity <NSObject>

@property (nonatomic, retain) Vector2 *velocity;

@end
```

Zdaj lahko enačbo za premikanje prepišemo še v vektorsko obliko

$$\vec{x} = \vec{x}_0 + \vec{v}\Delta t$$

V kodi, z uporabo metod razreda *Vector2* pa:

```
[item.position add:[Vector2 multiply:item.velocity by:gameTime.elapsedGameTime]];
```

Ker za simuliranje premikanja potrebujemo na objektu scene tako pozicijo kot hitrost, si pripravimo še protokol *IMovable*, ki preko vključevanja več protokolov zahteva oboje hkrati.

```
@protocol IMovable <IPosition, IVelocity>

@end
```

Tako lahko naredimo, da del fizikalnega pogona, ki skrbi za premikanje, deluje samo nad objekti, ki vključujejo protokol *IMovable*. Opozorimo le na to, da mora objekt izrecno uporabiti *IMovable*. Če ima posamično *IPosition* in *IVelocity* namreč ne pomeni, da bo samodejno viden tudi kot *IMovable*.

Končna simulacijska metoda za premikanje je torej:

```
+ (void) simulateMovementOn:(id)item withElapsed:(float)elapsed {
        id<IMovable> movable = [item conformsToProtocol:@protocol(IMovable)] ? item : nil;

        if (movable) {
            [movable.position add:[Vector2 multiply:movable.velocity by:elapsed]];
        }
}
```

Naloga: premikanje

Fiziko lahko v projekt dodamo na več načinov. Izvajamo jo lahko v sami metodi igre *updateWithGameTime* oziroma v istoimenski metodi ene od že izdelanih komponent (*Gameplay*, *Level*). Takoj ko gre za kompleksnejše obravnavanje, je smiselno ustvariti novo komponento *PhysicsEngine*, ki deluje nad sceno ali stopnjo ter pokliče vse ustrezne izračune glede na želeno obnašanje.

Možno je spisati tudi povsem splošen fizikalni pogon, ki glede na prisotnost protokolov na objektih scene računa vse potrebno (recimo premika vse objekte *IMovable*, vrti *IRotatable*, dela trke nad *IParticleCollider* itd.), še boljše pa je, če fizikalni pogon optimiziramo za zvrst igre.

V splošnem delu pogona si samo pripravimo posamezne izračune (premikanje, rotacija, računanje sil, trkov), ki jih v fizikalnem pogonu za konkretno igro po potrebi uporabljamo. Tako je v igri podiranja zidu na sceni naenkrat žoga in več deset opek, a nam ni potrebno delati detekcije trkov med vsemi pari opek, temveč le med žogo in opekami. Povsem splošen pogon, pri katerem samo pokličemo simulacijo nad celotno sceno, se brez naprednih tehnik (recimo ločevanje predmetov v različne skupine za trke) temu ne bi znal izogniti.

Ko se odločimo na katerem mestu in kako splošno bomo spisali fizikalni pogon, izdelamo najbolj osnovni del, to je računanje enačbe za premikanje. Da preverimo delovanje enemu od objektov, nastavimo začetno hitrost in, če je vse pravilno povezano, ga bo fizikalni pogon začel premikati po sceni.