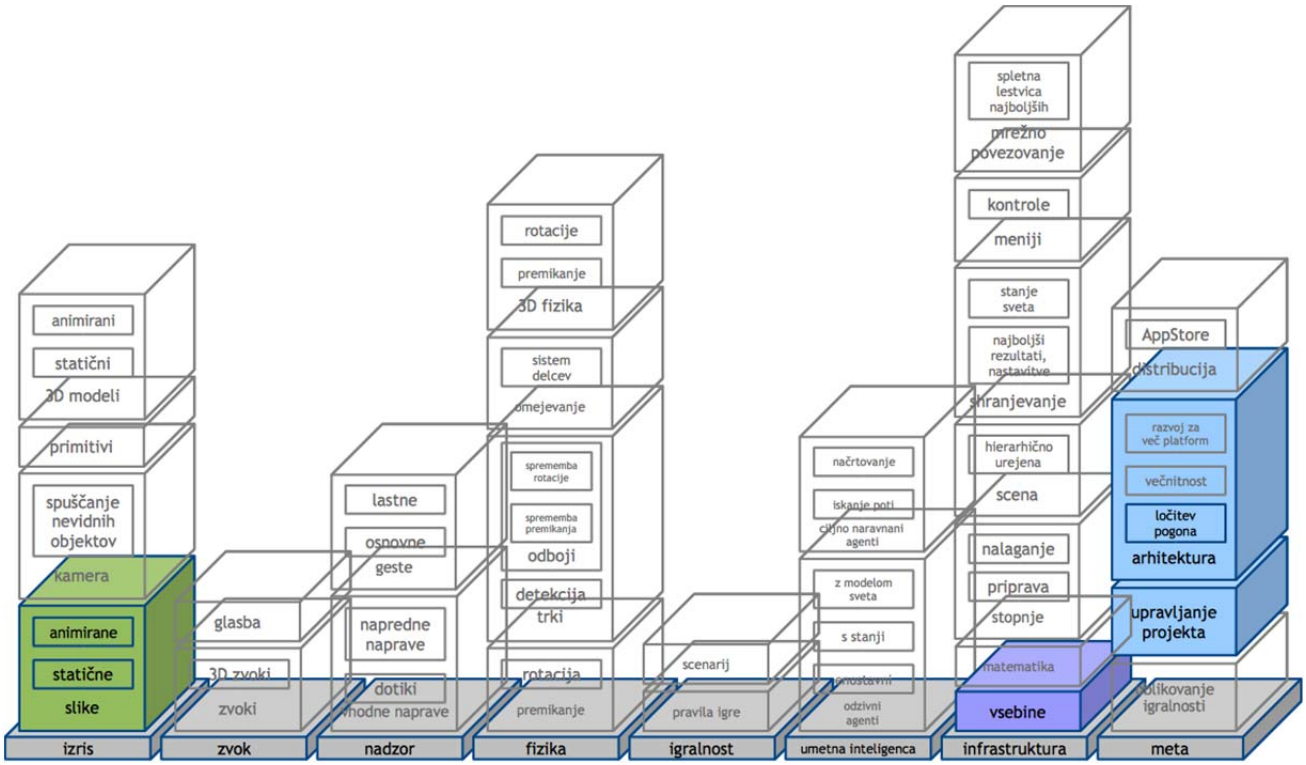
Sklop 2 (nadaljevanje)



Arhitektura

Kako bodo organizirani razredi naše igre, kakšna bo struktura objektov med njimi?

Kot smo povedali, je XNI več kot knjižnica, je ogrodje (angl. framework). To pomeni, da ne prinaša le veliko uporabnih funkcij in razredov, temveč ima izdelano že celotno arhitekturo za delovanje igre, katere osnova je izvajanje glavne zanke igre. Videli smo, da ustvarimo igro tako, da dedujemo iz razreda *Game* in prepišemo želene metode kot so *initialize*, *loadContent*, *updateWithGameTime* in *drawWithGameTime*.

Arhitektura ogrodja XNI

Da bomo razumeli delovanje naše igre, moramo najprej razumeti, kako deluje XNIjev objekt *Game*.

Konstruktor

V datoteki main.m smo zagnali aplikacijo z ukazom

```
UIApplicationMain(argc, argv, @"GameHost", @"MyGame");
```

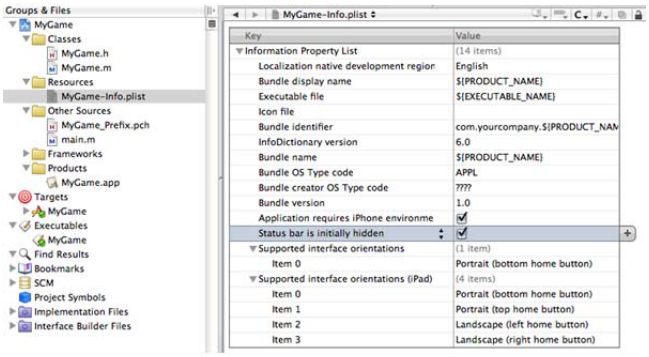
Ta bo za nas ustvaril objekt tipa *MyGame*, se pravi glavni razred naše igre, ki deduje iz razreda *Game*. V objective-cju je to združeno v alokacijo in klic konstruktorja metode *init* (če nima parametrov*)*. V ozadju torej XNI poskrbi, da se čisto na začetku torej pokliče

```
[[MyGame alloc] init];
```

Kot smo videli v pripravi projekta, smo v našem konstruktorju ustvarili objekt *GraphicsManager*. Po potrebi na tem mestu ustvarimo še vse ostale razrede, predvsem komponente (o njih malce kasneje). Prav tako lahko nastavimo lastnosti objekta *GraphicsManager* in *Game*, recimo, koliko naj traja želeni čas obhoda zanke (*targetElapsedTime*) in ali naj sploh uporablja fiksni čas obhoda (*isFixedTimeStep*). Privzete vrednosti so naslednje:

```
// Graphics manager properties
graphics.isFullScreen = NO;
graphics.supportedOrientations = DisplayOrientationPortrait;


// Game properties
self.isFixedTimeStep = YES;
self.targetElapsedTime = 1.0 / 60.0;
self.inactiveSleepTime = 1.0 / 5.0;
```

Na tem mestu omenimo še to, da se privzete vrednosti objekta GraphicsManager skladajo z nastavitvami v MyGame-plist.info datoteki v skupini Resources. Lastnost isFullScreen se nastavi glede na vrstico *Status bar is initially hidden* (samodejno je v datoteki plist ni, dodamo jo s klikom na plus ob strani vrstice). Podprte orientacije naprave pa se naložijo glede na vrednosti *Supported interface orientations*. Nastavitve orientacije naprave lahko spreminjamo tudi preko grafičnega vmesnika, če kliknemo na projekt in potem v zavihku *summary*.



Inicializacija

Po ustvarjenju ciljnega objekta, ki deduje iz razreda *Game*, XNI glede na nastavitve objekta *GraphicsManager* inicializira grafično knjižnico (v našem primeru OpenGL) s katero komuniciramo preko objekta *GraphicsDevice*.

Ko je to storjeno, pokliče metodo *initialize* na naši igri, le ta pa še metodo *loadContent*.

Priporočeno je, da v initialize postorimo vse potrebno glede nalaganja in nastavitve splošnih objektov, medtem ko v loadContent naložimo vse potrebne grafične objekte. To so tisti objekti, ki so odvisni od grafične strojne opreme in potrebujejo za inicializacijo delujoč objekt *GraphicsDevice*.

Ne pozabimo, da metodo initialize prepisujemo, torej moramo poklicati initialize tudi na očetu.

```
- (void) initialize {
        // Do any non-graphics related initialization code here.

        [super initialize]; // Parent initialization will call loadContent here.
}

- (void) loadContent {
        // Load all content with the ContentManager stored in self.content.
        someContent = [self.content load:@"someFile"];

        // Create all graphics resources using self.graphicsDevice.
        someResource = [[ResourceClass alloc] initWithGraphicsDevice:self.graphicsDevice];
}
```

Igrina zanka

Po zaključeni inicializaciji igra vstopi v izvajanje glavne zanke. Na tem mestu zaporedno kliče metodi updateWithGameTime in drawWithGameTime. V prvi imamo čas, da spreminjamo naš svet, v drugi pa igra pričakuje, da z grafičnimi objekti na zaslon izrišemo predstavitev našega sveta.

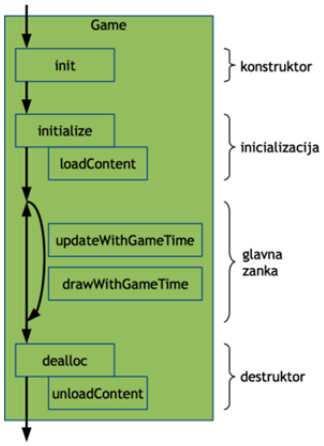
Update in draw se v primeru nastavitve isFixedTimeStep na YES kličeta v enakomernih intervalih, nastavljenih z lastnostjo targetElapsedTime. Če v igri porabimo manj časa od vsega, ki ga je na voljo, bo XNI začasno zaustavil izvajanje in s tem tudi bolj varčeval z baterijo naprave. Če je dejanski čas izvajanja update in draw metod daljši od ciljnega intervala, bo v objektu gameTime vrednost isRunningSlowly nastavlja na YES.

Če isFixedTimeStep spremenimo na NO, se update in draw kličeta nepretrgoma en za drugim, kolikor hitro je mogoče.

V vsakem primeru, tako za fiksen, kot za spremenljiv časovni interval, pa v lastnosti elapsedGameTime parametra gameTime izvemo, koliko časa je dejansko preteklo od prejšnjega obhoda zanke.

```
// Update your game world here.
- (void) updateWithGameTime:(GameTime *)gameTime {
        // Use elapsedGameTime to do your physics calculations.
        float elapsedSeconds = gameTime.elapsedGameTime;

        // Process input.
        // Run artificial intelligence.
        // Simulate physics.
        // Update scene objects.
        // Apply game rules.
        // ...
}


// Draw the game world.

- (void) drawWithGameTime:(GameTime *)gameTime {
        // Usually we first clear the whole screen to some color.
        [self.graphicsDevice clearWithColor:[Color white]];

        // Draw scene objects.
        // ...
}
```

Celotna slika izvajanja XNI igre izgleda takole:



Arhitektura igre

S povedanim o arhitekturi razreda *Game* v ogrodju XNI bi lahko izdelali celotno igro tako, da bi vso potrebno kodo za procesiranje in izris spisali v naš glavni objekt. Za manjši prototip ali tehnološki demo je to povsem sprejemljivo. Problem nastane, ko količina kode narase in se iz tako dolgega razreda (v angleščini poimenovanega tudi spaghetti code) težko znajdemo. Objektno orientirano programiranje na tem mestu priskoči na pomoč z ločitvijo zaključenih delov kode v svoje razrede.

XNI Komponente

Za razdelitv igre na ločene dele ima XNI že pripravljeno arhitekturo komponent.
Razreda *GameComponent* in *DrawableGameComponent* v veliki meri spominjata na strukturo razreda *Game*. Na voljo imamo metode *initialize*, *updateWithGameTime* in pri *DrawableGameComponent* še *loadContent* ter *drawWithGameTime*.

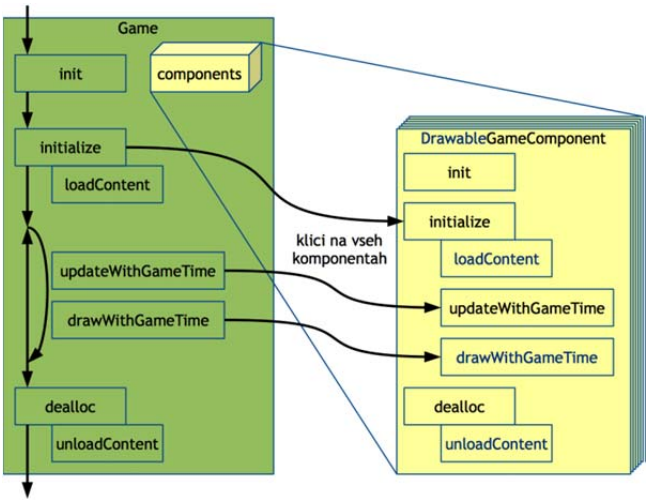
Na ta način lahko zelo enostavno iz same igre ločimo neko zaključeno funkcionalnost, recimo šteteje izrisanih slik na sekundo in jih zapakiramo v delujoč razred.

```
@interface FpsComponent : GameComponent {
        int fpsCounter;
        NSDate *countStart;

}

@end
@implementation FpsComponent

- (void) initialize {
        countStart = [[NSDate alloc] init];

}

- (void) updateWithGameTime:(GameTime *)gameTime {
        fpsCounter++;
        NSDate *currentTime = [NSDate dateWithTimeIntervalSinceNow:0];
        NSTimeInterval countDuration = [currentTime timeIntervalSinceDate:countStart];

        if (countDuration > 1) {
           NSLog(@"FPS: %i", fpsCounter);
           fpsCounter = 0;

           [countStart release];
           countStart = [currentTime retain];

           if (gameTime.isRunningSlowly) {
              NSLog(@"Game is running slowly!");
           }
        }
}

- (void) dealloc {
        [countStart release];
        [super dealloc];
}

@end
```

Zdaj nam v sami igri preostane le še to, da komponento ustvarimo in jo dodamo v seznam komponent. Običajno to storimo kar v samem konstruktorju igre.

```
[self.components addComponent:[[[FpsComponent alloc] initWithGame:self] autorelease]];
```

XNI bo za nas poklical intialize, updateWithGameTime in drawWithGameTime na vseh komponentah, ki smo jih ustrezno dodali. Slika izvajanja arhitekture je v primeru komponent sledeča:



Omenimo še, da je mogoče izvajati natančen nadzor nad tem, v kakšnem vrstnem redu se izvajajo klici na komponentah in ali se update ter draw sploh pokličeta. Za GameComponent sta na voljo lastnosti:

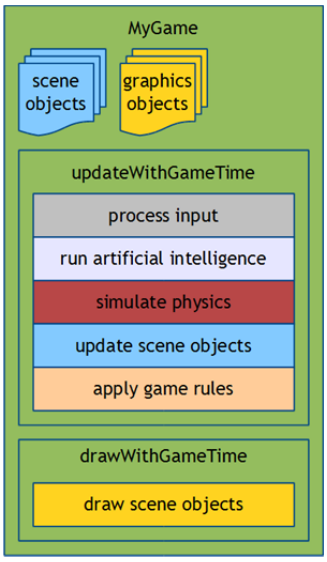
```
GameComponent *component;

component.enabled – (BOOL) controls if updateWithGameTime is called
component.updateOrder – (int) lower order components' update will be called first
```

ter pri DrawableGameComponent dodatno še:

```
DrawableGameComponent *component;

component.visible – (BOOL) controls if drawWithGameTime is called
component.drawOrder – (int) lower order components' draw will be called first
```

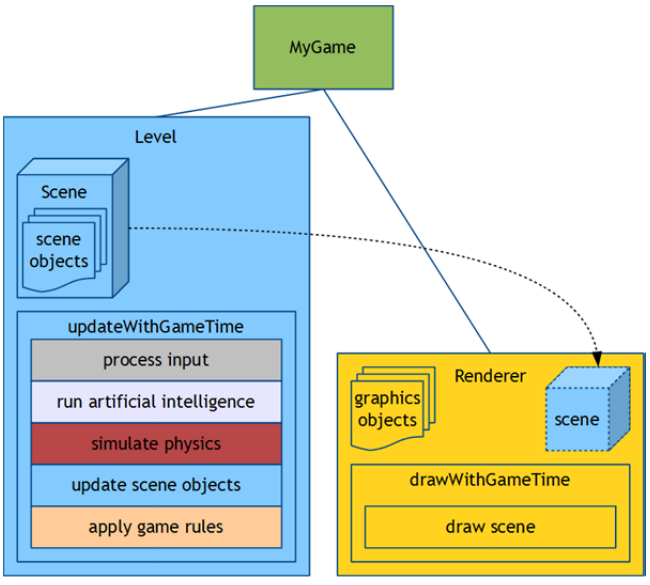
Komponente igre

S komponentami lahko logiko igre postopno ločimo v zaključene celote. Kot smo že omenili, bi lahko celotno igrino zanko spisali v glavnem objektu igre, ki deduje iz razreda *Game*. V tem primeru bi, kot smo nakazali v opisu igrine zanke, vso logiko združili v update in draw metodi enega samega objekta. Dobili bi takšno arhitekturo:

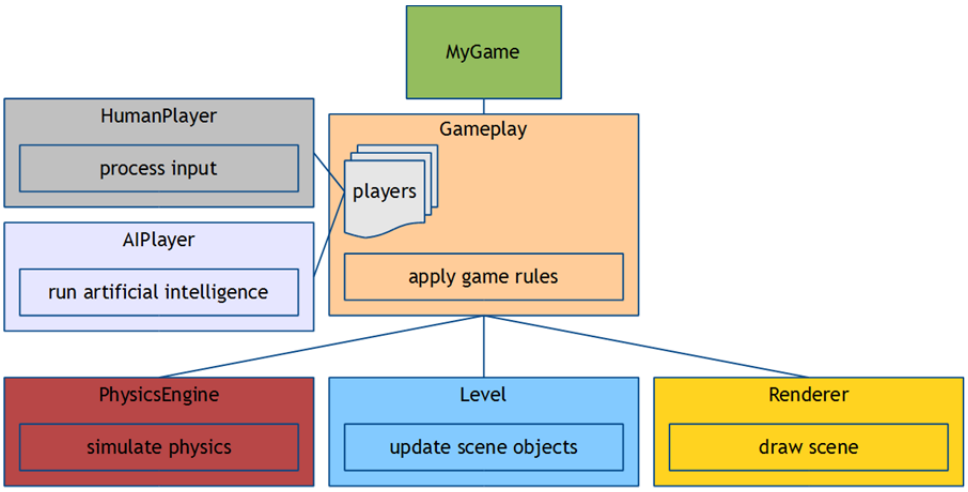


Neposredno v igri bi hranili vse spremenljivke za objekte na sceni, kakor tudi vse potrebne grafične objekte za izris. Metoda *updateWithGameTime* bi bila zelo dolga in z dodajanjem elementov vedno bolj kompleksna ter prepletena. Zato se takega načina programiranja drži izraz [spaghetti code](#).

Za začetek lahko stvari uredimo, če ločimo izris od podatkov. Vse objekte na sceni združimo v objekt *Scene*, katerega ustvari komponenta *Level*. Na drugi strani ustvarimo izrisljivo komponento *Renderer*, ki od stopnje prejme sceno in jo s pomočjo grafičnih objektov izriše. Tako se lahko v stopnji ukvarjamo s samim posodabljanjem sveta igre, medtem ko lahko izris poljubno zamenjamo in dograjujemo brez vpliva na samo simulacijo.



Preostane nam še, da samo procesiranje razdelimo v smiselne komponente in vse skupaj zapremo v eno samo komponento, *Gameplay*, ki zna ustvariti vse potrebne komponente za izvajanje igranja igre. Na podoben način bomo kasneje dodali še ostale vrhovne komponente kot so uvod, meniji in podobna stanja igre, med katerimi prehajamo v času od zagona do zaprtja aplikacije.



Vidimo, da ima *Gameplay* seznam igralcev, ki z interakcijo vplivajo na objekte na sceni. Na eni strani imamo človeške igralce, od katerih ukaze sprejemamo in interpretiramo preko vhodnih naprav ter na drugi strani umetno-inteligenčne nasprotnike, ki z različnimi algoritmi izračunajo katero dejanje naj stori lik, katerega upravljajo.

Fizikalni pogon, spet nad objekti scene, simulira potrebne fizikalne zakone, detektira trke in podobno. Sledi posodabljanje stopnje, kjer se običajno s scene odstranijo uničeni objekti in dodajo novi, glede na potek dogajanja v stopnji. Na koncu klicev update se izvede še komponenta *Gameplay*, kjer se preverijo pravila igre v obliki raznih pogojev za zmago ali izgubo.

Igrino zanko zaključi *Renderer*, ki trenutno stanje objektov na sceni izriše.

Na ta način imamo kodo lepo organizirano v zaključene enote, ki se ukvarjajo zgolj z določenim delom igre. To je glavni smisel objektno-orientiranega programiranja.

Ločitev pogona

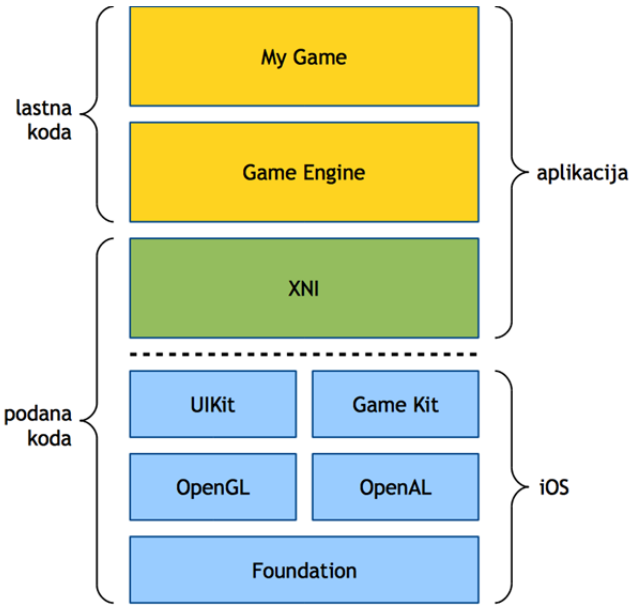
Čeprav smo igro horizontalno ločili na različne komponente igre: grafiko, fiziko, vhodne naprave in podobno, bo vsaka od teh komponent še vedno precej obsežna. Za simuliranje fizike in detekcijo trkov bomo morali spisati precej vrstic kode. Če bomo zopet vse spisali v en sam razred, se bo zgodba o kompleksnosti ponovila.

Poleg velikega obsega se koda deli še na tako, ki je specifična za konkretno igro, ter bolj splošno, večkrat uporabljivo kodo. Komponenta za izpis števila sličic na sekundo je že tak primer. Pri izrisu bomo potrebovali podatkovno strukturo Sprite, ki nam bi prišla prav v kakršnikoli igri. Del fizikalnega pogona, ki detektira prekrivanje dveh krogov ali spremembo energije pri trku, je prav tako neodvisen matematični ali fizikalni izračun.

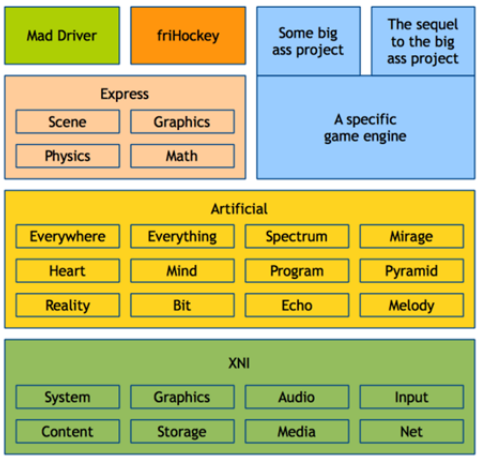
Na ta način lahko ločimo splošne komponente in razrede v pogon, ogrodje, oziroma knjižnico. Tako našo igro vertikalno razdelimo na dva ali več projektov, pri čemer je zgornji projekt konkretna igra, spodaj pa ležijo vedno bolj splošne knjižnice. Na samem dnu imamo XNI, ki je prav tako ogrodje v obliki statične knjižnice. Seveda je mogoče iti še nižje. Grafična knjižnica pod XNIjem je OpenGL, za

delo z uporabniškim vmesnikom iOSa je tu UIKit in za najosnovnejše razrede objective-cja skrbi Foundation.

Poudarimo, da lahko projekt na zgornjem sloju uporablja vse knjižnice pod njim in ne le tisto, ki je takoj pod njim. Iz naše igre, tudi če uporabljamo pogon, namreč vseskozi neposredno uporabljamo razrede XNIja in objective-c Foundationa.



Na ta način lahko svojo kodo ločimo v dva ali več slojev glede na njihovo splošnost in uporabnost v različnih projektih. Slika malce bolj zapoletenega sistema lastnih pogonov in knjižnic bi lahko bila na primer taka:



Na najnižjem nivoju je XNI, ki skrbi za povsem splošno ogrodje igrine zanke in komponent, ter prinaša vse potrebne sestavne dele za izdelavo igre: vmesnik do grafičnega in zvočnega čipa, branje stanja vhodnih naprav, nalaganje vsebin in podobno.

Artificial je primer lastne splošne knjižnice, v kateri so razširitve XNIjeve funkcionalnosti, kot tudi implementacija splošnih, povsod uporabnih algoritmov in razredov, od izpisovanja števila slik na

sekundo in komponente za izris črt, krogov in pravokotnikov, do splošnih geometrijskih izračunov in simboličnega glasbenega zapisa.

Nad tem je zgrajen že bolj specifičen pogon Express, ki je namenjen hitri izdelavi prototipov. Prinaša osnovno implementacijo delov fizikalnega pogona, protokole za objekte na sceni, strukture za predstavitev geometrijskih oblik za detekcijo trkov in uporabne grafične objekte.

S takim pogonom je moč spisati vrsto enostavnih iger, kot sta recimo Mad Driver in friHockey.

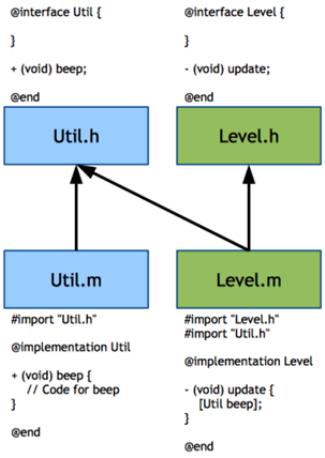
Po drugi strani bi za kakšen večji projekt spisali povsem specifičen pogon za to igro, ki bi se povsem prilagajal izbrani zvrsti in grafičnim potrebam. Še vedno bi igro ločil v dva projekta, pogon spodaj in konkretno vsebino igre zgoraj, tako da bi lahko na enostaven način kasneje z istim pogonom izdelali nadaljevanje ali razširitev z zgolj novimi vsebinami in dodatno funkcionalnostjo.

Organizacija razredov in zaglavne datoteke (angl. header files)

V programskih jezikih, kot sta c# in java, obstaja koncept združevanja objektov v imenske prostore (angl. namespace). Na ta način se izognemo težavam, če bi recimo v naši igri izdelali razred z istim imenom, kot že recimo obstaja v XNAju. Dodatno z vključevanjem (import) celotnega imenskega prostora pridobimo dostop za uporabo vseh razredov, ki so v njem definirani.

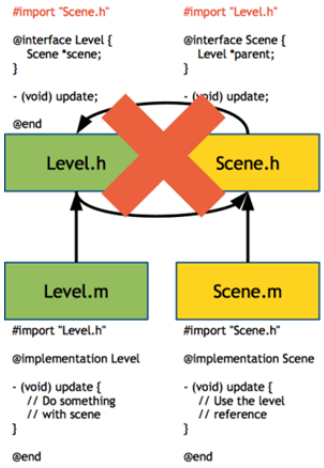
Tako c kot objective-c združevanja v imenske prostore ne poznata, zato moramo vedno paziti, da pri uporabi knjižnice XNI ne poimenujemo kakšnega svojega razreda enako, kot že obstaja v XNIju (recimo Game, Texture, Rectangle in podobno).

Po drugi strani za medsebojno uporabo razredov uporablja ločitev definicije objektov v zaglavne (angl. header) datoteke (.h) in sámo implementacijo (.m). Za to, da bo prevajalnik znal prevesti .m datoteko, potrebuje zgolj .h datoteke vseh razredov, ki se v implementaciji uporabijo.

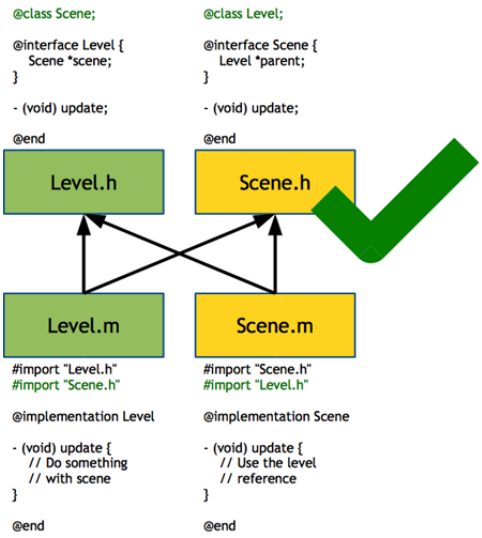


Level v implementaciji potrebuje metodo beep razreda Util, zato v Level.m preprosto uvozimo Util.h, da bo prevajalnik vedel za obstoj te metode in razreda.

Problem nastane, ko se moramo na nek zunanji razred sklicevati že v sami zaglavni datoteki. To je potrebno vedno, ko želimo deklarirati spremenljivko razdreda ali pa parameter ene od metod z zunanjim tipom. Naivna rešitev je, da premaknemo import zunanjega razreda v samo zaglavno datoteko. To deluje, vse dokler ne pride do primera, ko se dva razreda sklicujeta en na drugega. Ko nastopi cikel poskuša v spodnjem primeru Level.h vključiti Scene.h, ta zopet Level.h, spet Scene.h ...

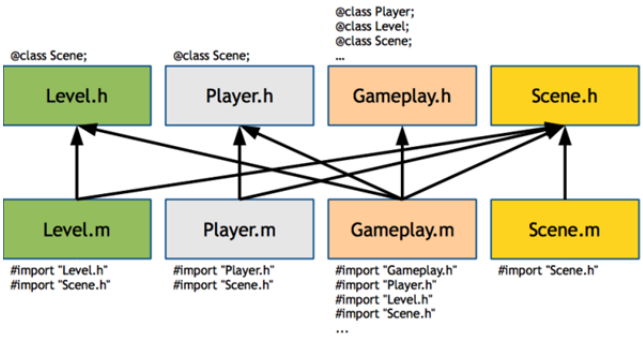


Za rešitev tega problema pozna objective-c direktivo @class, ki prevajalniku pove, da je Scene razred in naj pri obravnavanju spremenljivke sklepa, da gre preprosto za kazalec na nek objekt.

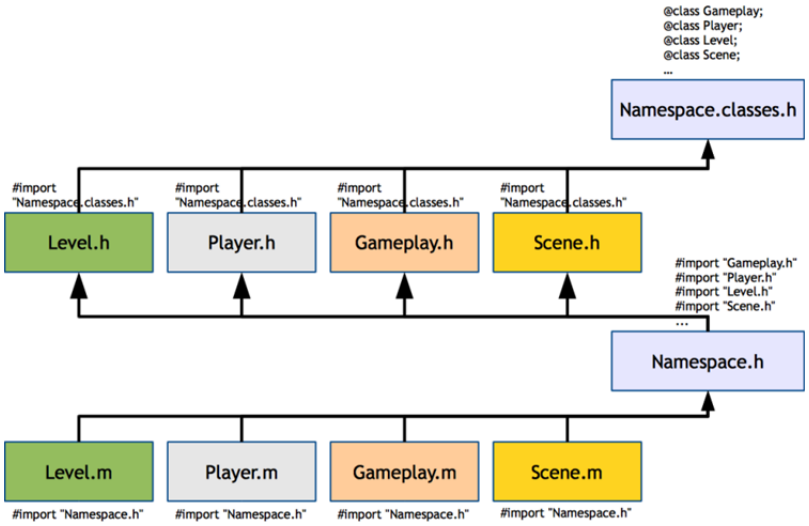


Ko bomo kasneje spoznali še protokole (isto kot interface v C# ali Javi), velja za njih isto, s tem da uporabimo direktivo @protocol.

Na ta način lahko sestavimo delček naše arhitekture, v kateri se že vidi naraščajoča kompleksnost vseh potrebnih importov in direktiv @class.



Da bo delo tudi v objective-cju čimbolj podobno preprostosti uporabe imenskih prostorov, si lahko vse potrebne importe združimo v zaglavno datoteko imenskega prostora (Namespace.h). Podobno storimo tudi z ukazi @class v ločeni datoteki (Namespace.classes.h). Na ta način zdaj ob dodajanju novega razreda v projekt zgolj dodamo ustrezni vrstici v obe zaglavni datoteki imenskega prostora, medtem ko v datoteki .h in .m uvozimo potrebno zaglavno datoteko.



Na ta način je organizirana tudi sama knjižnica XNI in odraža siceršnjo strukturo c# imenskih prostorov v XNAju. Kjer bi v c#u imeli

```
using Microsoft.Xna.Framework;
```

imamo v objective-cju

```
#import "Retronator.Xni.Framework.h"
```

Izjeme

Če uporabljamo uvoz datoteke "Namespace.classes.h" razredov naše igre, je torej v tej zaglavni datoteki povedano zgolj, kateri simboli (imena) predstavljajo razrede in kateri protokole. Če razred, ki ga opisujemo v tej zaglavni datoteki, deduje iz nekega drugega razreda ali implementira protokole, to ni dovolj. V tem primeru moramo izrecno vključiti tiste zaglavne datoteke razredov in protokolov, na katere se sklicujemo neposredno v vrstici @interface.

```
#import "Namespace.classes.h"
```

```
#import "SceneObject.h"
#import "IPosition.h"
#import "IRotation.h"

@interface Car : SceneObject <IPosition, IRotation> {
}

@end;
```

Druga izjema se nanaša na datoteke, v katerih deklariramo razne c-jevske strukture, enumeracije, define stavke in podobno. Ker te nove podatkovne tipe oziroma konstante največkrat rabimo v samih zaglavnih datotekah (za recimo definiranje spremenljivke nekega enum tipa), moramo to dotateko uvoziti tako v Namespace.h, kot tudi v Namespace.classes.h-ju.

Naloga: ločitev pogona

Ko dobimo občutek, kateri razredi v projektu so spisani splošno in bi jih lahko uporabili še v kakšnem drugi igri, jih lahko ločimo v svoj projekt tipa Static Library. Še lažje je, če imamo statično knjižnico že od samega začetka in ob samem ustvarjanju nov razred umestimo v primeren projekt (v igro ali v pogon/knjižnico pod njo).

Splača se zopet vzpostaviti svoj imenski prostor (ali več le-teh), s katerim na enostaven način uvozimo vse (ali pa samo smiselno združene) razrede pogona.

Ustvarjen Xcode-projekt zdaj preprosto povežemo v našo igro po istih navodilih, kot so opisana pri delu z XNI v obliki izvorne kode. XNI je namreč prav tako statična knjižnica, kot bo vaš pogon, torej vsa navodila za povezovanje z XNI v obliki Xcode-projekta veljajo za povezavo vaše igre z ločenim pogonom v svojem Xcode-projektu. Opozorimo, da moramo zaglavnim datotekam, katere naj se ob prevajanju prekopirajo v usr/local/include direktorij, z desnim klikom in izbiro SetRole nastaviti vrednost na Public.

Pri uporabi razredov iz knjižnice, ki je v celoti en nivo nižje pod našo igro (razredi igre se sklicujejo na razrede pogona, obratno pa ni nikoli res – pogon se ne zaveda igre nad njim), nam zato ni več potrebno ločevati med vključevanjem *@class* in *@protocol* deklaracij v zaglavnih datotekah in *#importi* v .m datotekah. Ker ni možno, da bi prišlo do navzkrižnega povezovanja, razrede pogona preprosto vključimo samo z uvozom glavne datoteke imenskega prostora (tiste z stavki *#import*) v sami zaglavni datoteki. Če malo pogledamo nazaj, vidimo, da smo na tak način delali že z XNIjem.

Izris

Kako na zaslon prikažemo sliko?

Za izris slike na zaslon potrebujemo dvoje: samo sliko, ki naj se izriše, shranjeno v objektu tipa *Texture2D,* in *SpriteBatch*, razred, ki zna z uporabo *GraphicsDevice* vmesnika izrisati teksturo na poljubno mesto na zaslonu.

Oboje običajno ustvarimo v metodi *loadContent* igre ali izrisljive komponente.

```
SpriteBatch *spriteBatch;
Texture2D *texture;

- (void) loadContent {
        spriteBatch = [[SpriteBatch alloc] initWithGraphicsDevice:self.graphicsDevice];
        texture = [self.content load:@"imageFilename"]; // Use self.game.content if you are in a
DrawableGameComponent
}
```

Pri tem smo v projekt seveda morali dodati datoteko imageFilename.png (podprti so tudi drugi tipi, recimo jpg, gif, bmp). Opozorimo, da pri nalaganju končnice ne pišemo, saj *ContentManager* pričakuje ime vsebine, to pa je privzeto enako imenu datoteke, iz katere se ta vsebina naloži v cevovod vsebine.

Zdaj lahko v klicu draw igre ali komponente teksturo izrišemo. *SpriteBatchu* moramo pri tem ustrezno povedati, naj začne sprejemati ukaze (*begin*) in naj jih na koncu tudi izvede (*end*).

```
- (void) drawWithGameTime:(GameTime *)gameTime {
        [self.graphicsDevice clearWithColor:[Color white]];

        [spriteBatch begin];

        [spriteBatch draw:texture to:[Vector2 vectorWithX:50 y:100] tintWithColor:[Color white]];

        [spriteBatch end];
}
```

Igra bo najprej zaslon pobrisala na belo barvo, saj moramo v vsakem ciklu zanke igre celotno sliko izrisati od začetka. *SpriteBatch* bo nato od koordinate (50, 100) pikslov navzdol in desno izrisal sliko, shranjeno v teksturi texture.

Parametri draw klica

*SpriteBatch*ov klic draw obstaja v 7 različicah, ki bolj ali manj natančno določajo, kako naj se izriše slika. V vsakem primeru na začetku povemo, katera tekstura se sploh izrisuje.

Pozicija in velikost

Naslednja stvar, ki jo lahko nastavimo, je, kam naj se izriše tekstura. Če uporabimo klic, ki sprejme parameter *toRectangle*, z objektom *Rectangle* točno določimo pravokotni del ekrana, katerega naj zapolni tekstura.

Po drugi strani lahko s parametrom *to*, ki sprejme objekt tipa *Vector2*, povemo samo točko, kam naj se izriše izhodišče slike. Izhodišče je običajno zgornja leva točka na teksturi s koordinato (0,0). Če želimo, da s pozicijo sovpada kakšna druga točka na teksturi, lahko izhodišče spremenimo s parametrom *origin*.

V primeru podajanja vektorja *to*, bo slika na ekranu zasedla točno toliko točk, kot je v izhodiščni datoteki. 128 pikslov široka tekstura, bo tudi na zaslonu merila 128 pikslov. Če želimo na to vplivati, moramo spremeniti parameter *scale*, ki sprejme *Vector2*. Z *x*-vrednostjo vektorja se bo pomnožila ciljna širina, z *y* pa višina izrisane slike. Če želimo povečevati ali zmanjševati proporcionalno, lahko namesto *scale* uporabimo parameter *scaleUniform*.

Izvirni del tekstura

V privzetem načinu se na zaslon izriše celotna tekstura. Če bi radi, da se izriše samo del le-te, lahko s parametrom *fromRectangle* povemo kateri del je to. Če v *fromRectangle* pošljemo vrednost *nil* se izriše celotna tekstura, sicer pa se za vse ostale operacije obravnava, kot bi bila že izvorna tekstura v osnovi manjša.

Možno je uporabiti širino in višino, ki presega meje izvirne tekstura. V tem primeru se bo, odvisno od nastavitev objekta S*amplerState* (o njem kmalu), tekstura večkrat ponovila.

Prebarvanje

Podatki o barvi tekstura se pri izrisu pomnoži z vrednostjo parametra *tintWithColor*. Običajno ga nastavimo na vrednosti [Color white], saj ima ta RGB vrednost (1,1,1), torej se zaradi množenja z 1 barva ne spremeni. Druge kombinacije ustrezno potemnijo izvorno teksturo z izbrano barvo.

Rotacija in zrcaljenje

Dodatno je možno izrisano sliko vrteti in sicer okoli izhodišča, ki smo ga določili s parametrom *origin*. Kot, za katerega se slika obrne, določa parameter *rotation*.

Na koncu lahko zahtevamo še uporabo zrcaljenja. V parameter *effects* običajno pošljemo konstanto *SpriteEffectsNone*, če pa želimo sliko zrcaliti, vrednost nadomestimo s *SpriteEffectsFlipHorizontally* ali *SpriteEffectsFlipVertically*.

Globina

Zadnji parameter je *layerDepth*. Ta v navezi s sortiranjem slik opisanim v naslednjem delu, skrbi za to, katera slika se pojavi pred katero. Vrednost 0 pomeni povsem spredaj, 1 povsem zadaj.

Parametri begin klica

Tudi s spreminjanjem klica *begin* lahko vplivamo na končni izris, tokrat na nivoji vseh slik, ki jih izrišemo v sledečem begin-end bloku *SpriteBatcha*. Vsem parametrom, razen vrednosti *SpriteSortMode*, lahko pošljemo konstanto nil, kar pomeni, naj *SpriteBatch* uporabi privzeto vrednost.

Sortiranje

Osnovna naloga *SpriteBatcha* je, da združi veliko zaporednih klicev draw v čim manj dejanskih klicev izrisa trikotnikov na *GraphicsDevicu*. To je običajno delovanje, ki ga dosežemo, če v klic *beginWithSortMode* pošljemo vrednosti *SpriteSortModeDeffered*.

Če želimo izključiti združevanje klicev, lahko uporabimo *SpriteSortModeImmediate*. V tem primeru se bodo vse ustrezne nastavitve na *GraphicsDevicu* zgodile že ob klicu begin, nakar lahko mi pred vsakim klicem draw ustrezno spreminjamo grafične nastavitve, za dosego naprednih grafičnih efektov. Za običajne stvari tega načina ne potrebujemo.

Naslednji način delovanja je *SpriteSortModeTexture*. Če pri klicu draw zamenjamo teksturo, s katero rišemo, bo SpriteBatch moral pri vsaki menjavi zaključiti trenutni niz poslanih slik. Če se kasneje ista tekstura še večkrat ponovi, bi lahko dosegli pohitritev, če bi vse izrise z eno teksturo združili v en sam klic. Ravno to pohitritev dosežemo z načinom *SpriteSortModeTexture*.

Zadnja dva načina sta *SpriteSortModeFrontToBack* in *SpriteSortModeBackToFront*. Pri njih namesto po teksturi, slike pred izrisom sortiramo po naraščajoči ali padajoči vrednosti *layerDepth*. Običajno si želimo, da sprednji predmeti prekrivajo oddaljene, zato jih moramo risati od zadaj naprej. Je pa možno, v kombinaciji z uporabo depth bufferja (opisan spodaj), prekrivanje doseči strojno. V tem primeru je boljše sortirati od spredaj nazaj, saj bodo slike hitro začele prekrivati celoten ekran, nakar se tistim v ozadju sploh ne bo potrebno izrisati.

Grafične nastavitve

Naslednji štirje parametri kontrolirajo stanje objekta *GraphicsDevice* in s tem pomemebno vplivajo na sam izris.

Glavni parameter za videz končnega izrisa je *blendState*. Opisuje, kakšne nastavitve za izračun barv pikslov naj uporablja *GraphicsDevice* pri izrisu. V razredu *BlendState* obstajajo že pripravljene konstante z najbolj običajnimi vrednostmi parametrov. Privzeto se uporablja vrednost *[BlendState alphaBlend]*, ki pri izračunu barve pikslov upošteva vrednosti kanala alpha, v katerem je shranjen podatek o prosojnosti piksla. Nastprotje temu načinu je *[BlendState opaque]*, pri čemer se kanal alpha ne uporabi. Tretji uporaben način je *[BlendState additive],* pri katerem se vrednost barve pikslov samo prišteva že trenutni obarvanosti. Ta način je precej uporaben za izris posebnih učinkov, kot so iskre, eksplozije, bleščanje sonca in podobno. Če poznate Photoshop je to ustreznica načinu "screen" pri nastavitvah sloja.

S parametrom *samplerState* izbiramo, kakšen filtrirni algoritem se uporablja pri pomanjševanju in povečevanju slike ter kakšne vrednosti naj imajo piksli, če smo *sourceRectangle* povečali izven izvirne velikosti teksture. Običajne nastavitve za filtriranje so anisotropic, linear in point, za zunajležeče piksle pa wrap in clamp. Pri načinu clamp, se bo izven teksture ponavljala barva robnih pikslov, z

wrap pa dosežemo zgoraj omenjeno ponavljanje teksture. Predpripravljene vrednosti so kombinacije teh dveh izbir, recimo privzeti *[SamplerState linearClamp]* ali na primer *[SamplerState anisotropicWrap]*, *[SampleState pointClamp]* ...

Parameter *depthStencilState* pove, ali naj se uporablja strojno globinsko prekrivanje izrisanih trikotnikov. *SpriteBatch* običajno deluje brez tega (vrednost *[DepthStencilState none]*), vključimo ga z *[DepthStencilState defaultDepth]*. Za določene efekte je uporaben tudi način, kjer samo upoštevamo do zdaj že izrisane trikotnike, novi pa ne spreminjajo globinskega medpomnilnika. To je mogoče z vrednostjo *[DepthStencilState depthRead]*.

Najosnovnejša naloga zadnjega parametra stanja *rasterizerState* je, da kontrolira ali se prikazujejo samo pozitivno ali samo negativno orientarirani trikotniki ali vsi. Običajen način je *[RasterizerState cullCounterClockwise]*, možni sta še konstanti *cullClockwise* in *cullNone*.

Uporaba lastnega senčilnika

Če potrebujemo še natančnejši nadzor nad izrisom, lahko zamenjamo privzeti senčilnik z lastnim, tako da pošljemo svojo instanco razreda BasicEffect s poljubnimi nastavitvami. Tako lahko naposredno spreminjamo projekcijsko matriko in matriko pogleda, če želimo npr. recimo izrisane slike vkomponirati v širšo sceno, običajno kar v 3D prostoru.

Za izhodišče lahko vzamete nastavitve BasicEffecta, ki ga uporablja SpriteBatch sicer.

```
basicEffect = [[BasicEffect alloc] initWithGraphicsDevice:theGraphicsDevice];
basicEffect.projection =
        [Matrix createOrthographicOffCenterWithLeft:0 right:self.graphicsDevice.viewport.width
         bottom:self.graphicsDevice.viewport.height top:0 zNearPlane:0 zFarPlane:-1];
basicEffect.textureEnabled = YES;
basicEffect.vertexColorEnabled = YES;
```

Transformacija vseh koordinat

Če želimo vse koordinate izrisa zgolj premakniti za nek faktor oziroma nad njimi izvesti kakršnokoli transformacijo, lahko z zadnjim parametrom transformMatrix storimo ravno to. Na ta način lahko zelo enostavno simuliramo premikanje kamere po sceni z ustvarjenjem ustrezne translacijske matrike na podlagi ciljne pozicije kamere.

Izdelava osnovne arhitekture za izris

Najosnovnejši deli arhitekture, ki jih potrebujemo za izris so: razred Scene, v katerega bomo združili vse objekte v igrinem svetu; komponenta Level, ki bo sceno ustvarila; ter izrisljiva komponenta Renderer, ki bo izdelano sceno sprejela in izrisala vse objekte na njej.

Scena

Za sceno lahko v najpreprostejši obliki vzamemo kar spremenljiv seznam objektov, NSMutableArray. Da pa bomo lahko s časom dodali naprednejše zmožnosti na sceno, uporabo seznama skrijemo za svoj razred Scene. Tako v osnovni verziji scena zgolj predaja ukaze notranjemu seznamu, čemur se reče tudi vzorec delegiranja (delegation pattern). Z uporabo protokola NSFastEnumeration omogočimo, da se bomo lahko čez sceno sprehajali z zanko "for each".

```
@interface Scene : NSObject <NSFastEnumeration> {
	NSMutableArray *items;
}

- (void) addItem:(id)item;
- (void) removeItem:(id)item;
- (void) clear;

@end
@implementation Scene

- (id) init
{
  self = [super init];
  if (self != nil) {
  items = [[NSMutableArray alloc] init];

  }
  return self;
}

- (void) addItem:(id)item {
  [items addObject:item];
}

- (void) removeItem:(id)item {
  [items removeObject:item];
}

- (void) clear {
  [items removeAllObjects];
}

- (NSUInteger) countByEnumeratingWithState:(NSFastEnumerationState *)state
	objects:(id *)stackbuf count:(NSUInteger)len {
	return [items countByEnumeratingWithState:state objects:stackbuf count:len];
}

- (void) dealloc
{
	[items release];
	[super dealloc];
}

@end
```

Stopnja

Komponenta Level kot rečeno poskrbi za ustvarjenje objekta Scene in ga napolni z ustreznimi objekti, ki naj bi bili na sceni (objekte nastavimo na izhodiščne položaje).

```
@interface Level : GameComponent {
	Scene *scene;
}
```

```
@property (nonatomic, readonly) Scene scene;

- (void) reset;

@end
@implementation Level

- (id) initWithGame:(Game *)theGame
{
        self = [super initWithGame:theGame];
        if (self != nil) {
           scene = [[Scene alloc] init];
        }
        return self;
}

@synthesize scene;

- (void) initialize {
        [self reset];
}

- (void) reset {
        // Remove everything from the scene.
        [scene clear];


        // Create an item, initialize its position and add it to the scene.
        Car *playerCar = [[[Car alloc] init] autorelease];
        playerCar.position = [Vector2 vectorWithX:50 y:100];
        [scene addItem:playerCar];


        // Repeat for all items
}

- (void) dealloc
{
        [scene release];
        [super dealloc];
}

@end
```

Objekti scene

Kot vidimo zgoraj, moramo imeti pripravljene tudi razrede za objekte na sceni. To so glavni akterji v naši igri: pacmani, duhci, bombermani, bombe, ovire, avtomobili in tudi stavbe, stene, stebri in podobno.

Da bodo lahko posamezni deli pogona obravnavali zelo veliko število različnih razredov, vstopijo v igro protokoli. Lastnosti objektov na sceni, kot so na primer pozicija, masa ali življensko energija, bomo namreč sestavili z uporabo večih protokolov.

Če ima objekt lastnost pozicijo, ga bo lahko igralec glede na vhodne naprave premikal po sceni, na drugi strani pa bo grafični pogon iz pozicije razbral, kam naj sliko za njegov tip objekta izriše.

Na ta način se nam v večini primerov ni potrebno ukvarjati s specifičnimi razredi, recimo avtomobili in stavbami, temveč fizikalni pogon zgolj obdela vse predmete, ki imajo pozicijo in hitrost. Grafični

pogon po drugi strani vzame tudi informacijo o tipu predmeta, na podlagi česar izbere katero sliko mora izrisati in jo združi s pozicijo, da ve kam ga izrisati.

Protokol za pozicijo je zelo preprost:

```
@protocol IPosition <NSObject>

@property (nonatomic, retain) Vector2 *position;

@end
```

Zdaj lahko izdelamo tudi zgoraj uporabljeni primer razreda Car:

```
@interface Car : NSObject <IPosition> {
	Vector2 *position;
}

@end
@implementation Car

- (id) init
{
	self = [super init];
	if (self != nil) {
	    position = [[Vector2 alloc] init];
	}
	return self;
}

@synthesize position;

- (void) dealloc
{
	[position release];
	[super dealloc];
}

@end
```

Izris

Na koncu postavljanja arhitekture je končno čas za izris. Renderer tipa DrawableGameComponent bo v inicializaciji sprejel sceno, ki jo bo v vsakem ciklu igrine zanke izrisal s pomočjo SpriteBatcha.

```
@interface Renderer : DrawableGameComponent {
	SpriteBatch *spriteBatch;

	Texture2D *carTexture;
	Texture2D *buildingTexture;

	Scene *scene;
}

- (id) initWithGame:(Game*)theGame scene:(Scene*)theScene;

@end
@implementation Renderer

- (id) initWithGame:(Game *)theGame scene:(Scene*)theScene {
	if (self = [super initWithGame:theGame]) {
```

```
            scene = theScene;
        }
        return self;
}

- (void) loadContent {
        // Create the sprite batch.
        spriteBatch = [[SpriteBatch alloc] initWithGraphicsDevice:self.graphicsDevice];

        // Load all textures.
        carTexture = [self.game.content load:@"car"];
        buildingTexture = [self.game.content load:@"building"];
}

- (void) drawWithGameTime:(GameTime *)gameTime {
        [self.graphicsDevice clearWithColor:[Color white]];


        [spriteBatch begin];

        // Draw scene.
        for (id item in scene) {
            // Check if an item has position.

            id <IPosition> itemWithPosition = [item conformsToProtocol:@protocol(IPosition)] ? item : nil;

            // Check which texture to use.

            Texture *texture = nil;

            if ([item isKindOfClass:[Car class]]) {
                texture = carTexture;
            } else if ([item isKindOfClass:[Building class]]) {
                texture = buildingTexture;
            }

            // If we have both an itemWithPosition and a valid texture we know how to draw this item.

            if (itemWithPosition && texture) {
                [spriteBatch draw:texture to:itemWithPosition.position tintWithColor:[Color white]];
            }
        }



        [spriteBatch end];
}

- (void) unloadContent {
        // Release all graphics objects.
        [spriteBatch release];
        [carTexture release];
        [buildingTexture release];
}
```

Uporaba slikovnega atlasa

Za konec lahko nadgradimo izris z uporabo slikovnega atlasa. Gre za teksturo, na kateri imamo združenih več posameznih slik, nakar pri izrisu z uporabo parametra fromRectangle SpriteBatchu povemo, kateri del teksture želimo tokrat izrisati. Ker moramo zdaj za opis slike poznati tako teksturo kot rectangle, si pripravimo nov podatkovni objekt Sprite. Poleg tega vanj shranimo tudi

vektor origin, ki bo SpriteBatchu povedal, katera točka na izvirni sliki naj sovpada s pozicijo, na katero ukažemo izris predmeta scene.

```
@interface Sprite : NSObject {
	Texture2D *texture;
	Rectangle *sourceRectangle;
	Vector2 *origin;
}

@property (nonatomic, retain) Texture2D *texture;
@property (nonatomic, retain) Rectangle *sourceRectangle;
@property (nonatomic, retain) Vector2 *origin;

@end
@implementation Sprite

@synthesize texture, sourceRectangle, origin;

- (void) dealloc
{
	[texture release];
	[sourceRectangle release];
	[origin release];
	[super dealloc];
}

@end
```

Zdaj lahko popravimo zgornjo kodo in namesto hranjenja tekstur za vsako vrsto objekta pripravimo objekt tipa Sprite.

```
// In interface:
Sprite *carSprite;
Sprite *buildingSprite;


// In loadContent:
spriteAtlas = [[self.game.content load:@"spriteAtlas"] autorelease];

carSprite = [[Sprite alloc] init];
carSprite.texture = spriteAtlas;
carSprite.sourceRectangle = [Rectangle rectangleWithX:0 y:0 width:20 height:40];
carSprite.origin = [Vector2 vectorWithX:10 y:20];

buildingSprite = [[Sprite alloc] init];
buildingSprite.texture = spriteAtlas;
buildingSprite.sourceRectangle = [Rectangle rectangleWithX:30 y:0 width:100 height:100];
buildingSprite.origin = [Vector2 vectorWithX:50 y:50];


// In draw:

Sprite *sprite = nil;

if ([item isKindOfClass:[Car class]]) {
	sprite = carSprite;
} else if ([item isKindOfClass:[Building class]]) {
	sprite = buildingSprite;
}


if (itemWithPosition && sprite) {
	[spriteBatch draw:sprite.texture to:itemWithPosition.position
	 fromRectangle:sprite.sourceRectangle tintWithColor:[Color white]
```

```
            rotation:0 origin:sprite.origin scaleUniform:1 effects:SpriteEffectsNone layerDepth:0];
    }
```

Naloga: statične slike

Da smo ustrezno postavili osnovno arhitekturo projekta lahko preverimo, ko na sceno dodamo nekaj objektov in jih z ustrezno spisanim Renderer-jem izrišemo.

Za izris statičnih slik ni potrebno veliko dela. Vzamemo SpriteBatch in mu naložimo izris vseh objektov na sceni. Podatke o teksturi in delu teksture, ki prikazuje določen objekt, si pred tem shranimo v objekte tipa Sprite. Kam na ekran narišemo določen objekt določa vektor pozicije, lastnost *origin* na Sprite-u pa določa, katera točka na sliki ustreza podani poziciji.

Ko je z uporabo podane arhitekture uspešno izrisanih nekaj testno postavljenih objektov je naloga opravljena.

Naloga: animirane slike

Risanje statičnih slik lahko nadgradimo na animirane slike, pri katerih s časom spreminjamo, katero sliko izrisujemo.

Potrebno je izdelati ustrezno podatkovno strukturo, ki bo hranila zaporedje slik in časov, ob katerih se pojavijo. Na drugi strani si je potrebno shraniti referenčno točko v času, oziroma izhodišče, glede na katerega se animacija izvaja. Včasih lahko uporabimo kar spremenljivko *gameTime.totalGameTime*, drugič si moramo to vrednost shraniti kot izhodišče za animacijo točno določenega objekta na sceni. Lahko pa si tudi pomagamo z vrednostjo, koliko časa je že preteklo od začetka animacije, ki mu v vsakem obhodu zanke prištejemo *gameTime.elapsedGameTime.*