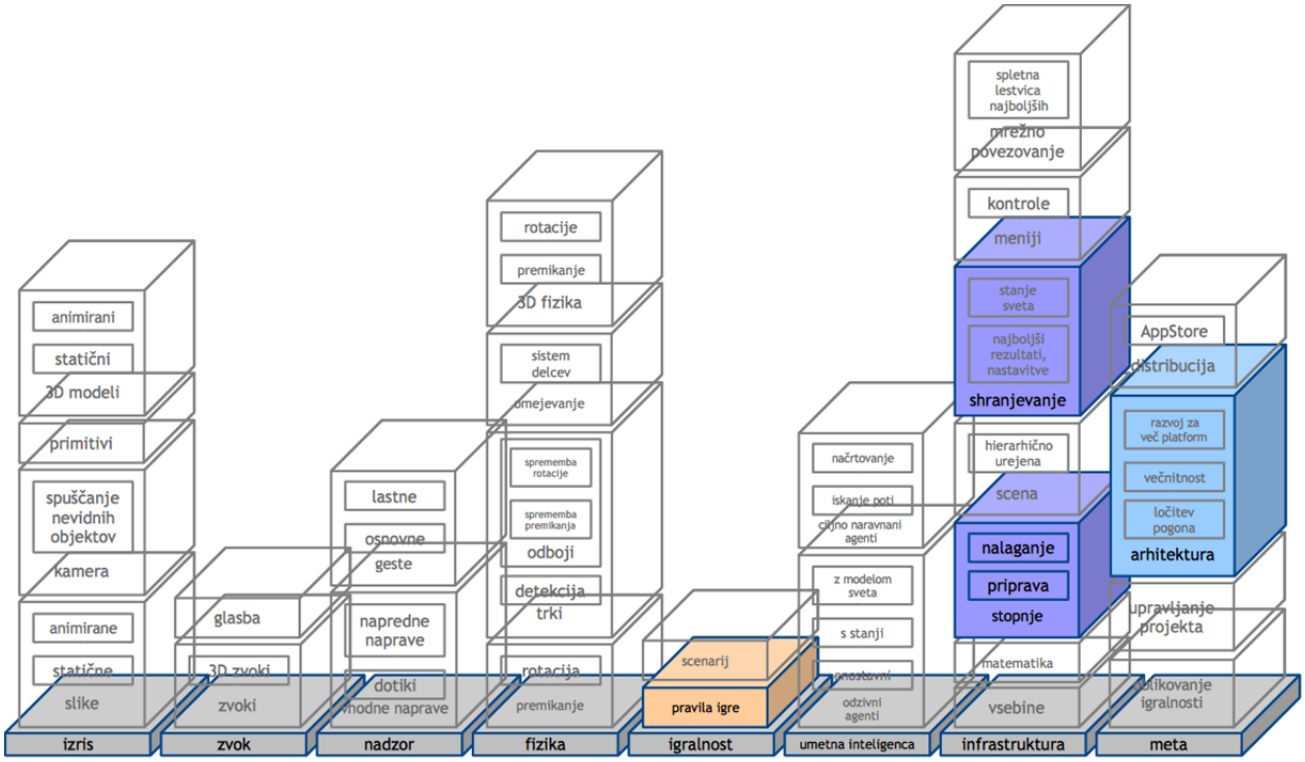
4. sklop


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
omejevanje
sprememba rotacije
sprememba premikanja
odboji
detekcija trki
rotacija
premikanje
fizika
scenarij
pravila igre
igralnost
načrtovanje
iskanje poti
cilj no 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

Igralnost

Kako iz tehnološke igrače ustvarimo igro?

Z ustvarjeno interakcijo in izdelanim fizikalnim pogonom je končno napočil čas za igralnost. Vsa pravila, ki smo jih v prvem tednu zapisali v koncept igre, moramo postopoma vključiti v projekt.

Priprava stopnje

Vzemimo za primer igro razbijanja zidu (Wall, Breakout) in pojdimo lepo po vrsti. Na sceni imamo štiri vrste objektov:

- lopar,
- žogico,
- opeke in
- stene.

Za vsakega ustvarimo svoj razred s primernimi podatki (pozicija, hitrost, geometrija za odboj). Z dosedanjim znanjem lahko naredimo premikanje loparja s prstom in fizikalno odbijanje žoge od sten, opek in loparja.

Ker bo po novem možno igro izgubiti in začeti od začetka, bo za osnovno postavitev vseh objektov na sceno odgovoren Level v novi metodi reset:

```
- (void) resetLevelWithBallSpeed:(float)speed {
	// Remove everything from the scene.
	[scene clear];
	bricksCount = 0;

	// Add ball and paddle.
	[scene addItem:ball];
	[scene addItem:paddle];

	// Add level limits.
	[scene addItem:[[[LevelLimit alloc] initWithLimit:
			[AAHalfPlane aaHalfPlaneWithDirection:AxisDirectionPositiveX distance:0]] autorelease]];
	[scene addItem:[[[LevelLimit alloc] initWithLimit:
			[AAHalfPlane aaHalfPlaneWithDirection:AxisDirectionNegativeX distance:-1000]] autorelease]];
	[scene addItem:[[[LevelLimit alloc] initWithLimit:
			[AAHalfPlane aaHalfPlaneWithDirection:AxisDirectionPositiveY distance:0]] autorelease]];

	// Add bricks.
	for (int i = 0; i < BrickStyles; i++) {
		for (int x = i % 2 ? 0 : 25; x <= bounds.width; x+=50) {
			Brick *brick = [[[Brick alloc] init] autorelease];
			brick.style = i;
			brick.position.x = x;
			brick.position.y = 100 + i * 25;
			[scene addItem:brick];
			bricksCount++;
		}
	}

	// Reset ball
	[self resetBallWithSpeed:speed];
}

- (void) resetBallWithSpeed:(float)speed {
	ball.position.x = bounds.width / 2;
	ball.position.y = bounds.height / 2;
	ball.velocity.x = ([Random float] - 0.5f) * 10;
	ball.velocity.y = speed;
}
```

Resetiranje žoge smo ločili od glavne postavitve stopnje, saj mora biti ta operacija ločeno na voljo ob izgubi življenja (ne pa tudi igre). Ti dve metodi bo Gameplay klical po potrebi, ob začetku in koncu igre. Tudi sam Gameplay razred ima metodo reset, ki poleg resetiranja objektov v stopnji poskrbi še za razne števce življenj, točk in ostalih podatkov povezanih z igralnostjo:

```
- (void) reset {
	lives = [Constants getInstance].startLives;
	difficultyLevel = 0;
	[level resetLevelWithBallSpeed:[self calculateCrrentBallSpeed]];
}
```

Če zdaj poženemo igro, se bo žogica odbijala od opek, a jih ne bo uničila. Če bomo zgrešili odboj, bo letela v neskončnost navzdol. Potrebno je vpeljati pravila igre.

Naloga: pravila igre

Pravila igre v projekt vključimo na treh mestih: v samih objektih scene po potrebi izdelamo odzive na trk in lastno posodabljanje. V posodabljanje stopnje dodamo logiko, ki se tiče posamezne stopnje, v kateri se nahajamo. Povečini tu dodajamo in odstranjujemo objekte scene v skladu s scenarijem stopnje. Nazadnje v sami komponenti Gameplay pregledujemo glavna pravila igre, ki določajo pozitivne in negativne posledice za igralca. V skrajnem primeru igralec pride do konca igre in zmaga ali pa na drugi strani izgubi vsa življenja in izgubi igro.

Primer: izguba življenja

Primer izdelave pravila v komponenti Gameplay je pregledovanje pozicije žoge, če je ta padla pod rob ekrana. V tem primeru igralcu vzamemo življenje in resetiramo žogico, če je izgubil tudi vsa življenja pa celotno igro:

```
- (void) updateWithGameTime:(GameTime *)gameTime {

	// Check life lose condition.
	if (level.ball.position.y > level.bounds.height + 50) {
		lives--;
		[level resetBallWithSpeed:[self calculateCrrentBallSpeed]];

		// Check game lost condition.
		if (lives < 0) {
			[self reset];
		}
	}
}
```

Reakcija na trke

Pravila igre velikokrat zahtevajo, da se nekaj zgodi ob ali po trku. Pri igri razbijanja zidu se žogica od loparja ne odbije fizikalno, ampak je njen odboj odvisen od tega, na kateri del loparja je padla. Da bi lahko to storili, mora fizikalni pogon obvestiti objekt, da se je znašel v trku.

Da bo objekt lahko sprejemal sporočila o trkih pripravimo nov vmesnik ICustomCollider:

```
@protocol ICustomCollider <NSObject>

@optional
- (BOOL) collidingWithItem:(id)item;
- (void) collidedWithItem:(id)item;

@end
```

Direktivo *@optional* smo uporabili, da lahko objekt po potrebi pripravi samo eno od obeh metod. *CollidingWithItem* bomo iz fizikalnega pogona poklicali pred trkom, objekt pa nam mora vrniti *BOOL* vrednost, ali naj se trk tudi zares izvede. Na ta način lahko recimo naredimo, da se žogica ne odbije, temveč potuje skozi več opek (npr. breakthrough power-up).

CollidedWithItem se pokliče, ko je fizikalni pogon že izračunal sile in popravil hitrosti objektov. Vse skupaj moramo vključiti v naš fizikalni pogon v metodi collisionBetween:

```
+ (void) collisionBetween:(id)item1 and:(id)item2 usingAlgorithm:(Class)collisionAlgorithm {
	if ([collisionAlgorithm detectCollisionBetween:item1 and:item2]) {
		if ([Collision shouldResolveCollisionBetween:item1 and:item2]) {
			[collisionAlgorithm resolveCollisionBetween:item1 and:item2];
			[Collision reportCollisionBetween:item1 and:item2];
		}
	}
}

+ (BOOL) shouldResolveCollisionBetween:(id)item1 and:(id)item2 {
	id<ICustomCollider> customCollider1 = [item1 conformsToProtocol:@protocol(ICustomCollider)] ? item1 : nil;
	id<ICustomCollider> customCollider2 = [item2 conformsToProtocol:@protocol(ICustomCollider)] ? item2 : nil;

	BOOL result = YES;

	if (customCollider1 && [customCollider1 respondsToSelector:@selector(collidingWithItem:)]) {
		result &= [customCollider1 collidingWithItem:item2];
	}

	if (customCollider2 && [customCollider2 respondsToSelector:@selector(collidingWithItem:)]) {
		result &= [customCollider2 collidingWithItem:item1];
	}

	return result;
}

+ (void) reportCollisionBetween:(id)item1 and:(id)item2 {
	id<ICustomCollider> customCollider1 = [item1 conformsToProtocol:@protocol(ICustomCollider)] ? item1 : nil;
	id<ICustomCollider> customCollider2 = [item2 conformsToProtocol:@protocol(ICustomCollider)] ? item2 : nil;

	if (customCollider1 && [customCollider1 respondsToSelector:@selector(collidedWithItem:)]) {
		[customCollider1 collidedWithItem:item2];
	}

	if (customCollider2 && [customCollider2 respondsToSelector:@selector(collidedWithItem:)]) {
		[customCollider2 collidedWithItem:item1];
	}
}
```

Sedaj lahko v igri razbijanja opek loparju dodamo vmesnik ICustomCollider in v reakciji na trk popravimo smer odbiti žogici:

```
- (void) collidedWithItem:(id)item {

	// Calculate horizontal velocity depending on where the paddle was hit.
	Ball *ball = [item isKindOfClass:[Ball class]] ? item : nil;

	if (ball) {
		float speed = [ball.velocity length];

		// Calculate where on the paddle we were hit, from -1 to 1.
		float hitPosition = (ball.position.x - position.x) / width * 2;

		// Calculate angle.
		float angle = hitPosition * [Constants getInstance].maximumBallAngle;

		// Rebound ball in desired direction.
		ball.velocity.x = sinf(angle);
		ball.velocity.y = -cosf(angle);
		[ball.velocity multiplyBy:speed];

		// Make sure the vertical velocity is big enough after collision.
		float minY = [Constants getInstance].minimumBallVerticalVelocity;
		if (fabsf(ball.velocity.y) < minY) {
			ball.velocity.y = ball.velocity.y < 0 ? -minY : minY;
		}
	}
}
```

Spreminjanje scene

Zelo pogosto pravila igre zahtevajo dodajanje in odstranjevanje objektov s scene, recimo pojavitev metka ob streljanju ter odstranitev ob trku. V primeru razbijanja zidu se mora opeka odstraniti, ko jo zadanemo z žogo. Opeka preko vmesnika ICustomCollider zdaj lahko izve, da je bila zadeta, nima pa dostopa do objekta scene v levelu, da bi se lahko odstranila. Naivna rešitev je, da opeka delo prepusti stopnji. Opeki zgolj dodamo spremenljivko destroyed, ki jo ob trku nastavimo na YES:

```
- (void) collidedWithItem:(id)item {

	destroyed = YES;

	// Make sure the vertical velocity is big enough after collision,
	// so we don't have to endlesly wait for the ball to come down.
	Ball *ball = [item isKindOfClass:[Ball class]] ? item : nil;
	if (ball) {
		float minY = [Constants getInstance].minimumBallVerticalVelocity;
		if (fabsf(ball.velocity.y) < minY) {
			ball.velocity.y = ball.velocity.y < 0 ? -minY : minY;
		}
	}
}
```

Level naj zdaj vseskozi gleda, ali je katera opeka razbira in jo odstrani:

```
- (void) updateWithGameTime:(GameTime *)gameTime {
	for (id item in scene) {
		Brick *brick = [item isKindOfClass:[Brick class]] ? item : nil;
		if (brick && brick.destroyed) {
			[scene removeItem:brick];
		}
	}
}
```

Pri tem naletimo na problem. Ker se z zanko for-in sprehajamo čez zbirko scene, ji ne smemo hkrati še dodajati ali odstranjevati elementov. Med prehodom si zato šele pripravimo seznam, katere objekte izbrisati ter jih odstranimo šele kasneje:

```
- (void) updateWithGameTime:(GameTime *)gameTime {
	NSMutableArray *removeBricks = [[NSMutableArray alloc] init];

	for (id item in scene) {
		Brick *brick = [item isKindOfClass:[Brick class]] ? item : nil;
		if (brick && brick.destroyed) {
			[removeBricks addObject:brick];
		}
	}

	for (Brick *brick in removeBricks) {
		[scene removeItem:brick];
		bricksCount--;
	}

	[removeBricks release];
}
```

Razbijanje opek zdaj sicer deluje, vendar je rešitev daleč od elegantne. Ne le, da za preprosto operacijo odstranjevanja rabimo deset vrstic, tudi koda je daleč proč od izvornega mesta, ki je v metodi collided opeke.

Lepši način je, da objektom omogočimo, da lahko neposredno uporabljajo sceno preko vmesnika ISceneUser:

```
@protocol ISceneUser <NSObject>

@property (nonatomic, retain) id<IScene> scene;

@optional
- (void) addedToScene:(id<IScene>)scene;
```

```
- (void) removedFromScene:(id<IScene>)scene;

@end
```

Objektu, ki uporablja ta vmesnik bomo ob dodajanju na sceno nastavili spremenljivko scene, da bo lahko preko nje kasneje izvajal operacije dodajanja in odstranjevanja. Opcijsko objektu tudi sporočimo, kdaj smo ga dodali ali odstranili s scene, kar je še en način, kako reagiramo na dogodke v igri. Metodi *addItem* in *removeItem* na sceni moramo zdaj razširiti na naslednji način:

```
- (void) addItem:(id)item {
	[items addObject:item];
	id<ISceneUser> sceneUser = [item conformsToProtocol:@protocol(ISceneUser)] ? item : nil;

	if (sceneUser) {
		sceneUser.scene = self;
		if ([sceneUser respondsToSelector:@selector(addedToScene:)]) {
			[sceneUser addedToScene:self];
		}
	}
}
```

Zdaj lahko opeka opremljena z vmesnikom *ISceneUser* odstrani samo sebe s scene ob trku:

```
- (BOOL) collidingWithItem:(id)item {
	[scene removeItem:self];
	return YES;
}
```

A ni vse tako enostavno. Tako kot smo prej naleteli na težavo pri odstranjevanju iz zbirke med sprehajanjem čez njo, tudi tokrat nismo odporni na ta problem. *CollidingWithItem* se namreč kliče iz fizikalnega pogona, ki med preverjanjem trkov ravno tako s zanko for-in prečesava isto zbirko.

Podobno kot smo prej ustvarili začasni seznam, katere opeke naj odstranimo, podobno bomo zdaj problem rešili znotraj same scene. Pripravili si bomo seznam ukazov, ki se bo med uporabo metod *addItem* in *removeItem* polnil, na koncu obhoda igrine zanke pa bomo vse ukaze izvedli. Da jo bomo enostavno vključili v cikel posodabljanja (update), tudi sceno nadgradimo v komponento (GameComponent):

```
/**
 A simple scene implementation that just uses an array as its backing.
 */
@interface SimpleScene : GameComponent <IScene> {
	// A list of items currently on the scene.
	NSMutableArray *items;

	// A list of adds and removes to be executed on the scene.
	NSMutableArray *actions;

	Event *itemAdded;
	Event *itemRemoved;
}

@end

@implementation SimpleScene

- (id) initWithGame:(Game *)theGame {
	self = [super initWithGame:theGame];
	if (self != nil) {
		items = [[NSMutableArray alloc] init];

		actions = [[NSMutableArray alloc] init];

		itemAdded = [[Event alloc] init];
		itemRemoved = [[Event alloc] init];
	}
	return self;
}
```

```
@synthesize itemAdded, itemRemoved;

- (int) updateOrder {return super.updateOrder;}
- (void) setUpdateOrder:(int)value {super.updateOrder = value;}
- (Event*) updateOrderChanged {return super.updateOrderChanged;}

- (BOOL) enabled {return super.enabled;}
- (void) setEnabled:(BOOL)value {super.enabled = value;}
- (Event*) enabledChanged {return super.enabledChanged;}


- (void) addItem:(id)item {
	[actions addObject:[SceneAction actionWithOperation:SceneOperationAdd item:item]];
}

- (void) removeItem:(id)item {
	[actions addObject:[SceneAction actionWithOperation:SceneOperationRemove item:item]];
}

- (void) clear {
	for (id item in items) {
		[self removeItem:item];
	}
}


- (void) updateWithGameTime:(GameTime *)gameTime {
	for (int i = 0; i < [actions count]; i++) {
		SceneAction *action = [actions objectAtIndex:i];

		// Retain the item so we guarantee it's alive during our operation.
		id item = [action.item retain];
		id<ISceneUser> sceneUser = [item conformsToProtocol:@protocol(ISceneUser)] ? item : nil;

		if (action.operation == SceneOperationAdd) {
			[items addObject:item];

			if (sceneUser) {
				sceneUser.scene = self;
				if ([sceneUser respondsToSelector:@selector(addedToScene:)]) {
					[sceneUser addedToScene:self];
				}
			}

			[itemAdded raiseWithSender:self eventArgs:[SceneEventArgs eventArgsWithItem:item]];
		} else {
			[items removeObject:item];

			if (sceneUser) {
				sceneUser.scene = nil;
				if ([sceneUser respondsToSelector:@selector(removedFromScene:)]) {
					[sceneUser removedFromScene:self];
				}
			}

			[itemRemoved raiseWithSender:self eventArgs:[SceneEventArgs eventArgsWithItem:item]];
		}

		// We've completed work with the item and can now release it.
		[item release];
	}

	[actions removeAllObjects];
}

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

- (void) dealloc
{
	[itemAdded release];
	[itemRemoved release];
```

```
        [actions release];
        [items release];
        [super dealloc];
}

@end
```

Naloga: scena

Za potrebe izdelave elementov igralnosti se pogosto ukvarjamo s tem, da moramo ob določenih dogodkih, recimo trkih ali ukazih, ki jih sproži igralec, spreminjati predmete na sceni. Pri tem gre tako za odstranjevanje in dodajanje novih predmetov, kot tudi za spreminjanje lastnosti obstoječih predmetov. Kakorkoli, za te potrebe mora naš razred za sceno podpirati sočasno sprehajanje čez vse objekte in dodajanje ter odstranjevanje njene vsebine.

Pristopov za dosego tega cilja je več, en od njih je prikazana uporaba seznama akcij, ki se naknadno izvedejo enkrat na prehod zanke igre. Uporabite lahko seveda tudi kakšen drug način. Naloga je opravljena, ko lahko z vašo arhitekturo dodajate in odstranjujete objekte na sceni med samim igranjem.

Naloga: stopnje

Tako kot smo naredili prvo stopnjo, lahko naredimo več različic razreda Level. Običajno z dedovanjem v vsaki konkretni stopnji spremenimo začetno postavitev elementov in ostala pravila, specifična za stopnjo.

Težji, a bolj prilagodljiv pristop je, da podatke o stopnji hranimo v zunanji datoteki, ki jo lahko enostavno spreminjamo. Inicializatorju stopnje v tem primeru podamo ime datoteke, nakar stopnja naloži ustrezno razporeditev elementov na sceni. Uporabimo lahko npr. preprosto tekstovno datoteko, XML, JSON.

Na začetku lahko v sami kodi nastavimo, katera stopnja naj bo aktivna, kasneje, ko bomo spoznali menije, pa naredimo pred igranjem vstopno točko z izbiro stopnje.