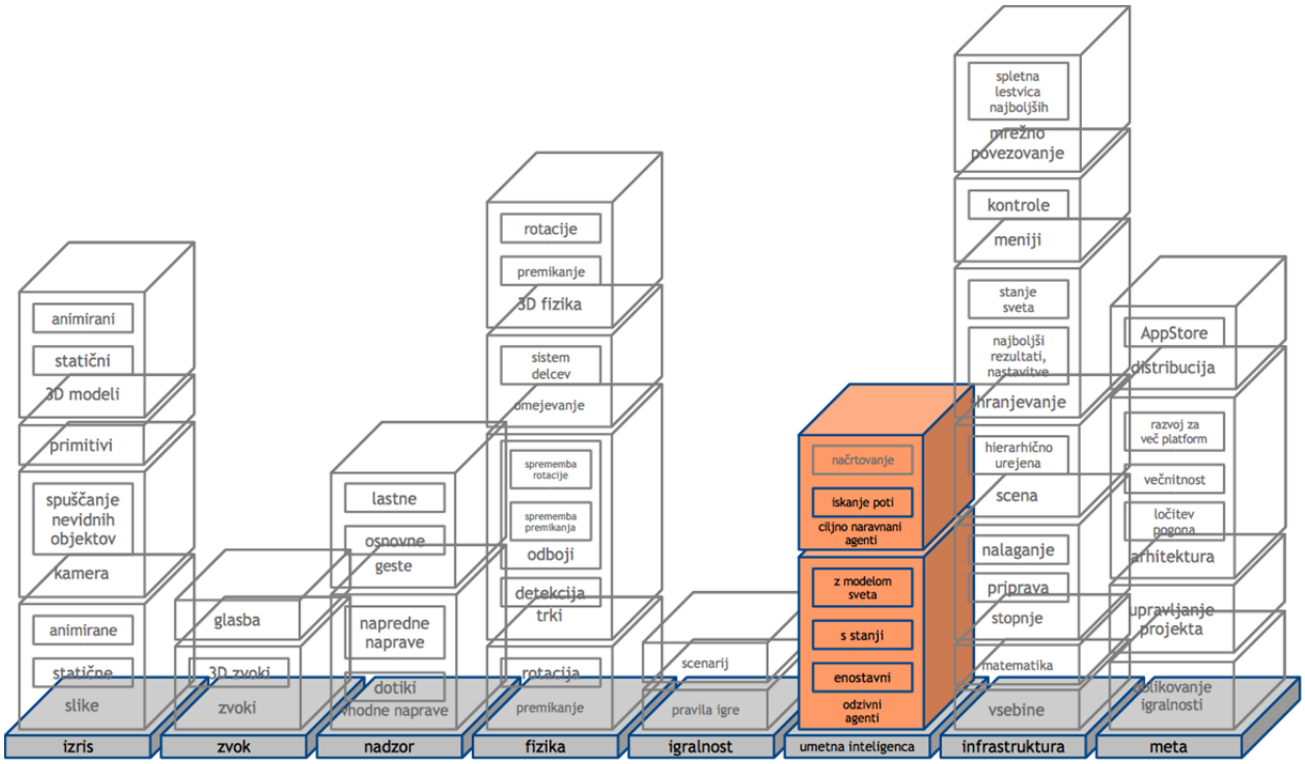
5. sklop


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
ciljno naravnani agenti
z modelom sveta
s stanji
enostavni
odzivni agenti
umetna inteligenca
spletna lestvica najboljših
prežno povezovanje
kontrole
meniji
stanje sveta
najboljši rezultati, nastavitve
hranjevanje
hierarhično urejena
scena
nalaganje
priprava
stopnje
matematika
vsebine
infrastruktura
AppStore
distribucija
razvoj za več platform
večnitnost
ločitev pogona
arhitektura
upravljanje projekta
oblikovanje igralnosti
meta

Umetna inteligenca

Kako lahko računalnik nadomesti človeškega nasprotnika ali vodi neigralčeve like?

Umetna inteligenca je eno od področij izdelovanja iger, ki mejijo na "črno magijo". Z drugimi besedami, podobno kot pri igralnosti je večina rešitev zelo specifičnih za samo igro. Določeni deli, kot npr. iskanje poti so vseeno zelo dobro raziskani in ponujajo splošne rešitve. Tukaj bomo pregledali predvsem arhitekturne rešitve vključitve inteligentnih agentov v igro.

Arhitektura umetno-inteligenčnega igralca

V poglavju o arhitekturi igre smo povedali, da v objektu Gameplay hranimo seznam igralcev. Gre za entitete zunaj sveta igre, ki s svojimi akcijami vplivajo na igro, ponavadi preko tega, da vodijo enega izmed likov znotraj sveta. Če vzamemo za primer igro zračnega hokeja (air hockey) igralec (abstraktna oseba zunaj igre) nadzoruje svoj plošček (objekt znotraj igrinega sveta, ki preko fizikalnega pogona odbija pak). Za samo igro je vseeno kako igralec spremeni lokacijo svojega ploščka. Player lahko to stori preko dotikov resničnega igralca ali po določenem algoritmu glede na pozicijo paka in nasprotnikovega ploščka izračuna, kam naj se premakne. Tako z dedovanjem ustvarimo hierarhijo abstraktnega igralca (*Player*) in dveh sinov: človeškega igralca (*HumanPlayer*), ki izvaja ukaze glede na vhodne naprave, ter umetno-inteligenčnega igralca (*AIPlayer*), kjer algoritem določi naslednjo akcijo. Oba to počneta v metodi update, saj dedujeta iz razreda *GameComponent*.

AIPlayer mora tako kot *HumanPlayer* imeti dostop do ploščka, da mu spreminja njegovo pozicijo. Dodatno potrebuje še dostop do stopnje, preko katere opazuje dogajanje. Resnični igralec namreč preko izrisane slike vidi dogajanje, računalniški pa bi si sliko lahko le bolj malo pomagal. Še več, pogosto je zgolj iz analiziranja objektov na sceni težko ugotoviti dobro strategijo za igranje. V primeru hokeja sceno sestavljata dva ploščka, pak in stene, ki omejujejo igralno polje. Da bi iz teh geometrijskih podatkov ugotovili, kje je gol nasprotnika, bi potrebovali občutno preveč programiranja. Veliko lažje je, če računalniškemu igralcu direktno povemo, kje so strateške točke na plošči (nasprotnikov gol za napad in lastni za obrambo).

Arhitektura neigralnih likov v igri

Podobno kot lahko zunanji igralec skrbi za premikanje in obnašanje lika v igri, lahko ta lik tudi sam direktno spreminja svoje delovanje. Podobno kot mora *AIPlayer* imeti dostop do stopnje ali vsaj scene, bo objekt na sceni običajno uporabil vmesnik *ISceneUser*, da dobi dostop do stanja igre. Poskrbeti moramo še, da bo dobil tudi priložnost procesirati te podatke.

Za objekte, ki morajo izvajati interno logiko pripravimo vmesnik *ICustomUpdate*:

```
@protocol ICustomUpdate <NSObject>

- (void) updateWithGameTime:(GameTime*)gameTime;

@end
```

V stopnji na vseh objektih na sceni s tem vmesnikom pokličemo update znotraj lastne metode update:

```
- (void) updateWithGameTime:(GameTime *)gameTime {
	// Update all items with custom update.
	for (id item in scene) {
		id<ICustomUpdate> updatable = [item conformsToProtocol:@protocol(ICustomUpdate)] ? item : nil;

		if (updatable) {
			[updatable updateWithGameTime:gameTime];
		}
	}
}
```

Kaj lik stori znotraj metode *update* je zelo odvisno od naših želja. Nasprotniki lahko sledijo igralčevemu liku, streljajo v njegovo smer ali pa so celo veliko bolj preprosti. Kakšen od nasprotnikov lahko ob trku spremeni svojo smer, kar storimo direktno v metodi *collided*. Drugi nasprotniki bodo

naprednejši in bodo poznali različne vrste obnašanja. Vse skupaj pokriva teorija umetno-inteligenčnih agentov, ki z naraščajočo kompleksnostjo omogoča vedno bolj inteligentno obnašanje.

Naloga: odzivni agenti

Inteligentne agente lahko v igro dodamo na dveh mestih. Če nadomestijo človeškega igralca, razred, ki predstavlja igralca (Player) z dedovanjem razdelimo na človeškega igralca (HumanPlayer), ki izvaja ukaze glede na vhodne naprave, ter umetno-inteligenčnega igralca (AIPlayer), kjer algoritem določi naslednjo akcijo.

Na drugi strani likom na sceni, ki se morajo inteligenčno obnašati, dodamo odločanje neposredno v njihov razred in metodo za odločanje kličemo podobno kot lastno posodabljanje. Pogosto lahko metoda updateWithGameTime skrbi kar za oboje skupaj.

Naloga je opravljena, ko določeni objekti na sceni reagirajo sami od sebe na okolico, tako v smislu neigralnih likov, kot tudi umetno-inteligenčnih nasprotnikov.

Naloga: odzivni agent s stanji

Veliko število agentov je možno spisati v obliki enostavnih odzivnih agentov, ki vsakič znova izvedejo popolnoma enak algoritem za odločanje. Odločitev je torej popolnoma neodvisna od prejšnjih odločitev ali stanja agenta.

Včasih za dosego želenega obnašanja to ni dovolj in moramo različna obnašanja razdeliti v stanja, ki jih agent izvaja čez serijo večih ciklov igrine zanke. Obnašanje agenta v tem primeru realiziramo s končnim avtomatom stanj.

Prvi korak je definiranje vseh stanj, katere običajno shranimo v konstante enum. V konstruktorju oziroma po potrebi v reset-metodi objekta nastavimo spremenljivko trenutnega stanja na začetno vrednost, torej stanje, v katerem se mora agent nahajati na začetku.

V metodi update nato glede na trenutno stanje izvedemo enega od različnih obnašanj, za kar običajno skrbi stavek switch, ki pokliče obnašanja spisana v posameznih podprogramih. Vsak tak podprogram izvede potrebno logiko, kot pri enostavnem odzivnem agentu. Dodatno mora vsebovati še logiko, ki pod določenimi pogoji spremeni trenutno stanje in s tem pove agentu, da naj v naslednjem ciklu začne z izvajanjem drugačnega obnašanja.

S tako izdelanim končnim avtomatom prehajanja med stanji agenta lahko več preprostih obnašanj enostavnih agentov združimo v kompleksnega agenta, ki zna spreminjati (izbirati) algoritem, po katerem se odziva.

Naloga: agent z modelom sveta

Pri umetni inteligenci v igrah si redko želimo, da se nasprotnik obnaša preveč "pametno" in natančno. Pogosto se ukvarjamo z ravno obratnim ciljem, kako narediti iz kirurško natančnega robota, ki ga je nemogoče premagati, čimbolj človeškega sogiralca. Človeško obnašanje je prav tako pogosto zaželeno pri neigralnih likih, ki velikokrat predstavljajo ravno ljudi.

Ena od zahtev, ki sledijo temu cilju, je, da agent, tako kot ponavadi igralec, ni vsevedoč. Agentu torej omejimo, da se ne zaveda celotnega sveta naenkrat, temveč lahko reagira le na stvari v svoji okolici.

Agentu z modelom sveta ne podamo kar scene stopnje, iz katere bi lahko ugotovil lokacije vseh nasprotnikov in objektov v stopnji, temveč ima agent svojo sceno, ki je na začetku prazna ter jo skozi branje vhodov postopno gradi. Zato rečemo, da agent gradi model sveta (posnetek dejanskega stanja scene).

Agentu z modelom sveta v vsakem ciklu zanke scena poda samo predmete, ki jih vidi s trenutne pozicije. Lokacije novih predmetov shrani v svojo lastno sceno in se glede na tako posodobljeno stanje odloči, kakšno akcijo bo izbral. Hkrati se s časom uči v kakšnem svetu se nahaja in lahko vedno pametneje izbira odločitve.

Pristop še zdaleč ni omejen na like, ki na ta način s premikanjem raziskujejo svet. Agent z modelom sveta je tudi vsak pametnejši algoritem za igranje iger s kartami, saj si mora v ozadju graditi svoj model, torej da predvideva, kakšne skrite karte imajo igralci v rokah.

Iskanje poti

Iskanje poti v grafu je eno prvih področij s katerimi se srečamo pri študiju umetne inteligence, zato tudi tu predvidevamo, da same algoritme poznate. Če ne drugega na medmrežju obstaja veliko dobrih "tutorialov" glede A*, če mogoče potrebujete osvežitev znanja. Tu si bomo pogledali, kako rezultate iskanja uporabimo v igri.

Naš cilj je ustvariti umetno-inteligenčni lik, ki bo znal sam najti pot mimo ovir skozi stopnjo. Za osnovo vzemimo agenta, ki se obrne direktno proti cilju in slepo hodi v to smer, dokler ga ne doseže:

```
@interface Agent : NSObject <IParticle, ICustomUpdate, ICustomCollider> {
	Vector2 *position;
	Vector2 *velocity;
	float radius;
	float mass;

	Vector2 *target;
}

@property (nonatomic, readonly) Vector2 *target;

- (void) goTo:(Vector2*)theTarget;

@end

@implementation Agent

- (id) init
{
	self = [super init];
	if (self != nil) {
		position = [[Vector2 alloc] init];
		velocity = [[Vector2 alloc] init];
		radius = 0.2f;
		mass = 1;
	}
	return self;
}

@synthesize position, velocity, radius, mass, target;

- (void) goTo:(Vector2 *)theTarget {
	[theTarget retain];
	[target release];
	target = theTarget;
}

- (void) updateWithGameTime:(GameTime *)gameTime {
	if (target) {
		[velocity set:[Vector2 subtract:target by:position]];
		float length = [velocity length];
		if (length > 0.01f) {
			[velocity normalize];
		} else {
			[velocity set:[Vector2 zero]];
			[target release];
			target = nil;
		}
	}
}

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

@end
```

Agentu lahko z metodo *goTo* ukažemo, proti kateri točki naj se premakne. V *update* bo vsakič usmeril svojo hitrost proti cilju, dokler se mu ne bo dovolj približal.

Za potrebe sledenja poti moramo agenta nadgraditi, da se bo znal premakniti med celotnim seznamom ciljnih točk. PathfindingAgenta izdelamo z dedovanjem Agenta, mu dodamo seznam točk in spremenimo obnašanje ob ukazu naj gre proti ciljni točki:

```
@interface PathfindingAgent : Agent <ISceneUser> {
	NSMutableArray *waypoints;
}

@property (nonatomic, readonly) NSMutableArray *waypoints;

@end

@implementation PathfindingAgent

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

@synthesize waypoints;

- (void) goTo:(Vector2 *)theTarget {
	// Clear current path.
	[waypoints removeAllObjects];

	// Insert your pathfinding algorithm here and fill the waypoints
	// [waypoints addObject:newWaypointToGoal ];

	[target release];
	target = nil;
}

- (void) updateWithGameTime:(GameTime *)gameTime {
	if (!target) {
		if ([waypoints count]) {
			[super goTo:[waypoints lastObject]];
			[waypoints removeLastObject];
		} else {
			[velocity set:[Vector2 zero]];
		}
	}

	[super updateWithGameTime:gameTime];
}

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

@end
```

Na ta način agent z iskanjem poti svojemu staršu zapovrstjo ukazuje kam naj se premika iz seznama svojih točk, ki jih je izračunal s pomočjo iskanja poti.

Še nasvet za samo izvedbo iskanja poti. Najlažje delo boste imeli, če imate že podatke o stopnji razporejene v dvo-dimenzionalnem seznamu. Za vsako polje lahko nastavite ali je prehodno ali ne, kar naj algoritem uporabi, ko generira povezave iz vozlišča v sosede.

Druga, težja možnost je posebna (hierarhična) scena, ki ob dodajanju objektov vanjo, sama v ozadju objekte razporeja v 2D mrežo. Najbolj prilagodljivo je, če za hranjenje uporabimo slovar, ki za vsako

koordinato (naslov večjega polja) hrani seznam objektov, ki se nahajajo na tem polju. Osnovna izvedba take funkcionalnosti je razred GridScene:

```
@interface GridScene : SimpleScene {
	NSMutableDictionary *grid;
}

- (NSArray*) getItemsAt:(XniPoint*)gridCoordinate;
- (NSArray*) getItemsAround:(XniPoint*)gridCoordinate neighbourDistance:(int)distance;

// Override this if you calculate grid coordinates form something other than IPosition or Vector2.
- (XniPoint*) calculateGridCoordinateForItem:(id)item;

@end

@implementation GridScene

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

		[super.itemAdded subscribeDelegate:
		 [Delegate delegateWithTarget:self Method:@selector(itemAddedToParent:eventArgs:)]];
		[super.itemRemoved subscribeDelegate:
		 [Delegate delegateWithTarget:self Method:@selector(itemRemovedFromParent:eventArgs:)]];
	}
	return self;
}

- (NSArray *) getItemsAt:(XniPoint*)gridCoordinate {
	NSMutableArray *itemsAtCoordinate = [grid objectForKey:gridCoordinate];
	return [NSArray arrayWithArray:itemsAtCoordinate];
}

- (NSArray*) getItemsAround:(XniPoint*)gridCoordinate neighbourDistance:(int)distance {
	NSMutableArray *itemsAround = [NSMutableArray array];

	for (int i = gridCoordinate.x-distance; i <= gridCoordinate.x+distance; i++) {
		for (int j = gridCoordinate.y-distance; j <= gridCoordinate.y+distance; j++) {
			NSMutableArray *itemsAtCoordinate = [grid objectForKey:[XniPoint pointWithX:i y:j]];
			[itemsAround addObjectsFromArray:itemsAtCoordinate];
		}
	}

	return itemsAround;
}

- (void) itemAddedToParent:(id<IScene>)scene eventArgs:(SceneEventArgs*)e {
	XniPoint *gridCoordinate = [self calculateGridCoordinateForItem:e.item];

	if (gridCoordinate) {
		NSMutableArray *itemsAtCoordinate = [grid objectForKey:gridCoordinate];
		if (!itemsAtCoordinate) {
			itemsAtCoordinate = [NSMutableArray array];
			[grid setObject:itemsAtCoordinate forKey:gridCoordinate];
		}
		[itemsAtCoordinate addObject:e.item];
	}
}

- (void) itemRemovedFromParent:(id<IScene>)scene eventArgs:(SceneEventArgs*)e {
	XniPoint *gridCoordinate = [self calculateGridCoordinateForItem:e.item];

	if (gridCoordinate) {
		NSMutableArray *itemsAtCoordinate = [grid objectForKey:gridCoordinate];
		[itemsAtCoordinate removeObject:e.item];
	}
}

- (XniPoint*) calculateGridCoordinateForItem:(id)item {
	id<IPosition> itemWithPosition = [item conformsToProtocol:@protocol(IPosition)] ? item : nil;
	Vector2 *position = [item isKindOfClass:[Vector2 class]] ? item : nil;

	if (itemWithPosition) {
```

```
                position = itemWithPosition.position;
        }

        if (position) {
                return [XniPoint pointWithX:floorf(position.x) y:floorf(position.y)];
        } else {
                return nil;
        }
}

- (void) dealloc
{
        [grid dealloc];
        [super dealloc];
}

@end
```

Novo metodo *getItemsAt* lahko uporabimo, da vidimo, če se na kateri od koordinat nahaja ovira ter v tem primeru algoritmu pri iskanju poti ne damo možnosti, da razširi vozlišče v to smer.

Naloga: iskanje poti

Iskanje poti je najpogostejša oblika uporabe iskanja pri umetni inteligenci v igrah; največkrat uporabljen algoritem pa je A*, saj imamo pri iskanju poti zelo enostavno hevristiko za oceno, koliko daleč od cilja smo (uporabimo npr. evklidsko ali pa manhattansko razdaljo).

Z iskanjem poti nadgradimo agenta, ki na ukaz, kam naj poskuša priti, namesto premikanja po direktni poti do cilja, vnaprej zgenerira zaporedje vmesnih točk, ki ga bo mimo ovir pripeljalo do ciljnega vozlišča.