NEAT algoritam u video igrama

Nuić, Luka

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij

NEAT ALGORITAM U VIDEO IGRAMA

Diplomski rad

Luka Nuić

Osijek, 2018.

SADRŽAJ

1. UVOD

Cilj ovog diplomskog rada je prikazati način rada NEAT algoritma - NeuroEvolucija Proširenih Topologija (engl. Neuroevolution of Augmented Topologies) na računalnim igrama. NEAT algoritam već je implementiran u C# okruženje stoga je potrebno samo napraviti vlastitu video igru i ispravno implementirati algoritam[2].

Izradit će se dvije računalne igre gdje će prva prikazivati kako algoritam radi u stvarnom vremenu. Algoritam će imati mogućnost kretanja i treba ispravno reagirati na promjene u svojoj okolini. Zbog toga, nakon što se izradi igra, potrebno je implementirati sustav detekcije okoline kako bi algoritam znao gdje se nalazi. Zatim treba definirati način nagrađivanja da algoritam zna što treba raditi, a što ne. Na kraju se za ispravno kretanje algoritma ispravljaju vrijednosti parametara i vrši se analiza treniranja populacije.

Druga igra će pokazati kako NEAT algoritam treba ispravno dizajnirati objekt za savladavanje prepreka. Algoritam će stvoriti svaki objekt te u ovisnosti o učinku pojedinog objekta odlučivati o tome tko će napredovati i kako. Zbog ovakvog pristupa implementacija algoritma je puno manje zahtjevna, dok je sama izrada igre puno zahtjevnija za razliku od prve igre. Potrebno je pomoću vrijednosti na izlazu, koje algoritam generira, stvoriti objekte s različitim varijacijama u dimenzijama.

Za razvoj video igara koristit će se Unity platforma sa skriptama pisanim u C# programskom jeziku u Microsoft Visual Studio razvojnom alatu. Kao algoritam koje će se implementirati oko igara koristit će se verzija SharpNEAT[2] implementacija za Unity platformu[3].

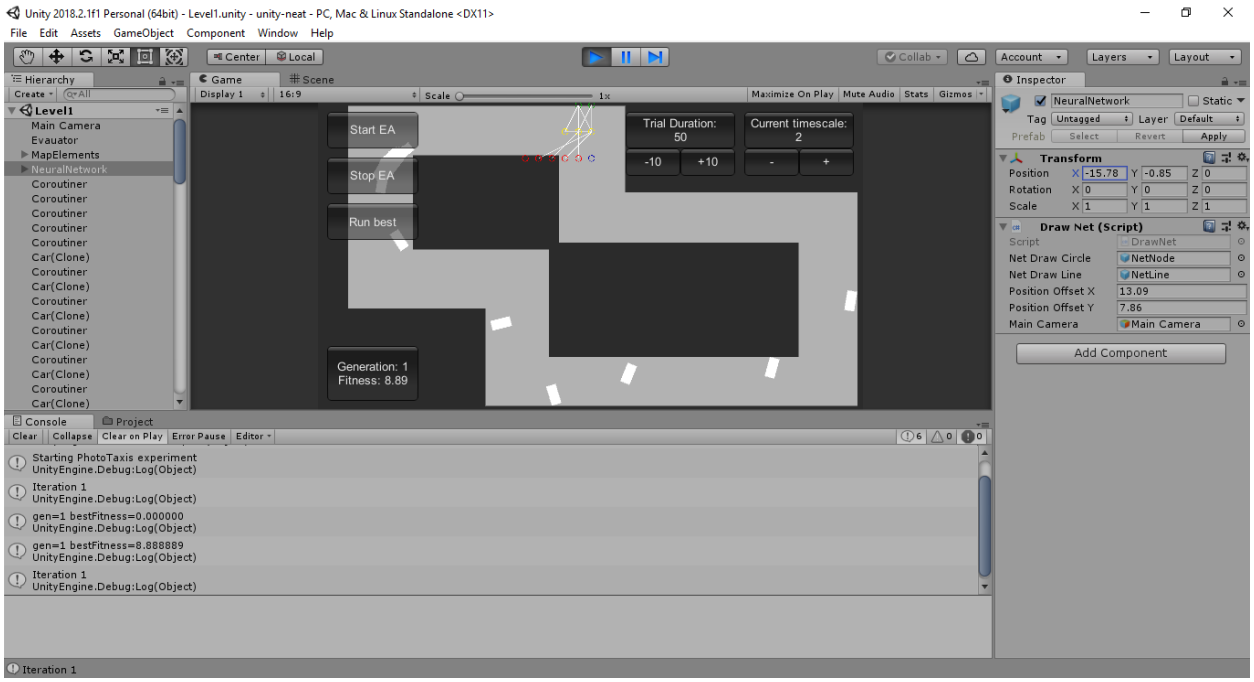
2. PRIMIJENJENE TEHNOLOGIJE I ALATI

2.1. Unity razvojno okruženje

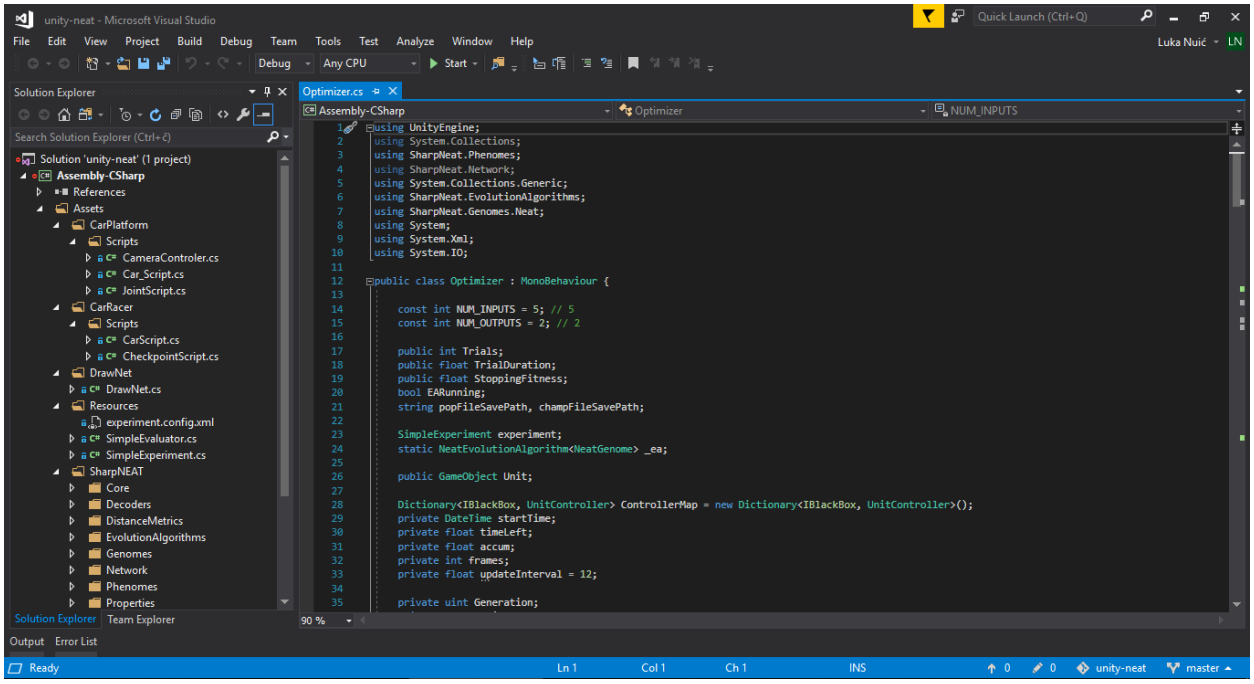
Unity je platforma za razvoj video igara koju je razvila Unity Technologies. Najviše se koristi za razvoj trodimenzionalnih i dvodimenzionalnih video igara za računala, konzole i mobilne uređaje. Unity omogućuje jednostavno sučelje za stvaranje scena u igrama te pisanje skripti u C# programskom jeziku. Također je podržavao i druge programske jezike kao Boo i JavaScript, ali više nisu kompatibilni s današnjim verzijama. Za uređivanje skripti u C# programskom jeziku najčešće se koristi Microsoft Visual Studio ili MonoDevelop kao besplatna verzija. Isto tako MonoDevelop je podržan na Linux i OSX operacijskom sustavu.

U početku Unity je bio namijenjen samo za OS X platformu, ali danas podržava 27 drugih platformi gdje su od najznačajnijih: Windows, iOS, Android i Mac. Za razvoj igara na pregledniku bio je podržan Unity Web Player samo na Windows i OS X, ali je zamijenjen WebGL. Velika prednost Unity okruženja je besplatna verzija aplikacije s velikim brojem značajki. Kod plaćenih licenci dobiva se bolja podrška pri razvoju kao i bolje usluge korištenja cloud računala.

U ovom radu izrađivat će se jednostavne 2D računalne igre koje će prvenstveno biti namijenjene za razvoj neuronskih mreža. Pisanje skripti s C# programskim jezik će se obavljati u Microsoft Visual Studio okruženju.



Slika 1. Sučelje Unity razvojnog okruženja



Slika 2. Microsoft Visual Studio razvojni alat

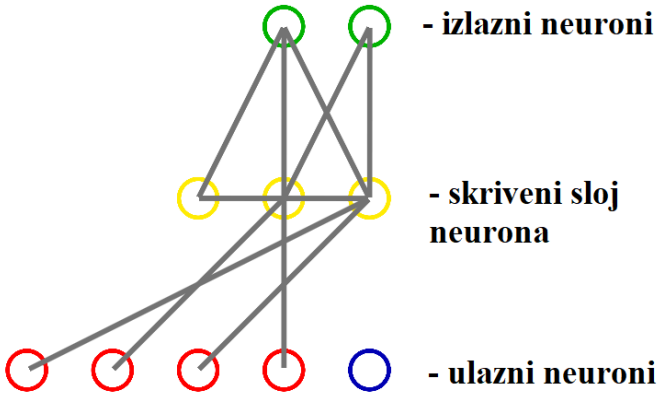
2.2. Umjetne neuronske mreže

Umjetne neuronske mreže su skup međusobno povezanih neurona koje „uče“ kako svladati određeni zadatak. Nadahnute su biološkim neuronskim mrežama kao kod životinjskog mozga zbog istog načina učenja, kroz iskustvo. Mreža se sastoji od ulaznih, izlaznih i skrivenih neurona koji su međusobno povezani vezom određene težine.

Vrstu mreže koju koristi NEAT algoritam je „Feed-forward“ što znači da informacija koja dođe na ulazne neurone će proći do izlaznih u jednom smjeru. Također postoje druge vrste neuronskih mreža poput ponavljajuća neuronska mreža (engl. Recurrent neural network) kod koje informacija može kružiti mrežom zbog postojanja povratnih veza. Takve mreže imaju mogućnosti pamćenja, ali ih je puno teže trenirati zbog kompleksnih radnji koje mogu učiniti.

Primjer treniranja mreže, kod procesa prepoznavanja slike, je kada se mreži daje skup slika i definira dali se na toj slici nalazi traženi objekt ili ne. Mreža se tako trenira na većem dijelu skupa slika te se nakon toga ostatak skupa koristi za provjeravanje ispravnosti treniranja. Izračunava se s kolikom vjerojatnošću neuronska mreža može prepoznati određeni objekt na slici. Tako mreža pomoću skupa podataka može samostalno generirati određenu funkciju rješavanja zadatka. Ovakav način učenja mreže se naziva nadzirano učenje (engl. Supervised Learning). Nasuprot tome također postoji nenadzirano učenje gdje mreža samostalno mora shvatiti kakav utjecaj imaju

informacije koje dobiva na ulazu. Način učenja koji NEAT algoritam koristi je „Reinforcement learning“ koji tijekom rada za svaku dobru radnju nagrađuje, a za svaku lošu kažnjava.



Slika 3. Umjetna neuronska mreža

2.3. NEAT algoritam

NeuroEvolucija Proširenih Topologija (engl. Neuroevolution of Augmented Topologies) je genetski algoritam (engl. Genetic Algorithm) korišten za evoluciju neuronskih mreža. Razvio ju je Ken Stanley 2002. godine u Sveučilištu Texas u Austin-u. Cjelokupnu implementaciju NEAT algoritma u C# / .NET okruženje je napravio Colin Green. U ovom projektu koristit će se verzija SharpNeat-a prilagođena za Unity platformu.

U tradicionalnim NeuroEvolucijskim pristupima pri evoluciji mreža uzimala se u obzir samo težinski parametri dok se topologija mreže odabirala prije nego evolucija započne. Ovakav pristup ima cilj samo optimizacije težinskih parametra mreže. Zatim se pokazalo da težinski parametri nisu jedini koji utječu na performanse mreže. Izmjena topologije, odnosno strukture mreže tijekom evolucije također utječe na funkcionalnost sustava. Tako algoritam traži ravnotežu između učinka razvijenih rješenja i njihove raznolikosti. Takva mreža se također naziva TWEAN (engl. Topology and Weight Evolving Artificial Neural network).

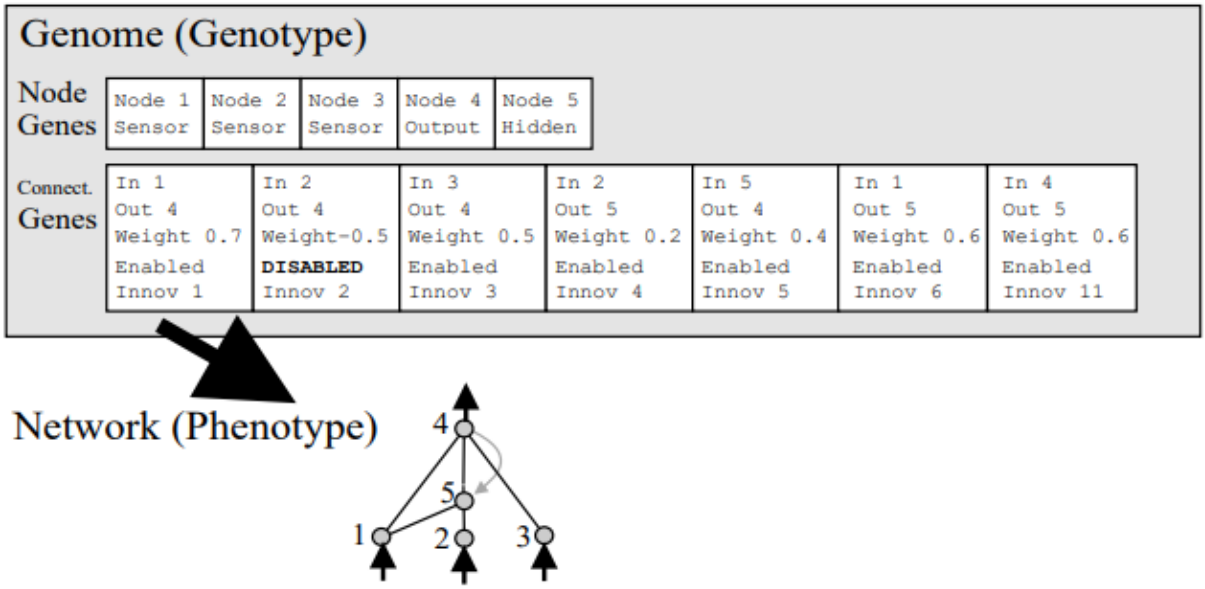
NEAT pristup započinje s mrežom koja se sastoji od samo ulaznih i izlaznih neurona. Nakon toga kroz evoluciju s diskretnim koracima, učinkovitost mreže se povećava. Povećanje učinkovitosti mreže postiže se s raznim promjenama unutar mreže kao što je izmjena težinskih vrijednosti veza između neurona, dodavanjem novih veza i dodavanjem novih neurona unutar skrivenog sloja.

NEAT algoritam temelji se na primjeni četiri ključne značajke:

- Evolucijski proces – NEAT koristi koncept vrsta (engl. species). Cjelokupna populacija je podijeljena u podskupove (vrste) gdje je pojedina vrsta povezana sa svojim genotipom te se koristi tijekom evolucije. Vjerojatnost povezivanja dvaju pojedinca iz različitih podskupa je vrlo mala što znači da će dijete imati manje šanse da bude lošiji od roditelja pošto su roditelji bili kompatibilni.
- Dijeljenje učinka pojedinca – učinak (engl. fitness) pojedinca se dijeli unutar vrste. Računa se prosječan učinak jedne vrste i ta vrijednost se smatra kao učinak svakog pojedinca unutar vrste. Zbog toga ako neka vrsta ima izvrsnog pojedinca ta vrsta se neće smatrati najboljom unutar populacije. Također promovira raznolikost unutar vrste, tj. što je vrsta veća, zbog dijeljenog učinka mogu postojati veće razlike između pojedinaca.
- Izbjegavanje konkurentskih inovacija putem povijesnih oznaka – Tijekom evolucije pojedinaca događaju se inovacije. Inovacije se mogu ponoviti iako su kod različitog pojedinca. Tako se u klasičnim algoritmima može dogoditi da se iste inovacije prenose više puta. NEAT to rješava tako da postavlja oznake za svaku inovaciju te se tijekom mutacija svaka inovacija gleda samo jednom.
- Kompleksifikacija (engl. Complexification) – početna neuronska mreža je što jednostavnija (npr. mreža sa samo ulaznim i izlaznim neuronima bez veza) nakon čega algoritam dodaje nove neurone ili veze između postojećih neurona ovisno o učinkovitosti sustava.

2.3.1. Programski kod NEAT algoritma

Programski kod algoritma moguće je napisati u bilo kojem programskom okruženju sve dok se prate značajke algoritma navedene u [1]. Tako će u ovom poglavlju biti objašnjena implementacija glavnih značajki algoritma u C# programskom jeziku. Implementacije u popularnim programskim jezicima i platformama već postoje kao što je navedenom pod [2] i [3] te su trenutno još u razvoju.



Slika 4. Parametri i sastav genoma[5]

Na početku je potrebno definirati način na kojem ćemo genome prikazati pomoću koda. Jedan genom predstavlja jedan organizam nekog sustava. Svaki genom se sastoji od čvorova i veza između istih. Čvor nekog genoma može po vrsti biti ulazni, izlazni i skriveni te mu uz tu značajku dodajemo i identifikacijski broj. Veze između čvorova trebaju sadržavati informacije o neuronima koje povezuju, težinski broj, je li navedena veza izražena ili ne te inovacijski broj. Zatim se definira genom koji se sastoji od skupa veza i neurona (Slika 4).

```
class ConnectionGene
{
    private int inNode;
    private int outNode;
    private float weight;
    private bool expressed;
    private int innovation;

    public ConnectionGene(int inNode, int outNode, float weight, bool expressed, int
innovation)
    {
        this.inNode = inNode;
        this.outNode = outNode;
        this.weight = weight;
        this.expressed = expressed;
        this.innovation = innovation;
    }
}
class NodeGene
{
    public enum TYPE
    {
        INPUT,
```

```
        HIDDEN,
        OUTPUT,

    }
    private TYPE type;
    private int id;

    public NodeGene(TYPE type, int id)
    {
        this.type = type;
        this.id = id;
    }
}
class Genome
{
    private Dictionary<int, ConnectionGene> connections;
    private Dictionary<int, NodeGene> nodes;

    public Genome()
    {
        nodes = new Dictionary<int, NodeGene>();
        connections = new Dictionary<int, ConnectionGene>();
    }
}
```

Kod 1. Podatkovna struktura genoma[4]

NEAT algoritam treba imati mogućnost izmjene težina neuronskih veza unutar genoma te izmjene strukture mreže. Zato je potrebno definirati metode koje će prilikom poziva dodati novu vezu ili novi čvor u genom. Mutacija prilikom koje se dodaje novi čvor u strukturu se obavlja tako da se prvo nasumično odabire veza u genomu i čitaju podaci o ulaznom i izlaznom čvoru. Odabrana veza postaje ne izražena (engl. expressed = false) te se dodaje novi čvor i dvije nove konekcije u genom. Ulaz odabrane veze se postavlja kao ulaz na jednu od novih veza dok je izlaz nove veze novostvoreni čvor. Druga veza spaja izlaz čvora za izlazom odabrane veze. Mutacija veza dosta je zahtijevnija za izvedbu naspram mutacije neurona. Tijekom ove mutacije nakon nasumičnog odabira dvaju čvorova provjerava se je li veza između njih moguća. Veza između dva čvora je moguća ako oba čvora nisu ulazni ili izlazni čvor. Zatim se provjerava postoji li veza između neurona te ako ne postoji stvara se nova veza s nasumičnom vrijednosti težine. U slučaju da veza nije moguća ili ako već postoji algoritam se vraća i odabire novi par čvorova. Također kao vrsta mutacije strukture mreže potrebno je definirati metodu za izmjenu težina neuronskih veza genoma. Pri ovoj mutaciji, u 90% slučajeva, težina veze se množi s nasumičnim koeficijentom gdje ostalih 10% poprima novu nasumičnu vrijednost težine[1].

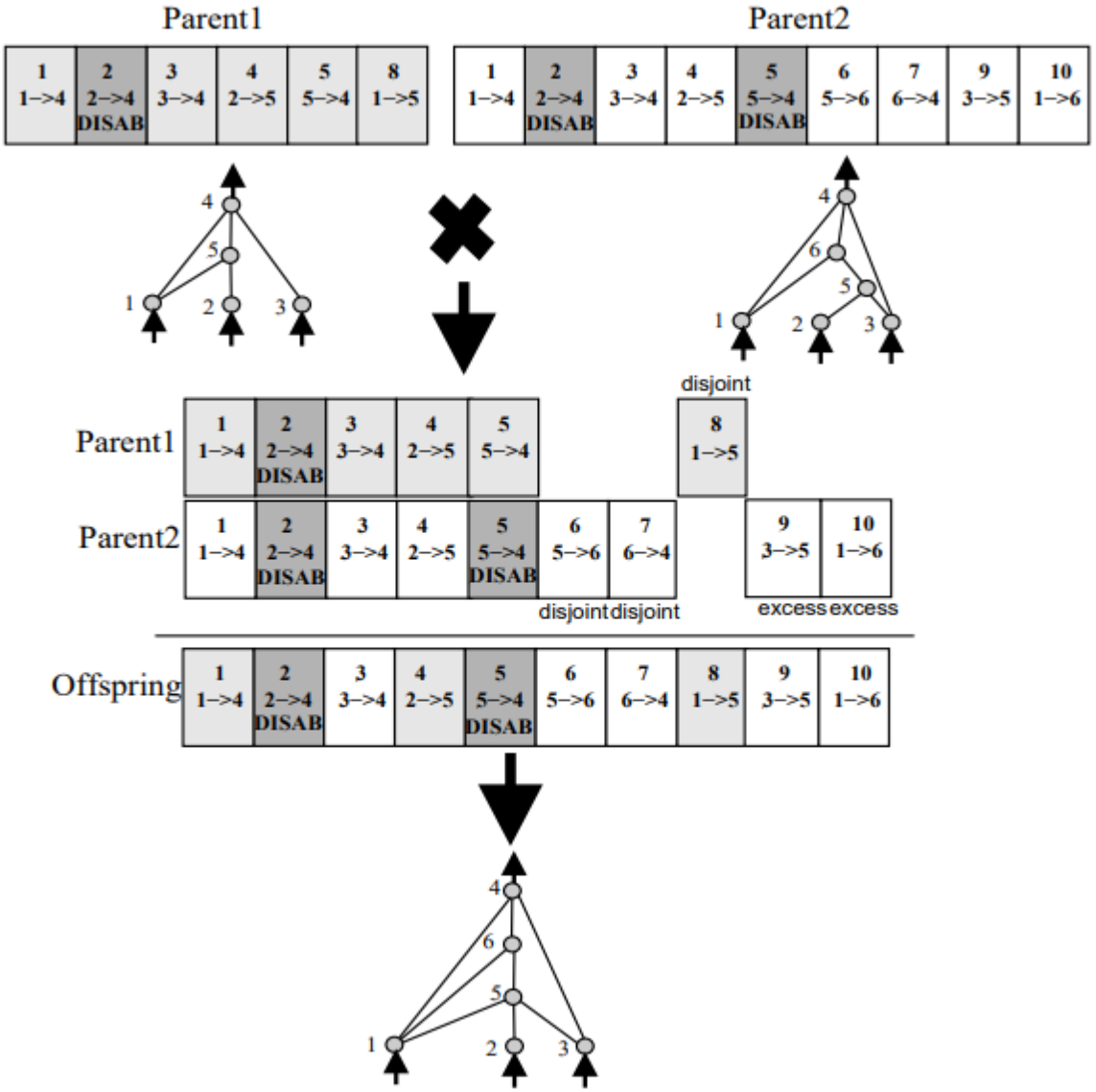
Nakon stvorenih metoda mutacija genoma izrađuje se metoda za preklapanje dvaju genoma. Tijekom preklapanja potrebno sve neurone s istim inovacijskim broj izjednačiti. Čvorovi dvaju genoma s istim inovacijskim brojem se smatraju podudarajućim te se direktno prenose na potomka. Čvorovi koji se ne podudaraju niti s jednim čvorom u drugom genomu smatraju se višak (engl. excess) ili odvojeni (engl. disjoint) neuron. Ovisno o tome prelazi li inovacijski broj čvora najveći

inovacijski broj ili ne, čvor postaje višak ili odvojeni neuron (Slika 5). Prilikom stvaranja potomka, kod podudarajućih čvorova, nasumično se odabere čvor bilo kojeg roditelja dok se svi čvorovi koji su višak ili odvojeni prenose na potomka od roditelja s većim učinkom.

```
public static Genome Crossover(Genome parent1, Genome parent2, Random r)
{
    Genome child = new Genome();

    foreach(NodeGene parent1Node in parent1.GetNodeGenes().Values)
    {
        child.AddNodeGene(parent1Node.copy());
    }
    foreach (ConnectionGene parent1Node in parent1.GetConnectionGenes().Values)
    {
        if (parent2.GetConnectionGenes().ContainsKey(parent1Node.GetInovation())) //matching
gene
        {
            ConnectionGene childConGene = Convert.ToBoolean(r.Next(2)) ? parent1Node.Copy() :
parent2.GetConnectionGenes()[parent1Node.GetInovation()];
            child.AddConnectionGene(childConGene);
        }
        else //disjoint or excess gene
        {
            ConnectionGene childConGene = parent1Node.Copy();
            child.AddConnectionGene(childConGene);
        }
    }
    return child;
}
```

Kod 2. Preklapanje dvaju genoma

Prikazana metoda (Kod 2) pretpostavlja da se prije njenog izvršenja zna koji je roditelj ima veći učinak. Veze s jednakim inovacijskom vrijednošću nasumično se uzimaju od jednog roditelja dok se ne podudarajuće veze uzimaju od razvijenijeg roditelja. Ovo je način preklapanja (Slika 5) gdje se preklapaju dva genoma različite strukture, ali pretežno istih čvorova (s istim inovacijskim brojem).



Slika 5. Način preklapanja dvaju genoma[8]

Sljedeći postupak ima cilj sačuvati inovacije različitih struktura pomoću podjele genoma u vrste. Tako se cjelokupna populacija dijeli u vrste gdje se sve slične strukture postavljaju u istu vrstu. Sličnost dvaju genoma će biti određena s brojem ne podudarajućih neurona gdje će genomi s više podudarajućih neurona biti kompatibilni. Razina kompatibilnosti dvaju genoma može se dobiti pomoću formule:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W} \qquad (1)$$

gdje je:

- δ – razina kompatibilnosti
- c_1, c_2 i c_3 – koeficijenti za definiranje važnosti pojedinih parametara

- N – koeficijent za normalizaciju većih genoma (ako je broj neurona u genomu manji od 20 koeficijent je 1)
- E i D – broj ne podudarajućih neurona između genoma
- W – prosječna težinska razlika između veza dvaju genoma

Kako bi se mogla izračunati razina kompatibilnosti potrebno je prvo prebrojati sve ne podudarajuće neurone kao i izračunati prosječnu težinsku razliku između veza genoma. Postupak brojanja ne podudarajućih neurona obavlja se tako da se prolazi kroz svaki neuron genoma. Kada se pronađe, ako je ne podudarajući neuron veći od maksimalnog inovacijskog broj genoma, dodaje se kao višak (engl. excess) odnosno ako je manji kao odvojeni čvor (engl. disjoint). Zatim treba izračunati težinsku razliku veza za što je također potreban broj podudarajućih neurona (Kod 3). Dohvaća se najveći inovacijski broj genoma te se prolazi kroz svaku neuronsku vezu. Svaka podudarajuća veza dodaje se u brojač i računa se težinska razlika između veza. Na kraju vraća se omjer težinske razlike veza i podudarajućih neurona.

```
public static float averageWeightDiff(Genome genome1, Genome genome2)
{
    int matchingGenes = 0;
    float weightDifference = 0;

    List<int> conKeys1 = new List<int>(genome1.GetNodeGenes().Keys);
    List<int> conKeys2 = new List<int>(genome2.GetNodeGenes().Keys);

    int highestInnovation1 = conKeys1[conKeys1.Count() - 1];
    int highestInnovation2 = conKeys2[conKeys2.Count() - 1];

    int indices = Math.Max(highestInnovation1, highestInnovation2);
    for (int i = 0; i <= indices; i++)
    {
        ConnectionGene connection1 = genome1.GetConnectionGenes()[i];
        ConnectionGene connection2 = genome2.GetConnectionGenes()[i];
        if (connection1 != null && connection2 != null)
        {
            matchingGenes++;
            weightDifference += Math.Abs(connection1.GetWeight() - connection2.GetWeight());
        }
    }

    return weightDifference / matchingGenes;
}
```

Kod 3. Metoda prosječne težinske razlike veza genoma

Nakon dobivenih parametara moguće je izračunati razinu kompatibilnosti dvaju genoma. Ako je razina u dogovorenim granicama genom se postavlja pod tu vrstu genoma. U slučaju da nije, uzima nasumični genom iz druge vrste i nastavlja uspoređivati dok ne nađe vrstu kojoj pripada. Ako genom ne pripada niti jednoj vrsti stvara se nova vrsta s tim genomom kao predstavnikom.

Kao mehanizam očuvanja vrsta u populaciji, NEAT algoritam koristi eksplicitno dijeljenje učinka (engl. explicit fitness sharing). Dijeljenje učinka između pojedinaca unutar vrste obavlja se sljedećom formulom:

$$f_i' = \frac{f_i}{\sum_{j=1}^{n} sh(\delta(i,j))} \tag{2}$$

gdje je:

- f_i' – dijeljeni učinak pojedinca
- $\delta(i,j)$ – razina kompatibilnosti dvaju genoma
- $sh(\delta(i,j)$ – funkcija koja vraća 1 ako je razina u dogovorenim granicama, u suprotnom vraća 0
- f_i – učinak pojedinca

Tako će vrsta s više pojedinaca dobivati manji učinak s čime se onemogućuje jednoj vrsti da prevlada cjelokupnu populaciju. Očuvanjem vrsta manjih veličina čuvaju se inovacije koje u kasnijim generacijama mogu biti ključne te prevladati populaciju.

3. REALIZACIJA VIDEO IGARA

U ovom radu izrađene video igre bit će prvenstveno namijenjene za NEAT algoritam. Cjelokupnim razvojem upravljat će algoritam gdje će korisnik po želji moći upravljati brzinom izvođenja algoritma, brojem pokušaja unutar jedne generacije kao i trajanjem pokušaja. Izradit će se dvije igre koje će biti različitog tipa. Prva igra služit će za prikaz stalnog rada algoritma kako reagira na promjene u okruženju dok će druga prikazati izmjenu upravljanog objekta s ciljem dobivanja boljeg rezultata.

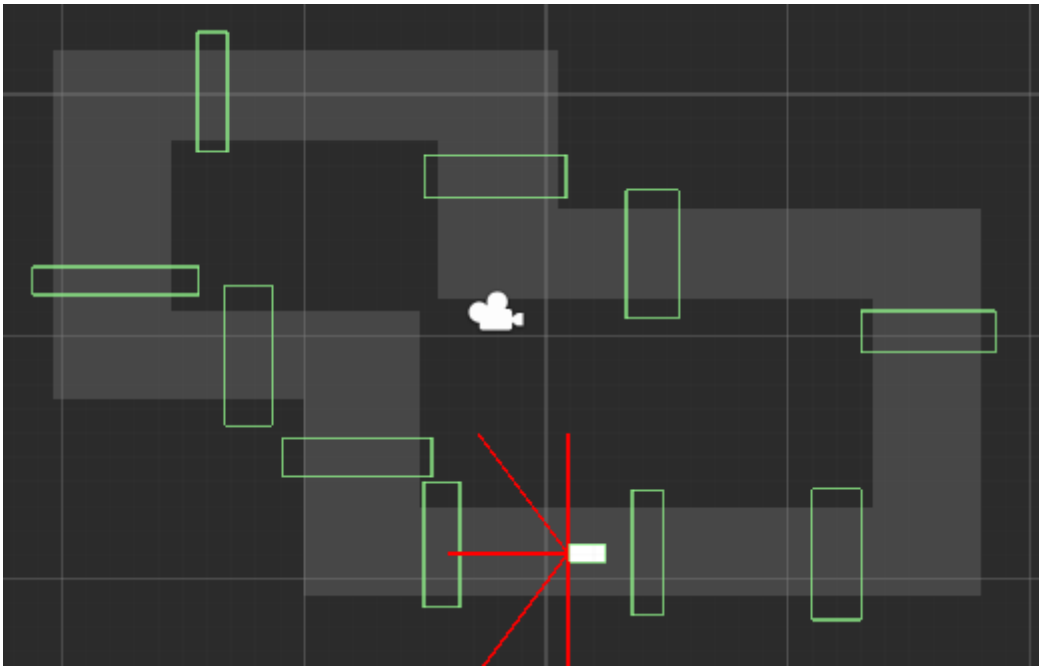
3.1. Izrada video igara

Na početku potrebno je izraditi objekt s kojim će algoritam upravljati. Zbog toga treba kontrole odmah ispravno definirati. Pošto algoritam sa svakom obradom na izlaze daje vrijednosti od 0 do 1 treba obratiti pozornost na ispravno skaliranje vrijednosti. Tako se unutar Unity sučelja izrađuje jednostavan objekt na kojega postavljamo sve komponente koje su mu potrebne za interakciju s okolinom. Isključuje se djelovanje gravitacije na objektu te postavlja se skripta koja će upravljati objektom. Unutar skripte definiraju jednostavne kontrole pomoću tipkovnice za pomoć pri daljnjoj izradi (Kod 4).

Zatim se postavljaju objekti unutar scene s kojim će upravljani objekt međudjelovati. Ovi objekti neće imati mogućnost kretanja već će služiti kao granice kretanja objekta i kao okidači za određene događaje. Tako je u sceni potrebno imati zidove koji određuju put kretanja kao i kontrolne točke koje će označavati napredak pojedinca. Za svaki objekt definiran je drugačiji sloj interakcije. Senzori koji detektiraju udaljenosti od zida trebaju detektirati samo zid odnosno ne smiju detektirati ostale članove populacije kao niti objekte na stazi (npr. kontrolne točke).

```
CarRotation += Input.GetKey(KeyCode.A) ? 3f : Input.GetKey(KeyCode.D) ? -3f : 0f;

if (Input.GetKey(KeyCode.W) && CarVelocity < maxCarSpeed + 10f) CarVelocity += 0.1f;

else if (CarVelocity > 0f) CarVelocity -= 0.1f;

if (Input.GetKey(KeyCode.S) && CarVelocity > -maxCarSpeed + 10f) CarVelocity -= 0.1f;

else if (CarVelocity < 0f) CarVelocity += 0.1f;

transform.rotation = Quaternion.Euler(new Vector3(transform.rotation.x, transform.rotation.y,
CarRotation));
gameObject.GetComponent<Rigidbody2D>().velocity = transform.up * CarVelocity;
```

Kod 4. Kontrole upravljanja objektom



Slika 6. Izgled scene prve video igre

Ideja druge igre je da algoritam generira određene objekte te da se ti generirani objekti samostalno kreću kroz prepreke. Objekt koji savlada najveći broj prepreka smatra se najučinkovitijim. Tako kroz evoluciju algoritam bi trebao stvoriti idealni objekt za rješavanja postavljenih prepreka. Stoga druga igra će biti puno zahtjevnija za izradu. Unatoč tome sama implementacija algoritma će biti puno jednostavnija jer nisu potrebni senzori okoline za razliku od prve igre.

Algoritmu se daje mogućnost kontroliranja triju parametara: broj kotača koje vozilo ima, duljina od kotača do središta vozila i promjer samih kotača. Kao i u prvoj igri postavlja se objekt upravljanja te se dodaju sve komponente za ispravno kretanje. Dodaje se skripta koja će od parametara koje joj algoritam preda generirati objekt. Tako smo stvorili objekt kojeg će algoritam koristiti u evolucijskom procesu. Pri stvaranju objekta čita se jedan od izlaza neuronske mreže i koristi ga kao broj kotača na tom objektu. Izvršava se petlja koja stvara navedeni broj kotača s objektom koji povezuje kotač s vozilom. Sam broj kotača također definira i koliko se dodatnih izlaza mreže čita. Za svaki kotač vozila čitaju se dodatna dva izlaza gdje prvi definira veličinu kotača, a drugi udaljenost kotača od vozila. Raspored čitanja izlaza može se vidjeti na sljedećem prikazu:

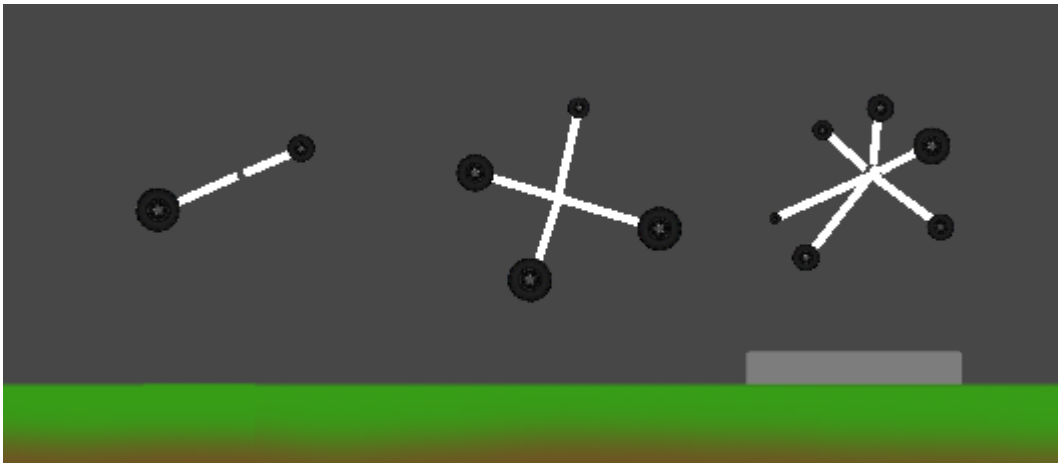
n_r	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	r_9	r_{10}	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}

gdje je:

- n_r – broj kotača vozila

- r_n – veličina kotača
- d_n – udaljenost kotača od središta vozila

Takav raspored omogućava vozilu da svaki kotač može biti drugačijih dimenzija što stvara veliki broj varijacija između vozila. Isto tako tijekom stvaranja svakog kotača vozila, čitanjem odgovarajućeg izlaza, kotač se postavlja na određenu udaljenost od središta vozila (Prilog A). Nadalje na objekt koji povezuje kotač s vozilom dodajemo skriptu koja će mijenjati dimenzije kotača. Skripta stvara predefinirani objekt kotača sa svim potrebnim komponentama te ga pozicionira na drugi kraj. Zatim nakon što se instancira objekt kotača prima se vrijednost dimenzija te mu se odmah dodjeljuje. Također brzina rotacije kotača postavlja se na konstantnu vrijednost kako bi se svako vozilo jednako kretalo.

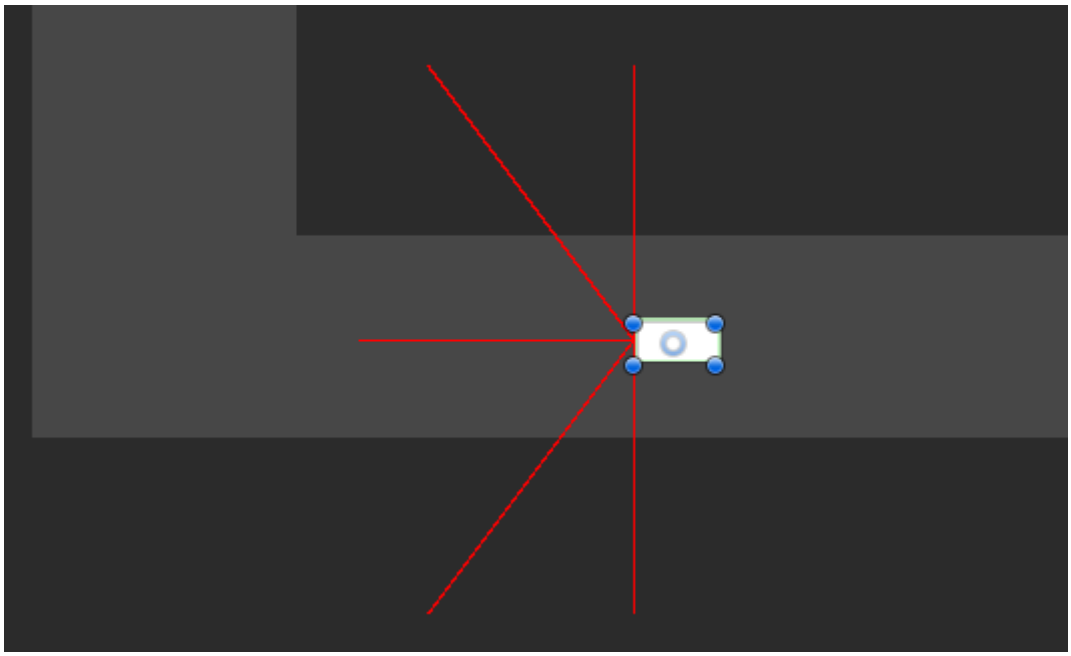


Slika 7. Stvoreni objekti s dva, četiri i šest kotača

3.2. Implementacija algoritma u video igre

Prvo je potrebno proučiti upute za stvaranje vlastite video igre s NEAT algoritmom [3] na Unity platformi. Navedeni repozitorij se preuzme i priključi u vlastiti Unity projekt. Zatim unutar scene dodaje se prazan objekt na kojeg se postavlja skripta „Optimizer“ kao komponenta. Također potrebno je izmijeniti klasu nasljeđivanja iz „MonoBehaviour“ u „UnitController“. Nakon toga unutar skripte možemo se služiti svim metodama i objektima potrebnim za rad s NEAT algoritmom. Glavni objekt s kojim će algoritam raditi prikačimo na „Optimizer“ skriptu. Pomoću iste skripte mijenjamo broj ulaza i izlaza mreže, trajanje jednog pokušaja generacije, broj pokušaja populacije te vrijednost učinka populacije na kojoj se algoritam zaustavlja. Isti proces se ponavlja za obje igre.

Kao prvi korak implementacije potrebno je na ulaze mreže dovesti ispravne parametre. Na ulaz se dovode podaci pomoću kojih će mreža donositi odluke. Tako u slučaju prve igre stvaraju se senzori koji će vraćati udaljenost objekta od prepreke. Kako bi se dobila udaljenost koristi se metoda „Raycast“ klase „Physics2D“ koja kao parametre prima točku iz koje će zraka senzora biti poslana, smjer slanja te zrake i domet zrake senzora. Ako u svom dometu senzor dođe u kontakt s objektom vraća se objekt koji u svojim parametrima ima udaljenost na kojoj se udaljenosti dogodio kontakt. Stoga se stavljaju pet senzora gdje će svaki imati drugačiju orijentaciju (Slika 8). Također treba postaviti da se zraka senzora odbija samo od zidova, a ne i od ostalih transparentnih objekata (npr. kontrolnih točaka staze ili drugih vozila tijekom treniranja). Zatim se udaljenosti spremaju u niz te se pokreće algoritam. Izlazi mreže služe za upravljanje objektom. U ovoj igri upravlja se brzinom i rotacijom vozila zbog čega je potrebno imati samo dva izlaza mreže. Nakon što se algoritam izvrši čitaju se izlazi mreže te se skaliraju na odgovarajuće vrijednosti (Kod 5).



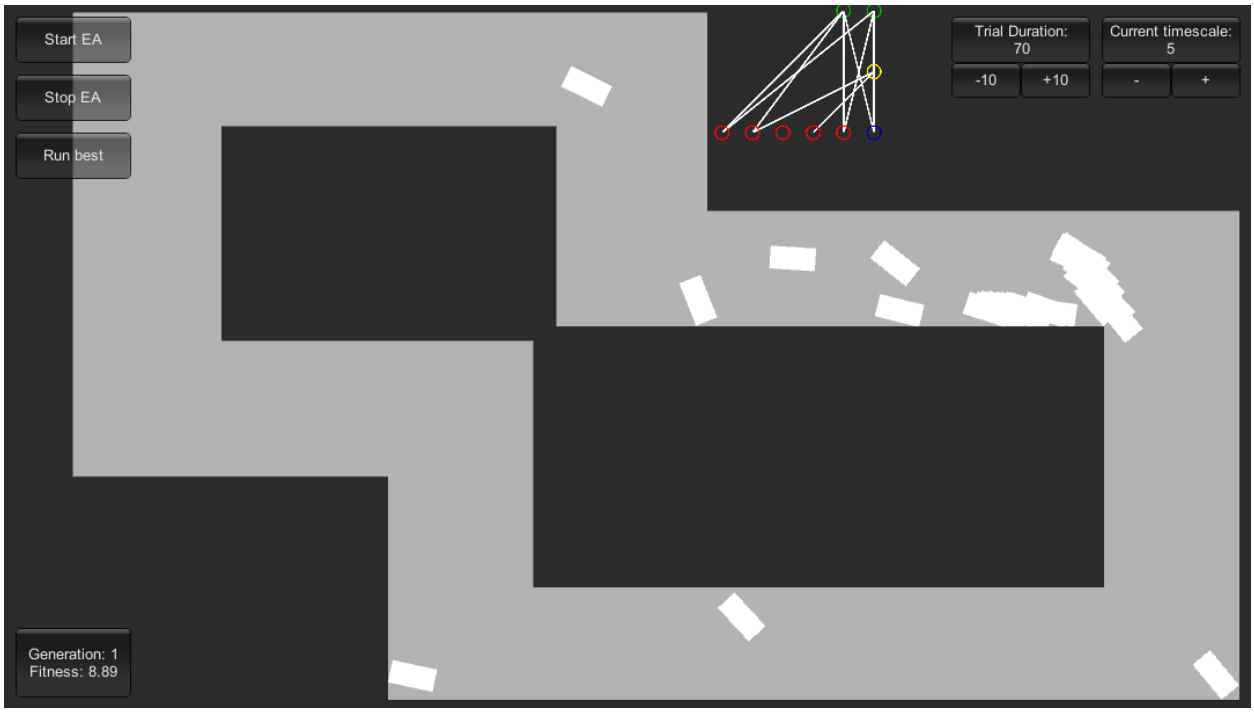
Slika 8. Raspored senzora na objektu

Zatim se definira funkcija za računanje učinka pojedinca. Ovdje se koriste prethodno napravljene kontrolne točke na stazi. Svaki prelazak vozila preko kontrolne točke donosi određenu vrijednost učinka gdje obilazak cijele staze donosi najveći učinak pojedincu. Isto tako potrebno je definirati redoslijed prolaska kontrolnih točaka odnosno da pojedinac učinak dobiva samo u ispravnom smjeru kretanja. Kako bi se smanjio broj sudara vozila sa zidom dodaje se brojač koji se povećava sa svakim sudarom. Navedeni brojač negativno utječe na učinak pojedinca što će poteći vozila da što manje dolaze u doticaj sa zidom. Formula za izračun učinka pojedinca je:

$$f_i = (10 \,/\, n_c) * \; n_{ci} + \; 10 * n_L - 0.1 * n_W \qquad (3)$$

gdje je:

- f_i – učinak pojedinca
- n_c – suma kontrolnih točaka
- n_{ci} – broj prijeđenih kontrolnih točaka
- n_l – broj obilazaka cjelokupne staze
- n_w – broj sudara vozila sa zidom



Slika 9. Treniranje populacije na prvoj igri

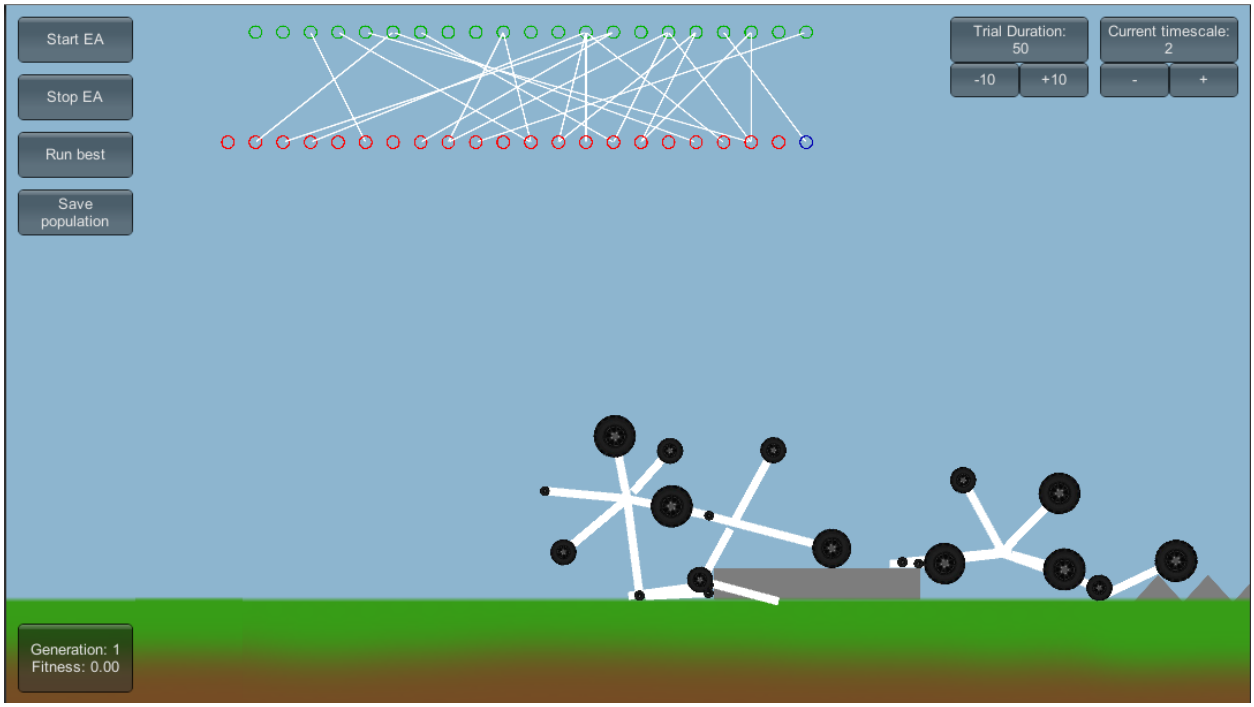
Na isti način se implementira algoritam i u drugu igru (Prilog A). Zbog načina rada na ulazu nisu potrebne nikakve vrijednosti. Usprkos tome vrijednosti su dodijeljene zbog toga što u protivnom na izlazu mreže pojavljuju konstantne vrijednosti. Zatim se pokreće treniranje te se čitaju vrijednosti s izlaza mreže. Pročitane vrijednosti u nastavku se koriste za definiranje dimenzija vozila (poglavlje 3.1). Učinak pojedinca u drugoj igri označen je s brojem prepreka kojih savlada. Stoga je samo potrebno dohvatiti udaljenost koju pojedinac prijeđe.

```
ISignalArray inputArr = box.InputSignalArray; //Network input array
inputArr[0] = frontSensor;
inputArr[1] = leftFrontSensor;
inputArr[2] = leftSensor;
inputArr[3] = rightFrontSensor;
inputArr[4] = rightSensor;

box.Activate(); //Activate network

ISignalArray outputArr = box.OutputSignalArray; //Network output array

//Scaling network output
var rotation = ((float)outputArr[0] * 2 - 1) * 5f;
var velocity = ((float)outputArr[1] * 2 - 1) / 20;

CarRotation += rotation;

if (Math.Abs(CarVelocity) < maxCarSpeed)
{
    CarVelocity += velocity;
}
```

Kod 5. Pisanje na ulaz i čitanje s izlaza neuronske mreže

3.3. Objašnjenje sučelja i analiza rezultata

Kada se pokrene igra dobiva se sučelje na kojemu je moguće pokrenuti i zaustaviti algoritam. Čim se algoritam pokrene prvo se pretražuje određeni direktorij unutar sustava te se provjerava postoje li već generirani podaci o populaciji i o najučinkovitijem pojedincu. Ako postoje učitavaju se i nastavlja s treniranjem gdje, ako ne postoje, stvaraju se novi. Tako je moguće spremiti generaciju koja se uspješno razvila kao i vidjeti izgled neuronske mreže svakog pojedinca. Osim pokretanja i zaustavljanja na sučelju možemo pokrenuti najboljeg pojedinca nakon što smo zaustavili treniranje. Sučelje također prikazuje trenutnu generaciju populacije i najveću vrijednost učinka cijele populacije. Za kontrole tijekom evolucije dodana je brzina izvođenja algoritma. Maksimalna brzina izvođenja ovisi o snazi računalna na kojem se algoritam pokreće. Dodana je mogućnost izmjene vremena trajanja jedne generacije. Nekada je potrebno povećati vrijeme trajanja jednog pokušaja kako bi se stigla svladati svaka postavljena prepreka. Osim toga povećanjem vremena najučinkovitiji pojedinci se mogu usavršiti. Nadalje izrađena je skripta za prikaz neuronske mreže najučinkovitijeg pojedinca. Nakon svake generacije skripta čita podatke najboljeg pojedinca i prikazuje na sučelju (Slika 10). Isto tako dodan je gumb za spremanje trenutne populacije i pojedinca s najboljim učinkom.

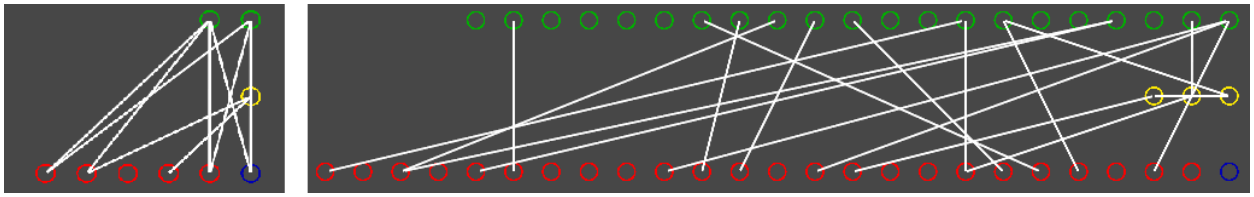


Slika 10. Sučelje tijekom razvoja

Kod treniranja prve video igre, algoritmu u većini slučajeva treba dosta generacija kako bi ispravno svladao stazu. U rijetkim slučajevima algoritam već nakon 20-tak generacija uspije

pogoditi ispravno kretanje i brzo se usavrši. Kao što je već objašnjeno, sve mutacije pojedinca su nasumične pa je i sam napredak učenja nasumičan. Također bi se mogao napraviti bolji sustav izračuna učinka pojedinca. Dosta često određene vrste u populaciji se zabijaju u zid ili se vrte u krug oko jedne točke. To bi se moglo ispraviti s većim brojem kontrolnih točaka ili kada bi svaki pomak u ispravnom smjeru davao pojedincu veći učinak. Isto tako može se povećati i broj senzora na vozilu kako bi se dobilo na preciznosti kretanja, ali onda se smanjuje brzina izvođenja algoritma. Zbog toga pri treniranju uvijek treba osigurati dovoljno informacija pojedincu da može savladati zadatak, a opet ne previše kako vrijeme treniranja ne bi bilo preveliko. Ovakva implementacija nije idealna, ali je dovoljna algoritmu da u određenom broju generacija ispravno izvrši zadatak.

Za razliku od prve igre, druga nije trebala nikakve izmjene pri računanju učinka pojedinca. Pojedinac koji svlada najveći broj prepreka je i najbolji. Tako da se je samo gledala udaljenost koju je pojedinac prešao. Veći problem je stvarao način generiranja svakog člana populacije. Zbog toga što cjelokupno generiranje počinje iz jedne skripte koja stvara druge objekte i treba njima slati podatke o dimenzijama drugih objekata. Također pošto se sve automatski generira svaki objekt treba postaviti na točnu poziciju i rotaciju. Nakon toga dosta je poteškoća stvarala fizika zbog velikog broja zglobova (engl. joints) koji bi stvarali nepredvidiva kretanja tako da je trebalo dobro uravnotežiti sve parametre objekata (masa, brzina kretanja, odstupanja zglobova, itd.). Objekti koji se stvaraju mogu imati od 1 do 10 kotača s različitom veličinom kotača i udaljenosti kotača od središta. Najučinkovitije vozilo se pokazalo vozilo s 2 kotača prosječnih dimenzijama dijelova. Svi pojedinci s većim brojem kotača često su zapinjali na užim preprekama i vrtjeli se na mjestu. Tako da je s ovom igrom pokazan način s kojim NEAT algoritam može stvoriti najboljeg pojedinca za određeni zadatak.



Slika 11. Neuronske mreže najboljih pojedinaca prve (lijevo) i druge (desno) igre nakon treniranja

4. ZAKLJUČAK

U ovom radu uspješno su izrađene dvije video igre pokretane NEAT algoritmom. Za razvoj igara korištena je Unity platforma gdje su skripte pisane u C# programskom jeziku pomoću Microsoft Visual Studia. Korištena je već izrađena implementacija NEAT algoritma u Unity platformi[3].

Definiran je način rada NEAT algoritma kao i sve značajke koje ga čine boljim od ostalih evolucijskih algoritama za određene slučajeve. Navedeni algoritam dijeli populaciju u vrste zbog čega može onemogućiti određenoj vrsti da prevlada populaciju. Pojedinci unutar vrste dijele učinak kako zbog jednog izvrsnog pojedinca vrste ne bi mogle naglo napredovati. Također tijekom evolucije algoritam na svaku promjenu (mutaciju) postavlja oznaku kako se iste inovacije ne bi ponavljale. Sve navedene značajke su objašnjene kao i način na koji bi se implementirale unutar koda. Osim toga prikazan je i primjer strukture podataka za NEAT algoritam u C# programskom jeziku.

Izrađene su dvije video igre gdje svaka igra prikazuje drugačiji način rada NEAT algoritma. Prva video igra izrađena je da prikaže način rada algoritma prilikom promjene okruženja za upravljani objekt. Algoritam za zadatak ima upravljati vozilom s ciljem obilaska cijele staze. Nakon određenog broja generacija algoritam ispravno upravlja vozilom što pokazuje ispravnu implementaciju algoritma. Zatim je izrađena druga igra koja prikazuje kako algoritam može kroz evoluciju odabrati najučinkovitije dimenzije upravljanog objekta. Postavljena je staza s preprekama na kojoj pojedinac s najviše svladanih prepreka dobiva najveći učinak. Algoritam određuje broj kotača vozila, udaljenost kotača od središta vozila te promjer kotača. Tako pomoću NEAT algoritma možemo dobiti vozilo koje je najbolje dizajnirano za savladavanje postavljenih prepreka. Također objašnjen je i cjelokupan proces izrade igara kao i implementacija algoritma u video igre.

Na kraju izrađene su određene izmjene u korisničkom sučelju tijekom treniranja. Korisnici mogu spremiti trenutnu populaciju, upravljati brzinom razvoja mreže, trajanjem jedne generacije te im je omogućen prikaz neuronske mreže najboljeg pojedinca trenutne generacije.

LITERATURA

[1] Kenneth O. Stanley, Risto Miikkulainen: Evolving Neural Networks through Augmenting Topologies, (26.8.2018.)
Link: http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf

[2] SharpNEAT is a complete implementation of NEAT written in C# / .NET, (26.8.2018.)
Link: http://sharpneat.sourceforge.net/

[3] UnityNEAT is a port of SharpNEAT from pure C# 4.0 to Unity, (26.8.2018.)
Link: https://github.com/lordjesus/UnityNEAT

[4] Evolving Neural Networks through Augmenting Topologies, (5.9.2018.)
Link: http://gekkoquant.com/2016/03/13/evolving-neural-networks-through-augmenting-topologies-part-1-of-4/

[5] Efficient Reinforcement Learning through Evolving Neural Network Topologies, (5.9.2018.)
Link: http://nn.cs.utexas.edu/downloads/papers/stanley.gecco02_1.pdf

[6] Using Genetic Algorithms to Evolve Artificial Neural Networks, William T. Kearney, (5.9.2018.)
Link: https://digitalcommons.colby.edu/cgi/viewcontent.cgi?article=1836&context=honorstheses

[7] A Hybrid Genetic Algorithm and Radial Basis Function NEAT, (5.9.2018.)
Link: https://www.researchgate.net/publication/262684992

[8] NeuroEvolution, NEAT Algorithm and My NEAT, (5.9.2018.)
Link: https://medium.com/datadriveninvestor/neuroevolution-neat-algorithm-and-my-neat-b83c5174d8b0

SAŽETAK

Naslov: NEAT algoritam u video igrama

U ovom radu implementiran je NEAT algoritam u video igre. Izrađene su različite vrste video igara kako bi se pokazala različita djelovanja algoritma. Objašnjen je NEAT algoritam kao i njegove bitne značajke. Uz svaku igru objašnjen je način implementacije algoritma kao i proces prilagodbe igara za NEAT algoritam. Također su opisani dijelovi koda ključni za rad s algoritmom te dijelovi koda za prikaz podataka tijekom razvoja mreže. Korisnicima je omogućeno sučelje za prikaz podataka o trenutnom stanju mreže i upravljanje s brzinom razvoja mreže, veličinom populacije i sl.

Ključne riječi: NEAT, video igre, C#, Unity, Visual Studio, Neuronske mreže, genetski algoritam

ABSTRACT

Title: NEAT algorithm in video games

In this thesis NEAT algorithm has been implemented in video games. Different types of video games were created to show different aspects of the algorithm. The NEAT algorithm as well as its essential features are described. The method of algorithm implementation to each game is explained and the process of a game adaptation for NEAT algorithm is shown. It also describes the parts of code that are essential for working with algorithms and parts of code to display data during network development. Users are provided with an interface to view data about the current state of the network and manage the speed of development, population size, etc.

Keywords: NEAT, video games, C#, Unity, Visual Studio, Neural networks, genetic algorithm

ŽIVOTOPIS

Luka Nuić rođen je 21. srpnja 1994. godine u Požegi. Od svog rođenja živi u mjestu Velika. U osnovnoj školi prvi put se susreće s računarstvom te mu to postaje najveća zanimacija. U 2009. godini upisuje smjer računalnog tehničara u tehničkoj školi Požega. Tijekom srednje škole otkriva razne programske jezike kao i razvoj web stranica. Srednju školu završava redovno u 2013. godini i upisuje preddiplomski studij računarstva na Elektrotehničkom fakultetu u Osijeku. Uspješno završava tri godine studija te upisuje diplomski studij računarstva. Obavlja stručnu praksu u tvrtki Mono gdje stječe iskustva s timskim razvojem većih aplikacija.

PRILOZI

Prilog A. Skripta vozila druge igre

```
using System.Collections;
using System.Collections.Generic;
using SharpNeat.Phenomes;
using UnityEngine;

public class Car_Script : UnitController {
    bool IsRunning;
    private int JointCount = 0;
    public GameObject JointPrefab;
    public float JointAngleDelta = 5f;
    IBlackBox box;

    void FixedUpdate()
    {
        if (IsRunning)
        {
            // input array assigned to constant because without it output is
constantly 0.5f
            ISignalArray inputArr = box.InputSignalArray;

            for (var i = 0; i < 21; i++) inputArr[i] = 0.5f;

            box.Activate(); // start network

            ISignalArray outputArr = box.OutputSignalArray; // read outputs

            JointCount = (int)Mathf.Round((float)outputArr[0] * 9 + 1); // get joint
count from output

            for (int i = 0; i < JointCount; i++) // spawning joints
            {
                // get joint position
                float JointPositionX = Mathf.Cos(i * 2f * Mathf.PI / JointCount);
                float JointPositionY = Mathf.Sin(i * 2f * Mathf.PI / JointCount);
                Vector3 JointPosition = new Vector3(JointPositionX, JointPositionY,
0f);

                // get joint rotation
                float JointRotationZ = 90f + i * (360 / JointCount);
                Quaternion JointRotation = Quaternion.Euler(new
Vector3(transform.rotation.x, transform.rotation.y, JointRotationZ));

                //create joint
                GameObject obj = Instantiate(JointPrefab, JointPosition,
JointRotation, transform);

                //change joint scale
                obj.transform.localScale = new Vector3(transform.localScale.x,
transform.localScale.y + (float)outputArr[i + 1], transform.localScale.z);

                //fix position based on scale
                float newObjPositionX = obj.transform.position.x + Mathf.Cos(i * 2f *
Mathf.PI / JointCount) * obj.transform.localScale.y / 3;
                float newObjPositionY = obj.transform.position.y + Mathf.Sin(i * 2f *
Mathf.PI / JointCount) * obj.transform.localScale.y / 3;
```

```
                obj.transform.position = new Vector3(newObjPositionX,
newObjPositionY, obj.transform.position.z);

                //send wheel radius to joint
                obj.GetComponent<JointScript>().SetWheelRadius((float)outputArr[i +
11]);

                SetupHingeJointComponent(obj);
            }

            Stop();
        }
    }

    void SetupHingeJointComponent(GameObject obj)
    {
        HingeJoint2D hj = gameObject.AddComponent<HingeJoint2D>() as HingeJoint2D;
        hj.connectedBody = obj.GetComponent<Rigidbody2D>();
        hj.useLimits = true;

        JointAngleLimits2D limits = new JointAngleLimits2D();
        limits.min = 180f - JointAngleDelta;
        limits.max = 180f + JointAngleDelta;
        hj.limits = limits;
    }

    public override void Stop()
    {
        this.IsRunning = false;
    }

    public override void Activate(IBlackBox box)
    {
        this.box = box;
        this.IsRunning = true;
    }

    public override float GetFitness()
    {
        return transform.position.x > 0 ? Mathf.Round(transform.position.x / 10) : 0;
    }
}
```