SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
V A R A Ž D I N

Toni Steyskal

Ponašanje i komunikacija agenata pomoću stabla odlučivanja

SEMINARSKI RAD

Varaždin, 2016.

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
V A R A Ž D I N

Toni Steyskal

Studij: Informacijsko i programsko inženjerstvo

Ponašanje i komunikacija agenata pomoću stabla odlučivanja

SEMINARSKI RAD

Mentor:

Doc. dr. sc. Markus Schatten

Varaždin, rujan 2016.

Sadržaj

1. Uvod 1

2. Stabla odlučivanja 2

2.1. Prolazak kroz stablo 3

2.2. Način rada stabla 3

2.2.1. Composite 4

2.2.2. Decorator 4

2.2.3. Leaf 5

3. Projekt 6

3.1. Akcije 7

3.1.1. Patrol 7

3.1.2. Follow 10

3.1.3. Shoot akcija 12

3.2. Conditionals 13

3.2.1. Scan conditional 13

3.2.2. Seek conditional 15

3.3. Definirana stabla odlučivanja 16

4. Literatura 19

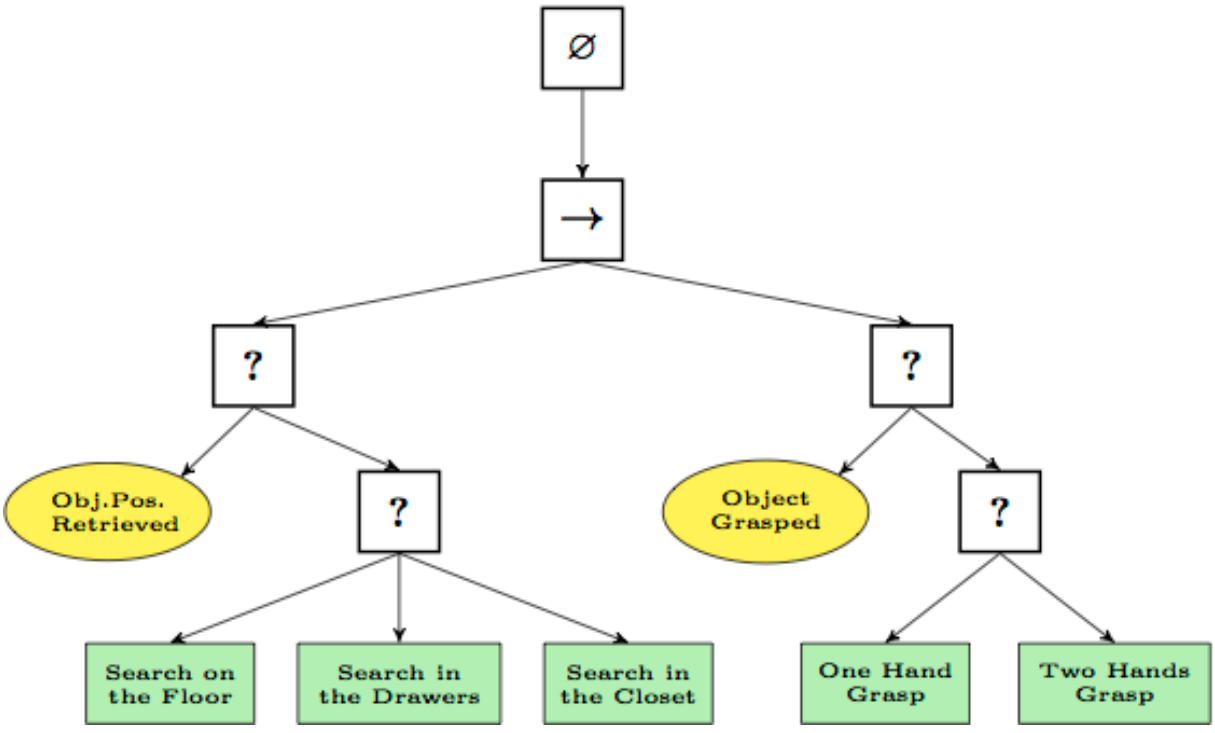
1. Uvod

Industrija video igara predstavlja najveću industriju zabave na svijetu, dio toga jest također i umjetna inteligencija koja se prikazuje preko raznih neprijatelja, ne igrivih likova i raznih sustava unutar video igara. Postoje različita rješenja za postizanje određenih razina kvalitete umjetne inteligencije, te ona s vremenom sve više napreduju. Jedno od takvih rješenja predstavljaju stabla odlučivanja koja izvode razne akcije i ponašanja agenata kako bi se dobilo željeno stanje ili reakcija na događaje unutar igre. Pomoću stabla odlučivanja agenti mogu simulirati kompleksna ponašanja i to rješenje za umjetnu inteligenciju je često primjenjivano u raznim žanrovima i tipovima igara. Kako su stabla odlučivanja česta pojava, i rješenje za umjetnu inteligenciju, ona su odabrana za tematiku ovog seminara. Objasnit će se što zapravo predstavljaju stabla odlučivanja i kako ih koristiti da bi se postiglo željeno ponašanje i reakcija na stimulaciju i prikazat će se njihova upotreba na jednostavnim agentima kao primjer jedne implementacije umjetne inteligencije pomoću alata Unity i programskog jezika C#.

2. Stabla odlučivanja

Stabla odlučivanja predstavljaju hijerarhiju čvorova u obliku stabla, čvorovi definiraju tok odluka s kojima se susreće umjetna inteligencija entiteta u video igri. Krajnji čvorovi, koji ne vode u daljnji dio stabla, se nazivaju lišća i predstavljaju odluke koje će kontrolirati ponašanje entiteta s umjetnom inteligencijom. Unutar stabla odlučivanja pojedini dijelovi formiraju grane koje predstavljaju razne tipove korisnih čvorova koji kontroliraju i utječu na prolazak kroz stablo kako bi se izvršio niz akcija koje najbolje odgovaraju situaciji u kojoj se agent našao.

Pojedina stabla mogu prikazivati duboku hijerarhiju te pritom unutar vlastitih čvorova pozivati pod stabla koja obavljaju specifične funkcije. Na taj način je moguće kreirati biblioteku ponašanja koja se mogu povezati zajedno da stvore jedan način ponašanja entiteta umjetne inteligencije. Razvoj ovog tipa je izuzetno podložan iteraciji, gdje se može početi sa izradom osnovnih ponašanja koja je moguće proširivati novim granama koje se bave alternativnim metodama za postizanje cilja agenta. Pojedine grane unutar stabla mogu se sortirati po prioritetu što omogućava umjetnoj inteligenciji niz rezervnih strategija ukoliko neka od ponašanja i akcije ne uspiju. Ovaj zanimljiv način definiranja umjetne inteligencije entiteta proizlazi iz definiranih prednosti tehnike stabla odlučivanja.



Slika 2.1. *Primjer stabla odlučivanja*

2.1. Prolazak kroz stablo

Posebna specifičnost za prolazak i tok kroz stablo odlučivanja zapravo predstavlja potreba stabla da otkucava mnogo puta tokom igre kako bi se pojedino stablo odlučivanja završilo. U najosnovnijoj implementaciji stabla odlučivanja, agent će prolaziti kroz stablo počevši od korijena stabla svaki pojedini okvir (eng. *frame*) provjeravajući svaki čvor niz stablo kako bi saznao koji je aktivan, provjeravajući bilo koji čvor na koji naiđe putem, dok ne dosegne do trenutno aktivnog čvora kako bi to sve opet ponovio.

Bitno je napomenuti kako ovo ne predstavlja naj efikasniji način rada, pogotovo ako se stablo odlučivanja kroz razvoj video igre sve više proširuje te ide sve više u dubinu.

2.2. Način rada stabla

Stablo odlučivanja se sastoji od različitih tipova čvorova, no postoje osnovne funkcionalnosti koje su jednake za svaki tip čvora unutar stabla odlučivanja. Ovo se odnosi na što ti pojedini čvorovi vraćaju kao status.

Vrste statusa koji se najčešće koriste su:

- uspjeh
- neuspjeh
- rad (eng. *running*)

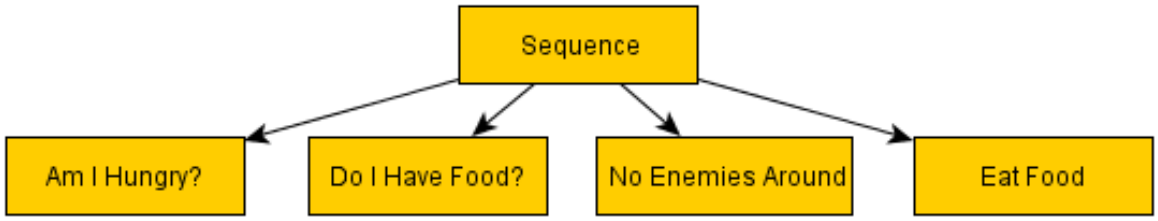
Prve dvije vrste statusa čvora informiraju svog roditelja da li je njihova operacija bila uspješna ili neuspješna. Dok treći status predstavlja informaciju da se uspjeh ili neuspjeh još nije dogodio, što zapravo znači da taj čvor još uvijek izvršava svoju logiku. Prilikom tog statusa čvor je provjeren ponovno prilikom sljedećeg prolaska kroz stabla, te će tada imat novu priliku da vrati uspjeh, neuspjeh ili rad kao status.

Ovakav način radan predstavlja pravu moć stabla odlučivanja jer omogućava da se jedan čvor izvršava kroz više prolaska kroz stablo tokom igre. Dobiveni statusi prilikom prolaska kroz

stablo definiraju tok stabla i pružaju niz događaja i različitih puteva izvršavanja kroz stablo kako bi se postiglo željeno ponašanje agenta umjetne inteligencije.

2.2.1. Composite

Composite čvor predstavlja čvor koji može sadržavati jedno ili više djece, te će provjeriti njihova stanja po definiranom nizu ili slučajnim odabirom. Također ovisno o vrsti čvora ovisno o uspjesima i neuspjesima djece će proslijediti status roditelju. Za vrijeme provjere statusa djece composite čvor vraća status rada. Primjeri vrste composite čvora su sekvenca i paralelni rad.



Slika 2.2.1.1 *Primjer composite čvora*

2.2.2. Decorator

Decorator čvor može sadržavati točno jedno dijete. Funkcija decorator čvora je da promijeni status koji je dobiven od djeteta, da prekine rad djeteta ili da ponavlja rad djeteta, ovisno o vrsti decorator čvora. Primjer vrste decoratora je Inverter koji vraća status koji je suprotan od onog koje je vratilo dijete decorator čvora.

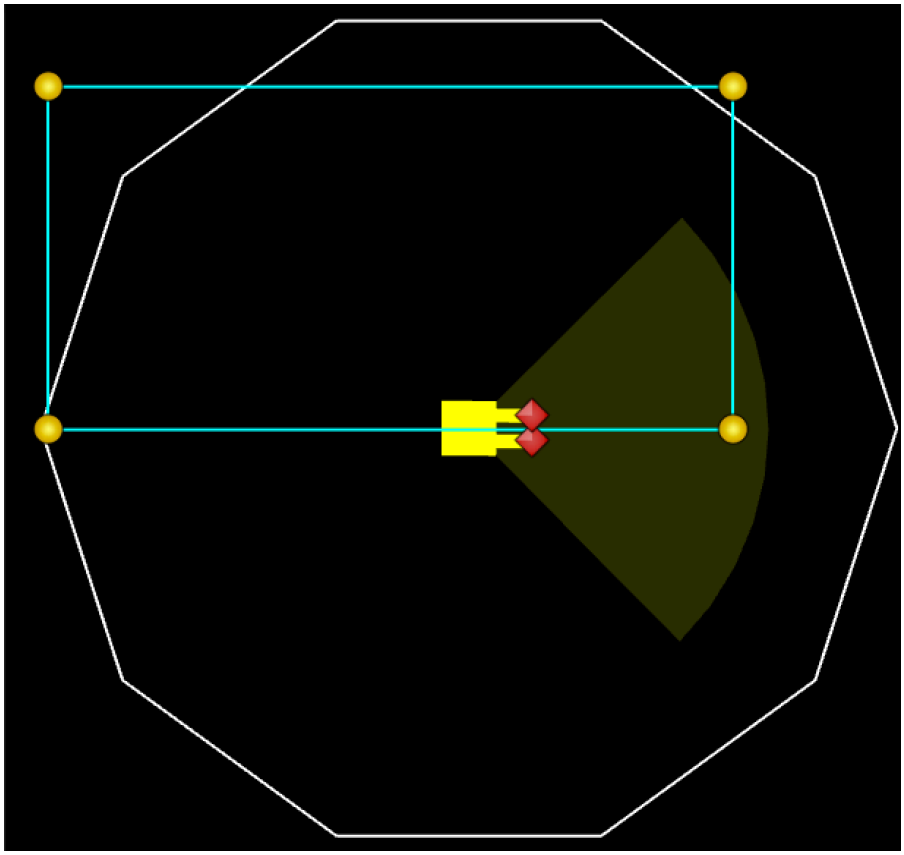
2.2.3. Leaf

Leaf predstavlja najniži tip čvora koji ne može sadržavati djecu, no predstavlja najmoćniji tip čvora. Leaf predstavlja implementaciju akcije koja je specifična za agenta ili općenito za stablo. Kako zapravo ovaj tip čvora omogućava vlastito definiranje akcije koja će se izvesti, on omogućava izradu moćnih stabla odlučivanja koja su u stanju izvesti prioritetna i slojevita ponašanja. To su zapravo čvorovi koji izvode specifičan kod unutar video igre i mogu kontrolirati kretnju i ponašanje agenta. Uz kontrolu agenta ovi čvorovi se mogu koristi i za provjeru okruženja u kojem se agent nalazi te daju određenu povratnu informaciju o daljnjem toku unutar stabla odlučivanja.

Dodatan bitan tip ovog čvora je onaj koji može pozvati drugo stablo odlučivanja te proslijediti podatke trenutnog stabla u novo stablo odlučivanja.

3. Projekt

Za potrebe ovog projekta kreiran je projekt koji se sastoji od vlastitih definiranih leaf (akcija) i conditional čvorova. Pomoću definiranih stabla odlučivanja kreirana su 3 prototipa agenata različitih ponašanja.



Slika 3.1. *Primjer agenta*

3.1. Akcije

U daljnjoj terminologiji akcije predstavljaju zapravo leaf čvorove koji se odnose na specifičan kod koji direktno utječe na sam objekt agenta. Kreirano je nekoliko novih akcija, neke od kojih su:

- Patrol
- Follow
- Shoot

3.1.1. Patrol

Patrol akcija definira kretnju agenta prema postavljenim točkama u prostoru (predefinirana putanja). Ukoliko pojedini agent uspješno završi kretnju definiranog puta status koji se vraća jest uspjeh, dok u suprotnom, ukoliko se agent i dalje kreće, vraća se status rada.

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using BehaviorDesigner.Runtime;
using BehaviorDesigner.Runtime.Tasks;

[TaskCategory ("CustomAI/Movement")]
[TaskDescription ("Patrols the agent around given waypoints." +
"\n@return Success: if the loop is unchecked and the agent has finished its
 route." +
"\n@return Running: if the agent is patrolling.")]
public class Patrol : Action
{
    [@Tooltip ("The movement speed of the agent.")]
    public SharedFloat
        MovementSpeed;
    [@Tooltip ("Deviation allowed when following route waypoints (0 means n
o deviation).")]
    public float
        AllowedRouteDeviation;
    [@Tooltip ("Time the agent will wait when it reaches its next waypoint.
")]
    public float
        WaitAtWaypointTime;
    [@Tooltip ("Should the agent loop this action.")]
    public bool
        Loop;
    [@Tooltip ("The parent of movement waypoints.")]
    public SharedTransform
        WaypointsParent;
    [@Tooltip ("The current position the agent is traveling to.")]
    public SharedVector3
        CurrentWaypointPosition;
    private List<Transform> _waypoints = new List<Transform> ();
    private int _waypointIndex;
    private bool _isMoving;
    private float _moveTime;
    private Transform _transform;
    private Rigidbody2D _rigidbody2D;

    public override void OnAwake ()
    {
        _transform = transform;
        _rigidbody2D = Owner.GetComponentInParent<Rigidbody2D> ();
        for (int i = 0; i < WaypointsParent.Value.childCount; i++)
        {
            _waypoints.Add (WaypointsParent.Value.GetChild (i));
        }

        _isMoving = true;
        FindTheClosestWaypoint ();
    }

    public override TaskStatus OnUpdate ()
    {
        if (Time.time >= _moveTime)
        {
            Move ();
```

```
            if (!_isMoving)
                return TaskStatus.Success;
        }

        return TaskStatus.Running;
    }

    private void FindTheClosestWaypoint ()
    {
        float distance = Mathf.Infinity;
        float localDistance;
        for (int i = 0; i < _waypoints.Count; i++)
        {
            localDistance = Vector3.Magnitude (_transform.position -
 _waypoints [i].position);

            if (localDistance < distance)
            {
                distance = localDistance;
                _waypointIndex = i;
            }
        }
    }

    private void MoveToPosition (Vector3 position)
    {
        Vector2 movementDirection = (Vector2)position -
 _rigidbody2D.position;
        movementDirection.Normalize ();
        _rigidbody2D.MovePosition (_rigidbody2D.position + movementDirectio
n * MovementSpeed.Value * Time.deltaTime);
    }

    private void Move ()
    {
        if ((_waypoints.Count != 0) && (_isMoving))
        {
            CurrentWaypointPosition.Value = _waypoints [_waypointIndex].pos
ition;
            MoveToPosition (CurrentWaypointPosition.Value);

            if (Vector3.Distance (CurrentWaypointPosition.Value, _rigidbody
2D.position) <= AllowedRouteDeviation)
            {
                _waypointIndex++;
                _moveTime = Time.time + WaitAtWaypointTime;
            }

            if (_waypointIndex >= _waypoints.Count)
            {
                if (Loop)
                    _waypointIndex = 0;
                else
                    _isMoving = false;
            }
        }
    }
}
```

Kod 3.1.1.1. *Patrol akcija*

3.1.2. Follow

Follow predstavlja akciju koja pokreće agenta prema nekom drugom objektu odnosno prilikom ove akcije agent će pratiti objekt koji mu je zadan. Ova akcija vraća status rada ukoliko agent i dalje prati svoj zadani objekt i neuspjeh ukoliko više ne postoji objekt koji je agent trebao pratiti.

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using BehaviorDesigner.Runtime;
using BehaviorDesigner.Runtime.Tasks;

#if !(UNITY_4_3 || UNITY_4_4)
using Tooltip = BehaviorDesigner.Runtime.Tasks.TooltipAttribute;
#endif

[TaskCategory ("CustomAI/Movement")]
[TaskDescription ("Follows the target." +
"\n@return Running: if the agent is following the target." +
"\n@return Failure: if there is no target set or scanned to be followed.")]
public class Follow : Action
{
    [@Tooltip ("The movement speed of the agent.")]
    public SharedFloat
    MovementSpeed;
    [@Tooltip ("The distance the agent will keep while following.")]
    public float
        DistanceToKeep;
    [@Tooltip ("The target that the agent is following.")]
    public SharedTransform
        Target;

    private Vector3 _offsetTarget;

    private Transform _transform;
    private Rigidbody2D _rigidbody2D;

    public override void OnAwake ()
    {
        _transform = transform;
        _rigidbody2D = Owner.GetComponentInParent<Rigidbody2D> ();
    }

    public override TaskStatus OnUpdate ()
    {
        if (Target.Value != null)
            Move ();
        else
        {
            Debug.LogWarning ("There is no target set, or scanned to be fol
lowed.");
            return TaskStatus.Failure;
        }
```

```
        return TaskStatus.Running;
    }

    private void MoveToPosition (Vector3 position)
    {
        Vector2 movementDirection = (Vector2)position -
 _rigidbody2D.position;
        movementDirection.Normalize ();

        _rigidbody2D.MovePosition (_rigidbody2D.position + movementDirectio
n * MovementSpeed.Value * Time.deltaTime);
    }

    private void Move ()
    {
        if (Vector3.Distance (Target.Value.position, _transform.position) >
 DistanceToKeep)
        {
            _offsetTarget = Target.Value.position -
 (DistanceToKeep * (Target.Value.position -
 _transform.position).normalized);
            MoveToPosition (_offsetTarget);
        }
        else
        {
            _rigidbody2D.velocity = Vector2.zero;
            _rigidbody2D.angularVelocity = 0.0f;
        }
    }
}
```

Kod 3.1.2.1. *Follow akcija*

3.1.3. Shoot akcija

Shoot akcija predstavlja jednostavnu akciju koja samo obavještava agenta da puca iz svojih oružja. Prilikom svakog uspješnog izvođenja vraća uspjeh, te ova akcija ne može vratiti drugi status.

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using BehaviorDesigner.Runtime;
using BehaviorDesigner.Runtime.Tasks;

#if !(UNITY_4_3 || UNITY_4_4)
using Tooltip = BehaviorDesigner.Runtime.Tasks.TooltipAttribute;
#endif

namespace BehaviorDesigner.LastEncounter
{
    [TaskCategory("CustomAI")]
    public class Shoot : Action
    {
        private Shooter _shooter;

        public override void OnAwake ()
        {
            _shooter = Owner.GetComponentInParent<Shooter> ();
        }

        public override TaskStatus OnUpdate ()
        {
            _shooter.Shoot ();

            return TaskStatus.Success;
        }
    }
}
```

Kod 3.1.3.1. *Shoot akcija*

3.2. Conditionals

Kako je u ovom primjeru potrebno da agent bude osviješten o svojoj meti preko definiranog vidnog polja definirani su dodatni čvorovi koji daju informaciju o postojanju mete unutar vidnog polja agenta.

3.2.1. Scan conditional

Scan conditional potražuje metu preko skenera agenta, koji mu ujedno služe i kao vidno polje. Vraća status uspjeha ukoliko pronađe drugi objekt te postavlja informaciju o istom, a ukoliko ne pronađe metu vraća status neuspjeha.

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using BehaviorDesigner.Runtime;
using BehaviorDesigner.Runtime.Tasks;

#if !(UNITY_4_3 || UNITY_4_4)
using Tooltip = BehaviorDesigner.Runtime.Tasks.TooltipAttribute;
#endif

namespace BehaviorDesigner.LastEncounter
{
    [TaskCategory("CustomAI/Scanners")]
    [TaskDescription("Scans for a target using given scanners." +
                     "\n@return Success, if a target is found (sets Target
to traget found)." +
                     "\n@return Failure, if no targets are found.")]
    public class Scan : Conditional
    {
        public SharedTransform Target;

        private FieldOfView _fieldOfView;

        public override void OnStart()
        {
            Target.Value = null;

            _fieldOfView = Owner.GetComponentInChildren<FieldOfView> ();
        }

        public override TaskStatus OnUpdate()
        {
            if (!Target.Value)
            {
                Target.Value = _fieldOfView.FindVisibleTarget();
            }
            else
            {
```

```
                if (!_fieldOfView.SeekVisibleTarget(Target.Value))
                {
                    Target.Value = null;
                }
            }

            if (Target.Value)
                return TaskStatus.Success;

            return TaskStatus.Failure;
        }
    }
}
```

Kod 3.2.1.1. *Scan conditional*

3.2.2. Seek conditional

Logički gledano seek conditional je vrlo sličan scan conditional čvoru kako koristi iste resurse no vraća drugačiji status ovisno o internoj logici. Ovaj čvor vraća status uspjeha ukoliko se prije definirana meta i dalje nalazi unutar dosega skenera agenata, a vraća status neuspjeha ako taj objekt nije više unutar vidnog polja, a prethodno je bio.

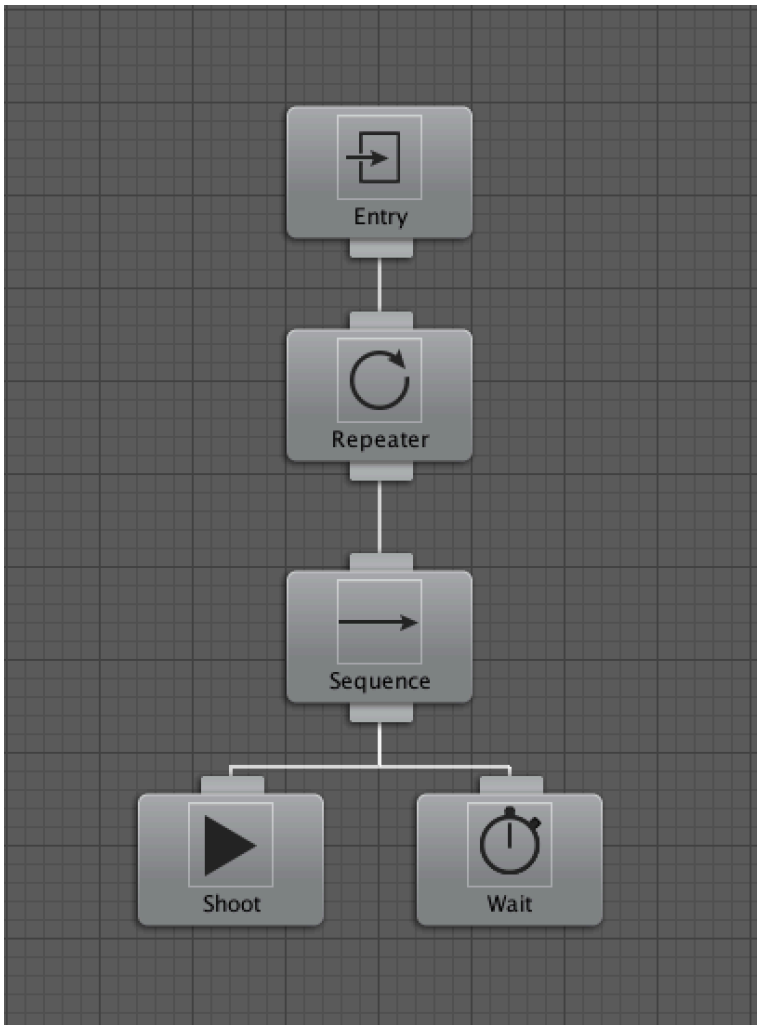
```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using BehaviorDesigner.Runtime;
using BehaviorDesigner.Runtime.Tasks;

#if !(UNITY_4_3 || UNITY_4_4)
using Tooltip = BehaviorDesigner.Runtime.Tasks.TooltipAttribute;
#endif

namespace BehaviorDesigner.LastEncounter
{
    [TaskCategory("CustomAI/Scanners")]
    [TaskDescription("Seeks an existing target using given scanners." +
                     "\n@return Success, if the target is in scan range." +
                     "\n@return Failure, if target is no longer in scan ran
ge.")]
    public class Seek : Conditional
    {
        public SharedTransform Target;

        private FieldOfView _fieldOfView;

        public override void OnStart()
        {
            _fieldOfView = Owner.GetComponentInChildren<FieldOfView> ();
        }

        public override TaskStatus OnUpdate()
        {
            if (Target.Value != null)
            {
                if (_fieldOfView.SeekVisibleTarget(Target.Value))
                {
                    return TaskStatus.Success;
                }
            }
            else if(Target.Value == null)
            {
                Debug.LogWarning("There is no target set, or scanned to be
seeked.");
            }

            return TaskStatus.Failure;
        }
    }
}
```

Kod 3.2.2.1. *Seek conditional*

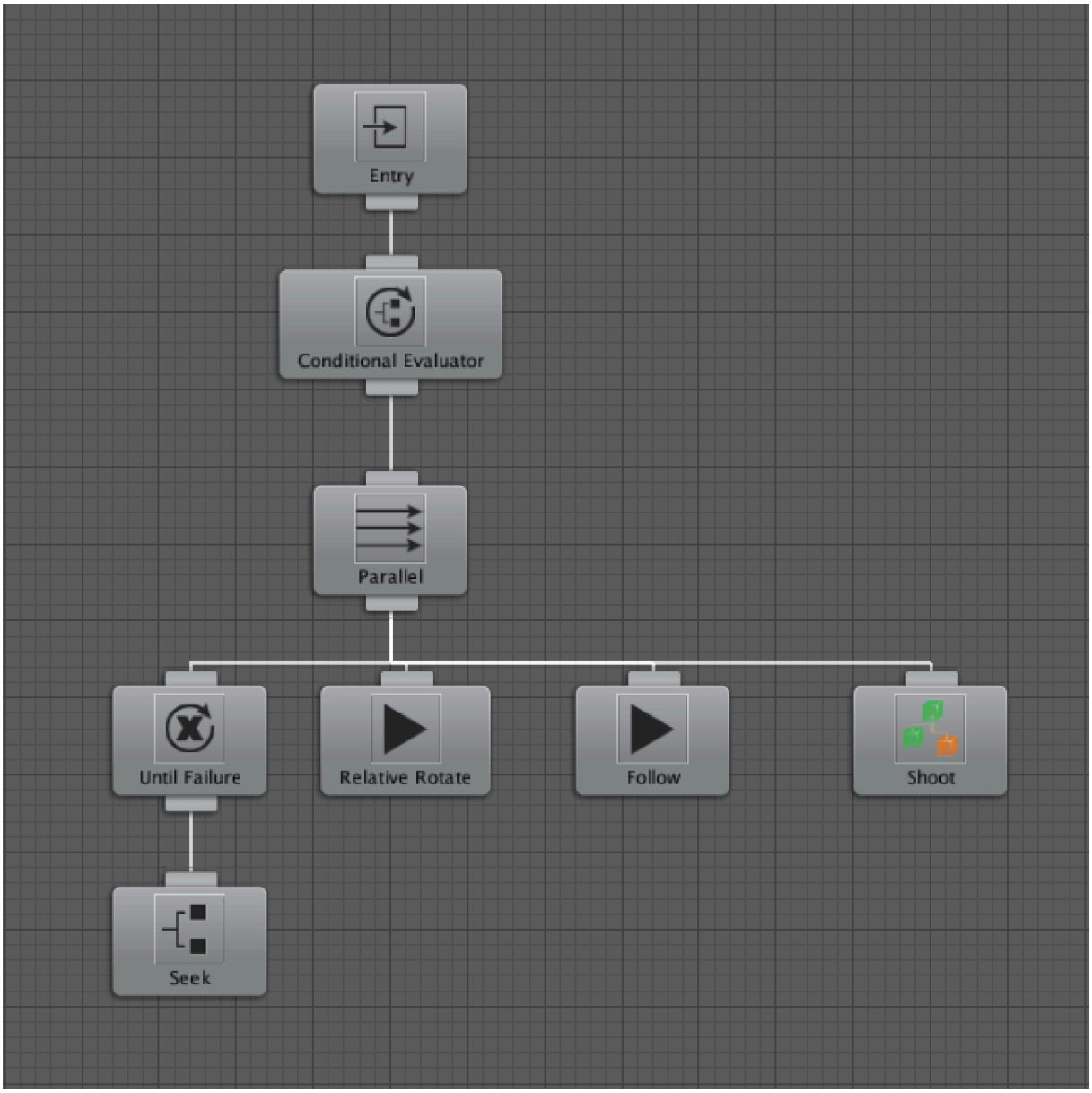
3.3. Definirana stabla odlučivanja

Kako postoje različita ponašanja agenata ona su definirana preko stabla odlučivanja te se mijenjaju kao primarno stablo agenta ovisno o njegovim statusima čvorova prijašnjih stabala. U narednim slikama će biti prikazane neka od svih stabla odlučivanja unutar projekta. Za potrebu ove logike kreirana su sljedeća stabla:

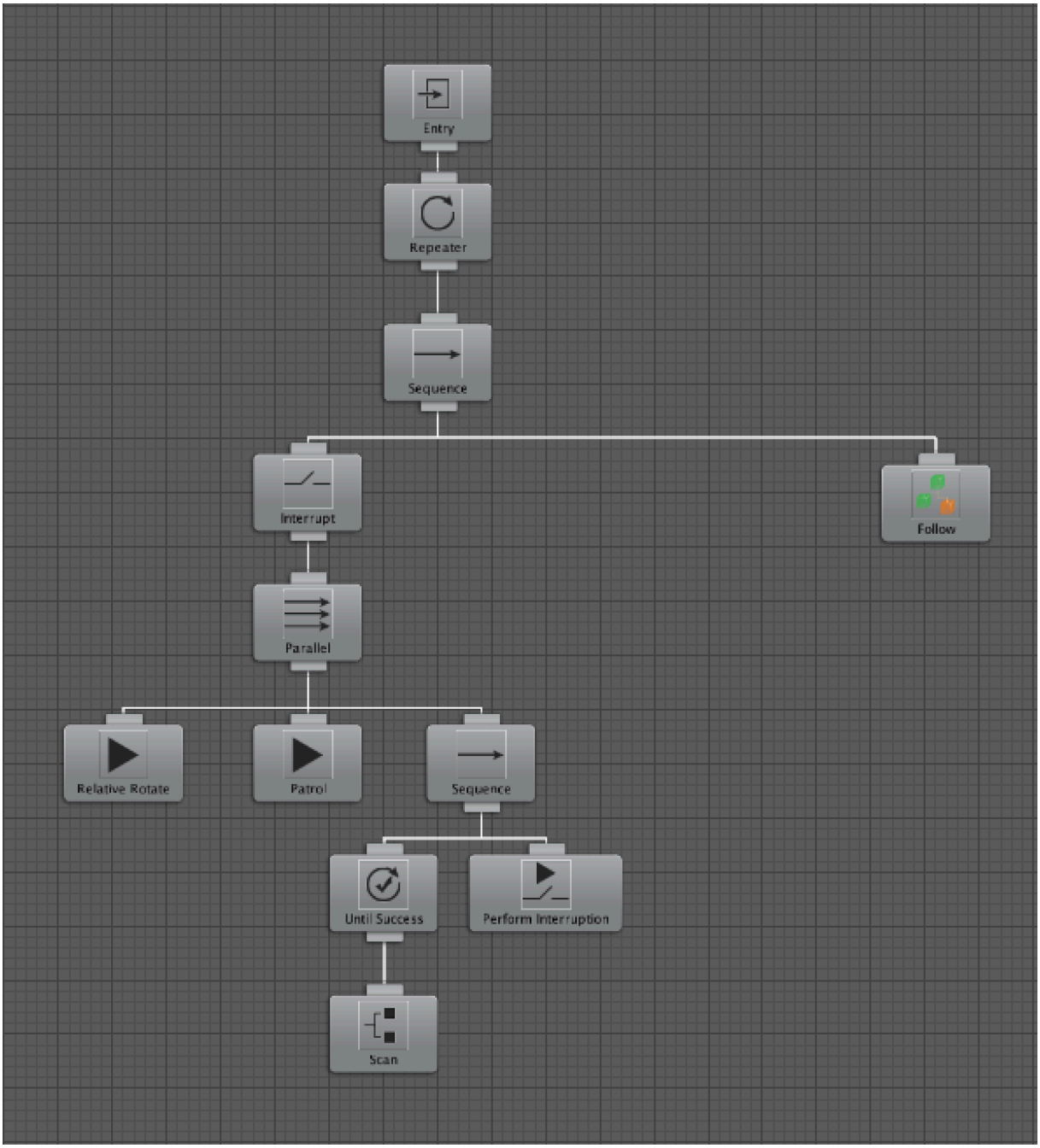
- FleeAlerted
- FollowAlerted
- Follow
- Patrol
- Shoot
- Turret
- TurretAlerted



Slika 3.3.1 *Shoot stablo odlučivanja*



Kod 3.3.2. *Follow stablo odlučivanja*



Kod 3.3.3. *Patrol stablo odlučivanja*

4. Literatura

[1] Simpson C. (2014), *Behavior trees for AI: How they work.* Dostupno 23.08.2016. na http://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php

[2] Wikipedia (2016), *Behavior tree (artificial intelligence, robotics and control).* Dostupno 23.08.2016. na https://en.wikipedia.org/wiki/Behavior_tree_(artificial_intelligence,_robotics_and_control)

[3] Opsive (2016), *Behavior Designer.* Dostupno 23.08.2016. na http://www.opsive.com/assets/BehaviorDesigner/