TAMPEREEN
AMMATTIKORKEAKOULU

HALLINNOIMATTOMAT DLL-TIEDOSTOT JA AUTOMAATIOTESTAUS

Lasse Jukola

Opinnäytetyö
Marraskuu 2015
Tietotekniikan koulutusohjelma
Ohjelmistotekniikka

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietotekniikan koulutusohjelma
Ohjelmistotekniikka

Jukola Lasse:
Hallinnoimattomat dll-tiedostot ja automaatiotestaus

Opinnäytetyö 19 sivua, joista liitteitä 2 sivua
Marraskuu 2015

Työn tarkoituksena oli tutustua hallinnoimattomien dll-tiedostojen käyttöön pelien automaatiotestauksessa. Hallinnoimattomien dll-tiedostojen avulla toteutetun automaatiotestaustekoälyn korvaaminen olisi mahdollista ilman itse pelin koodin uudelleen kääntämistä, jolloin pelin eri osia voitaisiin testata erilaisilla tekoälyillä. Tämä mahdollistaa testaustekoälyn iteroivan kehityksen itsenäisesti itse pelin kehityksen ohessa. Tätä varten toteutettiin pieni peliprojekti jonka tekoäly on toteutettu hallinnoimattomilla dll-tiedostoilla, jotka ladataan pelin ajonaikana.

Työssä käytetty tekoälyn toteutustapa on mahdollinen. Tekoäly on mahdollista ladata ajon aikana ja se toimii täysin itsenäisenä kokonaisuutena. Rajapinnan suurimpana ongelmana on tiedonsiirron kankeus pelin ja tekoälyn välillä. Tietojen muuntaminen sopivaan objektimuotoon vaatii huolellisuutta ja sen päivittäminen on vaivalloista. Siirtoa voidaan parantaa huomattavasti serialisoimalla data tekstimuotoiseksi ennen siirtoa.

Asiasanat: automaatiotestaus, dll-tiedosto, tekoäly

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree programme in ICT Engineering
Software Engineering

Jukola Lasse:
Unmanaged dll-files and test automation

Bachelor's thesis 19 pages, appendices 2 pages
November 2015

The purpose of this thesis was to explore automated game testing with unmanaged dll-files. Artificial intelligence created with dll-files would make it possible to replace it without recompiling the main game project. This would enable independent iterative development process of game and its testing artificial intelligence. For this purpose, small game project that implements artificial intelligence that can be dynamically loaded during run-time.

Implementation method of the artificial intelligence was successful. Artificial intelligence can be loaded during run-time as independent component. Interface has one significant problem, Data format being extremely rigid. This makes data updates into an arduous task. This could be significantly improved by serializing the data into text format before transferring it.

Key words: dll-file, test automation, artificial intelligence

SISÄLLYS

LYHENTEET JA TERMIT

DLL	Dynamic-link library, Dynaaminen-linkkikirjasto
RTS	Real time strategy, Reaaliaikastrategiapeli
XML	Extensible Markup Language, laajennettava merkintäkieli

1 JOHDANTO

Dll-tiedostoilla (Dynamic link library) voidaan jakaa ohjelmien kesken uudelleenkäytettävää kooditoteutuskokonaisuutta. Dll-tiedostoja käytetään peleissä erikseen toteutettujen kokonaisuuksien hallinnoimiseen ja liikuttamiseen. Esimerkiksi pelin konsoliohjaintuki olisi yksi kokonaisuus, joka voidaan koota ja siirtää projektista toiseen yksittäisenä dll-tiedostona.

Työn tarkoituksena oli tutustua dll-tiedostojen käyttöön automaatiotestauksessa. Tavoitteena oli toteuttaa yksinkertainen tekoäly, jonka logiikka toimii mahdollisimman itsenäisenä kokonaisuutena itse pelin rajapinnan ulkopuolelta. Työssä analysoidaan toteutuksen käyttökelpoisuutta erityisesti mobiiliapplikaatioiden testauksessa. Työssä pohditaan myös toteutuksen mahdollisia jatkokehitys mahdollisuuksia.

Työn peli on toteutettu Unity 3D -pelimoottorilla. Tekoäly on koottu dll-tiedostoksi Visual studio 2012:lla. Peli ja tekoäly ovat molemmat toteutettu C# -ohjelmointikieltä käyttäen.

2 AUTOMAATIOTESTAUS

Automaatiotestaus käsittää laajan skaalan erilaisia tekniikoita joista yksinkertaisin on yksikkötestit. Yksikkötesteillä testataan yksittäisen funktion toimintaa kutsumalla sitä ja vertaamalla sen aiheuttamaa ohjelman tilan muutosta (esimerkiksi muuttujan arvon vaihtuminen) tunnettuihin vertailuarvoihin. Yksikkötestit suoritetaan yleensä automaattisesti ohjelman kääntämisen yhteydessä, mutta ne voidaan myös suorittaa manuaalisesti. Yksikkötesteillä on kuitenkin hankalaa jollei mahdotonta löytää virheitä, jotka aiheutuvat useiden eri toimintojen kautta ellei näitä kaikkia tilanteita olla osattu ottaa huomioon jo yksikkötestejä kirjoittaessa. Tästä syystä yksikkötestit eivät voi olla kattavampia kuin niiden kirjoittaja on ennalta osannut ottaa huomioon. Tällöin yksikkötestit soveltuvat parhaiten reunaehto- ja regressiotestaukseen, mikä auttaa ehkäisemään samojen ongelmien toistumisen uudelleen ohjelmistokehityksen myöhemmässä vaiheessa. (Microsoft)

Laajempia kokonaisuuksia voidaan testata funktionaalisella testauksella, joka koostuu testi tilanteista, joilla varmistetaan osan tai koko ohjelman oikea toiminta. Tämä voidaan toteuttaa nauhoittamalla testaustilanne (käyttöliittymätestaus) ja toistamalla se ohjelmallisesti. Vaihtoehtoisesti voidaan testitapaus simuloida kokonaan sarjalla funktiokutsuja. Tämän työn tarkoituksena on tutkia mahdollisuutta toteuttaa tämä simulaatio kokonaan varsinaisen ohjelman ulkopuolelta dll-tiedostoilla. Näin saavutettaisiin erotus varsinaisen pelin ja sitä testaavan automaation välille. Lisäämällä tälle simulaatiometodille logiikkaa tehdä päätöksiä ohjelman tilan perusteella automaatio muuttuu kapeaalaiseksi tekoälyksi.

Automaatiotestauksen simulaatio tekoälyä parantamalla voidaan toteuttaa automaattipelaaja. Automaattipelaaja osaa ideaalitapauksessa pelata peliä ilman ihmisen vuorovaikutusta. Tällä tavoin toteutettu tekoäly voi pelata peliä yli-inhimillisellä nopeudella ja jopa ennen kuin pelin graafinen ympäristö mahdollistaa ihmispelaajan testaamisen. Automaattipelaajan etu ihmistestaajan nähden on juuri sen nopeus ja tämän mahdollistama toistojen määrä. Automaatiotestaus sopii siten esimerkiksi vuoropohjaisen pelin taistelu permutaatioiden testaamiseen. Esimerkiksi jos pelissä pelaajalla on kymmenen sankaria joista yhteen taisteluun voidaan ottaa neljä kerrallaan. Esimerkin eri sankaripermutaatioiden määrä on 210, jolloin jokaisen tapauksen manuaalinen läpikäyminen on käytännössä mahdotonta.

3 DLL-TIEDOSTO

Dll-tiedostot ovat linkkikirjastoja, jotka on tehty alun perin Windows-käyttöjärjestelmää varten. Niitä käytettiin resurssien, kuten graafisen käyttöliittymän ominaisuuksien jakamiseen, versioiden yhtenäistämiseen ja niiden dynaamisen lataamiseen. Tämä mahdollistaa sen, että jokaisen ohjelman ei tarvitse ladata tätä toiminnallisuutta muistiin useaan kertaan sekä varmistetaan ohjelmien toiminnallinen yhtenäisyys. Tämä taas vähentää ajonaikaisen muistin kulutusta ja mahdollistaa ladattujen resurssien yhteisen käytön. Resurssien jakamisen hyvänä puolena on niiden päivittämisen helppous, koska dll-tiedosto tarvitsee korvata vain yhdestä paikasta (Microsoft: What is a DLL?). Sama ominaisuus myös aiheuttaa jaettujen resurssien suurimman ongelman. Ohjelmistot eivät ole itsenäisiä kokonaisuuksia, jolloin uuden ohjelman toiminnan vaatiessa tietyn version dll-tiedostosta sen asentaminen saattaa aiheuttaa usean muun ohjelman toiminnan häiriintymisen tai kokonaan hajoamisen. Näiden ongelmien jäljittäminen saattaa olla erittäin hankalaa etenkin jos ongelma aiheutuu pitkän riippuvuusketjun aikana. (Dynamic Link Libraries)

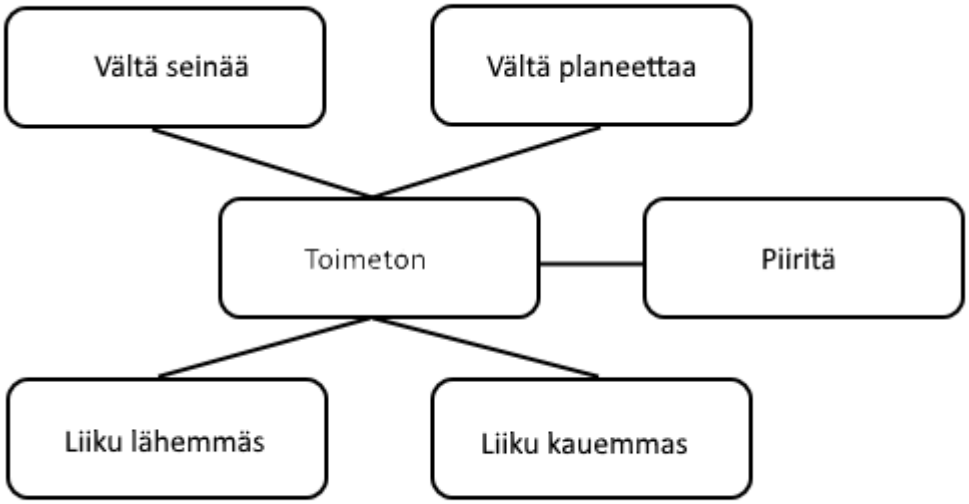
Dll-tiedostojen ominaisuudet, joita tässä työssä hyödynnetään, ovat niiden ajon aikainen lataaminen sekä korvaaminen itse pääohjelman kääntämistä. Työn dll-tiedostot on toteutettu hallinnoimattomasti. Tämä tarkoittaa että niillä on mahdollista tehdä käyttöjärjestelmää vahingoittavia toimintoja (Microsoft: Secure Coding Guidelines). Tämä on riski jos käytetään tuntemattomien ulkopuolisten lähteiden toteuttamia dll-tiedostoja. Riskiä voidaan pienentää jos niiden lähdekoodi on vapaasti saatavilla, mutta tämä taas vähentää yleispätevän kaupallisen

4 TEKOÄLY

Puhuttaessa tekoälystä voidaan tarkoittaa todella erilaisia asioita. Yksinkertaisimmallaan tekoälyllä tarkoitetaan mitä tahansa toteutusta joka tekee päätöksiä parametrien mukaan. Tekoälyllä voidaan myös tarkoittaa. ohjelmistoa, joka on tehty ratkaisemaan monimutkaisia ongelmia useilta aloilta tai jopa mallintamaan ihmisen kaltaista älykkyyttä (Goertzel & Pennachin 2007, 1). Pelien yhteydessä kuitenkin yleisesti tarkoitetaan erikoistunutta kapeaa tekoälyä, jonka tärkeimpänä tehtävänä on luoda mielekäs kokemus pelaajalle sen sijaan että tämä toimisi oikeasti älykkäällä tavalla (Portnow & Floyd 2012).

Tässä työssä keskitytään pääasiassa kapeaan tekoälyn toteutukseen. Projektin tekoälyn ei ole tarkoitus olla automaatiotestaustekoäly vaan toimia todisteena konseptin toteutettavuudesta. Tämän työn tekoälykomponentti on tehty komentajalähtöisesti. Tämä tarkoittaa sitä, että komentajaobjekti arvioi pelitilasta parhaan mahdollisen strategian (Rabin 2002 233–236). Tekoälykomponentin yksinkertaistamiseksi komentaja laskee myös komennot jokaiselle yksikölle erikseen, sen sijaan että jokaiselle yksikölle annettaisiin tavoite ja se laskee itse parhaan tavan saavuttaa kyseinen tavoite. Tällä tavalla toteutettu tekoälykäyttäytyminen on helpommin hallittavissa ja ennustettavissa, mutta sen sijaan tekoälyn laajentaminen ja muuttaminen toisenlaiseen peliin on hankalampaa verrattuna enemmän modulaariseen toteutukseen.

Projektin tekoäly toteuttaa hyvin yksinkertaista tilakonetta (kuva 1). Tilakoneessa ensisijaisesti toteutetaan esteiden väistöä, koska ne aiheuttavat aluksen välittömän tuhoutumisen. Sitten tilakone painottaa vastustajien lähestymistä, yksittäisen aluksen piirittämistä ja sen nopeaa tuhoamista. Tilakone mahdollistaa myös aluksen vetäytymisen kun aluksen elämäpisteet ja kilvet ovat tarpeeksi alhaisella tasolla.



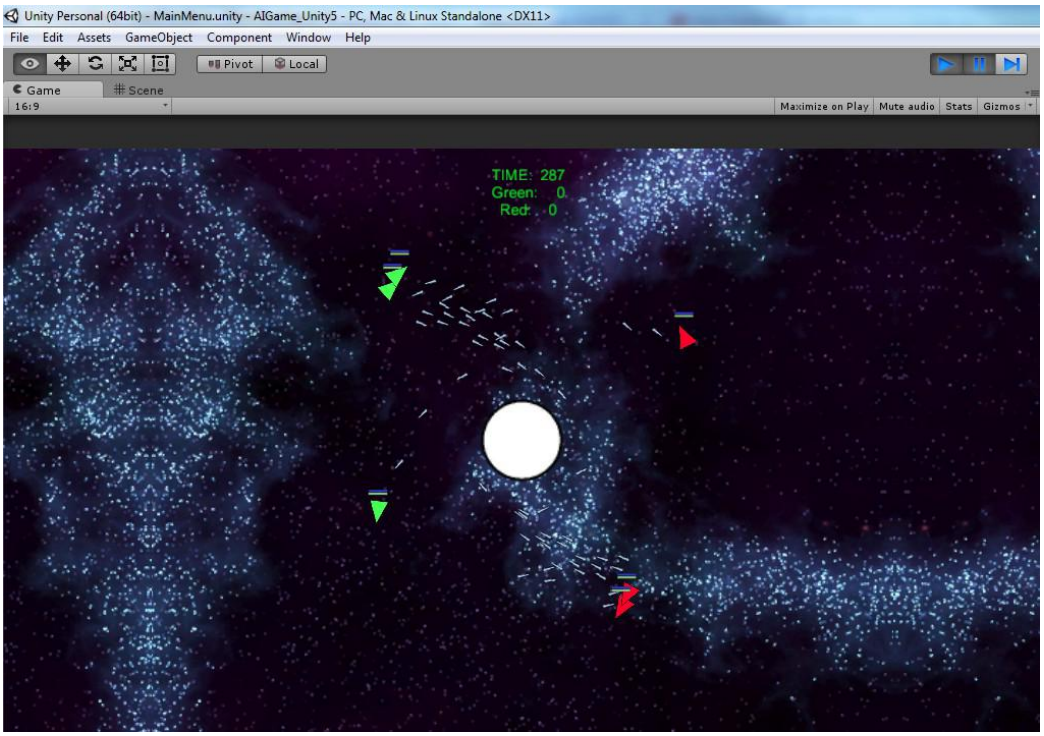
KUVA 1. Kuvaus projektissa käytetystä tilakoneesta.

Ajonaikaista tekoälyn lataamista voidaan hyödyntää esimerkiksi siten, että yhdellä tekoälyllä on kyky navigoida pelimaailmaa ja toinen tekoäly vastaa taisteluista. Tällöin yksittäisen ominaisuuden, kuten taisteluiden muuttuessa ainoastaan yksi osa testaustekoälystä täytyy korvata. Näin tekoälyn eri osia voidaan kehittää toisistaan riippumattomasti ja uuden osa-alueen lisääminen on yksinkertaisempaa, kuin jos koko tekoäly olisi yksi iso kokonaisuus.

5 PROJEKTI

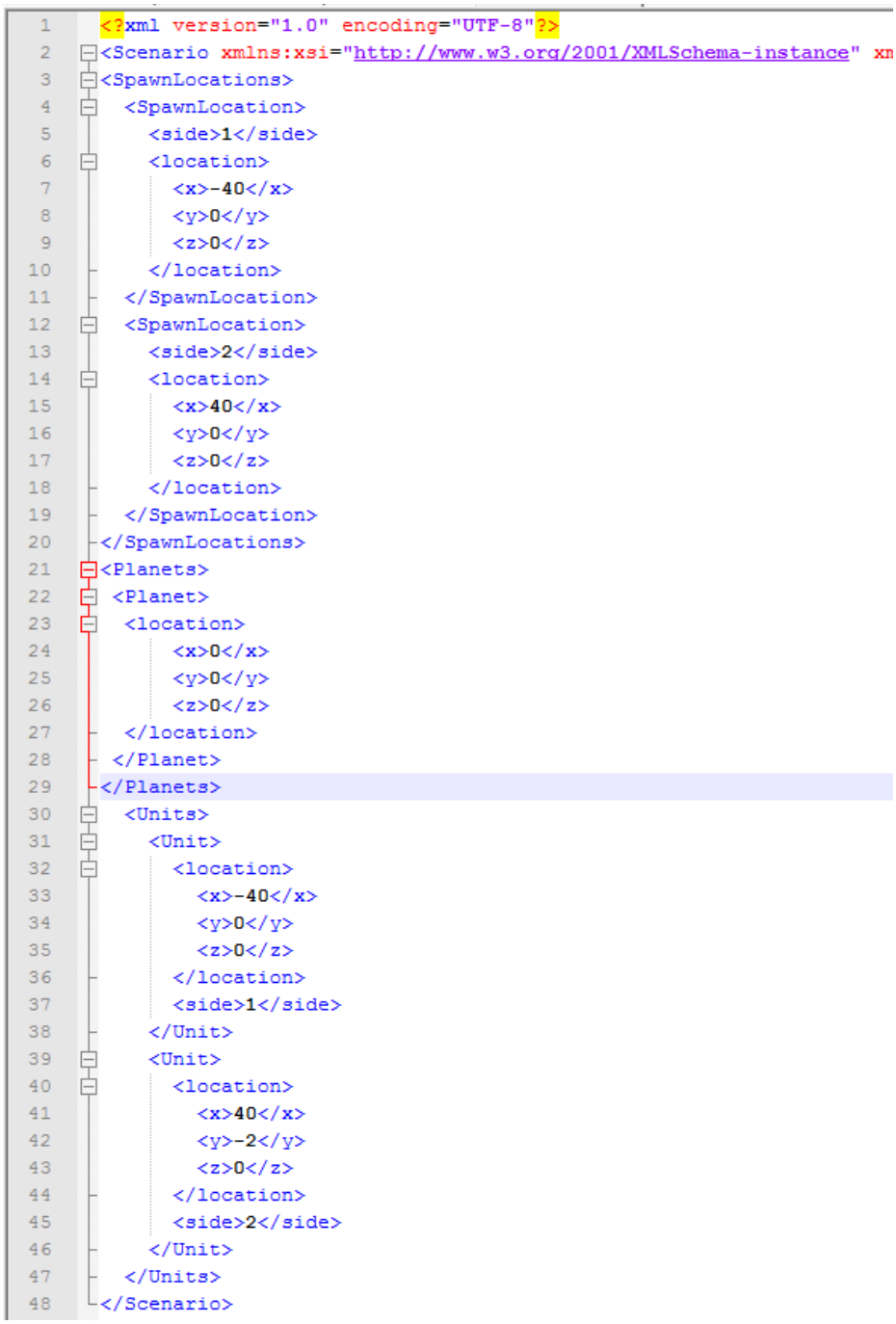
Työtä varten toteutettu peli on isometrinen 2D-avaruusreaaliaikastrategia, jossa 2 joukkuetta kilpailee pisteistä, joita saa vastustajien aluksia tuhoamalla aikarajan puitteissa. Alukset syntyvät uudelleen lyhyen ajan kuluttua tuhoutumisesta niille määritetyissä pisteissä. Aluksilla on elämän lisäksi kilpi, joka alkaa latautua uudelleen jos alus ei ota vahinkoa lyhyen ajan sisällä.

Pelikenttä muodostuu suorakaiteen muotoisesta alueesta, joka rajoittaa aluksien liikkumista. Kartalla on mahdollista olla useita planeettoja, joihin törmäys aiheuttaa törmänneen aluksen välittömän tuhoutumisen. (kuva 2)



KUVA 2. Esimerkki pelitilanteesta.

Pelin päävalikosta voidaan valita mitä tekoälykomponentteja ja skenaariotiedostoa halutaan käyttää. Tämän skenaariotiedoston parametreja pystyy muokkaamaan editoimalla pelin ulkopuolella olevaa xml-tiedostoa (kuva 3). Tiedostoa muokkaamalla voidaan pelissä testata lukemattomia erilaisia tilanteita. Tällä tavalla voidaan varmistaa toteutetun tekoälyn soveltaminen mitä erilaisimpien tilanteiden esittämiin haasteisiin.



```
<?xml version="1.0" encoding="UTF-8"?>
<Scenario xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xm
<SpawnLocations>
  <SpawnLocation>
    <side>1</side>
    <location>
      <x>-40</x>
      <y>0</y>
      <z>0</z>
    </location>
  </SpawnLocation>
  <SpawnLocation>
    <side>2</side>
    <location>
      <x>40</x>
      <y>0</y>
      <z>0</z>
    </location>
  </SpawnLocation>
</SpawnLocations>
<Planets>
 <Planet>
  <location>
      <x>0</x>
      <y>0</y>
      <z>0</z>
  </location>
 </Planet>
</Planets>
  <Units>
    <Unit>
      <location>
        <x>-40</x>
        <y>0</y>
        <z>0</z>
      </location>
      <side>1</side>
    </Unit>
    <Unit>
      <location>
        <x>40</x>
        <y>-2</y>
        <z>0</z>
      </location>
      <side>2</side>
    </Unit>
  </Units>
</Scenario>
```

KUVA 3. Esimerkki scenario xml:stä.

6 RAJAPINTA

Rajapintatoteutuksessa harkittiin kahta erilaista toteutusta. Ensimmäinen tapa oli toteuttaa rajapinta siten, että siinä on yksi funktio, jossa on parametrina yksi serialisoitu string-muuttuja ja reflektiofunktiot. Tämä string-muuttuja purettaisiin tekoälyn puolella ja se sisältää pelissä tapahtuneen eventin nimen ja kaikki oleelliset tiedot, joita tekoäly tarvitsee tehdäkseen päätökset. Toinen toteutustapa, jota päädyttiin käyttämään on hyvin lähellä ensimmäistä. Siinä siirrettävä data on valmiiksi muokattu tekoälyn käyttämään muotoon. Tämä yksinkertaistaa huomattavasti tekoälyn toteutusta.

Komponenttien välinen tiedonsiirto on ongelmallista. Hallinnoimaton dll-tekoäly komponentti ja pelialusta ei voi suoraan käsitellä toistensa määrittämiä olioita, koska niitä ei ole määritelty yhdessä käännösvaiheessa. Tästä syystä tieto täytyy siirtää joko primitiivisinä datatyyppeinä (int, string) tai serialisoida tieto molempien ymmärtämään muotoon (Json, xml). Mahdollista on myös ajon aikana etsiä tekstimuodossa rajapinnasta vastaava dataluokka, jonka mukaisina alustetaan tiedot sisältävä olio. Haettava luokka on käytännössä sama, joka on määritelty Unityn puolella, mutta olio on luotu ladatusta dll-tiedostosta, jolloin sen käyttö on yksinkertaista tekoälykomponentin puolella (Kuva 4).

DLLBridge.cs | PlanetData.cs | GameManager.cs | DLLInterface.cs | InputHandle

DLLBridge ▸ SendDataToDLL ()

```
145         try
146         {
147             //Team 1 (green) data send
148             {
149                 Action<object[]>callBack1 = Team1CommandDataCallback;
150                 Action callBack2 = Team1CommandDoneCallback;
151                 object[] unitData = new object[gameunitData.Count];
152                 Type type = team1AIDLL.GetType("AIDLL.GameUnitData");
153                 for(int i = 0; i< gameunitData.Count; i++)
154                 {
155                     unitData[i] = Activator.CreateInstance(type,DataTransferForm(gameunitData[i]));
156                 }
157
158                 //planetData
159                 object[] planetDLLData = new object[planetData.Count];
160                 if(planetData != null)
161                 {
162                     Type planetDatatype = team1AIDLL.GetType("AIDLL.PlanetData");
163                     for(int i = 0; i< planetData.Count; i++)
164                     {
165                         planetDLLData[i] = Activator.CreateInstance(planetDatatype,
166                                         (object)planetData[i].position, (object)planetData[i].radius);
167                     }
168                 }
169
170                 object[] teamArgs = new object[]{ unitData, 1, greenScore, RedScore,
171                                                 GameManager.Instance.RoundTimeLeft(),
172                                                 topCorner, botCorner, planetDLLData, callBack1, callBack2};
173                 team1CommandData.Clear();
174                 team1CommandsDone = false;
175
176                 team1AIDLLInstance.GetType().GetMethod("SendData").Invoke(team1AIDLLInstance,teamArgs);
177             }
178         }
179         catch(Exception e)
180         {
181             Debug.LogError("Errpr on dll1");
182             Debug.Log(e.ToString());
```

KUVA 4. Datan lähetys pelialustalta tekoälykomponentille.

Pelimoottorin ja tekoälykomponentin rajapinta on näin pystytty mahdollisimman yksinkertaisena. Rajapinta koostuu yhdestä funktiosta ja kahdesta data luokasta, jotka täytyy olla määritelty tekoälykomponentissa (liite 1). Rajapintafunktio hoitaa datan siirron ja kontrollin luovuttamisen tekoälykomponentille (Kuva 5). Funktion reflektioparametrit toimivat palautuskanavana, jossa tekoälyn tekemät päätökset ja kontrolli palautetaan pelin puolelle.

```
1
2
3   using System;
4   using UnityEngine;
5   using System.Reflection;
6   using System.Collections.Generic;
7
8   namespace AIDLL
9   {
10      interface DLLInterface
11      {
12          void SendData(object[] gameUnitData, int team, int ownTeamScore, int otherTeamScore,
13                        float timeLeft, Vector2 borderTopRight, Vector2 borderBottomLeft,
14                        object[] planetData, Action<object[]> callBack, Action commandsFinished);
15      }
16  }
```

KUVA 5. Rajapinta funktio.

Unity käyttää 2.0/3.5 .NET -kehyksen funktionaalisuutta, joka tukee korkeintaan 4 parametrin reflektiofunktioita. Tästä rajoitus on kierretty muuttamalla yksittäiset parametrit objektilistaksi. tästä syystä erityistä huomiota täytyy kiinnittää reflektiofunktioissa tapahtuvien virheiden tarkasteluun ja virheellisistä syötteistä toipumiseen. Toisena vaihtoehtona olisi käyttää oliota tai struct-muuttujia, mutta näiden käsittely on pelialustan puolella epäkäytännöllistä, koska jokaiseen muuttujaan täytyy viitata sen parametrin tekstimuotoisella nimellä.

7 JOHTOPÄÄTÖKSET JA POHDINTA

Työssä toteutettiin pieni peliprojekti ja dll-tiedostoilla toimiva tekoäly, joka toteuttaa määriteltyä rajapintaa. Työn pelin ja tekoälyn lähdetiedostot ovat vapaasti saatavilla bitbucket-sivustolla osoitteesta https://bitbucket.org/LasseJukola/ai-game.

Varsinaisessa itsenäisessä tekoälykomponentissa onnistuttiin. Se voidaan ladata pelin sisäisellä tiedosto valikosta ajon aikana ja sen korvaaminen uudella versiolla ei tuota komplikaatioita. Kuitenkin varsinaisen automaatiotestauksen kannalta tämänkaltainen tekoäly ei ole hyödyllinen, mutta se kuitenkin osoittaa että tämän kaltainen toteutus on mahdollinen.

Rajapinnan toteutuksessa onnistuttiin osittain. Rajapinta on hyvin yksinkertainen ja sen käyttö tekoälykomponentin puolella on todella helppoa. Suurimmaksi ongelmaksi osoittautui väärä valinta toteuttaa datan siirto valmiiksi muutetuilla objekteilla sen sijaan, että käytettäisiin tekstimuotoista dataa kuten json tai xml. Objekteiksi muokattu data mahdollistaa sen yksinkertaisen käsittelyn tekoälyn puolella. Käytännössä koko datatoteutuksen monimutkaisuus on vain siirretty pelin puolelle. Siirretyn datan lisääminen tai muuttaminen on huomattavasti monimutkaisempaa, kuin jos se olisi serialisoitu tekstimuotoon valmiilla kirjastoilla.

Mobiilialustoilla vastaavanlainen toteutus täyttyy kirjoittaa laitteen natiivilla ohjelmointikielellä eli ios laitteilla kieli on Objective-c ja androidi laitteilla vastaavasti Java. Apple kuitenkin estää sovellusten julkaisun, jonka toteutuksessa ladataan ajettavaa koodia itse ohjelman ulkopuolelta sen aiheuttaman turvallisuusriskin vuoksi. Tämä estää varsinaisen julkaisuversion automaatio automaatiotestauksen.

LÄHTEET

B Goertzel, C Pennachin. 2007 Artificial General Intelligence. Saksa. Springer.

Illinois State University, Dynamic Link Libraries. Luettu 10.11.2015. http://www.itk.ilstu.edu/staff/drathke/355/Web%20Content/reading/DLLs.htm

Microsoft, Walkthrough: Creating and Running Unit Tests for Managed Code. Luettu 05.11.2015. https://msdn.microsoft.com/en-us/library/ms182532.aspx

Microsoft, What is a DLL?. Luettu 05.11.2015. https://support.microsoft.com/enus/kb/815065

Microsoft, Secure Coding Guidelines for Unmanaged Code. Luettu 10.11.2015. https://msdn.microsoft.com/en-us/library/0e91td57.aspx

Portnow J, Floyd D.2012. Video. Extra Credits - The Singularity - When Technology Becomes Sentient 19.05.2012. katsottu 30.08.2015. https://youtu.be/6_BfYkqWHD8?t=1m35s.

Rabin S. 2002. Ai Game Programming Wisdom. Massachusetts: Charles river media,inc

LIITTEET

Liite 1. Rajapinnan dataluokat


```
using UnityEngine;
using System.Collections;
namespace AIDLL
{
 public class GameUnitData
 {
   public float maxHp = 100;
   public float maxShield  = 100;
   public float damage = 10;
   public float maxSpeed = 3;
   public float shieldRegenAmount = 10;
   public float shieldRegenTime = 5;
   public float shieldRegenInitTime = 10;
   public float maxRotateSpeed = 100;
   public float firingRange = 30;
   public float lastFired = 0;
   public float fireInterval = 0.5f;
   public float reSpawnTimer = 1;
   public GameUnit gameUnit;
   public bool canReceiveCommands = true;
   public Vector3 spawnPoint;
   public int id;
   public bool activeUnit = false;
   public int side;
   public float hp = 100;
   public float shield = 100;
   public bool fire = true;
   public Vector2 position;
   public float angle;
   public float speed = 2;
   public float rotateSpeed;
   public float rotateTarget;
   public Vector2 fireTarget;
   public GameUnitData(Vector3 spawnPoint,
                int id,
                bool activeUnit,
                int side,
                float hp,
                float shield,
                bool fire,
                Vector2 position,
                float angle,
                float speed,
                float rotateSpeed,
                float rotateTarget)
   {
          this.spawnPoint = spawnPoint;
          this.id = id;
          this.activeUnit = activeUnit;
          this.side = side;
          this.hp = hp;
          this.shield = shield;
          this.fire = fire;
          this.position = position;
          this.angle = angle;
          this.speed = speed;
          this.rotateSpeed = rotateSpeed;
          this.rotateTarget = rotateTarget;
   }
   public GameUnitData()
   {
   }
   public GameUnitData(int side, Vector3 spawnPoint, GameUnit gameUnit)
```

```
    {
      this.gameUnit = gameUnit;
  this.side = side;
      this.spawnPoint = spawnPoint;
    }
    public void UpdateData(Vector3 position, float angle)
    {
      this.position = position;
      this.angle = angle;
    }
  }
```

```
}
using UnityEngine;
using System.Collections;
namespace AIDLL
{
    public class PlanetData
    {
        public Vector2 position;
        public float radius;
        public PlanetData(Vector2 pos, float rad)
        {
            position = pos;
            radius = rad;
        }
}
```