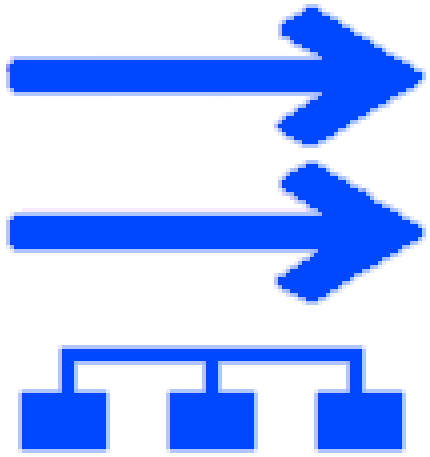
Marjaana Hirvonen

BT Builder: Käytöspuutyökalu Unity3D-pelimoottorille



Tradenomi

Tietojenkäsittely

Syksy 2016

KAJAANIN
AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

TIIVISTELMÄ

Tekijä: Hirvonen Marjaana

Työn nimi: BT Builder: Käytöspuutyökalu Unity3D-pelimoottorille

Tutkintonimike: Tradenomi, Tietojenkäsittely

Asiasanat: tekoäly, käytöspuu, Unity3D, työkalu, C#

Opinnäytetyön tavoite oli suunnitella ja ohjelmoida käytöspuu-rakennetta hyödyntävä tekoälytyökalu, jota voidaan käyttää Unity3D-pelimoottorissa tekoälyn päätöksenteon toteutukseen. Työkalu on lisäksi tarkoitus julkaista Unity Asset Store -kauppapaikassa.

Käytöspuuarkkitehtuuri on 2000-luvulla yleistynyt, varsinkin pelien tekoälyohjelmoinnissa käytetty rakenne. Siinä toiminnot ja niiden toteutumisen ehdot on erotettu siten, että itse puun rakenne ohjaa toimintojen valintaa. Tämän takia se soveltuu hyvin esitettäväksi visuaalisessa muodossa. Käytöspuuarkkitehtuurin perusrakenne on melko työläs toteuttaa vaikkapa tilakoneeseen verrattuna, ja valmiin työkalun käyttö vapauttaa ohjelmoijan keskittymään varsinaisen toiminnan toteuttamiseen.

Opinnäytetyöprosessin aikana työkalun toteutus ehti testausvaiheeseen asti. Kaikki suunnitellut ominaisuudet, joihin kuuluu visuaalinen editori, toiminnan toteutusjärjestelmä sekä visuaalinen tarkistusikkuna, on toteutettu ja työkalu toimii luotettavasti. Joitakin käytettävyyttä parantavia muutoksia on vielä tehtävä ennen julkaisuprosessin aloittamista.

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

ABSTRACT

Author: Hirvonen Marjaana

Title of the Publication: BT Builder: A Behavior Tree Tool for Unity3D Game Editor

Degree Title: Bachelor of Business Administration, Business Information Technology

Keywords: artificial intelligence, behavior tree, Unity3D, C#, visual editor

The goal of this thesis was to develop an artificial intelligence tool for defining behaviors using behavior tree architecture. The tool can be integrated in Unity3D editor and it is meant to be published in Unity Asset Store.

Behavior tree architecture is a decision making model that has become popular in video games in the recent years. In behavior trees the transition rules are decoupled from the actions, so that the tree structure itself defines which actions are executed. Due to its modularity it is also easy to visualize. The basic structure of this architecture is fairly complicated to implement compared to, for example, a state machine and using the tool lets the programmer focus on implementing the behavior itself.

All the main features including the visual editor, system for running the behaviors and visual debugger, were finished during the thesis process, and the tool works reliably. There is still testing to do, and some usability improvements will be made before the publishing process is started.

SISÄLLYS

1 JOHDANTO

Peliohjelmoinnissa minua on alusta asti kiinnostanut eniten tekoäly ja pelihahmojen käyttäytymisen rakentaminen. Vaikka tekoäly mielletään helposti älykkyyden simuloinniksi, todellisuudessa hyvinkin yksinkertaiset asiat voivat saada aikaan vaikutelman tietoisesta harkinnasta. Pelien tekoälyn tavoitteet ovatkin lähempänä tarinankerrontaa kuin yleistä tekoälyä. Olennaista ei ole se, miten tarkasti tietoja käsitellään tai millä perusteella päätöksiä tehdään, vaan tärkeintä on luoda illuusio ajattelevasta pelihahmosta.

Mitä enemmän pelissä on toimintavaihtoehtoja, sitä monimutkaisempi täytyy tekoälynkin olla. Yleisesti käytetty tilakone, joka on helppo määritellä yksinkertaisissa tilanteissa, käy nopeasti hankalaksi ymmärtää. Hienostuneemmat tekniikat, jotka pystyvät toimimaan joustavasti yllättävissäkin tilanteissa, voivat olla hankalia toteuttaa. Tämän takia halusin tehdä työkalun, jossa tämä hankala pohjatyö olisi tehty valmiiksi ja käyttäjä voisi keskittyä varsinaisen toiminnan ohjelmointiin.

Valitsin työkalun alustaksi Unity-pelimoottorin, koska se on suosittu sekä harrastajien keskuudessa että myös yrityksissä.

2 TEKOÄLYN OHJELMOINNISTA

Pelien tekoäly voidaan jakaa kolmeen osa-alueeseen: liikkumiseen, päätöksentekoon ja strategiaan. Kaksi ensin mainittua ovat yksittäisten pelihahmojen ominaisuuksia, kun taas kolmas liittyy pelihahmojen yhteistoimintaan. Läheskään kaikissa peleissä tekoälyn ohjaamilla hahmoilla ei ole yhteistä strategiaa, vaan ne toimivat itsenäisesti toisistaan riippumatta. [Millington 2009, 9.] Tässä osassa keskityn tekoälyn päätöksenteon rooliin peleissä ja siihen liittyviin vaatimuksiin. Esittelen myös tarkemmin käytöspuurakenteen toimintaa ja ominaisuuksia.

2.1 Pelien tekoäly

Pelien tekoälystä puhuttaessa ei yleensä tarkoiteta aivan samaa kuin tekoälyllä yleensä. Suurin ero on tekoälyn toiminnan tavoitteissa: kun akateeminen tekoälyn kehitys pyrkii pääasiassa parhaaseen mahdolliseen suoritukseen, peleissä tavoite on halutun pelaajakokemuksen aikaansaaminen. Tämä illuusio voidaan rakentaa myös yleisen tekoälyn tekniikoita käyttäen, mutta aina ne eivät sovi sellaisenaan. Pelien reaaliaikaisuus asettaa omat rajoituksensa. Aina onnistumiseen tähtäävä tekoäly taas voi olla liian ylivoimainen, jolloin pelaaminen ei enää ole mielekästä. On myös otettava huomioon pelaajan odotukset; tekoäly voi toimia täysin johdonmukaisesti, mutta jos pelaaja ei tiedä päätöksenteon perusteita, toiminta voi vaikuttaa järjettömältä ja siten rikkoa illuusion. Pelitekoälyn rakenteen kannalta tämä tarkoittaa sitä, että sen toiminnan on oltava helposti hallittavissa. [Dill 2013.]

2.1.1 Pelien tekoälyn haasteita

Koska tietokonepelien grafiikka kehittyy tällä hetkellä nopeasti yhä realistisempaan suuntaan, myös tekoälyn vaatimukset kasvavat jatkuvasti. Grafiikka vaikuttaa ennen kaikkea pelaajan odotuksiin: se, kuinka älykkäältä ja realistiselta hahmo näyttää, määrää sen, kuinka intuitiivisesti sen oletetaan käyttäytyvän. Pienetkin virheet voivat tällöin rikkoa älykkyyden vaikutelman. [Hayward 2007.]

Tekoälyn kannalta vaativimpia ovat hahmot, jotka toimivat tiiminä pelaajan kanssa ja joiden on voitava reagoida paitsi ympäristöön myös pelaajan toimintaan. Kuitenkin myös satunnaisten sivuhahmojen on luotava illuusio älykkäästä ajattelusta. Vaikka hahmon varsinainen tehtävä pelissä olisi hyvin yksinkertainen eikä vaatisi muuta kuin kaavamaista toimintaa, sen on silti voitava uskottavasti reagoida pelimaailman tapahtumiin. Kaikki yllättävät asiat eivät vaikuta pelin kulkuun eivätkä vaadi välitöntä toimintaa. Pelaajan kokemukseen vaikuttaa paljon, jos tekoälyn ohjaama hahmo osaa reagoida epätavallisiin asioihin myös näissä tapauksissa. Syntyy illuusio hahmosta, joka on tietoinen ympäristöstään. [Dill 2013.]

Älykkäältä vaikuttavan pelihahmon on myös kyettävä yhteistyöhön pelaajan ja muiden pelihahmojen kanssa. Kyetäkseen yhteistyöhön hahmon on saatava tietoja muiden pelihahmojen toiminnasta. Jos pelihahmo ei ota huomioon muiden hahmojen toimintaa, se ei vaikuta älykkäältä ja pelaajan illuusio rikkoutuu. [Garcés 2006.]

Inhimillinen päätöksenteko ei tapahdu juuri koskaan vain yhden kriteerin perusteella [Graham 2013]. Useimmiten päätöksiin vaikuttaa joukko asioita, joista mikään ei ole ehdoton edellytys. Toiminnan järkevyys arvioidaan siihen vaikuttavien asioiden yhteisvaikutuksen perusteella. Tämä asettaa haasteita tekoälyn ohjelmoinnille: miten määritellään ehto, joka ei suoraan riipu mistään yksittäisestä tekijästä? Kun on vielä otettava huomioon käytettävissä olevan laskentateho, päätöksentekoarkkitehtuurin merkitys kasvaa [Isla 2005].

Älykkäältä vaikuttavan pelihahmon on siis voitava ottaa huomioon varsin monenlaisia tietoja. Oikean toiminnan arviointi kussakin tilanteessa on sitä työläämpää, mitä enemmän mahdollisia olosuhteita ja toimintoja pelissä on. Tämän monimutkaisuuden hallinta on eräs suurimpia haasteita pelin tekoälyn rakentamisessa [Isla 2005]. Tämä ei kuitenkaan tarkoita, että monimutkainen ja yksityiskohtainen tekoäly olisi aina paras. Tärkeintä on, että se pystyy välttämään selviä virheitä. [Millington 2009, 19-21.]

2.1.2 Arkkitehtuureja

Tekoälyn toteutustapaa valittaessa on tärkeää ymmärtää, mitkä ovat eri tekniikoiden vahvuudet ja millaisia vaatimuksia juuri kyseinen tilanne tekoälylle asettaa. Vaaditun toiminnan monimutkaisuus, käytettävissä oleva laskentateho ja toteutukseen varattu aika vaikuttavat kaikki siihen, miten tekoäly kannattaa rakentaa. Yhtä yleispätevää ratkaisua ei ole. [Dawe et al 2013.] Joskus voi myös olla järkevää yhdistellä eri tekniikoita. Jos kuitenkin tällaisia poikkeustilanteita on projektissa paljon, voi olla kyse siitä, ettei ensin valittu tekniikka olekaan sopivin.

Tilakone

Tilakone on pelien tekoälyssä yleisesti käytetty rakenne. Siinä ohjattavalla hahmolla on määritelty tila, johon kuuluu tietty toiminto tai useita toimintoja. Jokaiselle tilalle on myös määritelty säännöt, joiden perusteella siirrytään seuraavaan tilaan. [Millington 2009, 310.] Vaikka tilakone on usein käyttökelpoinen rakenne, sillä on joitain rajoitteita, jotka vaikeuttavat monimutkaisempien päätöksentekorakenteiden ohjelmointia. Kun mahdollisia tiloja ja niiden välisiä siirtymiä on paljon, kokonaisuuden hahmottaminen käy vaikeammaksi. Jos suureen tilakoneeseen tehdään muutoksia, joudutaan ottamaan huomioon muutosten vaikutukset rakenteen muihin osiin. Tämä lisää virheiden mahdollisuutta ja vaikeuttaa editointia. Toinen tilakoneen ongelma on se, että se ei rakenteena salli yhtäaikaisia toimintoja. Kerrallaan voi olla voimassa vain yksi tila. [Dawe et al 2013.]

Hierarkkinen tilakone

Kun tiloja on paljon, voi olla järkevää jakaa tilakone pienemmiksi osiksi ja rakentaa niistä hierarkkinen tilakone. Siinä ylemmän tason kone hallitsee siirtymiä alemman tason koneiden välillä samaan tapaan kuin eri tilojen välillä. Tällä tavoin kokonaisuutta on helpompi hallita. Rakenne on kuitenkin edelleen melko joustamaton ja muutosten tekeminen työlästä [Dawe et al 2013].

Käytöspuu

Käytöspuussa siirtymälogiikka ja sen ehdot on erotettu toiminnoista, jolloin rakenne on helpommin muokattavissa. Varsinaiset toiminnot ovat lehtisolmuissa, jotka eivät sisällä tietoja edellisistä tai seuraavista toiminnoista. Se, mitkä toiminnot toteutuvat milläkin hetkellä, määritellään puun rakenteen avulla. Tämä mahdollistaa toimintojen ketjuttamisen ja myös useamman toiminnon suorittamisen samaan aikaan. Rakennetta voidaan helposti muokata lisäämällä ja poistamalla solmuja [Dawe et al 2013]. Käytöspuun rakenne ei kuitenkaan muutu itse ajon aikana, joten kaikki mahdolliset tilanteet ja toiminnot on edelleen suunniteltava etukäteen. Myöskään toimintojen tärkeysjärjestys ei muutu. Toisaalta puurakenne on helposti visualisoitavissa, jolloin käytöksen logiikan seuraaminen on helppoa. [Merrill 2013.]

Hyödyllisyysperiaate

Viime aikoina on yleistynyt hyödyllisyysarvoihin perustuva (utility-based) päätöksentekotekniikka. Siinä mahdollisille toiminnoille lasketaan hyödyllisyysarvo, jonka perusteella valitaan edullisin toiminto. Tyypillisesti hyödyllisyysarvo lasketaan eri tekijöiden vaikutusten keskiarvona. Tekniikan hyviä puolia on sen helppo laajennettavuus: koska toimintojen valintaperusteet ovat täysin itsenäisiä, uusia toimintoja on helppo lisätä. Kun toimintoja ja niihin vaikuttavia tekijöitä on paljon, mallin ennustettavuus vähenee, mikä parhaassa tapauksessa johtaa intuitiiviseen käytökseen. Mallin suurin haaste on tekijöiden vaikutusarvojen määrittely. [Graham 2013.]

Tavoitteellinen tehtäväsuunnittelu

Tavoitteellinen tehtäväsuunnittelu on rakenne, jossa jokaiselle toiminnolle on määritelty sen toteutumisen ehdot sekä vaikutukset. Kun haluttu päämäärä on tiedossa, seurataan siihen johtavien toimintojen ketjua, kunnes löydetään toiminto, joka voidaan toteuttaa. Koska toiminnot eivät sinänsä ole sidoksissa toisiinsa, uuden toiminnon lisääminen ei vaadi olemassa olevien muuttamista. Hyvin joustava

rakenne voi myös löytää odottamattomia ratkaisuja. Tämä voi toisaalta johtaa tekoälyn intuitiiviseen päätöksentekoon, mutta tekee siitä samalla vaikeamman hallita. [Dawe et al 2013.]

Alan kehitys

Koska pelien tekoälyn vaatimukset kasvavat jatkuvasti, myös uusia arkkitehtuureja ja tekniikoita kehitetään koko ajan. Edellä mainituista tekniikoista suurin osa on yleistynyt pelien tekoälyohjelmoinnissa vasta viime aikoina. Tämä ei kuitenkaan tarkoita, että aina olisi siirryttävä uusimpaan malliin ja hylättävä entiset. Jokaisella arkkitehtuurilla on omat hyvät ja huonot puolensa, ja käyttötarkoitus määrittää sen, mitä tekniikkaa kannattaa käyttää [Dawe et al 2013]. Myös tilakone, joka on rakenteeltaan melko joustamaton, on edelleen yleisesti käytössä ja soveltuu varsinkin tilanteisiin, joissa tarvitaan yksinkertaista ja kevyttä päätöksentekoa. [Millington 2009.]

Miksi juuri käytöspuut?

Olen valinnut tekoälytyökalun perustaksi käytöspuu-rakenteen, koska se on monipuolinen ja joustava, ja sopii sen takia hyvin monenlaisiin tilanteisiin. Käytöspuuhun voidaan myös yhdistää muita tekoälyrakenteita: esimerkiksi valitsijasolmussa voidaan käyttää hyödyllisyysperusteista tekniikkaa [Merrill 2013], tai solmu voi sisältää oppimisalgoritmeja [Pereira 2015]. Puumainen rakenne on helposti visualisoitavissa, ja muutokset voidaan siirtää suorittavaan koodiin solmujen välisenä hierarkiana, mikä helpottaa erillisen visuaalisen editorin käyttöä. Yleiskäyttöisen työkalun ohjelmointi on lisäksi hyödyllistä juuri käytöspuiden yhteydessä, sillä tämän arkkitehtuurin työläin yksittäinen tehtävä on sen perusrakenteen ohjelmointi, mikä ei riipu käyttötarkoituksesta tai yksittäisten solmujen sisällöstä.

2.2 Käytöspuut

Halo 2 oli ensimmäinen peli, jossa käytöspuurakennetta hyödynnettiin. Pelin hahmojen tekoälyn oli kyettävä mukautumaan moniin erilaisiin tilanteisiin ja otettava

huomioon paljon yksittäisiä tekijöitä, mikä olisi johtanut tilakonerakenteessa nopeasti hyvin monimutkaisiin ja hitaisiin ratkaisuihin. Tämän takia hahmojen tekoäly pyrittiin rakentamaan modulaariseksi. Eri hahmoille ja eri tilanteisiin ei tarvitse ohjelmoida kokonaan uutta käyttäytymislogiikkaa, vaan se voidaan rakentaa pienemmistä kokonaisuuksista tarpeen mukaan. [Isla 2005.] Toinen käytöspuita hyödyntävä peli on Spore. Sen tekoälyn pohjana on käytetty samaa järjestelmää, mutta varsinaiset toiminnot on erotettu ehto- ja valintalogiikasta [Hecker 2009].

2.2.1 Toimintaperiaate

Käytöspuun on usein määritelty olevan suunnattu asyklinen graafi. Sillä on alkupiste (juurisolmu), välisolmuja, joilla voi olla useita lapsisolmuja, sekä lehtisolmuja. Puun jokainen haara päättyy lehtisolmuun. Käytöspuu on kuitenkin sikäli erikoistapaus, että sen suoritus on kaksisuuntainen: jokainen käsitelty solmu saa palautusarvon, joka lopulta palautuu juurisolmuun. Vaikka siis rakenteessa voidaan tietyillä ehdoilla palata aiemmin käsiteltyyn solmuun (alipuun suoritus voi palautua sen juurena olevaan valitsijasolmuun), se ei voi koskaan muodostaa päättymätöntä kehää. Suorituksen kulun kannalta väli- ja lehtisolmuilla on käytöspuussa hyvin erilaiset tehtävät: lehtisolmu määrittää itse oman palautusarvonsa ja välittää sen takaisin kutsuvalle solmulle. Välisolmut taas kutsuvat lapsisolmuja sen perusteella, millaisia arvoja niille palautetaan. Seuraavaksi esittelen yleisimmät solmutyypit ja niiden toiminnan.

2.2.2 Solmut

Käytöspuun solmut voidaan jakaa valintasolmuihin, jotka ohjaavat puun läpikäyntiä, ja toimintasolmuihin, joissa on varsinainen toiminnan sisältö. Toimintasolmut ovat aina lehtisolmuja. Koska käytöspuumalli on vielä melko uusi, käytetyt termit vaihtelevat jonkin verran. Valintasolmujen perustyypit ovat kuitenkin selvästi erotettavissa. Solmun suoritus palauttaa aina arvon, joka määräytyy suorituksen onnistumisen mukaan. Yleisesti käytetään arvoja SUCCESS, FAILURE ja RUNNING

sekä ERROR. Joissain malleissa käytetään lisäksi arvoa, joka kertoo solmun olevan valmiina arvioitavaksi [Knafla 2011]. Onnistunut suoritus palauttaa arvon SUCCESS; mikäli suorituksen loppuunsaattaminen vie useamman kuin yhden läpikäynnin ajan, keskeneräinen suoritus saa arvon RUNNING. Jos suoritus epäonnistuu ennakoidusta syystä, palautusarvo on FAILURE. Odottamattomasta syystä, kuten tarvittavan arvon puuttumisen takia, epäonnistunut suoritus palauttaa arvon ERROR. Nämä arvot ohjaavat käytöspuun läpikäyntiä. [Marzinotto et al 2013.]

Valitsija

Priority (Valitsija) -solmu kutsuu lapsisolmujensa suoritusta määrätyssä järjestyksessä, kunnes jokin lapsista palauttaa arvon SUCCESS tai RUNNING, ja palauttaa ko. arvon hierarkiassa ylöspäin. Mikäli mikään lapsisolmuista ei onnistu, myös valintasolmu epäonnistuu. Priority-solmusta käytetään myös nimitystä Selector. [Pereira 2014.] Lapsisolmujen suoritusjärjestys määritellään toimintojen tärkeyden mukaan, niin että kriittisimmät toiminnot ovat aina etusijalla ja vähemmän tärkeät voivat toteutua vain, jos mikään tärkeämpi ei toteudu.

Sarja

Sequence (Sarja) -solmu kutsuu lapsisolmujaan, kunnes jokin niistä palauttaa arvon FAILURE tai RUNNING, jolloin sarja keskeyttää suorituksen ja palauttaa myös kyseisen arvon. Jos kaikkien lapsisolmujen suoritus onnistuu, myös sarja-solmu onnistuu.

Koristelija

Decorator (Koristelija) -solmu saa aina vain yhden lapsisolmun. Sen avulla voidaan lapsisolmun toimintaa muokata tarpeen mukaan. Koristelija voi olla esimerkiksi ajastin, joka kutsuu lapsisolmuaan vain, jos edellisestä kutsusta on kulunut tietty aika. Koristelijaa voidaan käyttää myös muokkaamaan lapsisolmun palautusarvoa; koristelijan avulla voidaan esimerkiksi palauttaa aina SUCCESS, jos sen

lapsisolmua on kutsuttu, vaikka lapsisolmun suoritus olisi epäonnistunut. Koristelijan toimintatapa määräytyy siis aina tapauksen mukaan.

Paralleeli

Paralleelisolmu kutsuu aina kaikkia lapsisolmujaan. Paralleelisolmun oma palautusarvo määräytyy lapsisolmujen arvojen perusteella, ennalta määritellyn kynnysarvon mukaan: mikäli tietty määrä lapsisolmuista onnistuu, paralleelisolmu onnistuu. Muitakin kriteerejä voidaan käyttää, kuten käynnissä olevien lapsisolmujen määrää. Rinnakkaisia toimintoja voidaan hyödyntää myös määriteltäessä suorituksen etenemiselle olosuhteita, joissa mikään yksittäisen ehdon toteutuminen ei sinänsä ratkaise jatkotoimia. Rinnakkaisuus ei tässä siis tarkoita toimintojen todellista yhtäaikaisuutta, vaan viittaa siihen, että paralleelisolmun lapsisolmujen suoritukset eivät riipu toisistaan.

Toiminta

Action (Toiminta) -solmu sisältää varsinaisen tehtävän suorituksen. Modulaarisuuden ja mallin läpinäkyvyyden varmistamiseksi toimintasolmut on hyvä suunnitella siten, että jokainen yksittäinen solmu suorittaa vain yhden tehtävän ja monimutkaisemmat rutiinit rakennetaan useiden solmujen avulla. Varsinaisen tehtävän lisäksi toimintasolmulla voi olla aloitus- ja lopetusfunktiot, jotka varmistavat, että tehtävästä toiseen siirtyminen tapahtuu halutulla tavalla [Pereira 2014 (2)]. Tämä on yleensä tarpeen vain silloin, kun varsinaisen tehtävän suoritus kestää useamman läpikäynnin ajan.

Ehto

Ehtoja voidaan käsitellä käytöspuissa useilla eri tavoilla. Yleisesti käytetty tapa on tarkistaa ehtojen toteutuminen lehtisolmussa, joka palauttaa arvon SUCCESS, mikäli ehto toteutuu, ja FAILURE, mikäli ei toteudu. Ehtosolmu ei tee muutoksia tarkistamiinsa tietoihin.

Erikoistapaukset

Valintasolmuista on erotettavissa myös erilaisia erityistapauksia, jotka poikkeavat perustyypeistä jonkin verran, mutta noudattavat silti samaa peruslogiikkaa. Muistava sarja -solmu toimii kuten tavallinen sarja, mutta sen sijaan, että sen lapsisolmujen läpikäynti alkaisi aina alusta, se jatkuu aina viimeksi käynnissä olleesta lapsisolmusta. Tämä on tärkeä ero sen takia, että näin voidaan määritellä useiden pitkäkestoisten toimintojen sarjoja. Samaan tapaan myös valitsija-solmusta voidaan toteuttaa muistava versio, joka muistaa viimeksi onnistuneen solmun ja aloittaa lapsisolmujen suorituksen siitä. [Pereira 2015.] Tarpeen voi olla myös satunnaisvalitsija-solmu, joka valitsee suoritettavan lapsisolmun jonkin satunnaisuustekijän perusteella.

2.2.3 Blackboard – Liitutaulu

Käytöspuun läpikäynnissä tarvittavat tiedot voidaan koota liitutaulu-objektiin (blackboard). Kun puu itse ei säilytä tietoja, voidaan samaa puuinstanssia käyttää useampien agenttien käytöksen määrittelemiseen [Mark 2010]. Tiedot voidaan jakaa kolmeen eri tyyppiin:

- agenttikohtaiset tiedot
- puukohtaiset tiedot: lähinnä suorituksen tilaa koskevat tiedot, kuten edellisen läpikäynnin avoimiksi jääneet solmut
- solmukohtaiset tiedot

[Pereira 2014 (2)]

Koska käytöspuun läpikäynnissä tarvittavat tiedot ovat usein hyvinkin raskaiden laskutoimitusten tulosta, on tehokkaampaa suorittaa nämä laskutoimitukset erillään käyttäytymisen määrittelystä. Varsinkin maailmaa koskevat tiedot, jotka ovat samat kaikille agenteille, kannattaa laskea valmiiksi ja tallentaa liitutaululle, mistä jokainen tietoa tarvitseva solmu voi saada sen suoraan. Näin voidaan myös kont-

rolloida ohjelman työkuormaa; vähemmän tärkeitä tietoja voidaan päivittää harvemmin, ja käytöspuu saa liitutaululta aina viimeksi lasketun tiedon. Tämä mahdollistaa myös monisäikeisyyden käytön. [Mark 2010.] Tällöin liitutaulun käytöstä on myös se etu, että käytöspuulla on aina jokin tieto käytettävissä, eikä päivityksen hitaus estä puun suoritusta, vaikka se vaikuttaakin lopputulokseen.

2.2.4 Edut

Käytöspuu-rakenteen suurimmat edut liittyvät sen modulaarisuuteen. Hyvinkin monimutkaisia rakenteita on helppo esittää graafisesti, mikä helpottaa suunnittelua. Kaikki toiminnan kulkuun vaikuttava informaatio on nähtävissä puun rakenteessa. Koska toiminnan kulku on määritelty itse rakenteessa, puuhun voidaan myös lisätä tai siitä poistaa solmuja ilman, että koko rakennetta olisi muutettava. Koska käytöspuulla ei ole varsinaisesti voimassaolevaa tilaa, kuten tilakoneessa, voidaan määritellä myös yhtäaikaisia toimintoja. Hyvin suuri käytöspuu voidaan myös jakaa pienempiin osiin, alipuihin, jotka voidaan esittää yhtenä solmuna. Näin voidaan monimutkainen käyttäytymismalli esittää helpommin ymmärrettävässä, pelkistetyssä muodossa, joka kuitenkin edelleen noudattaa itse koodin rakennetta. [Pereira 2015.] Modulaarisuuden takia myös graafisen käyttöliittymän suunnittelu käytöspuiden ohjelmointiin on melko yksinkertaista.

2.2.5 Haitat

Koska käytöspuun rakenne on rekursiivinen, monimutkaiset mallit voivat aiheuttaa ajonaikaisen pinon ylivuodon, varsinkin jos itse toimintasolmujen sisältö vaatii paljon sisäkkäisiä funktiokutsuja [Marzinotto 2013]. Tämä voidaan estää toteuttamalla puun läpikäynti ilman rekursiivisia funktioita, esimerkiksi tallentamalla solmut taulukkoon, jonka läpikäyntiä hallitsevat valintasolmujen sisäiset säännöt [Knafla 2011]. Mahdollisuus määritellä yhtäaikaisia toimintoja voi taas aiheuttaa tilanteen, jossa useampi toiminto yrittää muuttaa samaa arvoa. Yksinkertaisissa tapauksissa, joissa toimiva tilakone on helppo rakentaa, käytöspuu ei välttämättä ole tehokas ratkaisu. Tilanteissa, joissa tarvitaan useita erilaisia käyttäytymismalleja eri

olosuhteisiin, käytöspuun rakenteen määrittely voi olla hankalaa [Millington 2009, 371]. Käytöspuu on rakenteena staattinen, joten se ei sellaisenaan sovellu tilanteisiin, joissa tekoälyltä vaaditaan ennalta-arvaamattomuutta tai oppimiskykyä. Toteutuvan käytöksen määrittely eri tilanteissa on edelleen suunnittelijan vastuulla. [Pereira 2015.]

2.2.6 Toteutustapoja

Koska käytöspuurakenne on vielä verrattain uusi, sen toteutuksesta on useita versioita. Halo 2:ssa käytetty malli pohjautuu vielä melko paljon hierarkkiseen tilakonejärjestelmään [Isla 2005]. Sporessa käytetty malli erottaa valintasolmut varsinaisista toiminnoista [Hecker 2009]. Alex Champandard näkee toiminnan ehdon tarkistuksen yhtäaikaisena itse toiminnan kanssa [Champandard 2007], mikä mielestäni ei ole tarpeen. Bjoern Knaflan datapainotteinen lähestymistapa on tehokas, kun datan siirtonopeus nousee suuremmaksi pullonkaulaksi kuin laskentateho [Knafla 2011]. Myös useilla kaupallisilla pelimoottoreilla, kuten Unreal Enginella ja Cryenginella on omat käytöspuusovelluksensa. Edellä mainituista Unreal Enginen toteutustapa on sikäli joustamaton, että se käsittelee ehtoja koristelijasolmuina, mikä rajoittaa rakenteen suunnittelua. Myös paralleelien toimintojen käyttö on rajoitettu Champandardin esittämään ehtotarkistukseen. [UE4.]

Datan hallinta

Eräs keskeinen kysymys käytöspuurakenteen toteutuksessa on sen tarvitseman tiedon hallinta. Jos siirrettävää tietoa on paljon, voi olla tehokkainta erottaa tiedon varastointi ja muokkaaminen itse käytöspuun suorituksesta [Knafla 2011]. Tällöin tarvittava tieto voidaan ryhmitellä niin, että sen siirtäminen käyttömuistiin on mahdollisimman tehokasta. Samalla kuitenkin tiedon yksilöinti vaikeutuu [Shoulson et al, 2011].

Valittu versio

Olen valinnut työkalun pohjaksi Pereiran käyttämän mallin [Pereira 2014], joka on mielestäni yksinkertainen ja joustava. Puun läpikäynti aloitetaan aina juurisolmusta sen sijaan, että tarkistettaisiin ensin jo käynnissä olevat solmut. Tämä lisää työkuormaa, mutta mahdollistaa sen, että käynnissä oleva toiminto voidaan aina keskeyttää, mikäli jokin tärkeämpi toiminto valitaan sen sijaan. Pereira erottaa myös sarja- ja valitsijasolmuista omiksi solmutyypeiksi memory-versiot, jotka tallentavat tiedon viimeksi valitusta lapsisolmusta ja jatkavat seuraavalla läpikäynnillä suoritusta muistissa olevasta solmusta. Näin voidaan määritellä pidempikestoisten toimintojen sarjoja tai varmistaa, että valitsijasolmu valitsee uuden toiminnon vasta, kun edellinen on suoritettu loppuun. Pereiran mallissa liitutaulu, jolle solmujen tarvitsemat tiedot tallennetaan, on jaettu kolmeen skooppiin: solmukohtaisiin, puukohtaisiin ja yleisiin tietoihin. Tämän jaon olen osittain hylännyt: varsinainen liitutaulu käsittää tässä tapauksessa vain yleiset tiedot, joihin voi päästä käsiksi mistä tahansa ohjelman osasta. Koska puukohtaiset tiedot liittyvät etupäässä puun läpikäynnin hallintaan, nämä tiedot voivat sijaita puu-luokassa. Samoin solmukohtaiset tiedot, joita tarvitaan vain solmun sisällä, voivat sijaita solmussa. Tästä johtuen samaa puuinstanssia ei voi käyttää eri pelihahmojen ohjaamiseen, vaan jokaisella hahmolla on oltava oma instanssi. Tämä vastaa kuitenkin hyvin Unityn käyttämää komponenttijärjestelmää.

3 TOTEUTUS

Käytöspuutyökalun toteutus tapahtui projektiopintoina kevään ja kesän 2016 aikana. Suunnittelin aluksi käytännön osuudeksi pelidemoa, jossa käytöspuurakenne toimisi tekoälyn pohjana. Itse rakenteen toiminta oli kuitenkin se, mikä projektissa kiinnosti eniten. Vaikka käytöspuun muokkaaminen on helpompaa kuin tilakoneen, se vaatii enemmän pohjatyötä. Valmiista työkalusta olisi siten hyötyä myös muissa projekteissa, joissa tarvitaan tekoälyä.

3.1 Tavoitteet

Työkalun suunnittelun lähtökohtana oli sen toiminnallisuus. Päätavoite oli, että kaikki työkalun käytön kannalta olennaiset ominaisuudet toteutetaan. Käytettävissä olevan ajan rajallisuuden takia oli arvioitava, mitä ominaisuuksia voidaan jättää pois ilman, että käytettävyys kärsii. Koska projektin lopputulos on kaupallinen tuote, viimeistely ja toimintavarmuus ovat erityisen tärkeät. Myös hyvä dokumentaatio on välttämätön.

3.1.1 Rajapinta

Olen valinnut käytöspuutyökalun käyttöympäristöksi Unity3D-pelimoottorin, joka sekä harrastajien että kaupallisten pelikehittäjien yleisesti käyttämä alusta. Moottoriin kuuluu myös mahdollisuus rakentaa omia lisäosia ja työkaluja sekä kauppapaikka (Unity Asset Store), jossa käyttäjät voivat myydä tekemiään apuohjelmia tai vaikkapa grafiikkasisältöjä. Koska työkalun tarkoitus on toimia valmiin työskentely-ympäristön osana, sen on syytä noudattaa mahdollisimman hyvin ympäristön toimintaperiaatteita. Tästä syystä olen pyrkinyt pitämään työkalun visuaalisen ilmeen mahdollisimman samankaltaisena Unityn oman yleisilmeen kanssa ja lisännyt grafiikkaa vain, kun se on ollut välttämätöntä. Olen myös käyttänyt Unityn omia työkaluja aina kun se on käynyt päinsä. Unityn dokumentaatio on varsin kattava

[Unity3D 2016], ja koska pelimoottorilla on paljon käyttäjiä, myös dokumentoimattomista ominaisuuksista on helppo löytää tietoa. Olen suunnitellut työkalun toiminnan siten, että se sopii Unityn komponenttipohjaiseen järjestelmään.

3.1.2 Laajuus

Se, mitä tekoälyn on tarkoitus tehdä, riippuu aina sitä käyttävästä sovelluksesta. Vaikka joitain yleisiä toimintoja voidaan määritellä valmiiksi, kaikkea tarvittavaa ei mitenkään voida ennakoida. Valmiin toimintokirjaston rakentaminen vaatii paljon työtä eikä ole tämän projektin puitteissa mahdollista. Käytöspuun toimintaperiaatteeseen kuuluu kuitenkin myös peruslogiikkaa, joka ei ole riippuvaista käyttöympäristöstä. Tässä työssä keskityn käytöspuun sisäisen rakenteen toteuttamiseen. Valmiiden toimintojen puuttumisen takia työkalun käyttäjän on oltava ohjelmointitaitoinen, joten työkalu ei sovi aivan vasta-alkajille. Toisaalta tämä antaa käyttäjälle vapaammat kädet suunnitella käytöspuun käyttö aina tilanteen vaatimusten mukaan.

3.1.3 Tavoitteet

Työn tavoitteena on suunnitella ja ohjelmoida Unity3D-pelimoottoriin yhdistettävä työkalu, jolla voidaan rakentaa käytöspuu-arkkitehtuuria käyttävä tekoäly. Työkaluun kuuluvat seuraavat osat:

- visuaalinen editori, jolla voidaan luoda, muokata ja tallentaa käytöspuita käytettävissä olevista solmuista ja muokata solmuihin kuuluvia vakioarvoja
- rakenne, joka ajaa valmiita puita varsinaisessa sovelluksessa
- mahdollisuus ohjelmoida omia solmuja työkaluun
- visuaalinen vianetsintäohjelma, jonka avulla voidaan seurata puun suorituksen kulkua ohjelman ajon aikana
- dokumentaatio, jonka perusteella käyttäjä osaa käyttää ohjelmaa

3.2 Käytöspuutyökalun rakenne

Käytösmallin kokoaminen tapahtuu editori-ikkunassa. Käytöspuussa voidaan käyttää kaikkia projektissa olevia solmuja, jotka perivät BaseNode-luokan ja sijaitsevat määrätyissä solmukansioissa. Kansiorakenne vaikuttaa ennen kaikkea siihen, miten solmut näkyvät editorin valikoissa. Myös solmun tyyppi ja sen käsittely editorissa määräytyy solmun tiedostosijainnin mukaan. Kun haluttu puurakenne on valmis, se tallennetaan xml-tiedostona BT_Files-kansioon tai sen alikansioon. Tämän jälkeen tiedostoa voidaan käyttää BTree-komponentissa, joka muodostaa tietojen perusteella ajettavan koodin. Ajon aikana käytöspuun suorituksen kulkua voidaan seurata tarkistusikkunan avulla. Tiedon kulku käytöspuun ja muun pelimaailman välillä tapahtuu liitutaulun avulla. Liitutaulu täyttyy vasta ajon aikana, ja sen sisältöä voi tarkastella BT_Behave-komponentin inspector-näkymässä. Seuraavaksi esittelen ohjelman toimintaa ja käyttöä tarkemmin.

3.2.1 Toimintamallin tallennus

Editori-ikkunassa rakennettu käytöspuu tallennetaan xml-tiedostona. Läheskään kaikkea editorissa olevaa tietoa ei tarvita varsinaisen ohjelman aikana. Xml-tiedostoon tallennetaan vain olennaiset tiedot: solmun nimi, jonka perusteella solmuinstanssi luodaan; solmun välittömät lapsisolmut, ja editorissa määriteltyjen muuttujien arvot. Näiden lisäksi tallennetaan myös joitain tietoja, joita tarvitaan vain editorissa: solmun muistiinpanot, jotka helpottavat työskentelyä varsinkin suurempien puiden kanssa, sekä solmun näkyvyys. Näin piilotetut alipuut pysyvät piilotettuina myös puuta uudelleen avattaessa. Sekä puutiedoston luku että kirjoitus tehdään rekursiivisesti. Jotkin Unity-ympäristössä yleisesti käytetyt datatyypit, kuten paikkavektorit, on määritelty itse moottorissa eikä niitä voi sellaisenaan tallentaa. Nämä datatyypit (Vector2, Vector3, Quaternion, Color) tallennetaan float-tyyppisenä taulukkona, minkä lisäksi tallennetaan datatyypin nimi. Näin tiedot voidaan lukuvaiheessa rekonstruoida alkuperäiseen muotoon. Varsinaiseen puukomponenttiin xml-tiedosto ladataan TextAsset-objektina.

3.2.2 Puun luonti

Ajettava puuinstanssi luodaan aina ohjelman initialisointivaiheessa xml:n tietojen perusteella. Luonti tapahtuu rekursiivisesti ja itse puuluokassa on referenssi vain puun juurisolmuun. Jokaisesta solmusta luodaan instanssi solmun nimen perusteella. Jos sama solmu esiintyy useassa eri kohdassa puuta, jokainen sijainti saa oman instanssin. Tämän ansiosta solmun tiedot voidaan tallentaa itse solmuun. Mikäli solmun nimeä vastaavaa luokkaa ei löydy, puun luonti epäonnistuu.

Mikäli solmun initialisoinnissa tarvitaan tietoja liitutaululta, ne haetaan solmun _init-funktiossa. Tässä käyttäjän on varmistettava itse oikea kutsujärjestys: BT_Behave -komponentti, jossa liitutauluinstanssi sijaitsee, luo hallitsemansa puut Awake-funktiossa, joka kuuluu Unityn sisäänrakennettuihin delegaatteihin. Uusimmissa Unityn versioissa näiden funktioiden kutsujärjestystä voi muokata, mutta oletusarvoisesti järjestystä ei voida tietää. Mikäli liitutaulua halutaan käyttää solmujen initialisoinnissa, on tarvittavat tiedot vietävä liitutaululle ensin ja solmujen initialisointi suoritettava vasta sen jälkeen. Tämä tapahtuu BT_Behave-komponentin InitialiseNodes-funktion avulla.

Solmun käsittelyssä on useita vaiheita, jotka on koottu Execute-funktioon. Ensin solmu merkitään avatuksi; sen jälkeen tarkistetaan sen edellinen tila. Mikäli solmu on jäänyt auki (RUNNING-tilaan), kutsutaan suoraan Action-funktiota, jossa on solmun varsinainen sisältö. Muuten kutsutaan ensin StartAction-funktiota. Action-funktio palauttaa solmun viimeisimmän tilan. Mikäli tila ei ole RUNNING (solmun suoritus ei ole kesken), kutsutaan solmun EndAction-funktiota, joka palauttaa solmun valmiustilaan, ja solmu merkitään suljetuksi. Tiedot avoinna olevista solmuista tallennetaan kahteen bool-tyyppiseen taulukkoon, joissa on nykyisen ja edellisen läpikäynnin tilanne. Näin voidaan sulkea oikein ne solmut, jotka edellisellä läpikäynnillä ovat jääneet käyntiin, mutta joita ei ole nykyisellä läpikäynnillä avattu. Siten voidaan varmistaa, että kun solmu seuraavan kerran avataan, se on valmiustilassa.

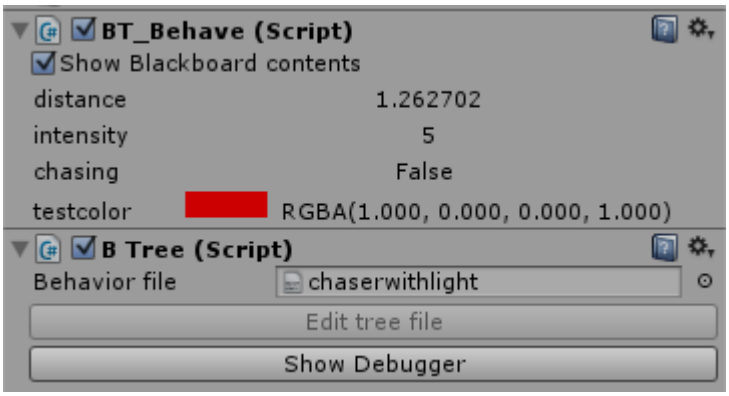
3.2.3 Komponentit

Managerikomponentti (BT_Behave), joka päivittää puukomponentteja, on toiminnan kannalta välttämätön ja lisätään automaattisesti BTree-komponentin kanssa, mikäli kyseisellä peliobjektilla ei sellaista jo ole. BTree-komponentteja voi peliobjektilla olla useita. BTree-komponentteja päivitetään siinä järjestyksessä, jossa ne näkyvät peliobjektin inspector-näkymässä. Toistaiseksi päivitystiheyttä ei voi erikseen määritellä. Käytöspuukomponentteja käsitellään samoin kuin Unityn muita komponentteja. Komponentit sisältävän objektin ei kuitenkaan tarvitse olla itse ohjattava peliobjekti. Koska työkalu sisältää vain käytöspuiden itsensä muokkaamiseen ja ajamiseen tarvittavat toiminnot, käyttäjä voi melko vapaasti päättää, miten tieto kulkee käytöspuun ja ohjattavan objektin välillä.

3.2.4 Komponenttien näkymät

Molemmilla komponenteilla on muokattu inspector-näkymä, joka poikkeaa Unityn automaattisesti luomasta näkymästä (kuva 1). BTree-komponentissa on kolme elementtiä: muokattava kenttä, johon voidaan asettaa haluttu xml-tiedosto; Edit tree file -painike, joka avaa käytöspuueditorin; sekä Show debugger, joka avaa tai sulkee visuaalisen tarkistusikkunan, josta suorituksen kulkua voi seurata. Lisäksi kyseisen puun saadessa virheellisen palautusarvon, mikä pysäyttää suorituksen, voidaan viimeisimmän suorituksen tilanne saada näkyviin Show last run -painikkeella. Tämä painike näkyy vain, jos suoritus on pysäytetty virheen takia.

Kun editori avataan Edit tree file -painikkeella, komponenttiin asetettu käytöspuutiedosto avataan automaattisesti. Tällöin tallennettaessa muutokset myös komponentin tiedosto päivittyy. Näin voidaan myös luoda ja liittää komponenttiin kokonaan uusi puu. Ajon aikana Edit-painike ei ole käytössä, koska käytöspuuinstanssin luonnin jälkeen sitä ei voida enää muuttaa.



Kuva 1. Käytöspuukomponentit ohjelman ollessa käynnissä.

BT_Behave-komponentin inspector-näkymä on perustilassa tyhjä. Ajon aikana siinä näkyy komponentin sisältämän liitutauluobjektin sisältö. Tämä helpottaa virheentarkistusta. Koska inspector-näkymät on toteutettu Unityn omaa järjestelmää käyttäen, ne voidaan myös piilottaa näkyvistä, jolloin komponentista näkyy vain sen nimi.

3.3 Solmut

Käytöspuun varsinainen toiminnallisuus sijaitsee solmuissa. Koska jokainen työkalua käyttävä sovellus on yksilöllinen, käyttäjällä on oltava mahdollisuus luoda tarpeen mukaan uusia, projektikohtaisia solmuja. Uusien solmujen luonti on toteutettava siten, että niitä voidaan käyttää editorissa ja ajettavan koodin osina kuten valmiita solmuja. Käyttäjän tarvitsee huolehtia vain solmun varsinaisesta sisällöstä.

3.3.1 Perussolmut

Käytöspuun suorituksen kulkua ohjaavat solmut on ohjelmoitu valmiiksi, koska niiden toiminta ei riipu ulkoisista tekijöistä. Näitä ovat valinta-, sarja-, paralleeli- ja satunnainen valintasolmu, sekä valinta- ja sarjasolmuista versiot, jotka pitävät muistissa edellisen suorituksen käynnissä olevan lapsisolmun. Näin voidaan määritellä usean eri pitkäkestoisen toiminnon sarjoja, kun lapsisolmujen läpikäynti ei

aina ala alusta. Myös joitain alipuun palautusarvoa muokkaavia koristelijoita on ohjelmassa valmiina, kuten onnistuu aina ja epäonnistuu aina -solmut. Sen sijaan ehto- ja toimintasolmut, jotka tyypillisesti lukevat ja muokkaavat käytöspuun ulkopuolista dataa, on käyttäjän määriteltävä itse.

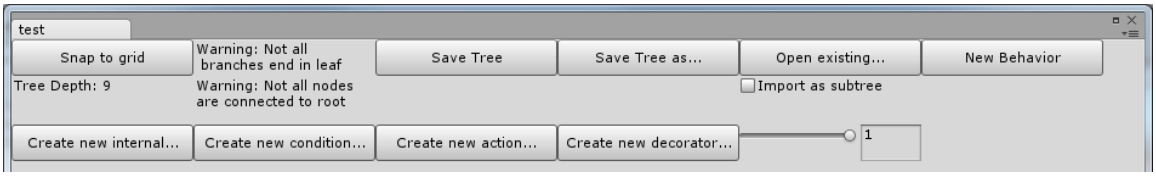
3.3.2 Omien solmujen luonti

Uusi solmu määritellään luomalla luokka, joka perii BaseNode-luokan. Uuden luokan on toteutettava ainakin Action-funktio, joka sisältää solmun varsinaisen toiminnan. Sen lisäksi voidaan määritellä StartAction- ja EndAction-funtiot, mikäli halutaan asioita tapahtuvan toiminnon alkaessa tai loppuessa. Näitä tarvitaan pääasiassa sellaisissa tapauksissa, joissa varsinainen toiminta kestää kauemmin kuin yhden puun läpikäynnin ajan. Hyvä käytäntö on suunnitella solmut niin, että yksi solmu suorittaa vain yhden tehtävän. Näin ohjelma pysyy modulaarisena ja käytöspuun muokkaaminen voidaan tehdä editorissa, koskematta koodiin. Editori lukee käytettävissä olevien solmujen tiedot suoraan kansioista, jonka jälkeen uusi solmu näkyy editorin valikossa ja sitä voidaan käyttää. Ehto- ja toimintasolmut on jaettu eri kansioihin käytettävyyden vuoksi. Molemmat ovat lehtisolmuja, jotka palauttavat suorituksen arvon, ja käyttäjä voi määritellä itse, kumpaan ryhmään mikäkin toiminto kuuluu. Koska erilaisia ehtoja ja toimintoja tarvitaan todennäköisesti paljon, tämä jako helpottaa oikean solmun hakemista. Solmut voidaan myös edelleen ryhmitellä alikansioihin. Kansiosijainti määrää myös sen, miten solmua voidaan käyttää editorissa: Internals-kansion solmut voivat saada useita lapsisolmuja, kun taas Decorators-kansioon sijoitetuilla solmuilla voi olla vain yksi lapsisolmu. Actions- ja Conditions-kansioiden solmut ovat aina lehtisolmuja.

3.4 Editori

Varsinaisen käytöspuun määrittely tapahtuu visuaalisen editorin avulla. Puun lähtöpisteenä on juurisolmu, joka on editorissa valmiina ja joita ei voi olla kerrallaan useampia. Editorin yläosassa on työkalurivi (kuva 2). Ylemmällä rivillä on puutie-

dostojen tallennukseen ja avaamiseen liittyvät toiminnot sekä Snap to Grid -toiminto, joka järjestää nykyisen puun solmut niiden keskinäisen aseman mukaan. Alemmalla rivillä on solmujen luontivalikot sekä zoomin hallinta. Lisäksi työkaluosassa näkyy puun syvyys sekä muita viestejä.



Kuva 2. Editorin työkalupalkki.

3.4.1 Yhdistäminen

Solmut voidaan yhdistää valitsemalla ensin toisen ja sitten toisen solmun nuolipainike. On huomattava, että jokaisella yhteydellä on suunta; vain vastakkaiset nuolet voidaan yhdistää toisiinsa. Mikäli yhteys on jo olemassa, se poistetaan. Yhdistäminen ei onnistu, mikäli puun rakenteen kriteerit eivät täyty, esimerkiksi jos haluttu yhteys muodostaisi silmukan. Juuri- ja koristelijasolmuilla ei voi olla kuin yksi lapsisolmu: useampaan lapseen yhdistäminen ei tällöin onnistu. Toimivan käytöspuun jokaisen haaran tulisi päättyä vähintään yhteen lehtisolmuun, joka voi määrätä palautusarvon; ohjelma tarkistaa tämän ja työkalurivin huomautusosassa näkyy varoitusviesti, mikäli näin ei ole. Samoin ohjelma varoittaa, jos kaikki editorissa näkyvät solmut eivät ole yhteydessä juurisolmuun.

3.4.2 Muokkaus

Solmun lapsisolmujen tärkeysjärjestys, joka on ohjelman suorituksen olennainen piirre, määräytyy solmujen visuaalisen sijainnin mukaan. Ylimpänä olevan solmun suoritusta siis kutsutaan ensin. Jotta puun suoritusjärjestyksen muokkaaminen ja lukeminen olisi mahdollisimman helppoa, näkymä voidaan järjestää automaattisesti. Kokonaisen alipuun voi siis siirtää eri paikkaan siirtämällä vain sen lähtösolmua. Toinen lukemista helpottava ominaisuus on alipuiden piilottaminen.

Lehtisolmuja lukuun ottamatta jokainen solmu voidaan kutistaa siten, että sen alla olevia solmuja ei näy. Tästä on hyötyä varsinkin suurempien puiden muokkauksessa, kun näkymän tarkkuustasoa ja siten luettavuutta voidaan muokata. Mikäli kutistetun alipuun juurisolmu poistetaan, tuhoutuvat myös sen alla olevat solmut. Laajennetussa näkymässä taas solmuja voidaan poistaa yksitellen. Jokaiseen solmuun voidaan myös liittää muistiinpanoja. Editorissa on myös zoom-toiminto, jolla koko näkymää voidaan suurentaa tai pienentää. Näkymän suurennustaso vaikuttaa siihen, mitkä solmujen muokkaustoiminnot ovat milloinkin käytettävissä.

3.4.3 Muuttujat

Solmujen tarvitsemat tiedot sijaitsevat pääsääntöisesti liitutaululla, jonka tarkempi esittely on seuraavassa luvussa. Monet solmut tarvitsevat kuitenkin etukäteen määriteltyjä vakioarvoja, jotka voivat vaihdella eri solmuinstanssien välillä, mutta pysyvät muuttumattomina itse ohjelman suorituksen aikana. Esimerkiksi se, kuinka moni paralleelisolmun lapsisolmuista saa epäonnistua, riippuu halutusta toimintalogiikasta. Samoin voivat toimia vaikkapa ehtojen raja-arvot tai muut käyttäytymistä hallitsevat arvot. Näitä arvoja on voitava muokata editorissa, jolloin käytöksen suunnittelija voi säätää toimintaa tarvitsematta koskea koodiin. Jotta muuttujan arvoa voidaan muokata editorissa, sen on oltava property-tyyppinen. Koska muokattava kenttä on erilainen eri datatyypeille, vain erikseen määritellyt datatyypit voivat näkyä editorissa [liite 1].

3.4.4 Tallentaminen ja avaaminen

Puun rakenne tallennetaan xml-muodossa. Xml on kuvauskieli, jossa tietojen väliset suhteet on esitetty rakenteen avulla [W3C 2008]. Tiedoston voi nimetä vapaasti. Tallentaminen on toteutettu rekursiivisesti juurisolmusta lähtien, joten koko näkymää ei tallenneta sellaisenaan, vaan vain juurisolmuun yhteydessä olevat solmut tallentuvat. Editorin huomautusosassa näkyy varoitusteksti, mikäli editorinäkymä sisältää solmuja, jotka eivät liity juurisolmuun. Olemassa olevan käytöspuutiedoston voi avata kahdella tavalla: uutena näkymänä tai alipuuna, jolloin

avattu puu näkyy yhtenä solmuna ilman juurisolmua ja sen voi liittää editorissa olevaan puuhun. Koska jokainen solmu edustaa konkreettista toimintoa, ohjelmasta on löydyttävä sitä vastaava luokka. Voi kuitenkin olla tarpeen avata puutiedostoja, joiden sisältämiä solmuja vastaavaa luokkaa ei ohjelmassa ole. Tällaisia solmuja voidaan käsitellä kuten muitakin, mutta ne on merkitty siten, että käyttäjä tunnistaa ne [liite 1]. Toimimattomia solmuja sisältävän puun voi myös edelleen tallentaa. Varsinaisessa ohjelmassa tällaisen puun initialisointi aiheuttaa kuitenkin virheen.

3.4.5 Tallennettavat tiedot

Kaikkea editorissa käytettävää tietoa ei tarvitse tallentaa varsinaista ajoa varten. On kuitenkin joitain tietoja, jotka on säilytettävä editointia varten. Xml-tiedostoon tallennetaan solmuluokan nimi, jota käytetään ajettavan puuinstanssin luomisessa, sekä mahdolliset editorissa asetettavat muuttujat. Tämän lisäksi editoria varten tallennetaan solmun muistiinpanot, joihin liitetään alipuun nimi, mikäli solmu on liitetty tallennettavaan puuhun erikseen avatun alipuun juurena. Myös solmujen näkyvyys tallennetaan. Jos näkymässä on piilotettuja solmuja, ne esitetään piilotettuina myös tiedostoa seuraavan kerran avattaessa. Jotkin Unityssa yleisesti käytetyt datatyypit, kuten paikkavektorit, perustuvat Unityn omiin strukteihin, joiden muuntaminen tallennettavaan muotoon vaatii ylimääräistä muokkausta. Tätä ei kuitenkaan tarvitse tehdä muuten kuin editorin tai käytöspuuinstanssin initialisointivaiheessa.

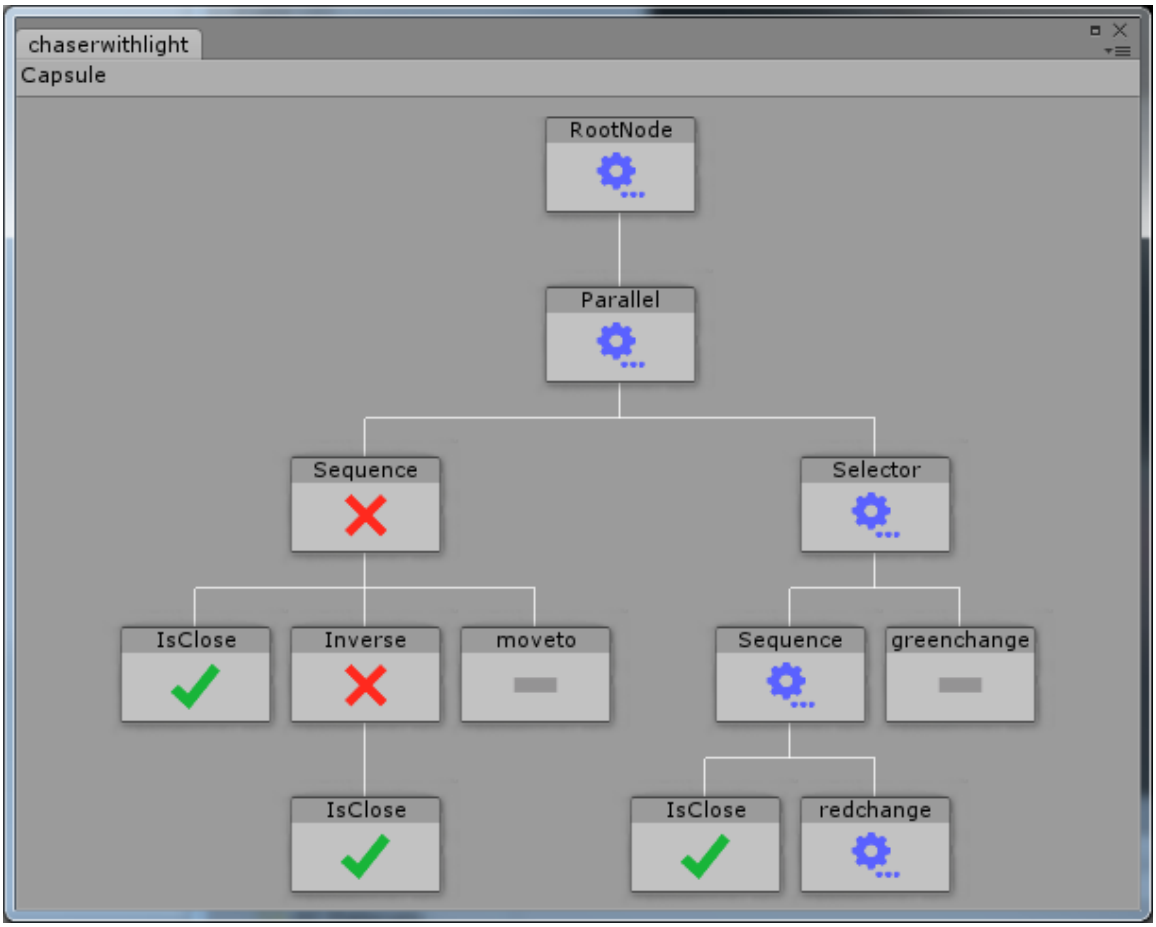
3.4.6 Ajonaikaiset tiedot

Ajonaikaisten tietojen käsittelyssä olen käyttänyt ns. liitutaulumallia. Tarvittavat muuttujat varastoidaan erilliseen tietorakenteeseen, ja niitä voidaan muokata mistä tahansa. Tiedon varastointi ja päivitys siis erotetaan sen käytöstä. Tietorakenteena olen käyttänyt dictionary-rakennetta (C#), joka perustuu avain/arvo-pareille. Luettavuuden vuoksi käytän string-tyyppistä avainta. Koska C# on vahvasti tyypitelty kieli, yhteen dictionary-objektiin voidaan varastoida vain yhdentyyppistä

tietoa. Tämä voidaan kiertää muuntamalla tieto object-tyyppiseksi varastoitaessa ja taas alkuperäisen tyypin mukaiseksi luettaessa, mutta tämä vaatii ylimääräisiä muistivarauksia, mikä voi olla ongelmallista varsinkin mobiilisovellusten yhteydessä. Tämän takia olen lisännyt liitutauluobjektiin erilliset tyypitellyt dictionaryt yleisimmin käytetyille tietotyypeille. Näin käyttäjä voi helpommin optimoida muistin käyttöä. BT_Behave-komponentissa on instanssi BT_Blackboard-luokasta, jossa on tarvittava toiminnallisuus tietojen varastointiin ja lukemiseen. Kaikki yhden BT_Behave-komponentin hallitsemat BTree-komponentit käyttävät samaa varastoa, joten peliobjektin yleisiä tietoja ei tarvitse varastoida useampaan paikkaan. Koska tietojen haku tapahtuu string-avaimella, joka annetaan erikseen aina tietoa haettaessa tai kirjoitettaessa, kirjoitusvirheen vaara on suuri. Tästä syystä liitutaulun sisältö päivitetään myös BT_Behaven inspector-näkymään.

3.5 Visuaalinen tarkistusikkuna

Jotta päätöksentekologiikan kulkua voidaan seurata ajon aikana, työkaluun kuuluu visuaalinen tarkistusohjelma. BTree-komponentin inspector-näkymän painikkeesta (”Show Debugger”) voidaan avata ikkuna, jossa näkyy puun rakenne. Jokaisen läpikäynnin jälkeen ikkunaan päivitetään solmujen palautusarvoja vastaavat symbolit [liite 1]. Mikäli solmu on ohitettu viimeisimmässä läpikäynnissä, se ei saa palautusarvoa. Näin voidaan nähdä, mitkä solmut on käsitelty ja kuinka suoritus on edennyt (kuva 3). Jos puussa on tapahtunut virhe ja sen suoritus on keskeytetty, viimeisimmän läpikäynnin tulokset saadaan näkyviin ‘Show last run’ -painikkeesta. Tämä toimii kuitenkin vain niin kauan kuin play-tilasta ei poistuta, sillä ajonaikaisia tietoja ei tallenneta pysyvästi. Virheen aiheuttanut solmu voidaan tarkistaa myös lokiviesteistä. Jokainen virhetuloksen lähettänyt solmu aiheuttaa oman virheviestin.



Kuva 3. Tarkistusikkuna toiminnassa. Symbolit vastaavat solmun palautusarvoja. Harmaalla palkilla merkityt solmut on ohitettu.

Ohjelmassa voidaan avata useita tarkistusikkunoita yhtä aikaa. Ikkunat sulkeutuvat automaattisesti, kun poistutaan play-tilasta. Kerrallaan avoinna olevien tarkistusikkunoiden määrälle ei ole erikseen määritelty ylärajaa. Ikkunan otsikkona on näytettävän puun tiedostonimi, ja lisäksi ikkunan yläreunassa näkyy puun omistavan peliobjektin nimi.

3.6 Dokumentaatio

Käyttäjän kannalta hyvä dokumentaatio on erittäin tärkeä. Vaikka ohjelman käyttöliittymän suunnittelussa pyritäänkin mahdollisimman selkeään ja yksiselitteiseen ulkoasuun, ei voida olettaa, että käyttäjä ymmärtää kaiken ilman lisätietoja. Kokematon tai asiaa ennestään tuntematon käyttäjä ymmärtää ohjelman toimintaperiaatteet parhaiten, jos saatavilla on esimerkkitapaus. Myös kokeneempi käyttäjä

haluaa todennäköisesti varmistaa, että ohjelma toimii oletetulla tavalla. Virheiden ja ongelmien selvittelyssä tarvitaan yksityiskohtaista tietoa siitä, miten ohjelma toimii. Tämän projektin dokumentaatio on sana- ja asiahakemisto, jossa esitellään käytettävissä olevat toiminnot sekä selitetään uusien solmujen ohjelmoinnin periaatteet ja vaatimukset. Tavoitteena on, että käyttäjä osaa dokumentaation perusteella käyttää ohjelmaa ja ratkaista yleisimpiä ongelmia ja virhetilanteita.

4 JÄLKIARVIOINTI

Vaikka suorituskoodi on rekursiivinen, tämä ei toistaiseksi ole tuottanut ongelmia. Koska käsitellyt solmut poistuvat aina kutsupinosta, puun laajuus ei sinänsä kuormita pinoa, vaan olennaisempi on puun syvyys. Puun rakenteen suunnittelulla voidaan vaikuttaa siihen, mitkä osat käsitellään, ja ehtojen perusteella epäolennaiset osat voidaan sivuuttaa kokonaan, mikä keventää puun läpikäyntiä. Tällä hetkellä käytän työkalua VR-pelissä Nemesis Perspective. Sekä editori että ajokoodi ovat toimineet luotettavasti.

Liitutaulun tyyppikohtaiset dictionaryt eivät kuormita heap-muistia, mikä on tärkeää mobiilisovellusten kannalta. Nykyinen liitutaulujärjestelmä on kuitenkin melko kömpelö ja vaatii käyttäjältä tarkkuutta. String-tyyppisten avainten käyttö ei myöskään ole optimaalista: niiden korvaaminen int-tyyppisillä keventäisi suoritusta.

Monissa tilanteissa käytöspuuta ei tarvitse tarkistaa jatkuvasti. Tällä hetkellä työkalussa ei kuitenkaan ole mahdollisuutta hallita läpikäynnin ajoitusta. Vähintään olisi voitava määritellä läpikäyntien väli harvemmaksi sekä oltava mahdollisuus tarkistaa tietty puu vain tarvittaessa.

Jos käytöspuu on hyvin suuri, visuaalisen tarkistusikkunan käyttö hankaloituu, koska tilaa tarvitaan paljon. Tällöin olisi hyvä olla mahdollisuus tuoda esiin vain tietty osa puusta. Ylimääräisen tyhjän tilan poistaminen solmujen välistä auttaisi myös (tällä hetkellä solmut järjestetään siten, että mitkään solmut eivät mene päällekkäin), mutta tätä vaikeuttaa se, ettei halutun tuloksen määrittely ole yksiselitteistä. Tyhjän tilan poisto voi myös vaikeuttaa puun lukemista.

Työkalun ongelmat siis liittyvät enimmäkseen käytettävyyteen. Varsinaisessa suorituksessa ongelmia ei ole ollut, joten projektia voi pitää onnistuneena.

LÄHTEET

Champandard, Alex (2007) Popular Approaches to Behavior Tree Design http://aigamedev.com/open/articles/popular-behavior-tree-design/ (Luettu 13.11.2016)

Dawe, Michael; Gargolinski, Steve; Dicken, Luke; Humphreys, Troy; Mark, Dave (2013) Behavior Selection Algorithms: An Overview Teoksessa Steve Rabin (toim.) Game AI Pro s. 47-60. Boca Raton, FL, USA: CRC Press

Dill, Kevin (2013) What is Game AI? Teoksessa Steve Rabin (toim.) Game AI Pro s. 3-9. Boca Raton, FL, USA: CRC Press

Garcés. Diego (2006) Achieving Coordination with Autonomous NPCs. Teoksessa Michael Dickheiser (toim.) Game Programming Gems 6 s. 223-234. Boston, MA,USA: Charles River Media

Graham, David (2013) An Introduction to Utility Theory Teoksessa Steve Rabin (toim.) Game AI Pro s. 113-126. Boca Raton, FL, USA: CRC Press

Hayward, David (2007) Uncanny AI: Artificial Intelligence In The Uncanny Valley http://www.gamasutra.com/view/feature/1436/uncanny_ai_artificial_.php?print=1 (Luettu 7.11.2016)

Hecker, Chris (2009) [My liner notes for spore/Spore Behavior Tree Docs]

http://chrishecker.com/My_liner_notes_for_spore/Spore_Behavior_Tree_Docs (luettu 15.10.2015)

Isla, Damían (2005) GDC 2005 Proceeding: Handling Complexity in the Halo 2 AI http://www.gamasutra.com/view/feature/130663/gdc_2005_proceeding_handling_.php (Luettu 28.11.2015)

Knafla, Bjoern (2011) In-Depth: Behavior Tree Entrails http://www.gamasutra.com/view/news/128548/InDepth_Behavior_Tree_Entrails.php (Luettu 28.1.2016)

Mark, Dave (2010) Damián Isla Interview on Blackboard Arch http://intrinsicalgorithm.com/IAonAI/2010/02/damian-isla-interview-on-blackboard-arch/ (Luettu 15.11.2016)

Marzinotto, A; Colledanchise, M; Smith, C; Ogren, P (2013) Towards a Unified Behavior Trees Framework for Robot Control http://www.csc.kth.se/~miccol/Michele_Colledanchise/Publications_files/2013_ICRA_mcko.pdf (luettu 16.10.2015)

Merrill, Bill (2013) Building Utility Decisions into Your Existing Behavior Tree Teoksessa Steve Rabin (toim.) Game AI Pro s. 127-136. Boca Raton, FL, USA: CRC Press

Millington, Ian & Funge, John: Artificial Intelligence For Games (2009) Burlington, MA, USA: Morgan Kaufmann Publishers

Pereira, Renato (2014) An Introduction to Behavior Trees - Part 2 http://guineashots.com/2014/08/10/an-introduction-to-behavior-trees-part-2/ (Luettu 7.11.2016)

Pereira, Renato (2014) Implementing a Behavior Tree - Part 1 http://guineashots.com/2014/09/24/implementing-a-behavior-tree-part-1/ (Luettu 7.11.2016)

Pereira, Renato; Engel, Paulo Martins (2015) A Framework for Constrained and Adaptive Behavior-Based Agents https://arxiv.org/pdf/1506.02312.pdf (Luettu 7.11.2016)

Shoulson, Alexander; Garcia, Francisco M.; Jones, Matthew; Mead, Robert; Badle, Norman I. (2011) https://people.cs.umass.edu/~fmgarcia/Papers/Parameterizing%20Behavior%20Trees.pdf (Luettu 13.11.2016)

Unity3D documentation (2016) https://docs.unity3d.com/ScriptReference/ (Luettu 15.11.2016)

Unreal Engine 4 documentation https://docs.unrealengine.com/latest/INT/Engine/AI/BehaviorTrees/HowUE4BehaviorTreesDiffer/index.html (Luettu 13.11.2016)

W3C Recommendation (2008) Extensible Markup Language https://www.w3.org/TR/REC-xml/ (Luettu 14.5.2016)

LIITTEET

HOW TO USE BT BUILDER

Adding Behavior Component

You can add a behavior to a GameObject or prefab just like any other component. The BT_Behave component, which manages the behavior execution, is added automatically, if there isn't already one. You can add several behaviors to the same GameObject.

Component menu

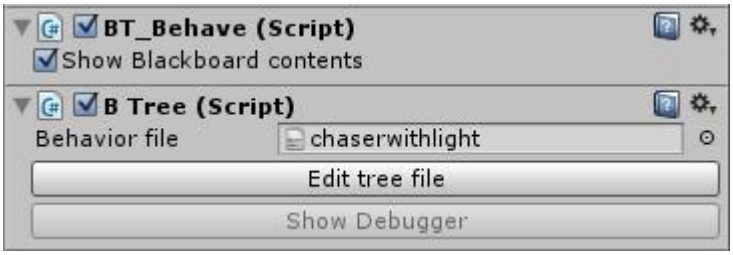
You can add a behavior to currently chosen game object from the Component menu (Component-> Behavior AI -> Behavior Tree).

Inspector

You can also use the Add Component -button in the inspector. Choose Behavior AI -> Behavior Tree.

Assigning a Behavior File

Behaviors made with the BT Builder are saved as xml files. To run a behavior, you have to assign a behavior file to the tree component. This can be done in several ways. You can open the editor from the component itself, use the object picker or just drag an existing file to the slot.



Edit Tree File

The tree component has an Edit tree file -button, which opens the behavior editor. You can either open an existing behavior file, or make a new behavior from scratch. When the behavior is saved, it is also assigned to the tree component.

This is also useful for quickly editing values that are set in the editor.
NOTE: Values set in editor are treated like constants and saved in the tree file itself. Changing values from one gameobject will change those values for all objects that use the same file. If you want to use the same behavior structure for different gameobjects with different value settings, you have to use different save files, or set values through the Blackboard, which is unique for each BT_Behave component.

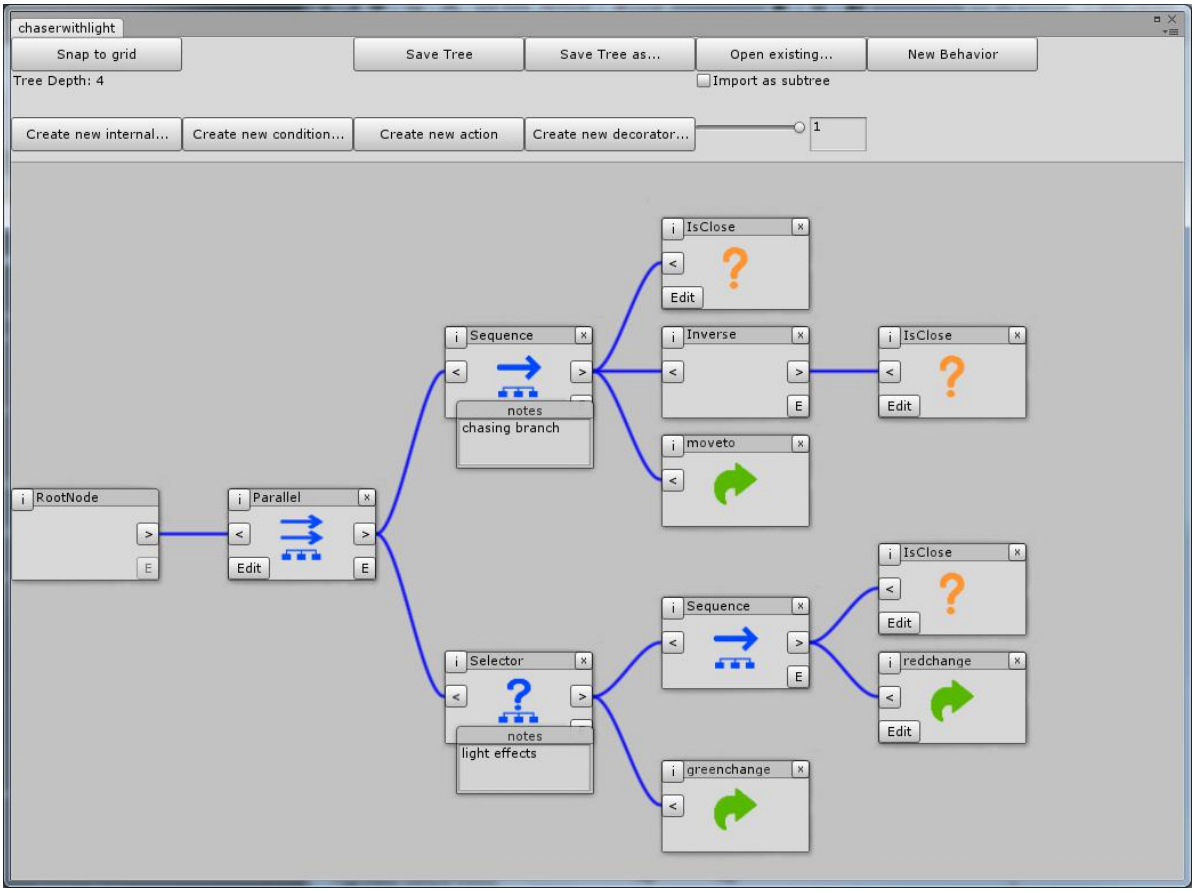
The behavior file is only used for creating and initialising the actual behavior instance, so editing the file during play mode has no effect. Edit Tree File -button is disabled during play mode.

Choose file from folder

You can assign a behavior file to the component like any other asset.
Clicking the circle to the right of the object field opens the object picker, where you can choose a behavior file.

You can also drag a behavior file from the project window and drop it to the file field.

EDITOR WINDOW



Creating a Behavior

When you open the editor or start a new behavior, there is always a RootNode placed on the left. This is the starting point of the behavior. The Root-Node can only have one child node.

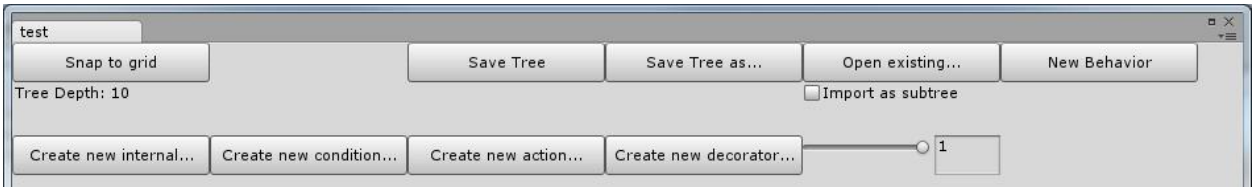
EDITING BEHAVIORS

Priority Order

The execution order of each node's children is determined by their position in the editor. The highest child is always the first in order. To change the execution order, you can simply drag a node to a different position. To move a whole branch, you

only need to move its root (the node where the branch starts). Clicking Snap to Grid -button arranges the nodes to the new order.

Toolbar



Toolbar's top row has controls and info for handling the behavior. Bottom row has buttons for adding nodes to the current behavior.

Snap to Grid

Snap to Grid -button aligns the nodes in the tree to their current order. It only affects the nodes that are part of the tree, i.e. connected to the RootNode. It is especially useful, when you need to move large parts of the tree. Snapping is not necessary for the running of the behavior, but it helps to keep the editor window ordered.

Save Tree

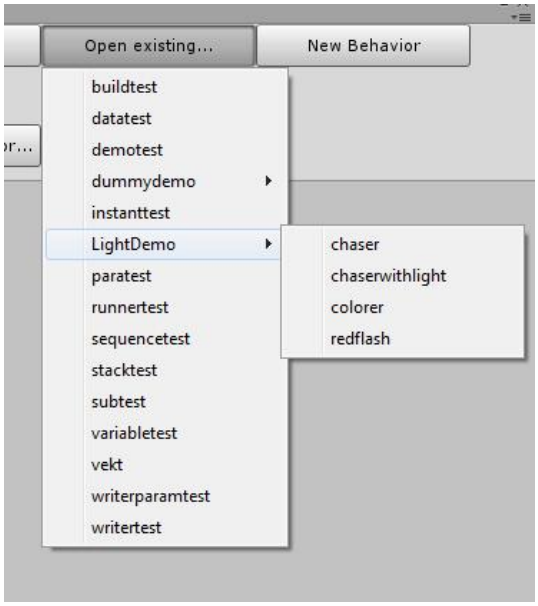
Saves the current behavior as xml file, using the current name and folder (name can be seen in the window title). If the behavior has no name yet, opens the saving dialog. If the editor was opened from a BTree component (see Edit Tree File), the behavior is also assigned to the component.

Save Tree As...

Opens the saving dialog window. You can type a name for the behavior and choose a folder to save it in. You can only choose the main BT_Files folder or an existing subfolder inside it. If you want to add a folder, you have to do it in Unity's project window. When you reopen the behavior editor, the folders will be updated.

Open Existing...

Opens a dropdown menu of existing behavior files in BT_Files and its subfolders. Choose a file in the menu to open it.



Import As Subtree

If the Import as subtree -box is checked, the chosen behavior file is added to the view as a subtree node (with hidden children, if it has any). You can then attach the node to the currently open behavior.

If the box is not checked, the behavior view is cleared and the chosen file is opened as the main behavior.

New Behavior

Clears the editor and adds a RootNode as the new behavior's starting point.

Create... -buttons

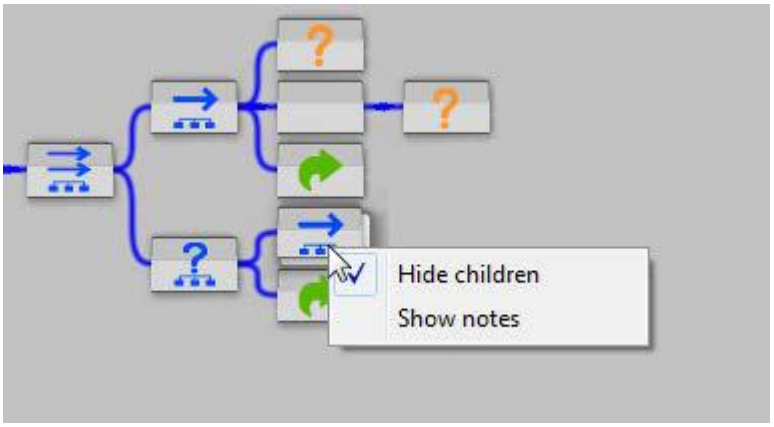
Use the Create buttons to add a new node to the behavior. Choose a node from the dropdown menu to add it to the view and then attach the node to the behavior.

Each basic type has its own folder. For example the Create new Action… - button will show all node scripts in the Actions-folder and its subfolders. You can place your own node scripts in any of the node folders, but keep in mind that the editor treats each type differently (see Creating Custom Nodes).

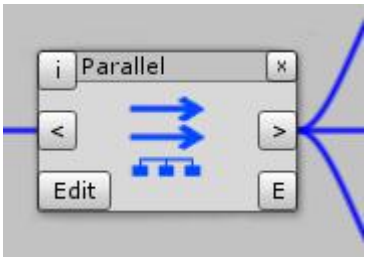
Zoom

Use the slider to zoom the behavior view. Default setting is 1 (largest). Not all controls are available, when the view is zoomed out. You can still change subtree visibility and show or hide notes by right-clicking a node. Use the zoom to get an overall view of a larger behavior.

You can also zoom in/out by pressing down control button and using the mouse wheel.



Node controls



Node Type Symbols

Most nodes have type symbols on them, which makes reading the behavior easier.

Action

Condition

Any kind of sequence

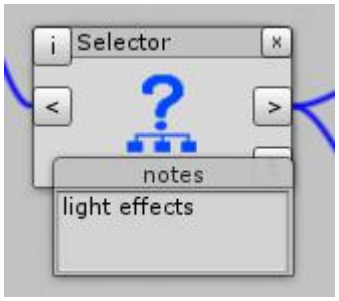
Any kind of selector

Parallel

Decorator nodes have no symbols on them.

Nodes with Notes

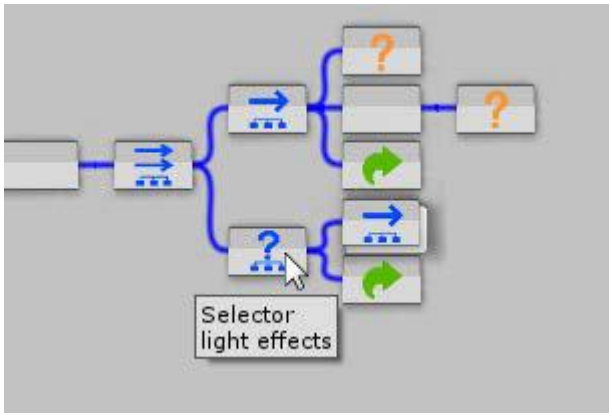
You can add notes to any node in the tree. Clicking the “i” button in the top left corner shows or hides the note field. You can keep the notes visible for any number or nodes. This can be very useful if you want to hide a big subtree, but want to keep in mind what it contains.



Notes are also shown as tooltip, when you hover over the node title.

When you save a behavior with imported subtrees in it (see Working with Sub-trees), the name of the imported file will be added to the notes. You can later edit the notes, if you don’t want to keep it.

Notes are saved with the behavior.



When the behavior view is zoomed out, the name of the node will appear on top of the notes in the tooltip.

Title Bar

The title bar has the class name of the node. If the editor view is zoomed down, the name is shown as a tooltip, when hovering over the title. The title text can't be changed. If you want to add information to a node, you can use the notes.

Deleting Nodes

You can destroy nodes by clicking the 'X' button in the upper right-hand corner of the node. You can also press 'delete' after clicking a node. Note that if the node has hidden subtrees, they will also be deleted.

Connecting Nodes

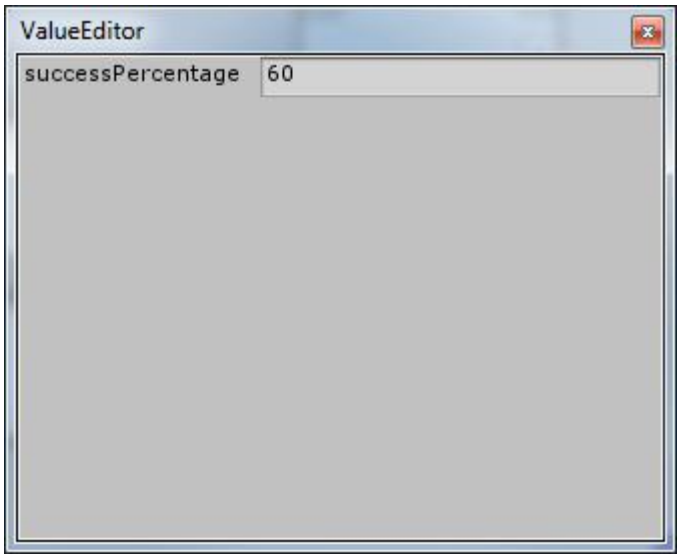
Nodes can be connected by clicking the arrow buttons on each side of the node. A left arrow can only be connected to a right arrow. Clicking the same node buttons again will break the connection.

There are some restrictions to the possible connections:

- RootNode and decorator nodes can only have one child. Trying to add another child to these node types will fail.
- Leaf nodes (actions and conditions) only have a left arrow, since they cannot have child nodes.
- Connections cannot form a loop. Trying to make a connection that would cause a loop will fail.
- All internal and decorator nodes must have at least one child.

Editing Variables

If a node class has variable properties of currently supported types, you can edit their values in the behavior editor. Clicking the Edit button on a node will open the editing dialog window.



Editable properties should be treated like constant values that will not be changed during gameplay, like limit values for conditions, timers etc., or initial values that can be referenced later.

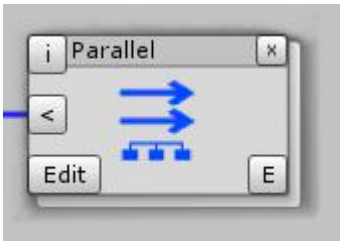
All objects that use the same behavior file, will also have the same property values.

Expanding Subtrees

If a node has children, it has an Expand button in the lower right hand corner (”E”). This is a toggle that shows or hides the child nodes and subtrees of the node. You can also hide subtrees by right-clicking a node and choosing Hide children.

When working with larger trees, it can be useful to hide parts of the tree.

Destroying a node with hidden subtrees or child nodes will also destroy all its hidden nodes.



Node with hidden children.

Hotkeys And Shortcuts

There are some shortcuts and quick access tools to help your workflow.

Space

Snap to grid.

Del

Destroy currently active node. If the node has hidden subtrees, they will also be destroyed.

Ctrl + S

Save the behavior using its current name and folder. If the behavior has no name yet, opens the saving dialog window.

Ctrl + Scroll

Use control + mouse scrollwheel to zoom the behavior view.

Right-click on Background

Open the behavior file menu by right -clicking on the background area. The chosen behavior is always added as a subtree node.

Right-click on Node

Right-clicking a node will show the visual settings of the node.

Hide children: Hide or show the node's subtrees.

Show notes: Hide or show the node's note window.

Saving / Opening Behaviors

Behavior files are saved as xml files. All behavior files are saved in the BT_Files folder. Only files inside this folder or its subfolders can be opened from the editor. You can create new folders inside the main
BT_ Files folder in the Project window and arrange your behavior files as you like.

When saving a new behavior or saving an existing behavior with a new name, you can also choose the folder where you want to place it. (see
Save Tree As…)

If you have opened the editor from the component's Edit Tree File -button, saving will also assign the behavior to the component.

All the values that can be changed in the editor (see Editing Variables) are also saved, so if you want to use the same behavior on another gameobject with different values, you have save it with another name. You can also use the gameobject's Blackboard to store unique values (see Blackboard).

Working with Subtrees

You can also create and save behaviors in smaller parts. Any behavior that you have saved can be added to another behavior tree. To add a saved behavior to another behavior, check the option "import as subtree" and choose the file you want to import. This will keep the currently edited behavior open. The opened behavior will appear to the window as a single node, with the name of the file at the bottom. The subtree node can then be attached to the edited behavior like any other node, and expanded to show the full branch.

When the full tree is saved, the imported subtree will be treated like any other part of the tree. The subtree name is added to the node's notes. The original subtree file is not changed.

WARNINGS

Tree Depth

Tree depth is the deepest level of nodes in the currently open behavior.
Since the executing code is recursive, very deep trees can cause a stack overflow.

There is no hard limit for maximum tree depth. In most cases you don't have to worry about this, but if you have problems with stack size, try reducing the tree depth.

Not Connected to Root

The editor saves the behavior recursively from the root node. Only nodes that are connected to the root will be saved in the tree file; any other nodes or subtrees in the window will simply be ignored.

Not Ending in Leaf

All branches of the behavior tree have to end in a leaf node, since only leaf nodes will define a return value. Calling an internal node or decorator that has no children will result in error.

Invalid Nodes

It is possible to open a tree file containing nodes that no longer exist in the project. Any nodes that do not represent an existing node class are
marked red. It is still possible to save the behavior for future editing, but trying to run it will result in error.

When you write a new custom node class, make sure that the script name and the class name are the same, and the node class inherits BaseNode
(see Creating Custom Nodes).

BLACKBOARD

Accessing the Blackboard

You can access the blackboard from anywhere in the game by getting a reference to the BT_Behave component.

The BaseNode class has a reference to its owner's BT_Behave, which you can use from inside a node:

```
agent.blackBoard.SetInt(5, "someInt");
```

Or if you need to use custom type data, you can use the object type dictionary:

```
agent.blackBoard.Set<customtype>(value,
"somevariable");
```

If the GameObject has several trees, they all use the same blackboard.

Typed Data

There are also typed versions of the containers. In the current version (1.1.2) these can be accessed from a BT_Blackboard instance in the BT_Behave component. They have get and set methods named by the data type (e.g. GetInt and SetInt for integers. Currently supported types are:

- int
- float
- string
- double
- bool
- Vector2
- Vector3
- Quaternion
- Color

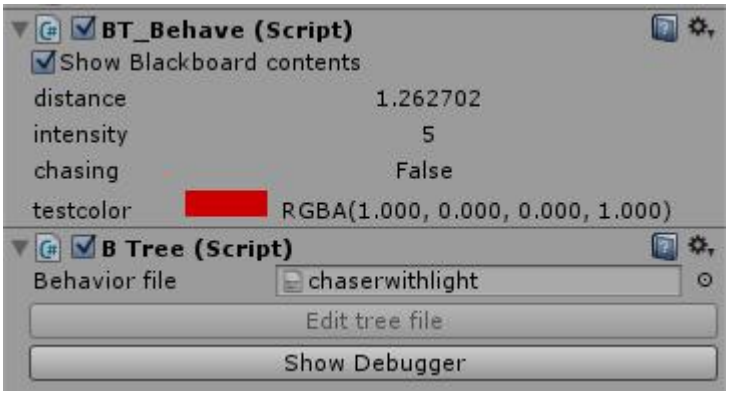
The benefit of typed containers is that they don't need boxing. Especially on mobile platforms using a lot of variables boxed to object type can cause problems.

Tips

Best practise for using the blackboard is to process the data and update it from outside the behavior itself. For example if behavior needs to know the distance to the closest enemy, this can be calculated elsewhere and only the distance saved to the blackboard. Then the behavior can use it directly.

If you need the same kind of nodes to use different values for a variable, you can define the individual variable name by storing it in a string property, which can be modified in the editor.

Choose the Show Blackboard contents –option to view the blackboard during play-mode.



VISUAL DEBUGGER

Using Visual Debugger

You can follow the behavior execution during play mode by opening the visual debugger window from the tree component. The debugger shows the tree graph, with symbols of the return value of each node after traversal.

 Return state: Success

 Return state: Failure

 Return state: Running

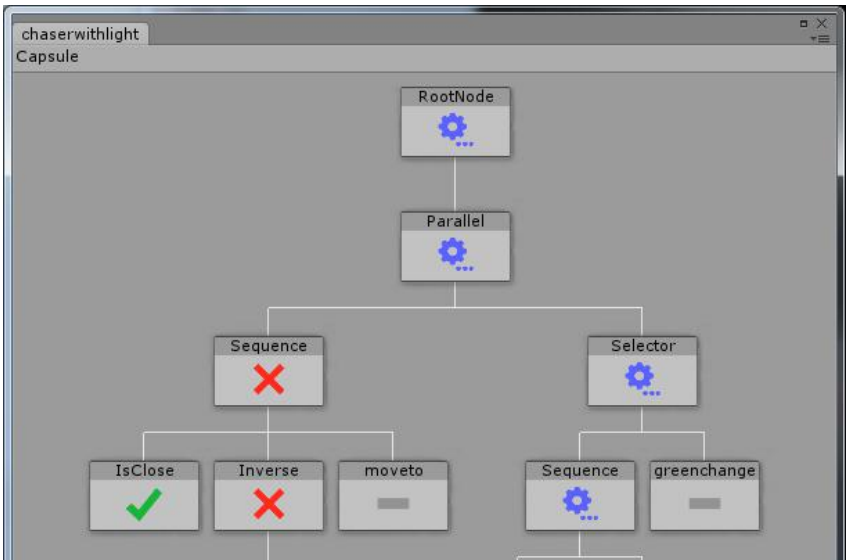
 Return state: Error

 Nodes that were not visited will show a grey rectangle.

Each tree can have its own debugger window open at the same time, even on different game objects.

The name of the tree file and the name of the owner object are displayed at the top of the debugger window.

Windows can also be moved and resized.



Show Last Run

If an error occurs in a tree, you can open the debugger from the “Show last run” -button. The button will only appear if there has been an error.

Since Unity3D does not save changes to game data during play mode, the last traversal info can only be accessed while the play mode is still on. Error info will also be shown in the log.

NODES

Creating Custom Nodes

To add a custom node into the behavior editor, you need to create a class that represents the node.

- Create a script in the appropriate folder (Actions / Conditions / Decorators / Internals). The script must have the same name as the class. Make sure the spelling is correct!
- Set the class to inherit from BaseNode.
- Override the Action() method.
    - Best practice is to keep it as simple as possible. Ideally one node should do only one thing, so the behavior is easy to modify by moving the nodes rather than rewriting them.
- Action() must return the result state of the node.

- BT_State.SUCCESS if the action is completed successfully
- BT_State.RUNNING if the action is not yet complete and should continue in the next traversal
- BT_State.FAILURE if the action fails in some expected way (e.g. a condition is not met)
- BT_State.ERROR if the action fails in an unexpected way (e.g. required data is missing). Error will stop the BTree component executing.

- You can also override other methods, if you need them:
  - _init() is for any initialization logic and is called when the behavior is first created. You can set variables to the Blackboard here. Trying to access a variable that is not found in Blackboard will create an error, but will not stop behavior execution.
  - StartAction() is for any logic needed to begin the action. It is mainly useful for actions that take more than one frame to execute. It is called once before the Action() method, but not again while it is running.
  - EndAction() is for any logic to end the running action. It is called when the Action() is finished successfully, and also if another branch interrupts the action.
- All variables that need to be updated or accessed from outside the node should be set to the Blackboard. However, you can also use local variables inside the node. If you need a value that can be set in the editor, you must make it a property.

Actions

These hold the actual functionality of the behavior. One behavior tree can execute several actions during one traversal, depending on its structure.
Actions can hold any sort of functionality.

Ideally, actions should not access other gameobjects or components directly, but save the resulting data to the blackboard.

Action nodes are mainly game specific, so they must be defined by the user (see Creating Custom nodes).

DebugWriter

DebugWriter node writes a log message every time it is accessed. The message can be edited in the Editor Window. There can be several DebugWriters in the same tree, and each can have a different message.

Conditions

It is best to put any condition checks into separate nodes, so that they can be easily moved or reused. Like action nodes, conditions are mainly user-defined (see Creating Custom nodes).

RandomCond

Returns BT_State.SUCCESS or BT_State.FAILURE depending on a random value. Success percentage can be edited in the Editor window. The range is 0-100, so that a value of 50 means a 50% chance of success.

Decorators

Decorators are nodes that modify tree execution order, or its child's return value.

A decorator can have only one child node. There are a few decorator nodes included in the behavior tool, but you can also create your own decorators.

AlwaysSuccess

Calls its child and returns BT_State.SUCCESS, unless the child returns BT_State.RUNNING or BT_State.ERROR. This can be useful for example when a failed action should not stop a sequence.

AlwaysFailure

Calls its child and returns BT_State.FAILURE, unless the child returns BT_State.RUNNING or BT_State.ERROR.

Inverse

Inverts the return value (BT_State.SUCCESS / BT_State.FAILURE). Error and running values are always returned as-is.

BT_WaitForSeconds

Only calls its child if the specified time has passed. The time limit can be set in the editor. Returns BT_State.RUNNING while the counter is running.

RepeatTimes

Calls its child the specified amount of times, unless the child returns BT_State.ERROR. The return value of the repeater is the last one it gets back from the child.

Internals

Internal nodes control the tree traversal. Internals call their children and use the returned state values to decide how to continue.

Parallel

Parallel node runs all its child nodes on every traversal. The return value of the parallel node depends on the collected values of its children: if number of failed children is greater than allowed limit, the parallel also fails. Otherwise if one or more child nodes return running, the parallel is also running. Error from any child node is returned immediately.

You can set the amount of accepted failures in the editor.

Selector

Selector runs its children in priority order, until it gets a return value success or running.

Plain Selector starts from the beginning of its children on every traversal, so a higher priority branch can interrupt a less important action.

SelectorMemory

SelectorMemory starts checking its children from the one that was successful on the last traversal.

RandomSelector

Picks one child at random and executes it.

Sequence

The Sequence node runs all its children in priority order, unless it receives a failure or error from a child node. There are two kinds of sequence nodes implemented in the editor.

Plain Sequence starts at the beginning of its children on every traversal.
This is useful for conditioned branches. A plain sequence can have a child or sub-tree that takes more than one traversal to complete; if the child is reached again

on the next traversal, it will continue its execution.

SequenceMemory

SequenceMemory is used for chaining actions that take longer to complete. It starts from the child that was left running on the last traversal. The previous children are not visited again, until the sequence has finished and can start again.