Datum: 22 juni 2017

Eindverslag afstudeerwerk

Creatures

simulatiespel met zelfvoorzienende AI

Auteurs:

Marlies Geerts — Willem Gillis

Karel de Grote-Hogeschool

Opleiding Multimedia en Communicatietechnologie

Schoolpromotor: Wim van WeyenBerg, Pieter Jorissen

Inhoudsopgave

Inleiding 3

Technische documentatie 4

Uitwerking AI 4

Wezen AI 4

Wat is Utility AI? 4

Onze implementatie van de AI 5

Overlevingsaspect 7

Reproductie aspect 9

Sociaal aspect 10

Geheugen 11

Sociaal geheugen 11

Ruimtelijk geheugen 11

Inventory 11

Genetica 12

Wezens genereren 12

Sensing 12

Pathfinding 13

Uitwerking look en feel 15

Wezen uiterlijk 15

UV scaling 15

Wezen animatie 16

Terrein 16

Uitwerking omgeving 17

Resources 17

Spawning 18

Statistieken 19

Overzicht gebruikte technieken en tools 20

Uiteindelijk gebruikte software, tools en talen 20

Bronnen 20

Opbouw eindwerk 20

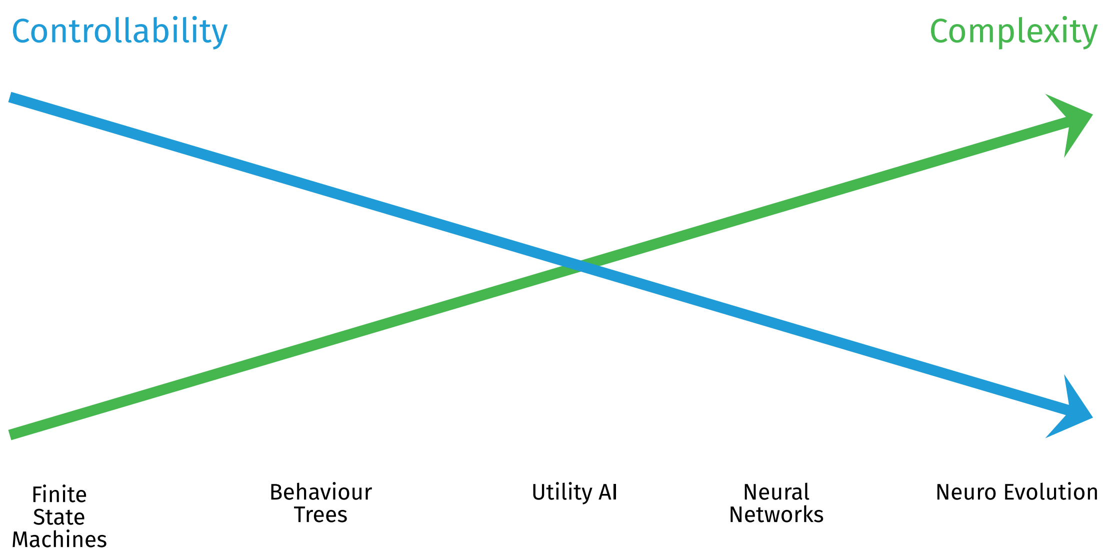
Initiële UML 22

Finale UML 23

Inleiding

We zijn begonnen aan deze thesis met het verlangen om ons te verdiepen in Artificiële Intelligentie en in het bijzonder rond het kader game development. Het merendeel van deze thesis wijdt zich hier dan ook aan. Artificiële Intelligentie of kortweg AI bevindt zich in zekere maten in alle computerspellen. Dit kan zeer eenvoudig zijn, zoals de spoken in Pac-Man, maar ook zeer complex, zoals tegenspelers in strategiespellen die menselijke tegenspelers moeten benaderen. Met de toenemende processorkracht en de trend om meer werk op de grafische kaart te leggen, groeien de mogelijkheden om steeds complexere AI te verwerken in games.

Er zijn verschillende manieren om AI te implementeren. In de afbeelding ziet u een eenvoudige vergelijking van AI technieken. In deze thesis hebben we gekozen voor Utility AI. Deze biedt een hoge complexiteit waardoor realistischer gedrag mogelijk is en is nog steeds flexibel genoeg om de AI in te stellen naar de noden van het spel. Hoe Utility AI precies werkt en wat de voor- en nadelen zijn kan u terugvinden in het hoofdstuk “Uitwerking AI”.



Bron: Unite Europe 2016 - Next Generation AI for Unity

De Utility AI hebben we geïmplementeerd in zelfgemaakte fictieve wezens. Deze wezens, lijkend op dinosaurussen, gebruiken deze AI en hun ingebouwde zintuigen om de wereld rondom hen zonder enige input van de gebruiker of centraal AI systeem te verkennen. Dat onze wezens volledig onafhankelijk kunnen rondlopen en voortplanten is uitzonderlijk voor een spel.

De wezens beschikken elk over een aantal eigenschappen; deze eigenschappen beïnvloeden hun gedrag met de omgeving en interacties met andere wezens. Sommige eigenschappen verhogen de kans dat het wezen blijft leven en als gevolg een partner kan vinden en zich kan voortplanten. De kinderen erven de eigenschappen over van de ouders en zo, door middel van Darwiniaanse selectie, blijven enkel de best aangepaste wezens over.

De data die hieruit gegenereerd wordt, tonen we op enkele grafieken in het spel. Hiermee kan er onderzocht worden welke eigenschappen na verloop van tijd succesvoller zijn dan andere. Ook zouden bepaalde situaties gesimuleerd kunnen worden, zoals hongersnood, waarvan je de effecten kunt zien in de grafieken. Welke eigenschappen zijn er dan in voordeel? De verschillende situaties kunnen dan vergeleken worden met elkaar, waardoor je eventuele voorspellingen kunt doen over een populatie dieren.

Onze doelgroep is dus tweezijdig. Langs de ene kant richten we ons op game studio’s. De AI die we hebben geschreven om onafhankelijk en realistisch te redeneren en persoonlijkheid kan tonen, kan gebruikt worden in games die een op zichzelf staande omgeving willen implementeren.

Langs de andere kant richten we ons ook in zekere mate op analisten. De AI en de fictieve wereld die we gecreeerd hebben kan gebruikt worden om populaties van dieren te simuleren.

KdG

Technische documentatie

Uitwerking AI

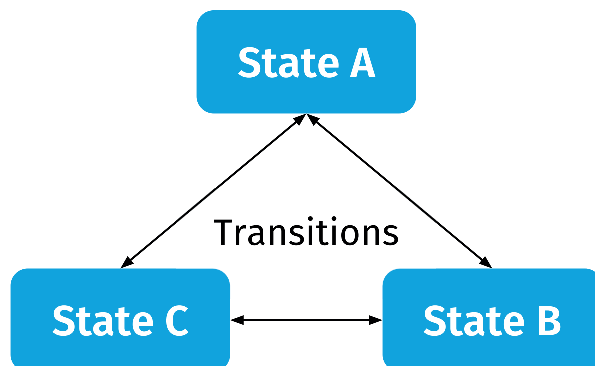
Wezen AI

Het doel van het eindwerk is om het gedrag van de wezens zo realistisch en natuurlijk mogelijk te maken. Om deze complexe AI te verwezenlijken hebben we eerst onderzoek gedaan naar AI technieken. Tijdens de lessen hadden we al kennis gemaakt met Finite State Machines, maar deze techniek voldeed niet aan onze eisen. Daarom zochten we andere technieken. Een stap dichter bij ons doel was behaviour trees. Deze zit standaard ingebouwd in Unreal Engine 4, de engine die we hebben gebruikt, maar we wilden nog meer flexibiliteit. Bovenal wilden we een AI creëren die zelfs met nieuwe, niet voorgeprogrammeerde situaties zou kunnen omgaan. Daarom hebben we gekozen voor Utility AI, een recentere AI techniek die complex gedrag toelaat terwijl het toch nog de maker ervan een degelijke hoeveelheid controle geeft.

Wat is Utility AI?

Om te begrijpen wat Utility AI anders maakt, is het belangrijk om te weten hoe enkele andere veelgebruikte technieken werken:

Finite State Machine (FSM): Veel oude games maar ook hedendaagse games maken gebruik van FSM voor de AI logica. Het is dan ook zeer geschikt om simpele AI mee te creëren.



In FSM is elke mogelijke overgang tussen de verschillende states of gedragingen (bv. slapen, zwemmen en lopen) op voorhand vastgelegd door de programmeur. Elke state heeft één of meerdere van deze overgangen. Via deze overgangen kan de AI in andere states terecht komen. Wanneer de AI in een state komt, worden enkel de mogelijke overgangen van die state nagekeken of een nieuwe overgang nodig is.

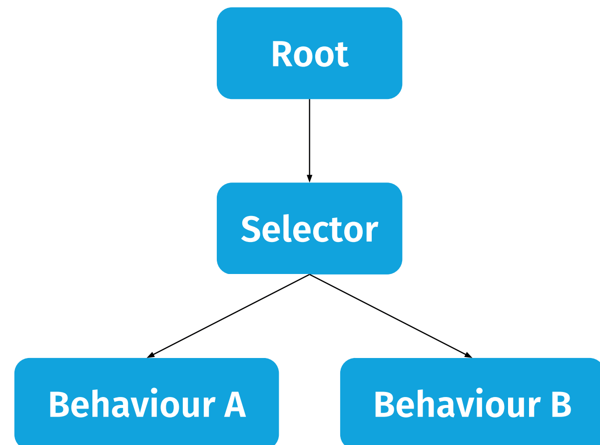
Voordelen

- Zeer eenvoudig
- Efficiënt

Nadelen

- Schaalt zeer slecht naar meerdere states
- Voorprogrammeren overgangen
- Kan vastlopen in een state

Behaviour tree: Hedendaagse games maken vaak gebruik van deze techniek. Het biedt de mogelijkheid om complexere AI te creëren dan FSM. In plaats om vanuit de huidige state verder te werken, start de behaviour tree elke keer vanuit de root "state". Door middel van selectors (bv. heeft de AI nood aan slaap? JA/NEE) gaat de AI door de hiërarchie van de behaviour tree tot de gewenste state gevonden is.



Voordelen

- Kan niet vastgeraken in een state
- Eenvoudig grafisch voor te stellen
- Schaalt beter dan FSM

Nadelen

- Schaalt nog altijd niet goed naar vele states
- Complexiteit verlaagt de performantie
- Voorprogrammeren overgangen

Utility AI: Recente games maken steeds meer gebruik van deze techniek vanwege de voordelen die het biedt tegenover FSM en behaviour trees. De techniek is behoorlijk nieuw en neemt een drastisch andere aanpak. In plaats van overgangen die opgesteld zijn door de programmeur, beslist de AI zelf wat de meest ideale state is voor de huidige situatie. De AI beoordeelt elke mogelijke state tegelijkertijd en kijkt welke state het best past bij de huidige nood. Dit werkt door van elke state een score te berekenen en daarvan de hoogste te nemen als actieve state. De score van elke state, ook wel de utility genoemd, wordt berekend door middel van zijn considerations. Considerations zijn factoren waar de state mee rekening moet houden. Dit kan bijvoorbeeld zijn hoeveel ammunitie een AI heeft en hoe ver de vijand van hem vandaan is. De considerations hebben een bepaalde input (bv. afstand in meter) en verkrijgen een genormaliseerde output (van 0-1) door middel van wiskundige curven. De effecten moeten dus niet lineair zijn. Een vijand met een mes die dichterbij komt, zal als veel gevaarlijker worden beschouwd als hij twee stappen doet en hij is redelijk dichtbij, dan als hij twee stappen doet terwijl hij nog ver weg is. Door middel van deze curves kan men dus realistische reacties instellen. De genormaliseerde scores worden dan uiteindelijk samengevoegd tot een finaal gewicht.

Voordelen

- Realistische keuzes en overgangen
- Schaalt goed
- Minder overgangsfouten door programmeur
- AI kan onverwachte oplossingen vinden
- Combinatie met FSM mogelijk

Nadelen

- Complexere initiële implementatie
- Debuggen en fine-tuning moeilijker dan FSM en behaviour trees

Onze implementatie van de AI

De implementatie van de AI die wij hebben geschreven is opgebouwd uit twee lagen. De bovenste laag wordt geregeld volgens de Utility AI theorie en de onderste door State Machines. Elke state heeft dus naast een aantal considerations ook een aantal substates. De transities tussen deze substates worden intern geregeld door de state. Op de volgende pagina staat een schema van elke state met zijn substates.

KdG

StateSystem

Chooses Utility state with highest Utility Score

Utility states

Consume
Gather
Explore
Sleep
SearchPartner
Breed
Fight
Flee
Communicate
Die

Wait
Consume

Seek
GatherItem
TakeItem

Wander
CheckResource

Seek
Sleep

Wander
Call
Seek
Confirm
Meet

Call
Wait
HomeCheck
Confirm
Breed

Seek
Fight
Search

Panic
Flee

SayHi
Wait
ShareInfo

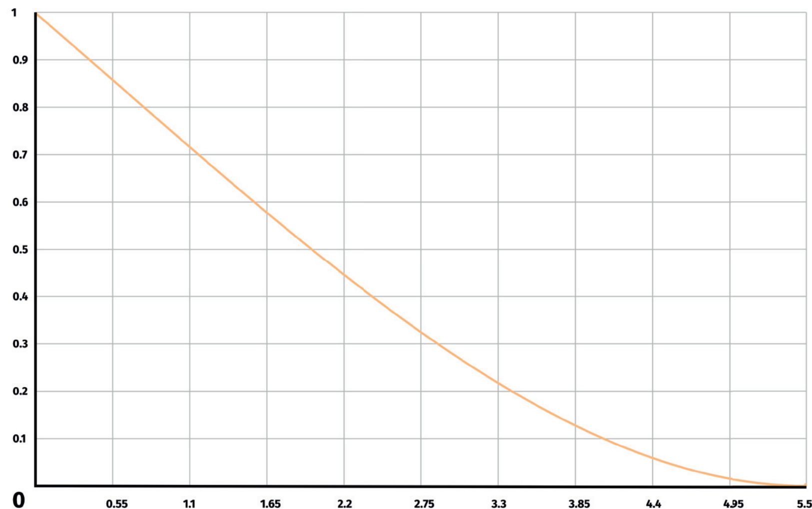
Substates of Utility states

De states van het wezen zijn onder te verdelen in drie groepen: overleven, reproductie en sociaal. Hieronder volgt een korte beschrijving van elke state met zijn considerations en/of vereisten.

Overlevingsaspect

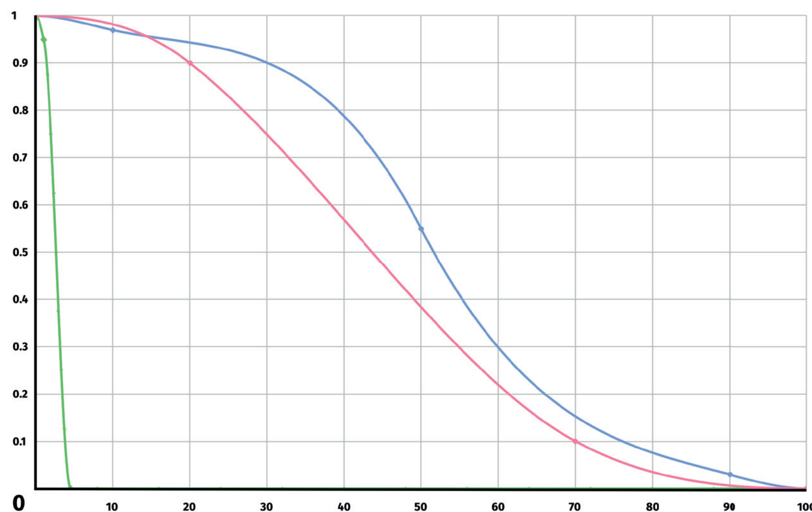
Verkennen (Explore): Het wezen verkent de omgeving en zoekt naar resources. Als het er één ziet, gaat het ernaar toe om te zien welk type het is.

- Aantal gekende locaties per type (eten, drinken, health - oranje curve)



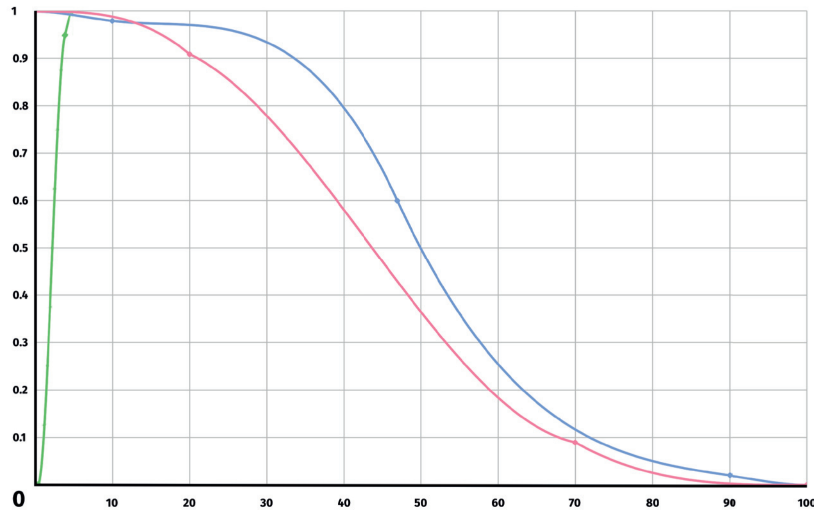
Verzamelen (Gather): Het wezen gaat naar een gekende resource om grondstoffen ervan te verzamelen. Deze state is enkel mogelijk als er resources gekend zijn.

- Honger (blauwe curve)
- Dorst (blauwe curve)
- Health (rode curve)
- Aantal items per soort (groene curve)
- Weet een relevante locatie



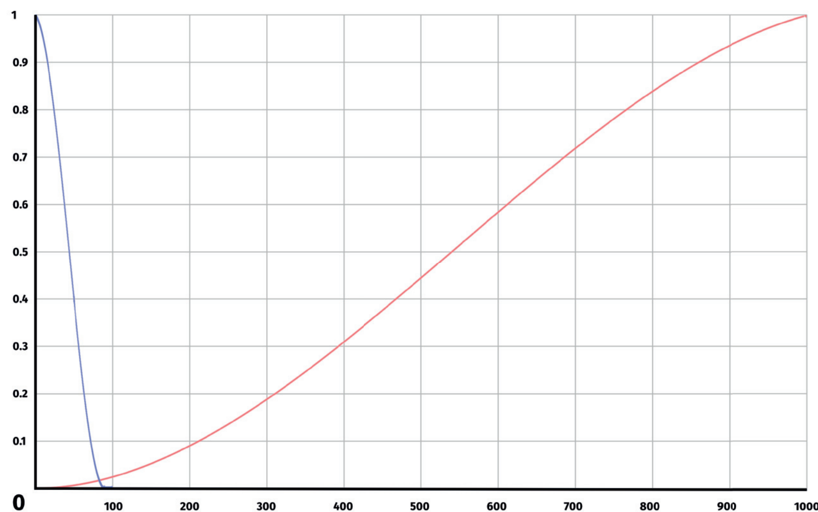
Consumeren (Consume): Het wezen eet één van de vergaarde grondstoffen van de resources op. Het duurt even voordat het gedaan heeft. Deze state is enkel mogelijk wanneer het wezen over grondstoffen beschikt in zijn inventory.

- Honger (blauwe curve)
- Dorst (blauwe curve)
- Health (rode curve)
- Heeft een relevante item (groene curve)



Slapen (Sleep): Het wezen gaat naar zijn thuislocatie en slaapt daar. Als hij niet op tijd op zijn thuislocatie geraakt, dan valt hij in slaap en vermindert zijn health.

- Slaperigheid (blauwe curve)
- Afstand tot thuis (rode curve)



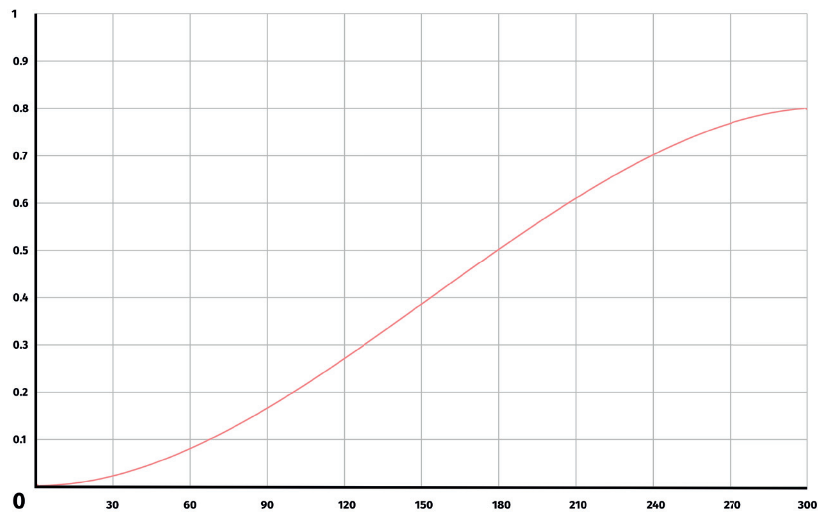
Sterven (Die): Het wezen is te oud of kon zijn basisbehoeften, zoals honger of dorst, niet op tijd voldoen. Het sterft direct. Het wezen laat een kruis achter als teken van een gestorven wezen.

- Te veel honger/dorst
- Te lage health
- Ouderdom

Reproductie aspect

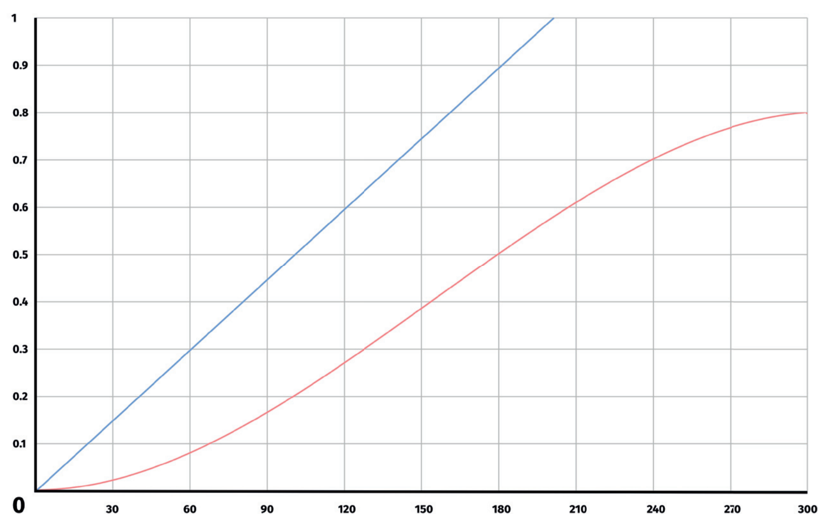
Partner zoeken (Search partner): Het wezen zoekt naar een potentiële partner: elk wezen dat nog geen partner heeft en geen rechtstreekse familie is. Het zoekt en vindt de partner door geluiden te maken, te luisteren en naar potentiële wezens toe te gaan.

- Heeft geen partner
- Volwassen
- Laatste breed tijd (rode curve)
- Hoort een wezen zoeken
- Aantal kinderen



Paren (Breed): Het wezen wil paren. Het zoekt zijn partner op in de wereld door geluiden te maken en te luisteren. Indien de partner niet gevonden wordt, gaat het naar hun thuis om te kijken of het zich daar toevallig bevindt.

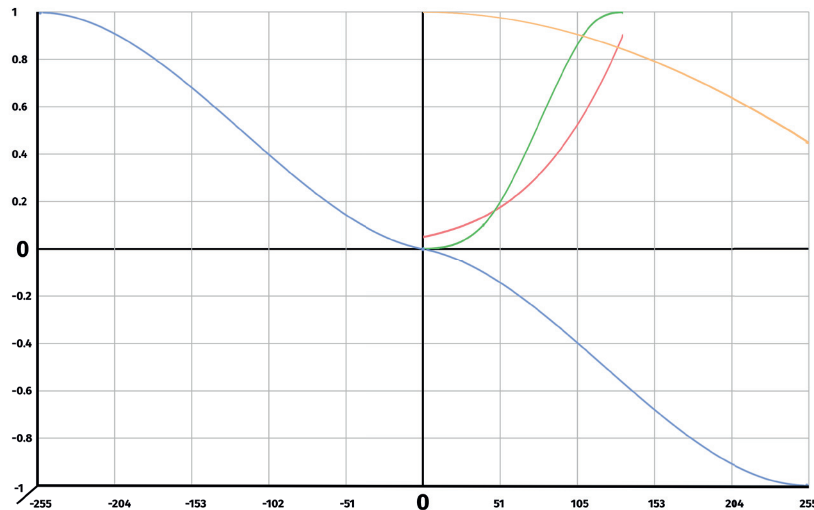
- Heeft een partner
- Laatste breed tijd (rode curve)
- Laatste tijd in breed state (blauwe curve)
- Hoort partner roepen
- Aantal kinderen



Sociaal aspect

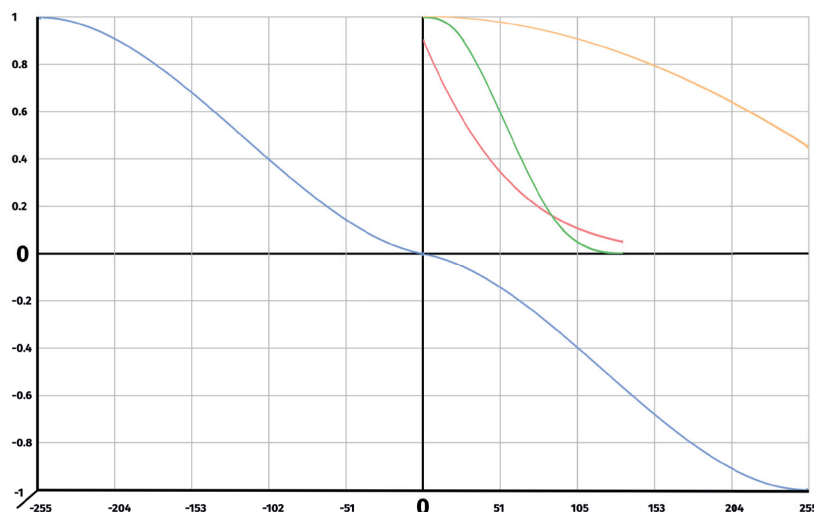
Vechten (Fight): Het wezen wilt vechten met een ander wezen. Het gaat ernaar toe en valt aan. Als het wezen wegloopt gaat het in achtervolging.

- Ziet of hoort ander wezen
- Nabijheid wezen (oranje curve)
- Relatiestatus (blauwe curve)
- Agressie (rode curve)
- Health (groene curve)
- Wezen is familie



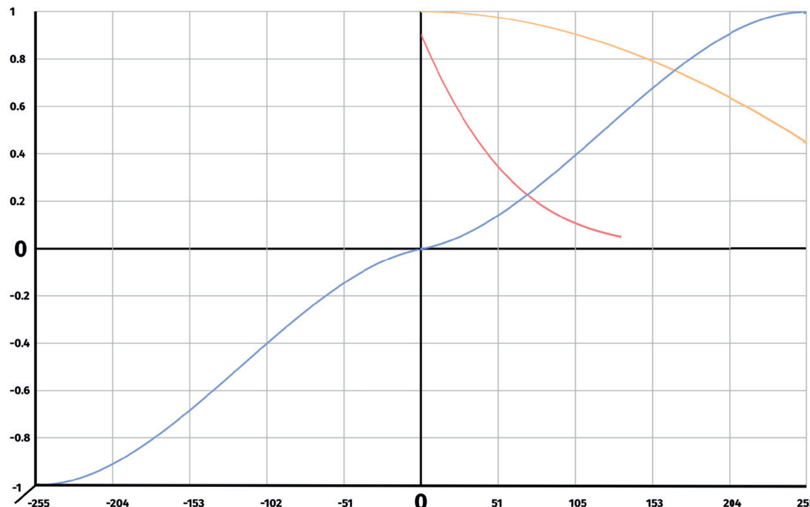
Vluchten (Flee): Het wezen heeft schrik van een ander wezen en/of is zwaar gewond en zal dus weglopen van elk gevaar in de buurt.

- Ziet of hoort ander wezen
- Nabijheid wezen (oranje curve)
- Relatiestatus (blauwe curve)
- Agressie (rode curve)
- Health (groene curve)
- Wezen is familie



Communiceren (Communicate): Het wezen gaat naar een ander wezen en zegt hallo. Als het een antwoord terugkrijgt, kan het besluiten of het informatie deelt over interessante resource plekken of niet.

- Ziet of hoort ander wezen
- Nabijheid wezen (oranje curve)
- Relatiestatus (blauwe curve)
- Agressie (rode curve)
- Wezen is familie



Geheugen

Om de AI op een realistische manier te laten werken, heeft het wezen een geheugen nodig. Dit geheugen heeft twee onderdelen: sociaal en ruimtelijk.

Sociaal geheugen

Om de contacten met andere wezens bij te houden is er een sociaal geheugen. Dit bevat de relatiestatus met elk ander wezen die het wezen ooit is tegengekomen en ook wie de kinderen, partner en ouders zijn.

- Relatiestatus: Een wezen kan positief of negatief kijken tegenover een ander wezen. Eerdere gebeurtenissen bepalen hoe positief of negatief. Hebben ze al een aantal keer gevochten met elkaar, dan kun je verwachten dat ze elkaar als vijanden bezien. Hebben ze daarentegen veel gecommuniceerd met elkaar en informatie gedeeld, dan zien ze elkaar als vrienden.

Ruimtelijk geheugen

Om de verschillende locaties bij te houden die het heeft gezien en de benodigde logica heeft elk wezen een ruimtelijk geheugen. Dit geheugen bevat de thuislocatie en de locaties van bekende en onbekende resources.

- Resources: Elke geziene resource, bekend of onbekend, wordt opgeslagen in dit geheugen. Het bevat een locatie en, indien het bekend is, ook het object type. Elk wezen weet hierdoor hoeveel resources van elk type het kent. Wanneer een wezen op een locatie terechtkomt van een resource die niet langer aanwezig is, wordt de locatie uit het geheugen verwijderd.
- Thuislocatie: Deze locatie is de slaapplaats van de wezens. Deze locatie wordt als ze jong zijn de eerste plek waar ze in slaap vallen (bij de eerste generatie) of is de thuis van de ouders (bij verdere generaties). Als ze volwassen worden kiezen ze een nieuwe plek en delen die met hun partner.

Inventory

Buiten het geheugen heeft het wezen ook een inventory. Hierin kan hij een beperkt aantal items steken om te eten of drinken onderweg. Deze items komen van de resources. Elke item neemt een bepaalde hoeveelheid plaats in in de inventory.

Genetica

Een wezen heeft bepaalde eigenschappen die vanaf zijn geboorte worden meegegeven en daarna niet meer wijzigen. Dit betreft zowel zijn uiterlijk als persoonlijkheid. De persoonlijkheidseigenschappen hebben allerlei effecten op zijn gedrag en op zijn mogelijkheden. Hieronder in het kort een lijst met de effecten van de zeven geïmplementeerde karakteristieken.

Agressie (Aggression):

- Heeft meer de neiging om te vechten dan te vluchten.

Behendigheid (Agility):

- Wandelt sneller over het terrein.
- Kan beter aanvallen vermijden.

Behulpzaamheid (Helpfulness):

- Is meer geneigd om een onbekende als vriend te zien.
- Is meer geneigd om informatie te delen met iemand.

Intelligentie (Intelligence):

- Kan sneller planten en vissen verkrijgen.
- Kan beter een weg zoeken doorheen het terrein.

Sterkte (Strength):

- Kan sneller konijnen verkrijgen.
- Doet meer schade bij een aanval.

Bestendigheid (Toughness):

- Krijgt minder schade van een succesvolle aanval.

Maximum leeftijd (Max Age):

- Kan langer leven.

Elk van deze eigenschappen heeft dus effect op de overlevingskans van een wezen in bepaalde situaties.

Wezens genereren

In het begin van het spel wordt een eerste generatie wezens aangemaakt. Hun uiterlijke eigenschappen worden volledig random bepaald binnen bepaalde grenzen. Ook hun persoonlijkheidseigenschappen worden willekeurig bepaald. Hierbij krijgt elk wezen evenveel punten die verdeeld worden over de verschillende eigenschappen.

Na de eerste generatie bepalen de eigenschappen van de ouders de eigenschappen van het kind. Voor de persoonlijkheidseigenschappen wordt het gemiddelde van elke eigenschap van de ouders genomen met een mogelijke kleine afwijking erbij om de eigenschappen van het kind te vormen. Wat betreft het uiterlijk worden telkens per onderdeel (de benen, de dikte van de onderkant, de torso, ...) de eigenschappen van één ouder overgenomen. Hierbij kan ook een kleine afwijking worden gegeven, maar niets te groot, aangezien kinderen op ouders moeten lijken. Hierdoor beginnen na een tijd de overlevende wezen op elkaar te lijken.

Sensing

Om de AI van de wezens volledig losgekoppeld te houden van de wereld en ze zo onafhankelijk mogelijk te maken, hebben we ervoor gekozen om de wezens zintuigen te geven: horen, zien en voelen. Hiermee kan elk wezen afzonderlijk de wereld leren kennen op een natuurgetrouwe manier.

Dit hebben we verwezenlijkt door middel van de AI Perception tools van Unreal Engine 4 met daar bovenop onze eigen implementatie.

Zicht: Elk wezen heeft verschillende zicht zones.

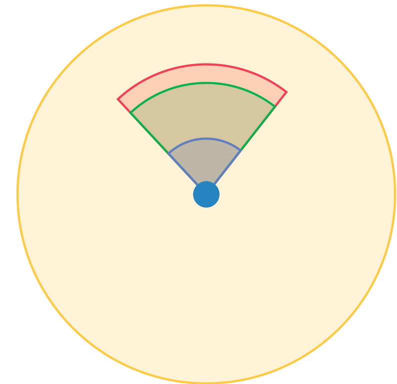
De groene zone is de eerste registratie van objecten in de omgeving. Alle objecten die hierin terecht komen worden geregistreerd als gezien. Vanaf dan weet het wezen dat er iets bestaat op de locatie van het geziene object. Deze locatie wordt ook opgeslagen in het geheugen. Het wezen weet echter nog niet wat het object is; het is nog te ver verwijderd.

De blauwe zone laat het wezen het geziene onbekende object herkennen. Wanneer het wezen dichter komt, komt het object in de blauwe zone. Nadat het object herkent is, wordt het object type samen met de locatie opgeslagen in het geheugen van het wezen.

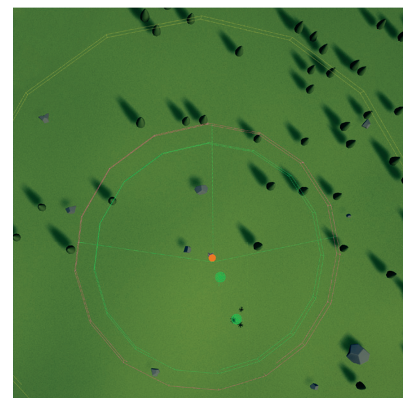
De rode zone is een bufferzone voorzien door de implementatie van Unreal zelf; objecten die eerst gezien zijn in de groene zone zullen nog steeds geregistreerd blijven als gezien zolang deze zich in deze zone bevinden. Als het object deze zone verlaat, wordt het verwijderd uit de geziene objecten. Maar eerdere interacties blijven opgeslagen in het geheugen van het wezen.

Gehoor: Door enkel gebruik te maken van zicht zouden de wezens elkaar zelden ontmoeten; de kans dat ze elkaar zien is te klein. Vandaar hebben we ook gehoor geïmplementeerd. Het gehoor werkt met een veel grotere cirkelvormige collider. Dit is de gele zone in de illustraties. Wezens kunnen geluiden maken met verschillende betekenissen, zoals bv. bij het zoeken van een partner. De wezens kunnen een onderscheid maken tussen deze geluiden en overeenkomstig reageren.

Aanraking: Wanneer de colliders van wezens of resources overlappen met elkaar wordt dat geregistreerd als aanraking. Zo kunnen de wezens interacteren met andere wezens of resources.



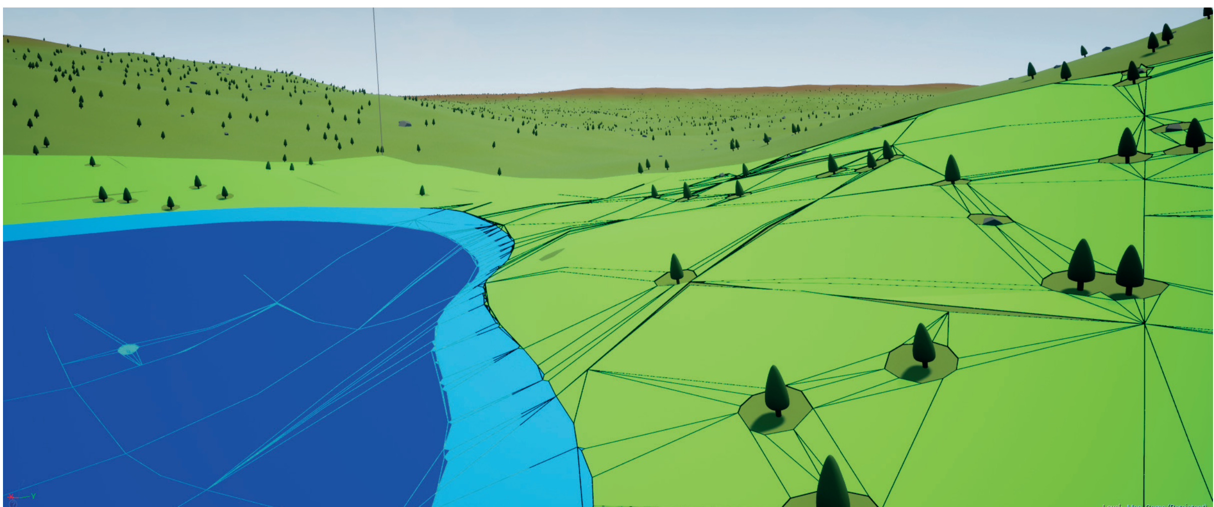
Vereenvoudigde voorstelling sensing wezens



Debug voorstelling sensing wezens

Pathfinding

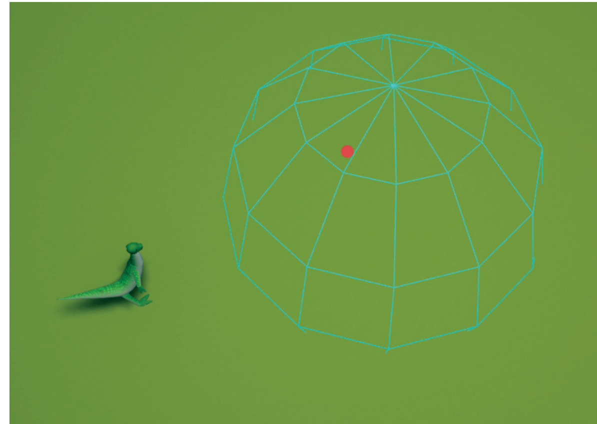
De pathfinding van de wezens wordt berekent via het ingebouwde navmesh systeem van Unreal Engine 4. Door middel van navmesh zones hebben we een opsplitsing gemaakt in water en begaanbare grond. Water op zich heeft ook een opsplitsing in twee nav area classes: één voor ondiep water en één voor diep. In ondiep water kan een wezen nog wandelen, maar dit heeft een grotere kost. Een wezen kan niet zwemmen, dus diep water is onbegaanbaar.



Debug voorstelling van de navmesh in Unreal

De navmesh maakt het mogelijk om objecten zoals bomen of stenen en steile hellingen automatisch als onbegaanbaar te registreren. Gecombineerd met de grotere kost van het water kunnen de wezens complexe paden vinden, rekening houdend met al de eerder vermelde factoren. Deze robuuste pathfinding heeft echter als nadeel dat de wezens nu altijd de meest efficiënte route nemen. De wezens moeten echter de omgeving verkennen en mogen geen absolute kennis hebben van de volledige wereld. Vandaar hebben we zelf beperkingen ingebouwd.

De belangrijkste beperking is dat wanneer een wezen naar zijn doel wil gaan, niet het volledige pad van te voren wordt uitgerekend. In plaats daarvan wordt telkens een pad berekend tot een dicht bijzijnde punt in de richting van het doel. Wanneer dit punt bereikt wordt of de timer verloopt, wordt een nieuw punt bepaald. Dit herhaalt zich tot het doel bereikt wordt.



Debug voorstelling punt

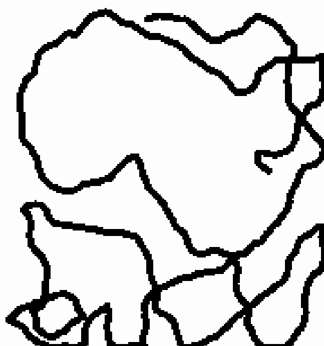
Dit heeft als voordeel dat het niet rekening kan houden met mogelijke obstakels onderweg die niet bekend waren, zoals meren of bergen. Dit geeft wat natuurlijke imperfecties in de pathfinding.

Het nadeel hiervan is dat wezens in speciale situaties kunnen vastraken. Om dit op te lossen laten we tijdelijk toe om het punt in de richting van het doel verder weg van het wezen te kiezen. Door deze zoek radius te vergroten kan het wezen een langer pad berekenen en kan het wezen hierdoor een uitweg kan vinden.

Tweede nadeel is dat de berekende route nog steeds zeer lang kan zijn. Bijvoorbeeld indien het punt zich net aan de overkant van een lang meer bevindt. Dit hebben we verholpen met een timer, welke een maximum duur voor de route bepaald.

In de Explore state hebben we ook het systeem van de tussen punten gebruikt. In deze state zorgen we ervoor dat de gekozen punten meer variatie hebben hierdoor kiezen ze sneller voor nieuwe wegen en ontdekken ze nieuwe plaatsen.

Een groot nadeel is dat de wezens hierdoor vaak in cirkels lopen. Dit hebben we geprobeerd te verhelpen door middel van render targets van Unreal. Elk wezen krijgt een RGB afbeelding toegewezen. Deze afbeelding heeft dezelfde proporties als het landschap waardoor de coördinaten daarvan overeen komen met 2D coördinaten op de afbeelding. Als het wezen verplaatst, wordt er op basis van zijn positie een punt getekend op de overeenkomstige plaats in de afbeelding. Dit doen we op twee kanalen: rood en groen. Het groene kanaal neemt over tijd in kleurwaarde af. Dit kanaal bevat de recente geschiedenis van de route van het wezen. Het rode kanaal, die niet afneemt, bevat informatie waar het wezen gedurende zijn volledige levensduur al is geweest. Bij het kiezen van een nieuw punt wordt er gekeken of die positie niet recent is bezocht en beslist of het al dan niet een goede keuze is.



Rood kanaal

Groen kanaal

RGB gecombineerd

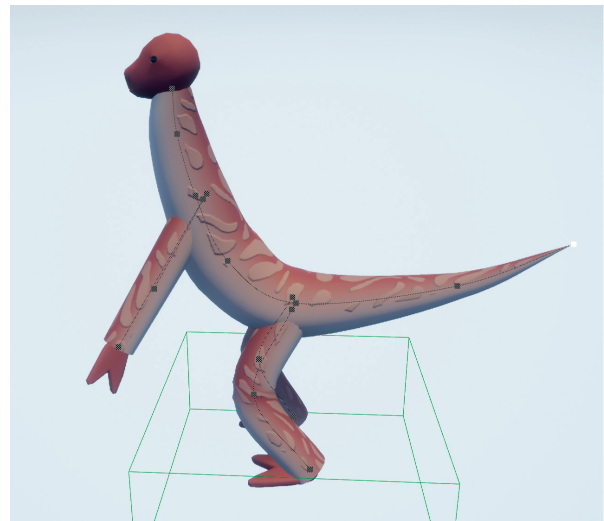
Uitwerking look en feel

Wezen uiterlijk

Om er voor te zorgen dat de evolutie visueel te volgen is en om de wezens op zich aantrekkelijker en interessanter te maken, hebben we gezocht naar manieren om het wezen te kunnen aanpassen via code. We zochten een techniek die zo veel mogelijk vrijheid bood; we stelden namelijk hoge eisen: volledige controle over de kleuren en patronen, mogelijkheid tot extra benen/armen, aanpasbare lichaamsverhoudingen, ...

De eerste concepten waren gebaseerd op een modulair systeem: we zouden verschillende modellen maken voor benen, armen en mogelijk zelfs handen en voeten. Maar ook al geeft dit veel opties, er zijn enkele beperkingen en het aanmaken van deze verschillende modellen zou te veel tijd in beslag nemen. Daarom zochten we naar andere opties.

Een ander concept was om gebruik te maken van tesselation. Unreal heeft goede ondersteuning voor tesselation en door middel van blending zouden we veel controle hebben over de vorm. Maar deze techniek was ook geen perfecte oplossing voor al onze vereisten. Uiteindelijk kwamen we op het idee om splines te gebruiken en omdat Unreal tools bevat om effectief splines en spline meshes in runtime te genereren, was de keuze snel gemaakt. De spline techniek werkt door middel van een 3D mesh, in ons geval een buis, die vervormd en gedupliceerd wordt om een spline te volgen. Voor elk punt dat gezet wordt op de spline, dus voor elk spline segment, komt er een mesh bij.



Spline wezen

Door vijf splines te combineren (torso, linkerarm, rechterarm, linkerbeen en rechterbeen) hebben we een basiswezen. De handen, voeten en hoofd zijn gewone 3D modellen, geplaatst op de uiteindes van deze splines. Met splines konden we het flexibele uiterlijk verwezenlijken waar we op hoopten. De genetica zou bepalen welke offsets tegenover het basisuiterlijk een wezen zou hebben en op basis daarvan de spline punten verplaatsen. Zo zou elk wezen er anders uitzien.

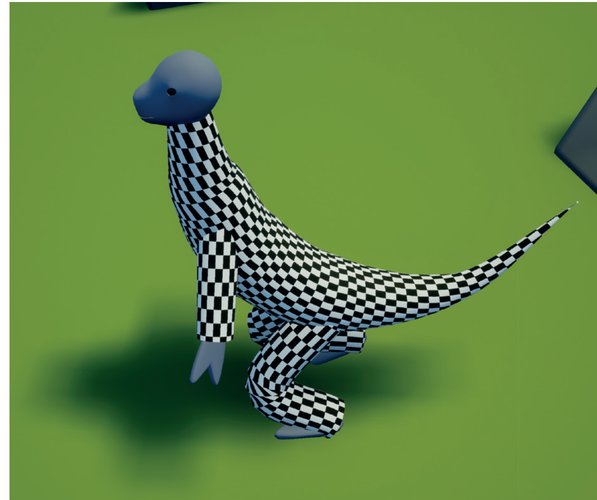
Later zouden we er ook makkelijk extra benen en armen aan kunnen toevoegen. Vanwege tijdsbeperking zijn we niet verder gegaan dan variaties in de lichaamsverhoudingen, maar de mogelijkheden met onze opzetting is er wel degelijk.

Voor de kleuren en patronen te maken hebben we gekozen voor Substance Designer. We hadden al ervaring met deze software en de procedurele workflow ervan was uitermate geschikt. Via Designer genereren we in run-time een unieke texture per wezen, zowel een albedo als een normal map. Wederom op basis van de genetica wordt de kleur en het patroon bepaald.

UV scaling

De splines van de torso en benen bestaan uit meerdere spline segmenten. Dit zorgde voor een probleem met de UV's van de complete splines. Elke onderdeel werd ofwel uitgerekt of geplet. Dit gaf lelijke artifacten en was niet bruikbaar voor de huidpatronen die we genereerden. Om dit op te lossen hebben we de data van het aanmaken van de spline doorgegeven aan het materiaal zelf. Het materiaal schaalt dan intern de UV's in de X/U en Y/V-as, afhankelijk van de grootte van het spline mesh segment en zijn omtrek. Dit lost de vervorming op, maar geeft seams omdat het begin en einde van de UV's niet meer aan elkaar grenzen. Dit hebben we opgelost door daarbovenop de UV's nog een translatie te geven op basis van de lengte van alle voorgaande spline segmenten. Hierdoor worden de UV's van de segmenten als het ware achter elkaar gelegd en vermijden we de seams. Het resultaat ervan ziet u op de afbeeldingen. De vierkanten zijn veel gelijkmatiger verdeeld rechts dan links.

Voor behandeling



Na behandeling

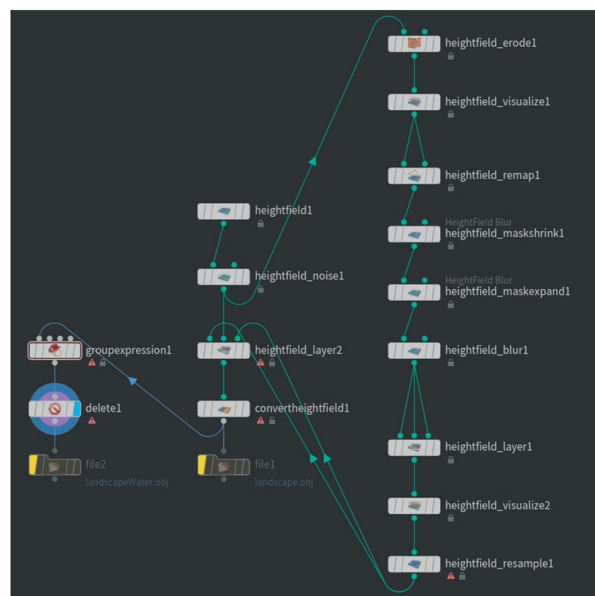
Wezen animatie

De animaties maken voor het spline wezen was niet voor de hand liggend. In een gewone workflow animeert men door middel van een bone skelet dat geskind is aan een mesh. Aangezien onze wezens bestaan uit meerdere meshes die in runtime worden gegenereerd, en dus geen bone skelet kunnen hebben, moesten we een creatieve oplossing zoeken. Onze eerste implementatie gebruikte uitsluitend code. Er zouden functies geschreven worden om de staart, armen, benen, ... op en neer te bewegen. Een wandelanimatie zou dan een combinatie zijn van al deze functies in een bepaalde volgorde. Op deze manier zouden we de animatie bibliotheek opbouwen. Dit zou uiteraard veel tijd in beslag nemen en het nog niet makkelijk maken om deftige animaties te maken. Daarna bedachten we een alternatieve manier: we exporteerden het spline wezen, maakte een conventioneel bone skelet en animeerde het op de normale manier. In Unreal vragen we enkel de locaties en rotaties op van de bones van deze animatie en geven deze door aan de spline punten. Op deze manier kunnen we de animaties snel en eenvoudig maken in een 3D software pakket en kan het toch werken met de splines, ondersteund het meerdere ledematen en werken de genetisch afwijkende lichaamsverhoudingen.

Terrein

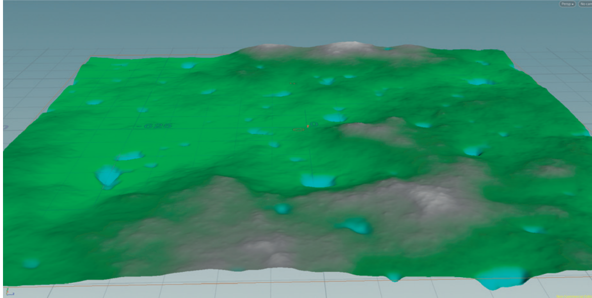
Om het terrein waar de wezens op gingen lopen te maken, wouden we eerst de landscape tool in Cinema 4D gebruiken om het simpel te houden, maar deze gaf niet behoorlijke resultaten in een beperkte tijd. Daarna hebben we Houdini 16 geprobeerd met zijn vernieuwde terrain tools. Hiermee lukte het al heel snel om een mooi landschap te maken.

Voor de meren hebben we verder gewerkt met deze Houdini file en wat geprobeerd om deze te maken via de erosie tools. Deze tools genereren mooie uitsnijdingen voor meren en ook rivieren op natuurlijke plaatsen. Het plaatsen van de meren in Unreal hebben we handmatig gedaan, aangezien we maar met één terrein gingen werken en het dus niet de moeite was om een automatisatiesysteem te bedenken. De heightinformatie van het gemaakte landschap hebben we gebaked in een afbeelding, dus een heightmap gemaakt, en gebruikt om in Unreal te importeren.

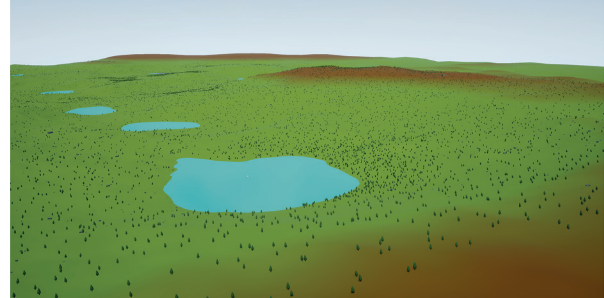


Node setup in Houdini

De texturing van terrein is een simpele lerp van twee kleuren en normals op basis van de hoogte. Bomen en stenen instanties werden handmatig geplaatst met de foliage tool van Unreal.



Resultaat in Houdini



Resultaat heightmap in Unreal

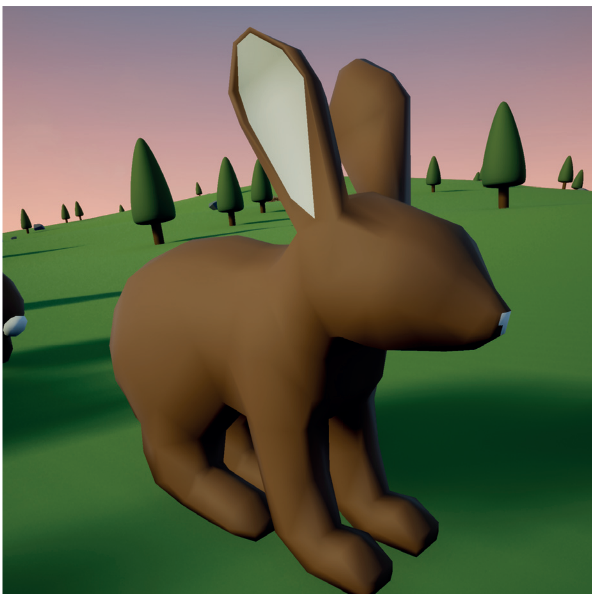
Uitwerking omgeving

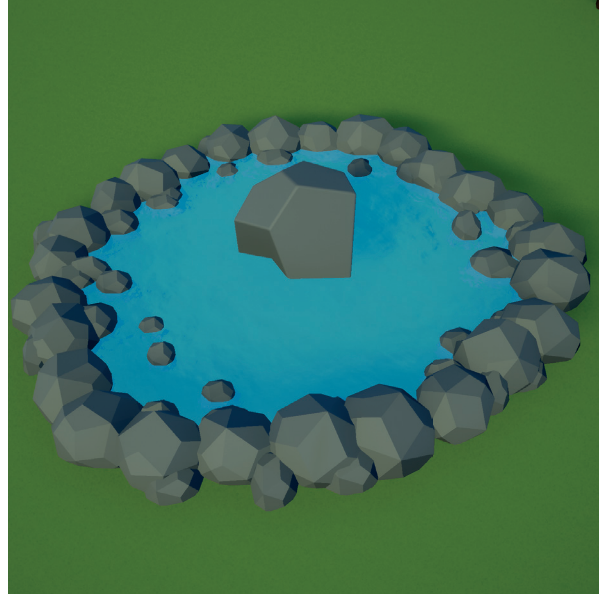
Resources

Resources zijn de objecten in de wereld waar de wezens hun basisbenodigdheden van kunnen verkrijgen. Er zijn vier type resources, elk met een eigen nut. Al deze resources zijn uitputbaar, wanneer de resources gebruikt worden neemt de hoeveelheid af. Het afnemen is ook visueel zichtbaar door het aantal dat verminderd. Indien de resource volledig uitgeput is wordt de resource verwijderd. Waarna er een nieuwe spawned op een andere random locatie.

- Konijnen: vult de body health van het wezen aan en geeft eten.
- Planten: maakt het wezen minder slaperig en geeft een beetje water en eten.
- Vissen: voornaamste bron van eten.
- Water: voornaamste bron van water.

Elk object van een resource heeft een bepaalde benodigde tijd om het te verzamelen en een bepaalde tijd om het op te eten.

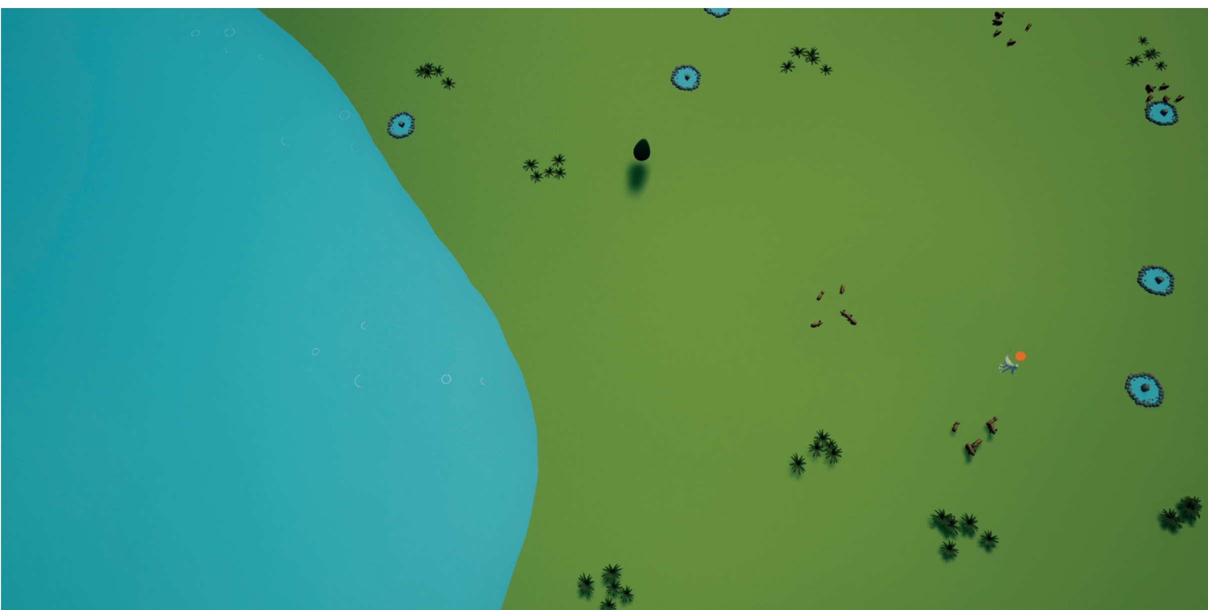




Spawning

Wezens en resources worden op willekeurige plaatsen in de wereld gespawned. Hiervoor hebben we de navmesh van de wezens hergebruikt (zie “Pathfinding” voor meer informatie). Door de spawning op basis van de navmesh te regelen voorkomen we dat objecten zich op onbereikbare plaatsen bevinden. Bovendien is de opsplitsing van water en begaanbare grond een grote hulp om bepaalde objecten enkel op land (planten) of enkel op water (vissen) te spawnen en het is eenvoudig uit te breiden voor extra gebieden (bv. bergen, moerassen, ...).

Unreal bevat ingebouwde functies om random punten te vinden in een nav area zone. Deze zijn echter niet betrouwbaar en hebben we moeten uitbreiden met eigen code. Nadat we een random punt hebben gevonden in de navmesh in een gewenste zone, moeten we het nog raycasten op het terrein om er zeker van te zijn dat het zich wel degelijk op de grond bevindt. Deze raycast gebruiken we ook om te testen of er zich op deze locatie al een resource bevindt. Indien er al één aanwezig is, zoeken we naar een nieuwe locatie. Een geslaagde raycast geeft ons ook de juiste normal ten op zichte van het terrein. Dit gebruiken we om objecten met een juiste oriëntatie te spawnen. Anders zouden de resources door de grond steken op een helling.



Spawning van de resources

Statistieken

Uiteindelijk kunnen we data vergaren. Dit doen we door de simulatie over een langere periode te laten lopen. Om sneller resultaten te bekomen kunnen we ook de simulatie tot twintig maal versnellen. Over tijd passen de wezens zich aan aan de omgeving, doordat wezens die betere karakteristieken hebben ook meer kans hebben om te overleven en kinderen te krijgen. Deze evolutie kunnen we volgen op drie grafieken:

Populatie (Population)

- Totale populatie
- Populatie met een partner
- Volwassen populatie

Karakteristieken (Characteristics)

- Gemiddelde van elke karakteristiek (zie hoofdstuk “Genetica”)

Ouderdom (Aging)

- Hoogste leeftijd
- Gemiddelde leeftijd

In het onderstaande voorbeeld zien we dat de maximum leeftijd en sterkte toeneemt. Dit komt doordat deze wezens sneller voedsel/konijnen kunnen vergaren en langer kunnen leven. Dit terwijl de agressie afneemt omdat deze wezens vaker vechten en bijgevolg schade oplopen.

KdG

Overzicht gebruikte technieken en tools

Uiteindelijk gebruikte software, tools en talen

Hieronder staat een lijst met alle uiteindelijk gebruikte software, tools en talen:

- Unreal Engine 4.15
- Cinema 4D R18
- Houdini 16
- Blender 2.78
- Substance Designer 6
- Adobe Illustrator CS5
- Visual Studio 2017
- Qt Creator 4.2.1
- C++
- Kanban (Trello)
- Git en Git LFS

Bronnen

- Bibliotheek voor onze naamgenerator van de wezens komt van de officiële US Census Data (www.census.gov/topics/population/genealogy/data/1990_census/1990_census_namefiles.html). Dit idee hebben we van een andere naamgenerator overgenomen.
- Enkele tutorials om Unreal Engine te leren kennen, zoals de officiële “Getting Started with UE4” (www.youtube.com/user/UnrealDevelopmentKit).
- Watermateriaal (www.youtube.com/watch?v=KPmRV1Z9ikY) en selectiemateriaal (www.youtube.com/watch?v=uwi6UDiNNX8).
- Belangrijke bronnen bij het leren van de Utility AI: “An Introduction to Utility Theory” van David Graham (www.gameaipro.com/GameAIPro/GameAIPro_Chapter09_An_Introduction_to_Utility_Theory.pdf) en Dave Marks presentaties (intrinsicalgorithm.com/IAonAI/2013/02/both-my-gdc-lectures-on-utility-theory-free-on-gdc-vault).

Opbouw eindwerk

Hieronder een opsomming van waar we in de verschillende fases van het project mee bezig waren.

Voorbereiding

- Lectuur gelezen en bekeken over C++ en Unreal Engine
- Veel besproken en gebrainstormd over mogelijke zaken die we erin konden steken
- Base en stretch goals opgemaakt: nagedacht over wat we specifiek erin zouden steken en welke dingen we zouden laten voor als we tijd over hadden
- Nagedacht over nodige klassen en klassendiagram aangemaakt (zie “Initiële UML”)
- Nagedacht over taken en Trello opgesteld met verschillende onderdelen

Start

- Structuur van project opgezet: klassen aangemaakt en aangepast waar nodig om in framework Unreal te passen
- De wereld opgebouwd voor de wezens: het terrein, de resources, de camera
- Schetsen gemaakt van het wezen
- Basisopbouw wezens: eigenschappen genereren, structs, ...
- Onderzoek AI

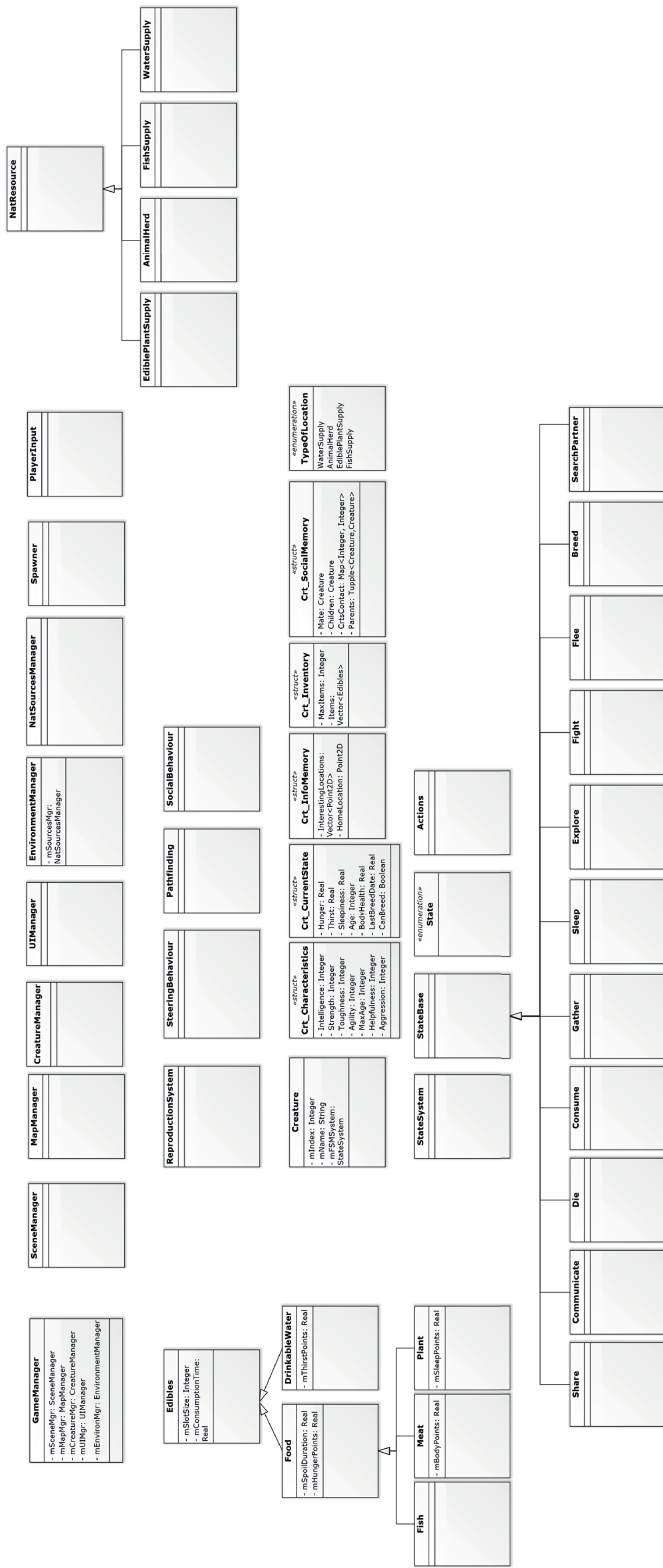
Midden

- Splines uitgeprobeerd en uitgewerkt voor basiswezen
- Pathfinding en sensing
- De verschillende states van het wezen elk apart uitgewerkt en getest
- Animatie splines
- Utility AI logica
- Random genereren van uiterlijk (Substance en splines)
- Grafieken

Afwerking

- Fine-tuning Utility AI
- Populatiebeheer: overleven en genoeg kinderen krijgen
- Bugfixing
- Data genereren

Initiële UML



Finale UML