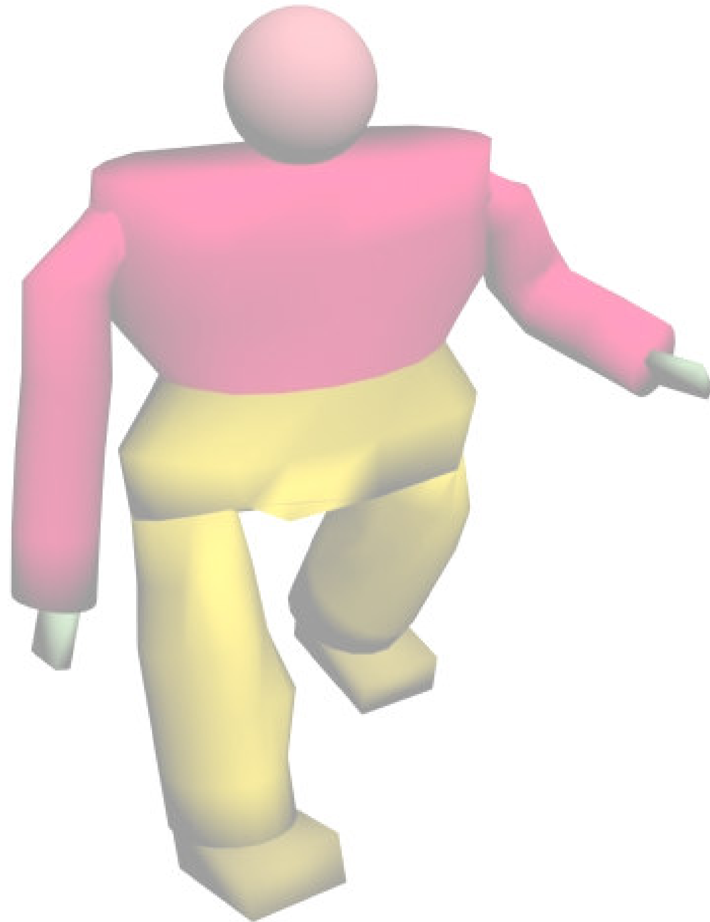
Virtual story Visualiser

ViVi

- Een add-on op de Virtual Story Teller -



Enschede, 20 juli 2005

Rick van der Werf
9906037
werf@cs.utwente.nl

Ivo Nouwens
9909273
nouwens@cs.utwente.nl

Samenvatting

Deze vrije opdracht behandelt het implementeren van een generieke visualisatie, die aansluit op een verhaal dat is gegenereerd door de Virtual Story Teller. Dit verhaal, eenmaal geparsed naar een leesbaar formaat, dient als invoer voor het programma Virtual story Visualiser (ViVi). Als deze input wordt uitgelezen, zullen hiervoor vooraf gedefinieerde locaties (die het decor vormen) en entiteiten (die de acteurs representeren) worden geladen. Tijdens de animatie zullen deze entiteiten de acties, zoals beschreven in het verhaal, in de juiste volgorden weergeven.

Deze implementatie is een van de laatste stappen die te ondernemen is in het ontwikkelproces van de Virtual Story Teller. Zodra er een verhaal gegenereerd kan worden, kan dit ook een grafische vorm aannemen, hetgeen een geheel extra dimensie biedt. Hoewel deze V.S.T. nog niet is voltooid, is het al wel mogelijk een voorstelling te maken van een mogelijke vorm van output, die de V.S.T. direct, of middels een parser zou kunnen leveren. Voorlopig wordt volstaan met een input van fictief formaat, waarbij elementaire acties en de uitvoerende entiteiten staan vermeld.

Het is een van de doelen om een generieke structuur aan de implementatie te geven, waardoor het mogelijk blijft om uitbreidingen te blijven maken, zodra eenmaal een basisversie beschikbaar is. Modellen en animaties van acteurs kunnen blijvend worden verbeterd en het scala aan acties kan worden uitgebreid.

Inhoudsopgave

Inleiding 4
Hoofdstuk 1. Opdracht en doelstelling 5
1.1 Ogre? 5
Keuze verantwoording *5*
Ogre: algemeen *5*
1.2 Doelen 6
Uitbreiding *6*
Graphics and Virtual Reality *7*
Hoofdstuk 2 ViVi: de werking 8
2.1 Input 8
actionlist.txt *8*
Locationlist.txt *9*
Unitlist.txt *10*
robot.txt, ninja.txt *10*
2.2 Plaatsing 10
2.3 Running 11
Hoofdstuk 3 ViVi: High level code beschrijving 12
3.1 Code walkthrough 12
Voorbereidingen *12*
Het animatieproces *12*
3.2 De afzonderlijke klassen 13
Hoofdstuk 4 ViVi: Low level code beschrijving 17
4.1 De afzonderlijke klassen: 17
Hoofdstuk 5 ViVi: Karakters en Animaties 23
5.1 De animaties 23
5.2 Humanoid 24
Hoofdstuk 6 Uitbreidingen 25
6.1 Een nieuw karakter met animaties 25
6.2 Een nieuwe actie in ViVi 25
6.3 Een nieuwe locatie in ViVi 25
6.4 Beperkingen 26
6.5 Overige uitbreidingen 27
Conclusies 28
Referenties 29

Inleiding

Op de leerstoel Taal Kennis Interactie van de afdeling Technische Informatica aan de Unitversiteit Twente wordt door verscheidene mensen gewerkt aan een project, dat ten doeleinde heeft een programma te maken dat verhalen gaat vertellen, waarbij multi agent aspecten worden gecombineerd met speech and language processing. Een welkome uitbreiding op dit werk in uitvoering zou een manier zijn om een gegenereerd verhaal ook weer te kunnen geven, te animeren, waarbij mogelijk de discipline graphics and virtual reality aan het project wordt toegevoegd. In deze vrije opdracht, uitgevoerd door 2 doctoraal studenten technische informatica, moet worden gezocht naar een manier om dit te realiseren. Tevens moet een eerste implementatie gemaakt worden, die de basis legt voor deze visualisatie.

Tijdens de zoek- en oriëntatiefase hebben verschillende manieren van realisatie de revue gepasseerd, elk met voor- en nadelen die verwacht kunnen worden. Maar er is uiteindelijk gekozen voor een oplossing, die een ruim scala aan mogelijkheden openstelt: 3D. Hierbij is het mogelijk eigen modellen en omgevingen te maken en eigen acties en animaties te ontwerpen. Met behulp van goed gedocumenteerde en uitgebreide software als 3dsMax en het Ogre framework kan, ook nadat eenmaal een basis is gelegd, nagenoeg onbeperkt worden verbeterd en uitgebreid.

Hoofdstuk 1. Opdracht en doelstelling

Voor de realisatie van een animatie is gekozen voor een 3D oplossing. Dit biedt aanzienlijk meer mogelijkheden (en ook moeilijkheden) dan een twee dimensionale realisatie. Gebruikmakend van 3D, kan ook nadat aan de opdracht (die een basis moet leggen) is voldaan, onbegrensd door anderen worden uitgebreid en verfijnd.

1.1 Ogre?

Keuze verantwoording
Na enig zoeken naar een manier dit te realiseren is gekozen voor het jonge Ogre framework [OGR05]. Dit is een programmeeromgeving in C++ waarmee 3D visualisaties gerealiseerd kunnen worden. Het is een zeer uitgebreid raamwerk, waarin een ruime basis gelegd, welke goed gedocumenteerd is [API05]. Rotaties, translaties, alle basis principes van graphics & virtual reality liggen hier al in geïmplementeerd dus Ogre maakt het mogelijk om direct aan de slag te gaan. Zo onstaat een zeer lage drempel voor alle mensen met een interesse voor G&VR en kan direct met het 'leuke' gedeelte worden gestart, zonder eerst een basis te hoeven leggen. Dit heeft tot gevolg gehad dat er wereldwijd grote belangstelling en participatie aan dit raamwerk is ontstaan, waardoor er een uitgestrekte hoeveelheid aan informatie beschikbaar is. Veel gestelde vragen, veel voorkomende problemen en een forum, waarop elke vraag gesteld kan worden en binnen zeer redelijke tijd (soms minder dan een dag) beantwoord wordt maken het erg toegankelijk. Aanvankelijk lijkt het een enorme aangelegenheid om met Ogre te starten, maar er is online een goede handleiding voor verschillende platformen en programmeeromgevingen beschikbaar, die stap voor stap toelicht wat er gedaan moet worden. Zodra de omgeving eenmaal werkt, zijn er een aantal tutorials, die de nieuwe gebruiker aan de hand nemen en heel geleidelijk de principes en mogelijkheden toelichten. Omdat het framework nog jong en (nu al) zeer populair is, worden er regelmatig bugs gefixed en uitbreidingen gemaakt. Voor deze basis versie is gewekt met Ogre3D versie 1.0.1

Ogre: algemeen.
Ogre kent een aantal voorgedefinieerde entiteiten (robot, ninja, ogrehead) en sommige hiervan hebben animaties zoals 'walk'. Het is mogelijk om zelf nieuwe entiteiten en animaties te maken met behulp van 3D modellering software als 3D Studio Max [3DS05]. Voor deze opdracht is een eigen model (entiteit) ontwikkeld (zie hoofdstuk 5) en is een tutorial gemaakt, die de basis van het maken van een dergelijk model in 3D Studio Max toelicht.

Wanneer de Ogre applicatie gestart is, zijn het deze entiteiten die zichtbaar zijn in de omgeving, de locatie, en dus moeten deze entiteiten het verhaal gaan weergeven. Een locatie is opgebouwd uit een 'boom' van Nodes en elke zichtbare entiteit hangt aan zo'n Node. In het midden van de scène zit de 'parent Node' en deze heeft een aantal 'child Nodes'. Wanneer een entiteit over het scherm beweegt, is het daadwerkelijk een van de child Nodes, die onderhevig is aan een verandering. Dit zullen voornamelijk translaties en rotaties zijn. Verder kan een entiteit een animatie aan of uit hebben staan: wanneer er een humanoid-karakter over het scherm loopt is dat een Entity ("humanoid"), met de animatie ("walk"), hangende aan een node, die elk frame (afhankelijk van de ingestelde loopsnelheid) over een bepaalde afstand wordt getransleerd in een bepaalde richting.

Ogre start standaard een applicatie klasse. Hierin ligt de link naar ViVi: er wordt een nieuwe klasse gedefinieerd, die erft van deze applicatie klasse, en zodoende gestart wordt. Zolang aan de juiste syntax voldaan wordt (zie hoofdstuk 2) en alle entiteiten, locaties en acties die een rol hebben zijn gedefinieerd, kan nu in principe elk verhaal worden opgegeven.

1.2 Doelen

Uitbreiding

Naast uiteraard het realiseren van de visualisatie van het verhaal (inmiddels ruim besproken), wat het primaire doel is, wordt ook gesteld dat de implementatie uitbreidbaar is. Er moet op doorgewerkt kunnen worden door andere mensen op latere tijdstippen. En er moeten wijzigingen kunnen worden gemaakt, wanneer de Virtual Story Teller anders zal uitpakken dan voorspelt en de implementatie van ViVi niet meer voldoet. Op dit doel wordt op de volgende 2 manieren rekening gehouden en geanticipeerd:

- De structuur, inhoud, naamgeving en opbouw van de code:
  - o Er is een duidelijke onderverdeling gemaakt in functionaliteit, die zich vertaalt naar een opsplitsing in verschillende klassen. (zie hoofdstuk 3 en 4). Voor alle grote processen en datatypen is een aparte klasse aangemaakt, en de functionaliteit is zoveel mogelijk hiernaar ingedeeld. Zodoende is een bepaald proces makkelijk in de code terug te vinden.
  - o Er is een uitgebreide en goede documentatie bij de code geleverd. Het gratis verkrijgbare programma “cppdoc” [CPP05] leest de documentatie uit de .h files en levert een html documtent op [DOC05], vergelijkbaar met javadoc.
  - o Voor het uitlezen van een verhaal wordt gebruik gemaakt van een tekstbestand opdat dit verhaal zich makkelijk laat wijzigen. Dit is verder doorgevoerd naar de acteurs, de StoryUnits. Elke (type) unit heeft zijn eigen tekstfile, met daarin de waarden van de eigenschappen van deze unit. Als nu een nieuwe wordt toegevoegd, kan dat eenvoudig middels een nieuwe file, met de juiste naam en daarin de juiste waarden.
- Er zijn een aantal tutorials geschreven. Eén geeft aan hoe er te werk gegaan moet worden bij het maken van een (actie) uitbreiding: waar krijgt de gebruiker mee te maken, wat moet hij zoal aapassen en waar moet allemaal opgelet worden? Een andere tutorial doet hetzelfde voor het maken van een nieuwe locatie. En er is een tutorial, die het maken van een nieuw, eigen karakter (zie ook hoofdstuk 5) in 3dStudioMax behandelt. Ook hierbij is goed omschreven waar aandacht aan besteed moet worden, wat gebruikelijke hindernissen zijn en hoe dit model tenslotte gebruikt kan worden in ViVi.

Zodoende wordt een mogelijke drempel om met deze implementatie verder te gaan aanzienlijk verlaagd en wordt het beginnen hiermee voor toekomstige gebruikers vereenvoudigd.

Graphics and Virtual Reality

Verder zijn er nog twee doelen gesteld; dit zijn meer persoonlijke doelen en hebben verder geen toegevoegde waarde voor de implementatie.

- Programmeren: een opdeling naar functionaliteit heeft enige betrekkingen met het toepassen van informatie systemen: data voor verschillende aspecten wordt in verschillende klassen opgeslagen. Ook de objectoriëntatie en hoe dit te realiseren in de C++ taal hebben een grote rol gespeeld.
- G&VR: beide studenten hebben met plezier deelgenomen aan het keuzevak dat dit onderwerp behandelt. In dit vak werd de basis behandeld en dat kan nu worden voortgezet. Eigen modellen en animaties kunnen worden ontworpen, en er wordt een heel verhaal geanimeerd.

Hoofdstuk 2 ViVi: de werking

Zoals reeds vermeld in hoofdstuk 1 zal dit gaan aansluiten op de Virtual Story Teller. Er is geanticipeerd op de output die deze zou kunnen leveren. Aangezien de daadwerkelijke output van de VST nog niet bekend is, zal daar waarschijnlijk t.z.t een nieuwe parser voor geschreven moeten worden, waar bij op een aantal zaken moet worden gelet. Deze punten worden hieronder behandeld. De huidige implementatie van de parser is zeer eenvoudig (en dus foutgevoelig) gehouden, omdat er uiteindelijk een nieuwe geschreven zal moeten worden

2.1 Input

Er zijn een aaltal tekst files, die de input vormen, daarnaast zijn er nog de StoryUnit files, die de eigenschappen van de verschillende (typen) acteurs dragen. Elke file moet aan een strikte syntax voldoen. Een mogelijke uitbreiding zou er in bestaan deze syntax wat losser te maken, door woorden te controleren, of erop te zoeken; aangezien nu 'blind' wordt gelezen, is de volgorde nog van groot belang. Zie ook hoofdstuk 6.3.

actionlist.txt
Dit is de voornaamste file: hierin ligt het verhaal besloten. De syntax is als volgt: actieNaam(argument1,argument2) en 1 per regel. Het aantal whitespaces tussen leestekens en argumenten maakt niet uit. In de basis versie zijn de volgende acties beschikbaar:

- arrivesAt(agent, location): wordt gebruikt bij het binnen lopen van een agent op een locatie.
- attack(agent1, agent2): wordt gebruikt om een acteur (agent1) een andere acteur (agent2) te laten aanvallen.
- die(agent1, agent2): wordt gebruikt om een acteur (agent1) om te laten vallen met het gezicht in de richting van een andere acteur (agent2), die hem waarschijnlijk heeft aangevallen.
- isAt(agent, location): wordt gebruikt om een agent op een locatie te zetten, zonder dat deze hier moet komen binnenlopen; als de locatie wordt geladen, zal deze agent al aanwezig zijn.
- kills(agent1, agent2): na "attack" en "die" behoeft kills nauwelijks nog uitleg: agent1 valt agent2 aan een agent2 gaat dood.
- movesTo(agent1, agent2): agent1 zal zich begeven in de richting van agent2.
- pickUp(agent1, object): agent1 zal het object oppakken van de grond.
- putDown(agent1, object): agent1 zal het object neerleggen op de grond.
- gives(agent1, agent2): agent1 zal het object dat zich in zijn rechterhand bevindt aan agent2 geven.
- travelsTo(agent, locatie): wordt gebruikt om agenten naar een andere locatie te laten afreizen.

Het besluit van scène te wisselen is afhankelijk van volgorde waarin deze acties elkaar opvolgen: elke scène heeft zijn eigen locatie. Het is dus belangrijk hoe deze lijst van acties is geordend, echter is dit nu nog niet van belang. Het is tenslotte de toekomstige parser die het verhaal gaat lezen en deze tekstfile produceert. Natuurlijk kan er al wel rekening mee worden gehouden wanneer deze scène wissel plaats zal vinden.

Een nieuwe scène wordt ingeluid met een isAt(unit,locatie) of een arrivesAt(unit,locatie), waarbij de locatie van deze actie een andere is dan die van de vorige locatie. Hierbij moeten alle isAt(unit,locatie) acties (indien aanwezig) vooraf gaan aan de eerste arrivesAt en alle andere acties binnen deze scène. Alle acties uit een voorgaande scène moeten óók hier aan vooraf gaan. Als er bijvoorbeeld 2 agenten zijn, ag1 en ag2, die beide van locatie loc1 naar locatie loc2 gaan, zullen eerst de acties die in de eerste scène plaats vinden moeten worden afhandelt, alvorens de actie arrivesAt te gebruiken.

Dit is als volgt weer te geven

Goed:		fout:	
	isAt(ag1,loc1)		isAt(ag1,loc1)
	isAt(ag2,loc1)		isAt(ag2,loc1)
	travelsTo(ag1,loc2)		travelsTo(ag1,loc2)
	travelsTo(ag2,loc2)		arrivesAt(ag1,loc2)
	arrivesAt(ag1,loc2)		travelsTo(ag2,loc2)
	arrivesAt(ag2,loc2)		arrivesAt(ag2,loc2)

Agenten 1 en 2 moeten eerst beide afreizen naar locatie 2, alvorens de scènewissel plaats moet vinden, geïnitieerd door arrivesAt(ag1,loc2).

Een ander voorbeeld:

Goed:		fout:	
	isAt(ag1,loc1)		isAt(ag1,loc1)
	travelsTo(ag1,loc2)		isAt(ag2,loc2)
	isAt(ag2,loc2)		travelsTo(ag1,loc2)
	arrivesAt(ag1,loc2)		arrivesAt(ag1,loc2)
	movesTo(ag1,ag2)		movesTo(ag1,ag2)

In dit geval gaat ag1, gestart op loc1 naar ag2, gestart op loc2. Ook hier moeten alle acties, die zichtbaar zijn in 1 scène (op loc1) eerst worden afgehandeld, voordat scene2 (op locatie2) begint. Deze 2e scène zal worden getriggered door de isAt(ag2,loc1) actie, dus deze zal moeten worden voorafgegaan door travelsTo(ag1,loc2).

Locationlist.txt

Deze file beschikt over alle locaties, die zich in het verhaal voor doen. Hierin hoeft alleen maar de naam van een locatie aan een bepaald type te worden gekoppeld, volgens de volgende syntax:

locatieType;locatieNaam voorbeeld: Forrest;Bos1

Er is 1 locatie per regel. De volgorde waarin dit wordt gedaan bepaald welke kant de acteurs de scène op of af lopen: voor de verbeelding worden de locaties, van boven naar beneden, van links naar rechts geconstrueerd. In werkelijkheid ligt alles op dezelfde plek in het coördinaten stelsel, als echter een vlakte na het bos is gelezen, zal een unit die reist van het bos naar de vlakte aan de rechterkant het bos uitlopen, en aan de linkerkant de vlakte binnen lopen. Omgekeerd geldt natuurlijk hetzelfde.

Unitlist.txt
Net als bij de Locationlist moeten alle units, die een rol spelen in het verhaal (ook de passieve) in de unitlist worden genoemd met een type. Dit moet volgens dezelfde syntax als bij de locaties: UnitType;unitNaam   voorbeeld: Newhumanoid;Hum1
Hierbij speelt de volgorde, waarin ze worden geschreven geen rol. Ook hier geld: 1 unit per regel.

robot.txt, ninja.txt
in de <unit type> tekst files, worden de speciale waarden van een unit opgeslagen. Als er een nieuwe unit wordt aangemaakt, en deze heeft voor de onderstaande punten andere waarden nodig dan de standaard (default) waarden, moeten deze in deze file worden opgeschreven. Ook hier is, vanwege de 'blinde' parser de volgorde en syntax zeer belangrijk. Op deze volgorde kunnen de volgende aspecten worden ingevuld (de getallen zijn de standaard waarden, die worden ingevuld als er geen tekstfile voor de unit wordt gevonden):

- Walkspeed: de snelheid waarmee de StoryUnit over het scherm word bewogen tijdens een move-actie.
  Volgens deze syntax: walkspeed : 35.0
- Scale: de schaling die de unit moet ondervinden, als deze niet 'goed' is ontworpen qua grootte. De getallen staan voor de schaling over de verschillende coördinaten; respectievelijk: x;y;z
  Volgens deze syntax: scale : 1.0;1.0;1.0
  Deze schaling kan alleen worden toegepast bij units die niet worden overgedragen (middels de transfer actie). Beter is het om de betreffende StoryUnit goed te ontwerpen, zodat deze niet hoeft te worden geschaald.
- De oriëntatie: de as (of richting) waarop de unit georiënteerd is. Onderstaand getal geldt voor de robot: deze 'kijkt' in de richting van de positieve x-as, als hij nog niet is geroteerd. Deze oriëntatie is belangrijk, omdat een unit de verkeerde kant op zal kijken als deze niet goed gedefinieerd is. In dat geval is natuurlijk wel de juiste oriëntatie te bepalen door 'slim' te gokken. De getallen staan voor de schaling over de verschillende coördinaten; respectievelijk: x;y;z
  Volgens deze syntax: orientation : 1.0;0.0;0.0

2.2 Plaatsing

Het is van groot belang dat de juiste files (unit tekst files, unit meshes, unit skeletten e.d.) niet alleen bestaan en de juiste naam hebben, maar ook op de juiste locatie opgeslagen zijn. Als dit niet het geval is, zal de applicatie logischerwijs afbreken. In het ideale geval breekt de compiler af met het bericht dat de gevraagde unit, mesh, skelet of animatie niet gevonden kan worden.

- tekstfiles: unitlist.txt, locationlist.txt, actionlist.txt en specifieke unit tekstfiles moeten in dezelfde directory staan als waar de executable van het programma staat. In dit geval is dat: \Common\bin\Debug
- Material files: \Media\materials\scripts
- Texture files: \Media\materials\textures
- Skeleton en Mesh files: \Media\models
- Het is mogelijk dat er nog meer files benodigd zijn, maar die zijn in deze basis versie niet gebruikt. In dat geval kan het beste op het forum of in de online tutorials worden gezocht naar de files van dat specifieke type. [OGR05]

Deze paden zijn opgeslagen in een configuratie file genaamd “resources.cfg”, welke zich bevind in dezelfde directory als de executable en de tekst files. Hierin liggen de relatieve paden (ten opzichte van de locatie van dit bestand) opgeslagen waar Ogre zal zoeken naar de files.
Een voorbeeld (uit “resources.cfg”): `FileSystem=../../../Media`. Dit geeft aan, dat de media directory (met daarin dus materials) zich 3 mappen hoger bevindt dan de executable.

2.3 Running

Als ViVi wordt gestart, zal eerst een standaard Ogre start scherm komen, waar wordt gevraagd naar de gewenste settings en dit kan naar wens (afhankelijk van het vermogen van het systeem) worden opgegeven. Als vervolgens de applicatie loopt, kunnen de richting en snelheid van de animatie op de volgende manier worden gecontroleerd:

- key ‘1’: de animatie speelt op normale snelheid, vooruit af
- key ‘2’: de animatie versnelt met 10 % van de huidige snelheid.
- Key ‘3’: de animatie vertraagt met 10% van de huidige snelheid.
- Key ‘4’: de animatie speelt achteruit met 20% normale snelheid.
- Key ‘7’: de autorun wordt aan gezet; deze staat standaard aan.
- Key ‘8’: de autorun wordt uit gezet; de animatie wordt nu gepauzeerd
- Key ‘9’: deze knop kan worden gebruikt als de autorun uit staat. De animatie zal zo lang deze toets is ingedrukt normaal verlopen, en zodra de toets weer wordt losgelaten weer pauzeren.
- Key ‘R’: met deze standaard Ogre3D toets kan worden gewisseld tussen verschillende ‘views’: de normale view, de polygoon view, en de hoekpunten view, waarin enkel de hoekpunten van de polygonen oplichten tegen een donkere achtergrond.
- Key ‘P’: hiermee kan informatie over de camera onder in het beeld worden weergegeven: de eerste 3 coördinaten representeren de positie op het veld, de laatste 4 de oriëntatie van de camera
- Key ‘F’: deze knop kan gebruikt worden om de ‘overlay’ informatie (eventuele camera gegevens, het Ogre3D logo en de FPS rate) aan en uit te zetten.
- Key ‘W’: net als ‘pijl omhoog’ wordt deze toets gebruik om de camera vooruit in de kijkrichting te bewegen.
- Key ‘S’: net als ‘pijl omlaag’ wordt deze toets gebruik om de camera achteruit in de kijkrichting te bewegen.
- Key ‘A’: deze toets wordt gebruik om de camera naar links ten opzichte van de kijkrichting te bewegen.
- Key ‘D’: deze toets wordt gebruik om de camera naar rechts ten opzichte van de kijkrichting te bewegen.
- Key ‘PageUp’: deze toets wordt gebruik om de camera omhoog ten opzichte van de kijkrichting te bewegen.
- Key ‘PageDown’: deze toets wordt gebruik om de camera omlaag ten opzichte van de kijkrichting te bewegen.
- Key ‘pijl links’: deze toets wordt gebruik om de camera naar links te draaien.
- Key ‘pijl rechs: deze toets wordt gebruik om de camera naar rechts te draaien.
- Muis: deze kan worden gebruikt om de kijkrichting te veranderen.

Hoofdstuk 3 ViVi: High level code beschrijving

In dit hoofdstuk zal oppervlakkig worden ingegaan op de code. Eerst wordt een soort walkthrough gegeven waarin dwars door alle code heen de relevante dingen worden vermeld, die plaats vinden bij het starten van het programma (de voorbereiding op het animatieproces) en het eigenlijke animatieproces. Vervolgens worden per klasse de inhoud en functionaliteit besproken. Dit hoofdstuk dient alleen om (snel) een overzicht te ontwikkelen. Voor een grondiger toelichting wordt, als aanvulling hier verwezen naar hoofdstuk 4: hier wordt dieper op de code ingegaan.

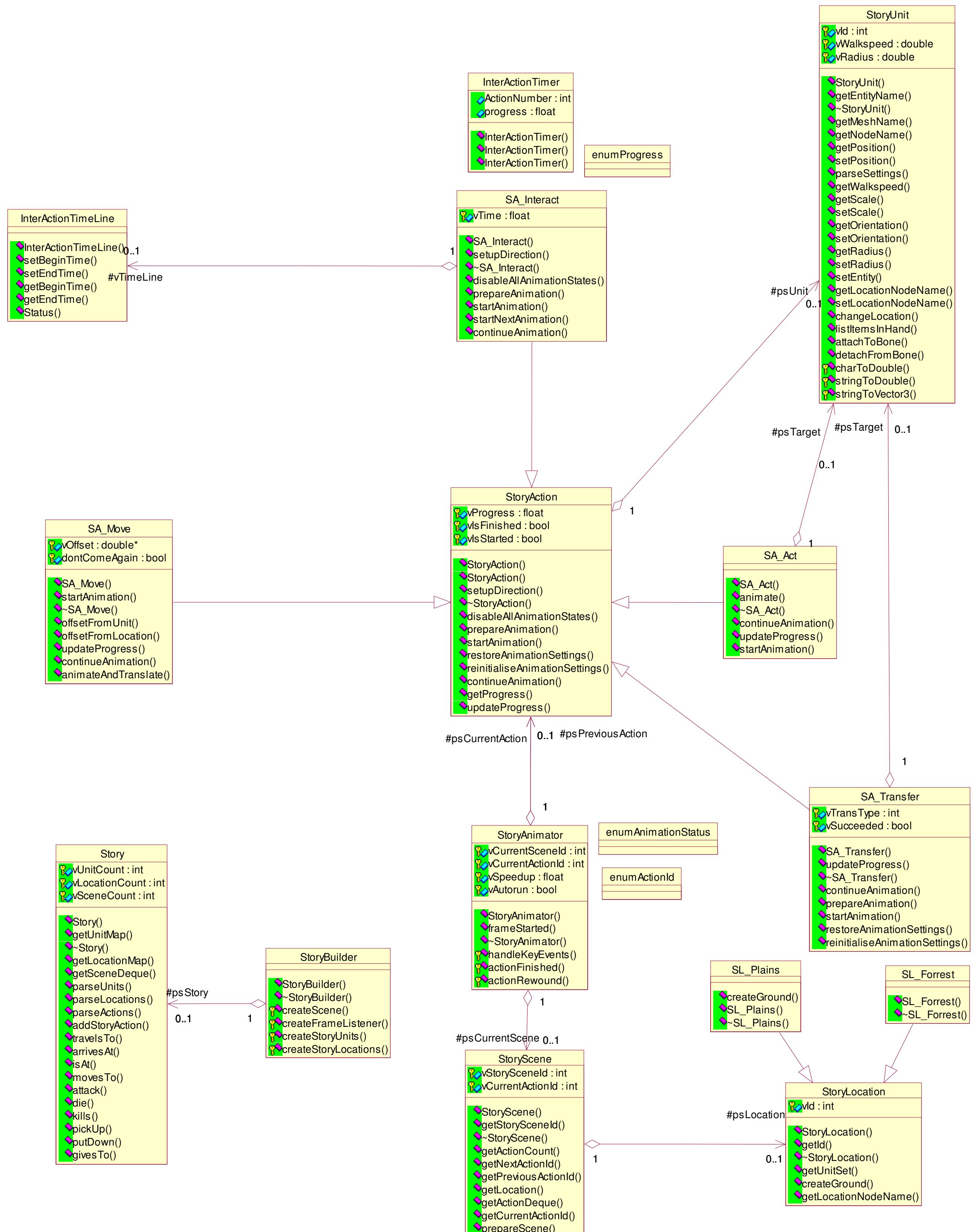
3.1 Code walkthrough

Voorbereidingen
Als het programma gestart wordt, zal nadat de link is gelegd van Ogre naar ViVi, aangevangen worden met een verhaal - een Story - aan te maken. Een Story beschikt over een aantal lijsten, waarin alle benodigde gegevens van entiteiten, locaties en scènes worden opgeslagen. Story zal tekst files (zie 2.1) uitlezen voor de entiteiten in dit verhaal, voor de locaties in dit verhaal en voor de acties, die plaats vinden in de scènes, waarbij deze lijsten worden opgevuld.
Na het maken van de Story, kunnen de unit en locatie lijsten worden gebruikt om de benodigde entiteiten voor de units en de locaties te laden. Als laatste stap in de voorbereiding worden deze lijsten dus weer uitgelezen, waarop de entiteiten worden aan gemaakt en aan een Node worden gehangen. Tenslotte wordt een StoryAnimator gestart, wat tevens het volgende proces inluidt.

Het animatieproces
Tijdens de animatie kunnen de snelheid en richting (vooruit, achteruit) worden bepaald tijdens het afspelen van de scène. In de StoryAnimator wordt bijgehouden welke scène en actie op elk moment tijdens de animatie weergegeven moet worden, en is dus verantwoordelijk voor het voorbereiden en starten van de animatie. Tevens wordt afgevangen in welke volgorde alles moet worden afgespeeld. Het is namelijk aan de gebruiker om een bepaalde scène of actie terug te spoelen, of te versnellen. Met het starten van een animatie wordt de laatste link gelegd naar de StoryActions; er zijn verschillende StoryActions voor verschillende typen. Hieronder zal worden toegelicht hoe dit onderscheid gekozen is. Hier volstaat het te vermelden dat in deze StoryActions ligt opgeslagen wat de StoryUnits precies moeten laten zien: dit zal een animatie van een bepaalde mesh (een StoryUnit) zijn, waarbij eventueel over een locatie wordt bewogen.

3.2 De afzonderlijke klassen

Dit is een klasse diagram van de source code in de basis versie. Het kan als structureel overzicht gebruikt worden bij de toelichting van de afzonderlijke klassen hieronder en in hoofdstuk 4.



Main
Deze klasse - verreweg de kleinste - bevat enkel een aantal benodige regels om Ogre te laten draaien. Er wordt een StoryBuilder aangemaakt en gestart. Dit slaat een brug naar de eigen applicatie.

StoryBuilder
Als StoryBuilder gemaakt wordt, zal een Story worden gemaakt, waarin alle input wordt verwerkt tot formaten waar Ogre mee kan gaan werken. Als dit eenmaal klaar is worden achtergrond/lucht en camerahoek ingesteld. Vervolgens zullen de in Story aangemaakte lijsten worden uitgelezen waarbij de benodigde entiteiten gecreëerd worden. Tot zover de voorbereidingen op het animatieproces. Storybuilder zal nu een StoryAnimator construeren, die de animatie initialiseert.

Story
Zoals reeds vermeld is Story verantwoordelijk voor het verwerken van de input (zie paragraaf 2.2). De gegevens in unitlist.txt, locations.txt en actionlist.txt zullen worden opgeslagen in de volgende dataformaten:

- De map psUnitMap zal alle entiteiten bevatten die aan het verhaal deelnemen; dit zijn de acteurs. Deze lijst zal op een later tijdstip, bij de voorbereiding voor het grafische gedeelte, in zijn geheel worden uitgelezen. Hierin worden de units opgeslagen met als key de naam van de unit om efficiënt de juiste unit te kunnen opzoeken.
- De map van StoryLocations zal alle locaties bevatten waarop een scène zal worden afgespeeld. Hierin worden de locaties opgeslagen met als key de naam van de locatie om efficiënt de juiste locatie te kunnen opzoeken.
- De 'double ended queue' psSceneDeque zal alle scènes bevatten, waaruit het verhaal is opgebouwd; de volgorde is hierbij van belang en daarom is gekozen voor het deque formaat. Elke scène heeft een lijst van acties, die bepaalt wat er geanimeerd zal worden, zodra de StoryUnits zichtbaar worden.

Bij het aanmaken van de scènes wordt gebruik gemaakt van verscheidene hulpfuncties, die de naam van de actie lezen en zodoende het juiste type actie creëren (een move, act of interact).

StoryUnit
StoryUnit is de klasse, die voornamelijk dient als opslag van de verzameling gegevens voor alle typen van StoryUnits. Een deel van deze gegevens wordt gegeven zodra een StoryUnit wordt aangemaakt. De rest wordt gelezen uit de input van een tekstfile, corresponderend met het type van de StoryUnit. Als dit is opgeslagen kan het nog worden opgevraagd en veranderd worden, indien nodig. Zo houdt StoryUnit ook bij op welke locatie hij zich bevindt.

StoryLocation: SL_Forrest en SL_Plains
StoryLocation is de superklasse van de specifieke locatie klassen SL_Forrest en SL_Plains. Deze bevat de methoden en datastructuren, die alle locaties gemeen hebben. Er is een methode die een standaard vloer aanmaakt, waar elke specifieke locatie vervolgens gebruik van kan maken en er wordt een set aangemaakt, waarin alle entiteiten opgeslagen zullen worden, die het decor van de specifieke locaties vormen. Dit zijn dus geen uitvoerende karakters, maar passieve StoryUnits. Er wordt gebruik gemaakt van hetzelfde type, omdat het evengoed entiteiten betreft, die een uiterlijk hebben en een locatie moeten hebben.

SL_Forrest vult deze lijst van StoryUnit op met een verzameling bomen: elke boom is een aparte entiteit en heeft een eigen coördinaat. Forrest maakt gebruik van de standaard beschikbare vloer. SL_Plains heeft daarentegen een eigen vloer gekregen en overschrijft daartoe de methode van de superklasse, die hiervoor verantwoordelijk is. Verder zijn er in een SL_Plains locatie een ogrehead en een appel entiteit te vinden.

StoryScene
Een StoryScene bestaat uit alle acties, die uitgevoerd moeten worden op één locatie. Hiertoe bevat een StoryScene dus een lijst van StoryActions, een StoryLocation en een getal dat de scène (er kunnen er natuurlijk meerdere zijn) identificeert. Wat functionaliteit betreft is deze klasse verantwoordelijk voor het initiëren van de voorbereiding van elke actie in de scène. Tevens voor het beheren en aangeven welke actie (op het moment dat daarom wordt gevraagd) wordt geanimeerd, welke de vorige was en welke de volgende zal zijn.

StoryAction
De klasse is de superklasse van de 4 verschillende actie typen. Er zijn sub-klassen voor verplaatsingen ('moves'), voor acties, die op de huidige plaat plaatsvinden ('acts'), voor het overdragen van entiteiten tussen StoryUnits ('transfers') en er is een sub-klasse voor interacties ('interacts'), welke een verzameling is van deze 3 elementaire acties. De opdeling is gemaakt omdat de uitvoering van deze verschillende typen van acties een specialisatie van een aantal functies vereisen, terwijl een (nog groter) aantal functies algemeen toegepast kan worden. StoryAction bevat dan ook de functionaliteit, die voor alle sub-klassen gelijk is. Vanwege het specifieke gebruik van gespecialiseerde functies en het afwijken van benodigde argumenten is ervoor gekozen deze verschillende actietypen te erkennen door ze fysiek van elkaar te onderscheiden. Zodoende is het niet nodig om steeds te controleren met welk type er gewerkt wordt en wordt implementeren van verschillende acties vergemakkelijkt. Hieronder zijn de sub-klassen specifiek toegelicht.

SA_Move
Deze klasse is de specialisatie van StoryAction, die alle acties afhandelt, waarin een StoryUnit over een bepaalde afstand moet bewegen. Er wordt onderscheid gemaakt tussen:

- Het uitlopen van de scène: een unit moet naar een andere locatie lopen en zal aan de linker of rechterkant de set (locatie) af lopen.
- Het inlopen van de scène: een unit komt van een andere locatie en gaat naar een willekeurige plaats binnen een bepaalde afstand van het midden van de locatie.
- Het bewegen naar een andere unit of coördinaat: hierbij wordt rekening gehouden met de radius, die deze doel-unit heeft (een StoryUnit, moet niet 'in' een andere unit komen te staan).

SA_Act
SA_Act is verantwoordelijk voor alle acties die StoryUnits op hun plaats uitvoeren, oftewel: de units staan stil en een animatie uitvoeren in de richting van een andere StoryUnit of plek.

SA_Transfer
Deze klasse behandelt het koppelen / ontkoppelen van passieve entiteiten (dit zijn ook StoryUnits) aan / van acterende StoryUnits (actieve StoryUnits). Er vindt geen animatie plaats, maar er wordt enkel van 'bezitter' gewisseld. Dit omvat niet alleen het overgeven van iets tussen 2 StoryUnits, maar ook het oppakken (de locatie 'geeft' een entiteit aan de acteur) en neerzetten (de acteur 'geeft' een entiteit aan de locatie) van objecten.

SA_Interact
Interact behandelt alle acties, die niet als elementair (zoals de 3 hierboven) gezien kunnen worden. Het betreft meestal één of meerdere acties waarbij meerdere StoryUnits betrokken zijn. Vandaar ook de naam. Deze interact-actie bestaat dus uit een verzameling elementaire acties (move, act, transfer), die in elke gewenste volgorde kunnen worden uitgevoerd, waarbij de acties onderling elkaar kunnen triggeren op een bepaald tijdstip. Elke elementaire actie kan worden getriggered door de mate waarin een voorafgaande actie is gevorderd. Zodoende kunnen complexe acties worden gedefinieerd. De enige voorwaarde waaraan voldaan moet worden is dat een StoryUnit slechts 1 actie tegelijk laat zien.

StoryAnimator
StoryAnimator is verantwoordelijk voor het animeerproces en het bijhouden van de wensen van de gebruiker tijdens animatie. Deze gebruiker heeft namelijk controle over vooruit of achteruit spoelen en de snelheid van de animatie. De animator zorgt ervoor dat de juiste scène en acties worden gekozen, zodra het tijd is voor een volgende (dan wel vorige) actie.

StoryConstants
Hierin worden de constanten opgeslagen, die door de hele code heen worden gebruikt.

Hoofdstuk 4 ViVi: Low level code beschrijving

Dit hoofdstuk moet gezien worden als eventuele aanvulling op het vorige. In hoofdstuk 3 wordt in vogelvlucht beschreven hoe het programma werkt en hoe de functionaliteit is opgesplitst over de verscheidene klassen. Dit is voldoende om een overzicht te geven betreffende de opbouw en de globale functionaliteit. Als het echter gewenst of vereist is de code te lezen (vanwege mogelijke uitbreidingen), is enige toelichting nog wel wenselijk.

In dit hoofdstuk zijn nogmaals dezelfde, afzonderlijke klassen toegelicht, alleen nu wordt dieper op de code zelf ingegaan. Het kan naast de documentatie als toelichting op de code gelezen worden. En omdat er gebruik gemaakt zal worden van dezelfde termen (methoden, typen en variabelen), die gebruikt zijn in de broncode, is het een significant naslag werk. De toegevoegde waarde op de documentatie is de samenhang binnen de klassen in dit hoofdstuk.

4.1 De afzonderlijke klassen:

Main.cpp
De main bestaat uit een aantal regels code dat standaard is voor een Ogre applicatie. Voordat Ogre wordt gestart wordt er een StoryBuilder aangemaakt. Dit is de link naar deze (ViVi) applicatie.

StoryBuilder.cpp / Storybuilder.h
Bij het construeren van een StoryBuilder, zal een story worden aangemaakt. Het is uiteindelijk in *createScene* waar deze story middels de methoden *createStoryUnits* en *createStoryLocations* wordt uitgelezen. *createScene* wordt door het Ogre framework wordt aangeroepen. Hierin wordt de camera ingesteld en een omgeving (lucht en achtergrond) aangemaakt.
createStoryUnits zal door een map van StoryUnits heen itereren en voor elke StoryUnit hierin een Entity aanmaken en aan een Node hangen. Voorlopig worden deze entiteiten op onzichtbaar gezet, omdat ze nog geen deel uitmaken van een scène.
createStoryLocations itereert door een map van StoryLocations en maakt voor elke locatie in deze map een Node aan voor alle entiteiten die bij een bepaalde locatie horen. Deze Nodes worden dan gekoppeld aan de Node van de locatie zelf. Elke locatie beschikt over een vloer en een eigen set van entiteiten: ook deze worden hier gecreëerd.
Als dit klaar is, zal vanuit het Ogre framework de methode *createFrameListener* worden aangeroepen, die de brug slaat naar het volgende proces: animatie. Er wordt een StoryAnimator aangemaakt, die alle zojuist geconstrueerder gegevens (de 'wereld', alle scenes en de sceneManager) mee krijgt.

Story.cpp / Story.h
Story heeft een map van StoryUnits; een map van Locations en een double-ended-queue van StoryScenes en een overeenkomstig stel counts, die worden opgehoogd, zodra een entiteit in deze mappen of deque wordt aangemaakt. Deze counts houden zo bij hoeveel units, locaties of scènes er zijn en worden gebruikt in de generieke naamgeving van deze entiteiten. Verder zijn er nog 'standaard' get-functies die deze mappen en deque retourneren voor de StoryUnits, StoryLocations en StoryScenes respectievelijk.

Bij het maken van een nieuwe story wordt in de methode *parseUnits* eerst een lijst uitgelezen waar alle participerende entiteiten (agenten) in staan; voor elke entiteit wordt een StoryUnit aangemaakt (met eventueel een initiële locatie) en deze wordt toegevoegd aan de StoryUnit map, met als key de naam van de StoryUnit. Hetzelfde gebeurt voor alle locaties van het verhaal in de methode *parseLocations*.

Tenslotte worden de acties gelezen. Voor elke scène wordt een actielijst (van type deque) aangemaakt. Aan deze lijst worden alle acties (SA_Move, SA_Act en SA_Interact per scène toegevoegd. Dit wordt uitgevoerd door de methode *parseActions*. Binnen deze methode wordt beslist aan de hand van de naam van de actie en de argumenten hiervan of de actie binnen de huidige scène valt, of een volgende scène inluidt. Van elke gelezen actie (1 per regel in de tekst file) worden naam en argumenten door gegeven aan de hulpfunctie *addStoryAction*, die aan de hand van deze gegevens beslist welke specifieke StoryAction aangemaakt dient te worden en roept vervolgens de functie aan, die dit realiseert:
travelsTo: maakt een SA_Move aan, waarbij de unit naar een andere locatie (en scène) gaat lopen;
arrivesAt: maakt een SA_Move aan, waarbij de unit een locatie (en scène) binnen komt lopen;
isAt: stelt de positie van een unit gelijk aan het midden van de locatie
movesTo: maakt een SA_Move aan, waarbij een unit naar een andere unit beweegt. Hierbij wordt rekening gehouden met de radius van de target unit.
attack: maakt een SA_Act aan en wordt gebruikt om een unit een aanval te laten uitvoeren op een andere unit.
die: maakt een SA_Act aan, waarin de unit op zijn huidige plek dood gaat.
kills: maakt een SA_Interact aan, bestaande uit een verzameling SA_Acts, waarbij gebruikt gemaakt wordt van timing en actiontriggering. Hiermee wordt de variëteit van de interact-actie gedemonstreerd.

Als alle acties voor één scène aan bod gekomen zijn, wordt deze actielijst – samen met de locatie waar de scène zich afspeelt en een getal (later de ID van de scène) als argument mee gegeven voor het maken van een nieuwe StoryScene. Deze StoryScene wordt toegevoegd aan de StoryScene deque.

StoryUnit.cpp / StoryUnit.h

Een StoryUnit heeft een aantal variabelen en een verzameling corresponderende get- en set-functies. Zo zijn er: het ID, de naam (van entiteit, Node en mesh), de positie, het type entiteit, de oriëntatie, de schaalfactor en de beweegsnelheid. Bij het aanmaken van een nieuwe StoryUnit worden de naam, het ID en de positie meteen uit de argumenten verkregen en geset.
De overige relevante gegevens voor een StoryUnit worden uit een file uitgelezen. Dit uitlezen gebeurt in *parseSettings*, die met behulp van de hulpfuncties *stringToVector3*, *stringToDouble* en *charToDouble* een tekst file met dezelfde naam als de naam van de nieuw aangemaakt StoryUnit regel voor regel doorloopt. Het voordeel van het uitlezen van een file, is dat er geen speciale klasse hoeft te worden gemaakt per type entiteit; de specifieke eigenschappen voor bepaalde typen kunnen zo worden uitgelezen. Op deze manier kan er zonder dat er opnieuw gecompileerd moet worden iets toegevoegd of veranderd worden. Dit vergemakkelijkt het introduceren van StoryUnits van een nieuw type aan het programma ook aanzienlijk.

StoryLocation.cpp / StoryLocation.h

StoryLocation is de superklasse van de specifieke locaties. Bij het maken van een StoryLocation wordt een "default" naam en een set van StoryUnits aangemaakt. Deze set zal bij het maken van een specifieke locatie pas gevuld worden. Zodat alle locaties gebruik kunnen maken van dezelfde vloer, heeft StoryLoaction de *createGround* methode, welke (gebruik makend van de SceneManager) een entiteit voor de grond aan maakt en het ID van deze locatie gebruikt in de naamgeving. Het midden van de vloer ligt in de oorsprong van het coördinatenstelsel dat Ogre gebruikt. Het ID van de locatie is de LocationCount, die wordt meegegeven bij het maken van een nieuwe locatie. StoryLocation beschikt tenslotte nog over een methode waarmee zijn set van StoryUnits kan worden opgevraagd.

SL_Forrest.cpp / SL_Forrest.h

Een Forrest is een locatie met een verzameling bomen, die elk moeten worden aangemaakt, geplaatst en toegevoegd aan de set van StoryUnits voor deze locatie. De entiteiten in deze set maken slechts deel uit van de locatie en participeren uiteraard niet (live) in het verhaal.

SL_Plains.cpp / SL_Plains.h

Een Plains is een locatie met enkel een appel en een ogrehead toegevoegd aan de set van StoryUnits. Net als bij een Forrest zijn deze entiteiten alleen maar bedoeld als 'decor'. De locatie plains beschikt over een andere vloer dan de standaard vloer, die gebruikt is in SL_Forrest. De methode createGround krijgt in deze klasse een eigen invulling, zodat een andere vloer-mesh gebruikt kan worden.

StoryScene.cpp / StoryScene.h

Als een nieuwe StoryScene wordt aangemaakt, krijgt deze een StoryLocation, een ID en een lijst van StoryActions mee en deze argumenten worden direct op locale variabelen geset. Voor deze variabelen zijn ook in deze klasse standaard get-funties en er is een *getActionCount* functie, die de grootte van de actie lijst retourneert.

De methode *attachAllUnits* itereert door de lijst van StoryActions heen en kijkt voor elke StoryAction of de StoryUnit van die StoryAction al aan een Node hangt. Als dat niet het geval is, wordt deze aan een Node gehangen op de positie van de entiteit zelf. Was er al wel een Node voor deze entiteit (dit was dan in een andere scène) dan wordt de entiteit van die Node losgekoppeld en in de huidige scène geplaatst op dezelfde positie, gespiegeld over de Z-as. Dit heeft als doel dat als een StoryUnit in de vorige scène aan de linkerkant van de locatie weg liep, deze in de huidige scène aan de rechterkant weer komt binnen lopen en omgekeerd. *PrepareScene* roept voor elke actie in de actielijst de methode *prepareAnimation* aan, waar de daadwerkelijke animatie wordt voorbereid (zie StoryAction). Hierbij wordt de SceneManager meegegeven.

De methoden *getNextActionId* en *getPreviousActionId* beheren de CurrentActionId. Dat wil zeggen: deze methoden setten en retourneren de waarde van CurrentActionId, afhankelijk van de hoeveelheid resterende scènes bij het vooruit spelen en achteruit spoelen. Wanneer deze CurrentActionId nodig is kan dat bij een StoryScene worden opgevraagd met de methode *getCurrentActionId*.

StoryAction.cpp / StoryAction.h

De klasse StoryAction is de superklasse van 4 verschillende actie typen: dit zijn:

- SA_move voor het afhandelen van beweeg acties;
- SA_Act voor acties die plaatsvinden op het huidige coördinaat;
- SA_Transfer voor het (ont)koppelen van nodes van StoryUnits
- SA_Interact voor het specificeren van gecompliceerde acties, bestaande uit een verzameling van bovenstaande acties, waarbij deze op elkaar kunnen worden getimed.

StoryAction beschikt over 2 constructors: één simpele, waarin slechts een unit en actienaam worden meegegeven en opgeslagen, en één waarbij verder nog een doel wordt meegegeven. Ook deze waarde wordt opgeslagen.

StoryAction bevat verder enkel de functionaliteit, die voor een paar (of alle) sub-klassen gelijk is. Deze functies (lijst van get- en setmethoden, *setupDirection*, *diableAllAnimationstates*) wordt in de sub-klassen geërfd en gebruikt. Verder zijn functies, die in de sub-klassen een verschillende inhoud hebben, in deze klasse virtual gemaakt. Dit geldt bijvoorbeeld voor *updateProgress*, die bijhoudt hoever een animatie gevorderd is. Ook *prepareAnimation*, *startAnimation*, *continueAnimation*, *reinitialiseAnimationSettings* en *restoreAnimationSettings* hebben (deels) per sub-klasse een eigen indeling.

De enige methode, die niet virtueel is en door alle klassen gebruikt kan worden is *getProgress*, die de specifieke *updateProgress* de progress variabele zal laten bijwerken alvorens deze te retourneren.

SA_Move.cpp / SA_Move.h

SA_Move zal alle acties afhandelen waarin er bewogen moet worden naar een bepaalde bestemming. In bestemmingen wordt onderscheid gemaakt tussen:
Het uitlopen van een scène: er wordt naar een vast coördinaat aan de zijkanten van een locatie gelopen. Of dit links of rechts is, wordt bepaald van de volgorde waarin deze locaties tijdens de voorbereiding zijn geparsed. Deze worden van boven naar beneden gelezen en krijgen zo oplopende locationId's, waarbij hogere id's (voor de verbeelding) rechts liggen van de locaties met lagere id's. In werkelijkheid liggen alle locaties op dezelfde plek, maar dat zal een gebruiker er niet aan kunnen aflezen.
Het binnenlopen van een scène. Er wordt bewogen naar een van tevoren gedefinieerd coördinaat. Deze bestemming wordt vervolgens willekeurig veranderd binnen een bepaald vlak op het midden. Dit vlak is 300 bij 150 groot; ter referentie: een StoryLocation is 1500 bij 1500. Zodoende komen StoryUnits elke keer dat het(zelfde) verhaal wordt afgespeeld, ergens anders in de scène te staan (binnen die bepaalde offset) en – belangrijker – ze komen niet meer allen op hetzelfde coördinaat terecht.

Het lopen naar een andere unit in de scène. Als er wordt bewogen naar een andere unit op dezelfde locatie, moet de StoryUnit niet op exact dezelfde positie terechtkomen, maar voor het doel stoppen met transleren. Hiertoe wordt er bewogen tot op 50% van de radius van de target agent. Deze radius wordt optioneel aan de SA_Move meegegeven, waarmee ook wordt aangegeven of er sprake is van een beweeg actie tussen twee locaties of naar een andere StoryUnit.

De realisatie van deze offsets vindt plaats tijdens de *startAnimation* en wordt uitgevoerd door de methoden *offsetFromUnit* en *offsetFromLocation*. *UpdateProgress* wordt in deze klasse gerealiseerd door de resterende afstand te delen op de initiële afstand en dat van 1 af te trekken; zodoende geeft vProgress op een lineaire schaal tussen 0 en 1 aan hoever de animatie is gevorderd.

SA_Act.cpp / SA_Act.h

In tegenstelling tot de SA_Move klasse gaat het hier om een animatie op de plaats die éénmaal wordt uitgevoerd. Een SA_Act wordt altijd uitgevoerd door 1 unit in de richting van een andere unit. SA_Act heeft een *animate* methode, aangeroepen door de eigen variant van *continueAnimation*, die *restoreAnimationSettings* aanroept, zodra een actie is terug gespoeld en de tijdspositie bijhoudt. *Animate* beheert de animatie settings en retourneert constanten, die aangeven hoe het ervoor staat met de actie: rewound (klaar als er achteruit wordt afgespeeld), pending (bezig, vooruit of achteruit), finished (klaar bij vooruit afspelen).
De progressie van een SA_Act wordt middels *updateProgress* bijgehouden aan de hand van de ratio (hoeveelheid animatie verstreken) / (lengte van de animatie).

SA_Transfer.cpp / SA_Transfer.h

Deze klasse wordt gebruikt voor het overdragen van entiteiten. Er kan een overdracht plaatsvinden tussen 2 (actieve) StoryUnits en tussen een entiteit, die deel uitmaakt van een locatie en een (actieve) StoryUnits. (In principe zijn alle entiteiten StoryUnits; met 'actieve' worden hier agenten bedoeld.) Omdat er niks geanimeerd hoeft te worden zijn de methoden die deze settings beheren zeer eenvoudig gebleven in deze klasse. Als de actie uitgevoerd wordt, is deze in hetzelfde frame al weer afgelopen.
continueAnimation is verantwoordelijk voor het loskoppelen (*dettachFromBone*) en aankoppelen (*attachToBone*) van entiteiten en het beheer van de animatiesettings. Afhankelijk van vooruit of achteruit afspelen (respectievelijk positieve of negatieve vSpeed) worden isStarted en isFinished beide op true of false gezet, respectievelijk. Wat de voortgang van deze actie betreft hoeft er niet veel te worden bijgehouden; als u*pdateProgress* wordt aangeroepen is de actie nog niet begonnen of al direct klaar.

SA_Interact.cpp / SA_Interact.h

Deze klasse maakt het mogelijk andere acties te combineren tot (voor de 'buitenwereld') één complexe StoryAction. Het gaat hier om een lijst van acties met een bijbehorende timing. De eerste waarde in de timing lijst wordt genegeerd, de overige waarden bestaan uit een verwijzing naar een andere actie samen met een percentage (een getal tussen 0-1), dat de voortgang van die andere actie aangeeft. De index van deze vector correspondeert met de actie waar de opgeslagen waarden voor moeten gelden. De vector met acties is van gelijke lengte als de timing lijst; vandaar dat de waarden uit de timing lijst op plaats *i* behoren bij de actie op plaats *i* in de actie vector.

Zo zal bijvoorbeeld, een waarde van (2 , 0.5) op plaats *i* (die correspondeert met de huidige actie) betekenen dat deze (huidige) actie gestart gaat worden zodra actie nummer 2 op 50% is. Tijdens het vooruitspoelen van een SA_Interact zal een tijdlijn worden gemaakt waarop wordt bijgehouden op welk tijdstip welke animatie is begonnen en gestopt en wat de huidige status is van deze actie. Ook hier geldt, dat het nummer van de actie correspondeert met de index in de tijdlijn. De tijdlijn is gespecificeerd in een aparte klasse, die bevat is binnen de SA_Interact.h. Deze klasse

bezit een drietal vectoren, die voor elke actie in de Interact opslaan, wanneer deze is begonnen (tijdswaarde), geëindigd (tijdswaarde) en wat zijn huidige status is. Per actie bekeken bevat de tijdlijn dus een drietal gegevens. Deze gegevens zullen gedurende het vooruit dan wel achteruit spoelen van de animatie worden bijgehouden. Voor de tijden geldt dat de begintijd van actie nummer 0 altijd op 0 staat en voor de andere gevallen geldt dat een tijd op -1 wordt gezet zolang deze nog niet bekend is. Aan de hand van deze tijdswaarden wordt de status bij gehouden, deze kan op *Pre*, *Active* of *Post* staan. Deze status wordt gebruikt om efficiënt bij te houden welke acties aan bod moeten komen voor de animatie. Verder zorgt de tijdlijn ervoor dat er ook teruggespoeld kan worden als een deel van de acties reeds is afgelopen.

Terug naar de SA_Interact klasse: Het opbouwen en beheren van de tijdlijn (het hart van de klasse) zal worden uitgevoerd door *continueAnimation*. De overige 'beheer' methoden, zoals *setupDirection*, *diableAllAnimationstates*, *prepareAnimation* itereren door de vector van StoryActions en zullen voor al deze specifieke StoryActions dezelfde methode aanroepen. *StartAnimation* zoekt de StoryAction aan het begin van de vector op en roept hiervoor de *startAnimation* methode aan.

StoryAnimator.cpp / StoryAnimator.h

StoryAnimator zorgt voor het aansturen van de gehele animatie. Vanuit het Ogre framework worden een aantal functies geërfd van de klasse ExampleFrameListener. De functie waar het om draait is *frameStarted*, deze wordt aan het begin van elk frame door Ogre aangeroepen. Deze functie stuurt allereerst *handleKeyEvents* aan en kijkt vervolgens of er een signaal is binnengekomen dat aangeeft of de huidige actie al voltooid is. Wanneer dit niet het geval is, wordt de animatie voortgezet; indien dit wèl het geval is, wordt de volgende animatie voorbereid en gestart.

De functies *actionFinished* en *actionRewound* zorgen ervoor, door het beheren van de variabelen CurrentActionId, CurrentSceneId en de actieve Locatie, dat de juiste scène en actie worden gezet zodra het tijd is voor een volgende (dan wel vorige) actie. De *handleKeyEvents* functie wordt gebruikt om de snelheid en richting van de animatie te zetten zodat er met een bepaalde snelheid heen en weer gespoeld kan worden.

StoryConstants.h

Hierin worden de constanten opgeslagen, die door de hele code heen worden gebruikt. Zo worden er bijvoorbeeld in het programma print regels gebruik om de werking van een specifiek onderdeel te controleren. Om deze print regels staat een if-statement, dat controleert of de 'debug-constante' gedefinieerd is of niet. Zodoende kan met 1 regeltje de desbetreffende debug-print-code worden aan- of uitgezet.

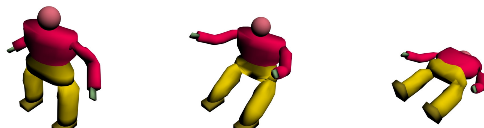
Hoofdstuk 5 ViVi: Karakters en Animaties

Zoals reeds eerder vermeld zijn er een aantal standaard modellen bij Ogre3D. Een aantal van de modellen zijn voorzien van één of meerdere animaties. Binnen ViVi zijn echter ook animaties gewenst die niet bij deze modellen beschikbaar zijn, vandaar dat er één zelf gemaakt model is toegevoegd.

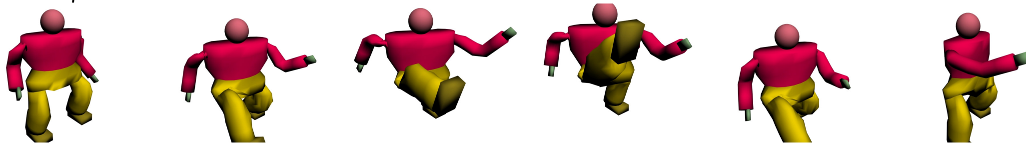
5.1 De animaties

Zoals reeds beschreven zijn de standaard animaties niet voldoende voor de huidige versie van ViVi, dit is de reden dat er een zelf gemaakt model is toegevoegd. Deze paragraaf zal ingaan op de animaties die in de huidige versie van ViVi beschikbaar zijn. Er zal per animatie kort worden uitgelegd waarbij deze op dit moment wordt gebruikt en er zal getoond worden hoe deze animatie eruit ziet bij de Humanoid (het zelf gemaakte model).

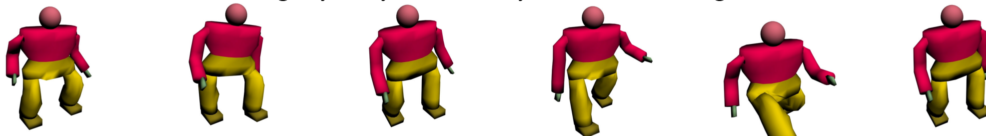
- **Die**
  Wordt alleen gebruikt in combinatie met "Attack1" in het geval er een "kills" actie plaatsvindt.



- **Attack1**
  Wordt alleen gebruikt in combinatie met "Die" in het geval er een "kills" actie plaatsvindt.

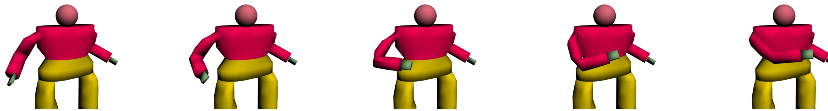


- **Walk**
  Op dit moment nog de enige animatie die gebruikt wordt binnen een SA_Move. Deze animeert het model op zijn plaats en wordt dus met behulp van SA_Move tegelijkertijd met de juiste snelheid getransleerd.

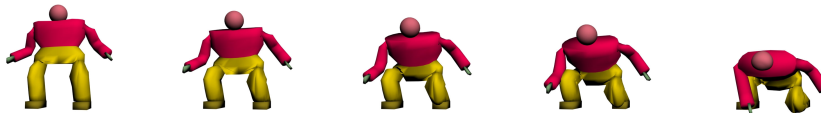


- **Eat**
  Ongebruikt totnogtoe, dit is een relatief simpele animatie die de hand vanaf de standaard plaats richting de mond beweegt en zou gebruikt kunnen worden wanneer een Unit iets uit zijn hand eet.

- **StretchArm**
  Bij een transfer tussen 2 StoryUnits wordt deze animatie gebruikt, hierbij wordt de rechterarm uitgestrekt om iets aan te pakken of te geven.



- **Grab**
  Wanneer een transfer wordt gebruikt om iets op te pakken dan wel neer te zetten wordt deze animatie gebruikt. Hierbij maakt de Humanoid een bukkende beweging en een greep naar de grond.



5.2 Humanoid

In de vorige paragraaf zijn de verschillende animaties van het Humanoid character reeds getoond en toegelicht, er zal hier nog kort worden ingegaan op hoe het character is gerealiseerd.
Het Humanoid character is ontworpen met behulp van 3D Studio Max [3DS05], aangezien er reeds enige ervaring aanwezig was met 3D Studio. Daarnaast is, naast nog een aantal andere exporters, in de download sectie van Ogre3D [OGR05] een exporter te vinden voor 3D Studio. Tevens zijn er een aantal tutorials in omloop met betrekking tot 3D Studio en deze exporter. Het character is bewust simpel gehouden zodat deze niet zo snel beperkt zal zijn om te bewegen en daarnaast is het geen doel van de opdracht is om alles grafisch hoogstaand te maken. Er zal hier daarom geen uitgebreide uitleg van 3D Studio volgen, aangezien dit veel te omvangrijk zou worden.

Voor verdere informatie over het modelleren en animeren binnen ViVi kan de bijgeleverde tutorial [MOD05] worden geraadpleegd. Het animatieproces kan in een aantal stappen worden onderverdeeld. Deze stappen zullen in bovengenoemde tutorial [MOD05] verder worden uitgediept, hier volgt een kort overzicht:

- Maken van een model.
- Toevoegen van een biped skeleton.
- Maken of laden van een biped animatie.
- Exporteren naar XML.
- Skeleton.xml bestand evt. samenvoegen met reeds bestaand bestand.
- Converteren naar het Ogre formaat.
- Bestanden op de juist plek zetten.

Wanneer deze bovenstaande stappen worden gevolgd kan er een nieuw model worden gemaakt welke gebruikt kan worden binnen Ogre en ViVi. Indien nodig kan er nog een mijnmodel.txt tekst file worden gebruikt om schaling, oriëntatie, etc. om instellingen te corrigeren (zie hoofdstuk2).

Hoofdstuk 6 Uitbreidingen

Tijdens de ontwikkeling, zijn er naast een aantal gebreken ook een aantal zeer nuttige uitbreidingen aan het licht gekomen. Het is, gezien de tijd, niet mogelijk geweest deze allen te realiseren, maar het kan wellicht worden besproken, opdat anderen de draad hier kunnen oppakken. Dit hoofdstuk zal deze punten behandelen. Eerst wordt het maken van een uitbreiding kort aangestipt, vervolgens een paragraaf over beperkingen waar rekening mee gehouden dient te worden en tenslotte mogelijke ideeën om de implementatie mee te verrijken.

6.1 Een nieuw karakter met animaties

Het huidige karakter en zijn animaties zijn ontwikkeld met 3dStudioMax. Omdat het werken met dit omvangrijke software pakket voor een nieuwe gebruiker van het programma niet bepaald zelfevident is, is er een tutorial gemaakt, die het maken van een nieuw model toelicht. [MOD05] Er wordt aangegeven wat er moet gebeuren en hoe dit bereikt kan worden. Daarnaast wordt in dit document doorverwezen naar een serie vrij verkrijgbare tutorials, bestaande uit films waarin te zien is hoe een model met skelet gemaakt en verandert kan worden.

6.2 Een nieuwe actie in ViVi

Om uitbreidingen te kunnen doen, zullen ook nieuwe actie typen moet worden gemaakt, die eventueel middels een SA_Interact gecombineerd kunnen worden met bestaande (elementaire) acties om complexer gedrag te vertonen. Ook hiervoor is een tutorial geschreven [ACT05]. Hierin wordt toegelicht hoe een actie uitgelezen zal worden (het proces van aanmaken) en welke stappen hierbij ondernomen worden. Hierbij is vermeld welke methoden een uitbreiding moeten krijgen, om de nieuwe actie te kunnen parsen. Er wordt verder toegelicht waar rekening mee gehouden dient te worden bij het maken en implementeren van de nieuwe acie klasse en dit alles wordt gedemonstreerd aan de hand van de laatste toevoeging: SA_Transfer, waarbij objecten worden overgegeven.

6.3 Een nieuwe locatie in ViVi

Locaties zijn ook erg verhaal afhankelijke objecten en de basis verzameling van 2 (Forrest en Plains) zal zeker moeten worden uitgebreid. Hiertoe is een derde tutorial geschreven [LOC05]. Hierin wordt, net als bij een nieuwe actie aangegeven hoe een nieuwe locatie tot stand zal komen, en wat hiertoe gedaan zal moeten worden. Het creeren van een nieuwe locatie is minder complex dan het maken van een nieuw type actie, omdat er slecht 1 nieuwe klasse gedefinieerd dient te worden en mogelijk een regel in een (locationlist.txt) tekstfile. Er hoeft minder diep door verscheidene klassen te worden gespit, en het maken van een nieuwe locatie, is goed reproduceerbaar, wanneer naar een reeds bestaande klasse (bijvoorbeeld SL_Forrest) gekeken wordt. Omwille van de volledigheid is echter toch een handleiding gemaakt, die mooi in het bestaande straatje van tutorials past en het een beginnende gebruiker vergemakkelijkt

6.4 Beperkingen

In de laatste fase van ontwikkeling zijn nog een aantal beperkingen en punten die voor verbetering vatbaar zijn naar voren gekomen:

- De namen van animaties van karakters: Voor de basisversie is het vereist dat bepaalde animaties voor alle karakters dezelfde naam dragen. Voor nieuwe karakters is dit geen groot probleem, immers kan hierop worden gelet bij het aanmaken ervan. Echter heeft Ogre ook een aantal eigen karakters meegeleverd (bijvoorbeeld een robot en ninja), waarvoor dit niet geldt: de robot heeft een "shoot" animatie, die gezien kan worden als aanval en de ninja heeft een aantal "attack" animaties, gevolgd door een getal. Nu zou het geschikt zijn, een actie-mapper te maken, die afhankelijk van het type unit, de juiste animatie naam retourneerd. Dit kan dan in de story klasse, bij het parsen: na het uitlezen van de actie naam en voor het aan aanmaken van de StoryActions en de gegevens die hiervoor nodig zijn (bijvoorbeeld mapOnTo(shoot;attack) kan worden opgeslagen in de specifieke unitType tekstfile; in dit geval robot.txt. Met deze uitbreiding hoeft niet zo zeer meer op een juiste benaming worden gelet en kunnen er meer units (de meegeleverde robot) in het verhaal betrokken worden.
- De huidige parser controleert niet wat hij aan het lezen is. Als er dan een verkeerde volgorde in de tekstfiles gehanteerd is, kan moeilijk worden voorspelt, waar er iets scheef gaat lopen (misschien wel letterlijk), maar dat er iets mis gaat is redelijk zeker. Een verbetering van de parser zou kunnen inhouden dat hij bij het lezen van bepaalde getallen op zoek gaat naar het keyword hiervoor: bijvoorbeeld "scale".
- In de basis versie worden objecten enkel nog met de rechterhand aangepakt, omdat er niet mee animaties zijn. Als die er wel komen, is wederom een mapper denkbaar, die controleert of en waar er al een object aan het lichaam van een StoryUnit hangt, die vervolgens de actie bepaald: neem het nieuwe object in de linker i.p.v. rechterhand; plaats eerst het huidige object op de grond, voordat het nieuwe wordt aangenomen / opgepakt enzovoorts.
- In het ideale geval zijn de unit tekst files (met oriëntatie, schaling ed.) niet meer nodig; deze aspecten kunnen beter bij het ontwerp al goed gekozen worden en een parser zal dit ook niet uit een verhaal kunnen halen. Walkspeed behoudt op dit punt wel zijn toegevoegde waarde. Er kunnen bijvoorbeeld ook andere snelheden worden gedefinieerd voor andere typen van move, die middels een mapper worden gekoppeld aan een actie (swim, fly, crawl, run).

6.5 Overige uitbreidingen

Naast nuttige verbeteringen is er nog een ruime hoeveelheid aan uitbreidingen denkbaar. Hier worden er een paar besproken, die het verhaal gevarieerder kunnen doen weergeven.

- Een koppeling tussen de Story klasse en een externe parse module, zodat deze los ontwikkeld en getest kan worden. Er is dan wel een vorm van communicatie nodig tussen story en de parser.
- Als beide mappers, genoemd in de bovenstaande paragraaf worden gecombineerd, kan worden gecontroleerd wat er in een hand van een StoryUnit is aangekoppeld en afhankelijk hiervan een geschikte actie worden gekozen. Stel bijvoorbeeld dat een attack wordt uitgevoerd. Op dit moment zal een humanoid karakter een schop/sla beweging maken. Als deze echter een zwaard in de handen zou hebben is het natuurlijk veel leuker dat hij hier mee uithaalt. Of als er een appel aan zijn hand hangt, dat hij deze dan zal gooien ...
- Een variatie met camerahoeken. Met een druk op de knop wordt een andere positie en hoek ingesteld, of de camera wordt aan de actieve StoryUnit gehangen
- Voertuigen: het is al mogelijk een voertuig te ontwikkelen (dat is tenslotte ook een StoryUnit), in het verhaal te laden en rond te laten reizen. Een interessante uitbreiding hierbij is de plaatsing van units hierop of in. Er zal dan moeten worden nagedacht over een aanpassing in de parser die t.z.t. de uitvoer van de V.S.T. verwerkt, aangezien het niet voldoende is om de unit te bewegen, als deze zich in (bijvoorbeeld) een auto zit. De Node van de unit zal dan waarschijnlijk aan de node van de auto hangen, waardoor de unit zal meebewegen met de auto, maar niet andersom. Als de parser dan dus leest: "Cees stapt in de auto en ging naar het bos"), zal de paser niet moeten genereren: getsIn(Cees,auto) maar: getsIn(Cees,auto)
TravelsTo(Cees,bos) TravelsTo(car,locatie))

Het is inmiddels wel duidelijk dat er nog genoeg mogelijk is, en zodra de V.S.T. klaar zal zijn, er nog wel het een en ander toegevoegd kan worden.

Conclusies

In een concluderend hoofdstuk van een verslag, bij een implementatie kan niet veel worden geconcludeerd, behalve dat het werk is verricht. Wel kunnen de aanvankelijk gestelde doelen worden beschouwd en er kan worden vooruit geblikt.

Er is veel ervaring opgedaan en is er met plezier gewerkt aan een eigen ontwerp van een model en animates. Er is immers met praktisch niets begonnen en nu kan er een verhaal in 3D worden weergegeven. Echter is het precies dit punt, waar nu vooral op uitgebreid zou kunnen worden en deze lat en doelstelling wordt vrij gemakkelijk te hoog gelegd aangezien de grafische mogelijkheden min of meer onbegrensd zijn. Daarbij ligt de ‘funfactor’ sterker bij het maken van nieuwe en betere ‘eyecandy’. Uiteindelijk zullen er Nieuwe modellen en acties (en dus animaties) nodig zijn om gevarieerd verhaal te kunnen weergeven. Echter het belang hiervan is ondergeschikt aan het maken van een complete en goed werkende basis versie en er gaat al snel te veel tijd zitten in het maken van deze ‘eyecandy’. Wat betreft Graphics & virtual reality is het doel daarom slechts gedeeltelijk bereikt.

Er is veel tijd gestoken in het uitbreidbaar en generiek houden van de opzet en implementatie. Op dit punt kan wel gesteld worden dat de gewenste doelen bereikt zijn. De meeste toekomstige uitbreidingen zullen acties zijn; er is extra zorg besteed aan het mogelijk maken hiervan.

Met het oog op uitbreidingen is er ook een drietal tutorials ontwikkeld, die het maken van een model in 3D Studio Max, het maken van extra acties en het maken van nieuwe locaties behandelt. Omwille van toegankelijkheid zijn deze tutorials in het Engels opgesteld. Uiteraard hebben deze tutorials, hun meerwaarde nog niet bewezen, maar ze bieden een helpende hand bij het overschrijden van de drempel voor personen, die dit project willen voortzetten.

Hopelijk zal dit het geval zijn.

Referenties

[OGR05] Ogre3D website : http://www.ogre3d.org

[API05] The Ogre3D online API reference :
http://www.ogre3d.org/docs/api/html/

[MOD05] Werf, R.J. van der, How To Model For ViVi, tutorial
ViVi DVD\Tutorials\tutorial - How to model for ViVi.doc, 2005

[ACT05] Werf, R.J. van der, How to make your own action, tutorial
ViVi DVD\Tutorials\tutorial - How to make your own action.doc, 2005

[LOC05] Nouwens, I.C.C., How to make your own location, tutorial
ViVi DVD\Tutorials\tutorial - How to make your own location.doc, 2005

[3DS05] 3D Studio Max website : http://www.discreet.com

[DOC05] Werf, R.J. van der & Nouwens, I.C.C., ViVi source Documentation
ViVi DVD\ViVi\ViVi API\index.htm, 2005

[CPP05] Cppdoc website : http://www.cppdoc.com
ViVi DVD\Software\cppdoc2.zip