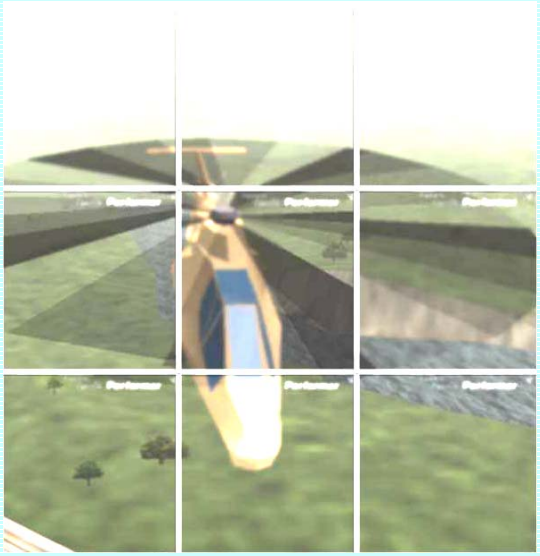
Realtime Distributed Visualisation for Commercial Off The Shelf Clusters



Versie 3.01 1/7/04

Edgar Herbie Antonius van Tetering

TNO FEL

Nederlandse Organisatie voor Toegepast Natuurwetenschappelijk Onderzoek
TNO Fysisch Electrotechnisch Laboratorium

TNO-rapport
RDV.1.0.3

Realtime Distributed Visualisation

TNO Fysisch en Elektronisch Laboratorium

Oude Waalsdorperweg 63
Postbus 96864
2509 JG 's-Gravenhage

Telefoon 070 374 00 00
Fax 070 328 09 61

Datum
4 maart 2005

Auteur
E.H.A. van Tetering

Opdrachtnummer	001
Opdrachtgever	FEL
Organisatieonderdeel	0204
Projectbegeleider	H. Jense
Organisatieonderdeel	Command, Control & Simulation

Rubricering	Ongeclassificeerd
Titel	Realtime Distributed Visualisation
Managementuittreksel	3
Conclusies	87
Rapporttekst	4-86
Bijlagen	Models, Headers, Manuals
Vastgesteld door	H. Jense
Vastgesteld d.d.	R. Krijnen

Exemplaarnr.	1
Oplage	4
Aantal pagina's	91
Aantal bijlagen	3

TNO Fysisch en Elektronisch Laboratorium is onderdeel
van TNO Defensieonderzoek waartoe verder behoren:

TNO Prins Maurits Laboratorium
TNO Technische Menskunde

Nederlandse Organisatie voor toegepast-natuurwetenschappelijk onderzoek TNO

Verklaring

Naam: E.H.A. van Tetering

Studentnummer: 96007322

Hierbij verklaar ik dat ik deze scriptie en afstudeeropdracht geheel zelfstandig heb uitgevoerd en geen andere bronnen of hulpmiddelen heb geraadpleegd dan degenen die in dit verslag staan aangegeven.

Den Haag, dinsdag 2 december 2003.

Managementuittreksel

Probleemstelling
Binnen het TNO Fysisch Electrotechnisch Laboratorium wordt onderzoek en ontwikkeling gedaan naar visualisatie en simulatie op clusters van civiele computers. Voor het realiseren en implementeren van simulators op een cluster van niet militaire civiele systemen in een realtime omgeving dienen bestaande componenten van deze simulators geparallelliseerd te worden op verschillende niveaus. De geparallelliseerde onderdelen dienen voor elk simulator component in realtime gesynchroniseerd te worden zodat de simulator onderdelen samen hetzelfde gedrag als een niet geparallelliseerde simulator vertonen.

Beschrijving van de werkzaamheden
Specifieke onderzoeks- en ontwikkelings gebieden zijn geweest het ontwikkelen van een realtime synchronisatie algoritme, een algoritme voor het interpoleren van processen, een applicatie programmeertaal voor het communiceren van gegevens, een applicatie programmeertaal voor het ontwikkelen van grafische clusters, een applicatie programmeertaal voor het uitlezen van randapparaten en een groep modellen. Om het resultaat van de werkzaamheden te demonstreren, is er een parallelle simulator ontwikkeld.

Resultaten en conclusies
Uit metingen is gebleken dat de synchronisatie algoritmen voldoen aan de eisen voor realtime prestaties in een commerciële omgeving. De realtime synchronisatie algoritmen benaderen een van te voren ingestelde doel simulatie- en swapbuffer synchronisatietijd en blijft binnen een afwijkingstijd die voldoet aan de eisen voor realtime applicaties in een niet realtime omgeving. Verder is gebleken dat de dynamische frequentiebepaling van datasynchronisatie- en swapbuffer processen de applicatie naar de ideale omstandigheid stuurt.

Detaillering in de richting van lokaal gestuurd object management zal de controle op de simulatie objecten verbeteren, mede omdat gelocaliseerde controle op globale creatie- en destructie processen het instantiëren van objecten over meerdere instanties van een applicatie in een cluster mogelijk maakt. Verfijning van de applicatie programmeertalen op het gebied van dynamisch instantieerbare kanalen, schermen en grafische pijpen vergroot de scaleerbaarheid van de applicatie op lokale machines. Verder kan ontwikkeling van interpolatie- en realtime algoritmen voor proces interpolatie en frequentie controle op realtime hardware een bijdrage leveren aan een significante verbetering van de realtime applicatie prestaties.

Toepasbaarheid

- Datamining applications
- Simulation applications
- Urban planning applications
- Situation awareness systems
- Medical visualisation systems
- Graphics simulation systems

Voorwoord

Deze scriptie is het resultaat van een afstudeerstage bij het TNO Fysisch Electrotechnisch Laboratorium te Den Haag. Voor de afstudeerstage is er een simulator ontwikkeld die gebruik maakt van een netwerk van civiele computers. Er is gedurende de ontwikkeling een intensieve literatuurstudie verricht naar gedistribueerde systemen, realtime netwerken en simulators. Tijdens de stage heb ik samengewerkt met mijn stagebegeleider en een aantal wetenschappers die werkzaam zijn aan het TNO Fysisch Electrotechnisch Laboratorium. Ik ben iedereen die aan de afstudeerstage heeft meegewerkt veel dank verschuldigd.

Ik wou in het bijzonder mijn stagebegeleider Jan Schramp bedanken voor zijn persoonlijke adviezen en waardevol technisch inzicht. Ook wou ik Jan van Geest, Mario Smeenk, Janjelle Boomgaardt, Ronald van Maarseveen en mijn baas Hans Jense bij het TNO Fysisch Electrotechnisch Laboratorium bedanken voor hun inzicht op het gebied van respectievelijk simulator interconnectiviteit, realtime stuuralgoritmen, realtime hardware, simulatie van natuurkundige processen en besturingssystemen.

Ik hoop dat de producten die ik heb ontwikkeld bij het TNO Fysisch Electrotechnisch Laboratorium worden doorontwikkeld. Met de producten zoals die nu bestaan kunnen gedistribueerde simulatie demonstraties worden ontwikkeld. Toevoegingen op het gebied van configuratie-, shader-, effects-, simulator- en interconnectiviteits bibliotheken kunnen de producten doen uitgroeien tot een volwaardig systeem voor gedistribueerde simulators.

E.H.A. van Tetering.

Inhoudsopgave

1 Inleiding

Deze scriptie is het resultaat van een acht maanden durende stage bij het TNO Fysisch Electrotechnisch Laboratorium te Den Haag. Gedurende de stage is er onderzoek gedaan naar gedistribueerde applicaties, grafische clusters, en gedistribueerde simulaties. Tijdens het onderzoek zijn er een gedistribueerd grafisch cluster, een groep applicatie programmeertalen en een simulatie onwikkeld voor een netwerk van civiele machines.

Een eindproduct in de vorm van applicatie programmeertalen voor de ontwikkeling van gedistribueerde grafische applicaties en object georiënteerde modellen is opgeleverd ter beoordeling door een onafhankelijke groep wetenschappers. Van de groep wetenschappers zijn er een aantal actief als onderzoekend medewerker aan het TNO Fysisch Electrotechnisch Laboratorium te Den Haag. In deze scriptie komen de begrippen en de gedachtegang naar voren die een rol hebben gespeeld tijdens ontwikkeling van de applicatie programmeertalen en object georiënteerde modellen.

Het tweede hoofdstuk bevat de probleemstelling. De probleemstelling definieert en maakt een afbakening van het te onderzoeken probleem. In de probleemstelling komen de elementen naar voren die een belangrijk effect hebben gehad op de ontwikkeling van het gedistribueerd grafisch cluster en de applicatie programmeertalen.

Het derde hoofdstuk beschrijft een simulatie applicatie die is geprogrammeerd met de ontwikkelde applicatie programmeertalen voor gedistribueerde grafische applicaties. Het hoofdstuk beschrijft tevens de broncode van de simulatie applicatie en gaat dieper in op de verdeling van elementen in het grafisch cluster.

In het vierde hoofdstuk wordt er een simulatie framework beschreven dat is ontwikkeld door onderzoekers aan het TNO Fysisch Electrotechnisch Laboratorium. Het simulatie framework bevat een grafische applicatie programmeertaal die kan worden gebruikt voor het ontwikkelen van de grafische onderdelen in simulatie applicaties. Parallellisering van deze grafische applicatie programmeertaal is een aanzienlijk onderdeel van het onderzoek in deze scriptie.

Het vijfde hoofdstuk is een uitwerking van een onderzoek naar bestaande grafische clusters en parallelle applicatie programmeertalen. Het onderzoek in het vijfde hoofdstuk is voornamelijk gericht op de generieke functionaliteit van de bestaande grafische clusters. Tijdens het onderzoek zijn er afwegingen gemaakt die een invloed hebben gehad op de uiteindelijk ontwerpkeuzen van de ontwikkelde programmeertalen.

In het zesde hoofdstuk is er een analyse gemaakt van de grafische clusters en parallelle applicatie programmeertalen die zijn besproken in het vijfde hoofdstuk. Tijdens de analyse komen er concepten van de bestaande grafische clusters naar voren die gebruikt zijn bij de ontwikkeling van het grafisch cluster en de applicatie programmeertalen voor het TNO Fysisch Electrotechnisch Laboratorium.

Het zevende hoofdstuk beschrijft de kernproblemen die in het kader van de ontwikkelingen zijn onderzocht. Het zevende hoofdstuk introduceert tevens een aantal ontwikkelde concepten die gebruikt zijn voor het oplossen van de kernproblemen uit het tweede hoofdstuk. De concepten worden in de latere hoofdstukken verder uitgewerkt.

Het achtste hoofdstuk beschrijft een omgeving die is ontwikkeld voor het configureren en installeren van applicatie programmeertalen voor gedistribueerde applicaties. De configuratie en installatie omgeving biedt een basis voor de doorontwikkeling van configuratieapplicaties voor realtime gedistribueerde grafische clusters.

In het negende hoofdstuk wordt er ingegaan op de ontwikkeling van een niet gedistribueerde grafische applicatie programmeertaal die is geschreven met een gespecialiseerde applicatie programmeertaal voor grafische applicaties. Het negende hoofdstuk introduceert ook een algoritme dat gebruikt wordt voor synchronisatie van applicatie processen in de tijd.

Het tiende hoofdstuk bespreekt de ontwikkeling van een applicatie programmeertaal voor gedistribueerde grafische applicaties. Verder bespreekt het tiende hoofdstuk de ontwikkeling van applicatie programmeertalen voor een datacommunicatie machine en gedistribueerd bestandssysteem. Het hoofdstuk gaat dieper in op de applicatie programmeertalen die zijn ontwikkeld.

Het elfde hoofdstuk bevat een groep object georiënteerde modellen ontworpen om de bestaande grafische applicatie programmeertaal zoals die is ontwikkeld door het TNO Fysisch Electrotechnisch Laboratorium uit te breiden met klassen die de applicatie voorbereidt op parallelle omgevingen.

Het twaalfde hoofdstuk is een evaluatie van een ontwikkelde simulatie applicatie. In dit hoofdstuk komen metingen naar voren die zijn verricht op het grafisch cluster tijdens de uitvoering van de ontwikkelde simulatie omgeving. De metingen worden tijdens de evaluatie kwalitatief besproken.

In het dertiende hoofdstuk wordt er teruggekeken op de ontwikkelingen en wordt er ingespeeld op de doorontwikkeling van toekomstige versies van de applicatie programmeertalen. Ook worden er in dit hoofdstuk conclusies getrokken over de ontwikkeling van commerciële gedistribueerde simulatie applicaties in het algemeen.

2 Probleemstelling

Voor het realiseren van een grafische applicatie op een cluster van civiele computers in een realtime omgeving dienen bestaande componenten van een niet parallelle grafische applicatie geparallelliseerd te worden. Het gevolg van de parallellisering is een grafisch cluster, met gedistribueerde beelden, scèneobjecten en grafische processen. De grafische applicatie moet parallelliseren, dynamisch verdelen en in realtime communiceren over een cluster van civiele machines. Bij een gedistribueerd beeld worden er meerdere schermen gebruikt om naar een enkele scène te kijken. De objecten in de scène zijn ook gedistribueerd maar lijken als één object in de scène aanwezig te zijn.

Alle beelden of een groep beelden in het grafisch cluster moeten samen te voegen zijn tot één enkel beeld en de processen van de applicatie dienen te worden geparallelliseerd en gedistribueerd. De geparallelliseerde en gedistribueerde processen moeten voor elk grafisch component gesynchroniseerd worden zodat alle processen bijdragen aan één grafisch hoofdproces.

De beelden kunnen slechts één enkel beeld genereren als de grafische processen die de beelden genereren tegelijkertijd of gesynchroniseerd uitgevoerd worden. Voor een gesynchroniseerde uitvoer van de grafische processen moeten er algoritmen ontworpen worden voor de synchronisatie van beeldpunten, beelden en gegevens; respectievelijk genlock-, swapbuffer- en gegevens synchronisatie algoritmen. Hierbij spelen drie procestypen een rol, twee zijn verantwoordelijk voor het tekenen van beeldpunten en beelden, en één voor het verwerken van gegevensstromen.

Bij het synchroniseren van processen die verantwoordelijk zijn voor het tekenen van beeldpunten is een vorm van synchronisatie nodig waarbij het tekenen van een bepaalde groep beeldpunten op hetzelfde moment begint als het tekenen van een gerelateerde groep beeldpunten op een ander beeldscherm. Het maximale verschil tussen de tekenmomenten van een gesynchroniseerde groep beeldpunten noemt men de genlock latency. Tijdens synchronisatie dient de genlock latency zo laag mogelijk gehouden te worden.

Bij het synchroniseren van processen verantwoordelijk voor het tekenen van beelden is een vorm van synchronisatie nodig die de buffers waarin de beelden zijn opgeslagen gelijk doet overgaan. Het maximale verschil tussen de momenten waarop het tekenen van ieder beeld begint in een gesynchroniseerde groep buffers wordt de swapbuffer latency genoemd. Tijdens synchronisatie moet de genlock latency gesynchroniseerd worden met de swapbuffer latency zodat de hoeveelheid beeldpunten per beeld gelijk blijven lopen.

Bij het synchroniseren van processen die verantwoordelijk zijn voor gegevensstroom verwerking dienen gegevens behorende bij gedistribueerde scèneobjecten gelijk te blijven over alle machines in een cluster. Tijdens gegevens synchronisatie zal er een globale tijdsafwijking ontstaan die de werking van de gedistribueerde grafische processen beïnvloedt. De ontstane tijdsafwijking mag geen vertraging veroorzaken van de swapbuffer en genlock processen.

3 Prototype applicatie

Om de ontwikkelde applicatie programmeertalen te demonstreren is er een applicatie ontworpen met de ontwikkelde applicatie programmeertalen. De demonstratie applicatie bestaat uit een simulatie met daaraan gekoppeld een aantal externe en interne controllers. De controllers besturen verschillende simulatie componenten zoals een gedistribueerd camera systeem en een Commanche gevechtshelikopter.

De applicatie is een commerciële demonstratie van de mogelijkheden met grafische clusters voor de civiele markt. De applicatie is ontwikkeld voor een civiele machine met een gesimuleerd cluster of een cluster van civiele machines. De applicatie is ontwikkeld op een enkele civiele computer en getransporteerd naar een civiel cluster in een ruimte van het TNO Fysisch Electrotechnisch Laboratorium. De ontwikkelde applicatie programmeertalen worden in de applicatie gedemonstreerd.

De volgende paragrafen geven een gedetailleerde beschrijving van de simulatie entiteiten, simulatie architectuur, simulatie omgeving, en simulatie broncode. De simulatie entiteiten zijn de entiteiten die een rol spelen in de simulatie. De beschrijving van de simulatie architectuur geeft een beschrijving van de architectuur van de simulatie applicatie. De beschrijving van de simulatie omgeving bevat een schermvoorbeeld van de simulatie en de omgeving waarin de simulatie wordt uitgevoerd. De beschrijving van de simulatie broncode geeft een gedetailleerde uitleg van de simulatie broncode.

3.1 Simulatie entiteiten

De simulatie entiteiten bestaan uit een Commanche helikopter entiteit, een lichtpunt, achtergrond mist en een stad. De Commanche helikopter entiteit genereert in de simulatie omgeving positie en oriëntatie berichten die omgezet kunnen worden in gespecialiseerde simulator berichten voor koppeling aan externe simulators. Indien de helikopter entiteit gekoppeld is, kunnen andere simulators gebruik maken van de gegevens die het dynamisch helikopter model genereert. De Commanche helikopter entiteit bestaat uit een zorgvuldig gemodelleerd dynamisch model en een grafisch model. Het dynamische model is gekoppeld aan een stuurknuppel die wordt gebruikt voor besturing van de Commanche helikopter. Het grafische model is een drie dimensionale representatie van de Commanche helikopter.

De virtuele wereld wordt bekeken door het oog van een dynamische camera die distribueert over de verschillende machines in het grafisch cluster. De positie en oriëntatie van de camera worden vanaf een lokale machine bestuurd. De camera maakt gebruik van een dynamisch en gesynchroniseerd camera model waarmee soepele transities naar camera posities gerealiseerd worden. De beelden die door de camera bekeken worden zijn gesynchroniseerd met behulp van een groep synchronisatie algoritmen. Vanwege de synchronisaties ontstaan er soepele camera transities waardoor gebruikers ruimtelijk inzicht krijgen in de omgeving. Met de stuurknuppel en een muis kunnen de camera zichtvelden ingesteld worden.

De simulatie applicatie distribueert gegevens van de verschillende dynamische entiteiten over het cluster. De simulatie applicatie bepaalt lokaal de positie en oriëntatie van de camera en de Commanche helikopter entiteit en distribueert deze over het cluster door gebruik te maken van een datacommunicatie machine. De datacommunicatie machine houdt hierbij alle gegevens gesynchroniseerd en gedistribueerd, zodat de dynamische entiteiten bij elke simulatiestap gesynchroniseerde en gedistribueerde eigenschappen hebben. In de latere hoofdstukken wordt er dieper ingegaan op de synchronisatie algoritmen en datacommunicatie machine.

Voor het dynamische model en dynamisch camera model is de penalty methode [10] gebruikt om bewegingen van camera en helikopter entiteit simulaties te bepalen. Bij camera bewegingen is een additionele natuurlijk gedempte ruisvector geïntroduceerd om de illusie te wekken dat de camera door een mens bediend wordt. Bij het dynamische model van de Commanche helikopter entiteit is een hybride model met behulp van discrete en niet discrete methoden [10] ontworpen om het dynamische gedrag van de entiteit zo realistisch mogelijk te simuleren. De volgende paragrafen bespreken de voorheen genoemde elementen in meer detail.

3.2 Simulatie architectuur

De onderstaande simulatie architectuur geeft aan hoe de simulatie is ontworpen. De muis en de stuurknuppel zijn gekoppeld aan het camera model en simulator model. De stuurknuppel bepaalt hierbij de camera positionerings typen en de muis de relatieve camerahoeken. Het camera model bevindt zich op de eerste machine en distribueert de camera positie en oriëntatie over negen machines. Het simulator model bevindt zich op de eerste machine en distribueert de positie en oriëntatie van de Commanche helicopter over negen machines.

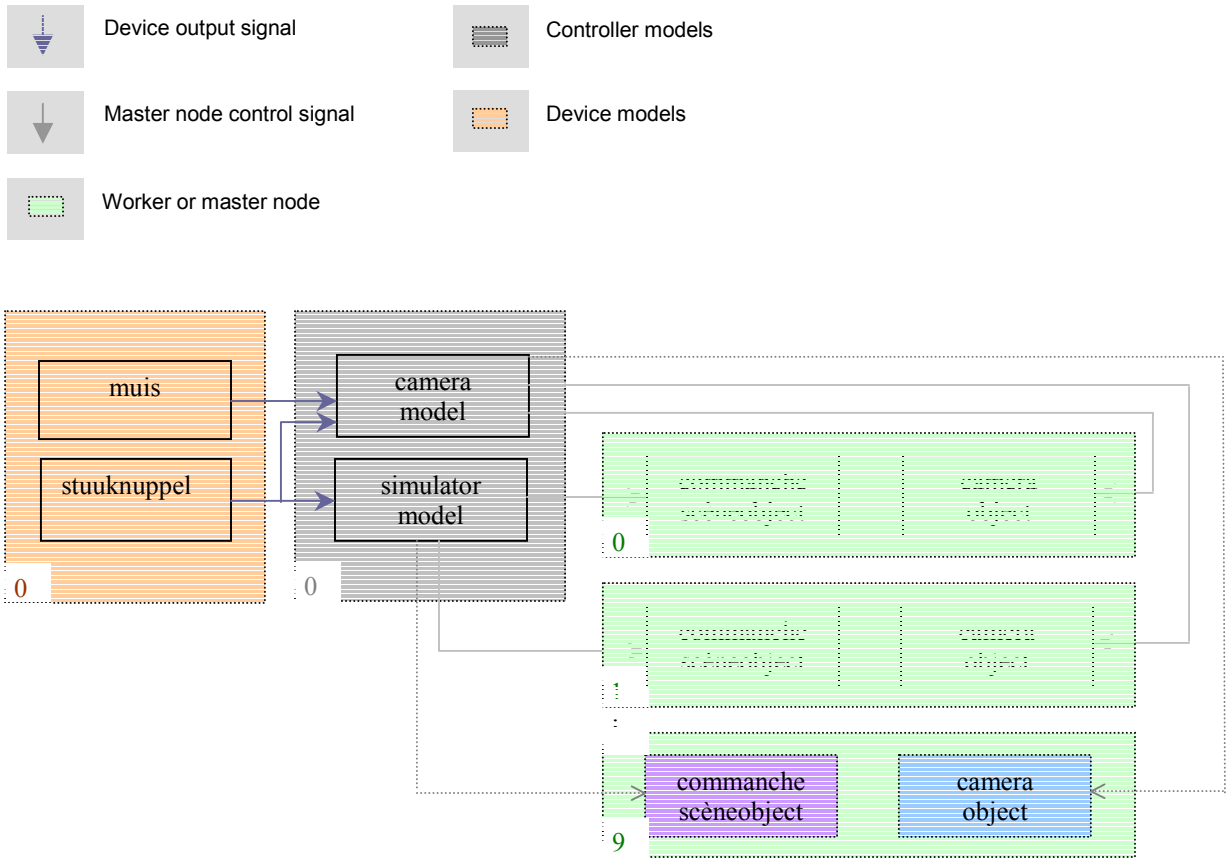


Fig0.1

Uit de simulatie architectuur is op te merken dat de gerepliceerde gedistribueerde objecten het camera object en commanche object zijn. De objecten zijn gedistribueerd over negen nodes dat in het bovenstaande model vereenvoudigd is. Voor het camera model geldt dat de camera berekeningen op de eerste machine gedaan worden. Ook voor het simulator model geldt dat de simulator berekeningen op de eerste machine gedaan worden. De camera en simulator worden vervolgens gedistribueerd en gesynchroniseerd over het netwerk met instellingen voor de camera die er een samengesteld enkelvoudig beeld genereren.

3.3 Schermvoorbeeld simulatie

Het grafisch cluster bestaat uit meerdere machines die zijn gekoppeld middels een communicatie netwerk. Elke machine heeft een apart beeldscherm en grafische kaart. Het onderstaande schermvoorbeeld is een voorbeeld van een simulatie voor negen machines in een grafisch cluster. Het schermvoorbeeld van de simulatie is weergegeven op een enkele machine waarbij ieder scherm een apart beeldscherm voorstelt. Alle beeldschermen geven in een grafisch cluster hetzelfde beeld als in de afzonderlijke schermen in het schermvoorbeeld.

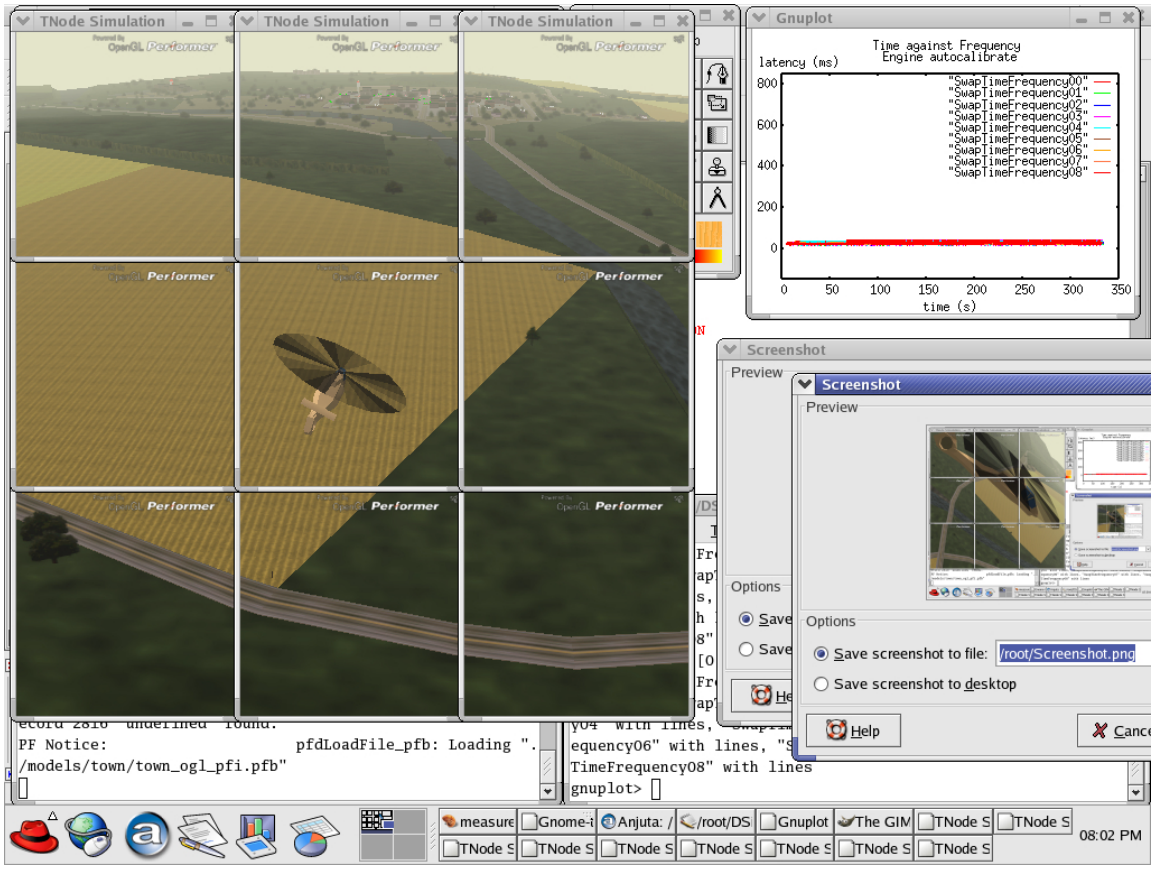


Fig0.2

Het schermvoorbeeld is een representatie van de ontwikkelomgeving die is gebruikt om de applicatie programmeertalen voor het grafisch cluster te ontwikkelen. De ontwikkelomgeving representeert een cluster van civiele machines. De bovenstaande schermafbeelding is een beeld van het gerepresenteerde cluster. De applicatie is zo ontwikkeld dat het op machine en data niveau scaleert afhankelijk van de hoeveelheid data in de simulatie en de hoeveelheid machines in het cluster. In het bovenstaande geval zal de applicatie scaleren over een cluster van negen machines die zijn gekoppeld middels een netwerk. Er zal in de latere hoofdstukken dieper worden ingegaan op de scalering van de applicatie in een cluster.

3.4 Simulatie broncode

De onderstaande broncode is een representatie van de gedistribueerde simulator zoals die in het schermvoorbeeld is getoond. De toelichtende tekst verklaart de betekenis van de broncode en legt uit welke processen achter de broncode worden gerealiseerd. De broncode wordt lineair doorlopen en wordt bij elke stap besproken in de toelichtende tekst. Alvorens een simulatie of parallelle applicatie te programmeren geeft de ontwikkelaar aan welk globaal gedrag de grafische applicatie moet vertonen. In de onderstaande code bepaalt de applicatie zelf op dynamische wijze frequenties voor gegevens- en beeldsynchronisatie en worden processen geïnterpoleerd.

```
static double dTimer = 0;

VisualEngine_Initialise(iArgumentCount, pacArgumentContents);
VisualEngine_SetDynamicFrequencyOn();
VisualEngine_SetDynamicInterleaveOn();
```

Na initialisatie en instelling van de applicatie worden de schermen voor de verschillende machines ingesteld. Omdat de simulatie op een enkele machine wordt uitgevoerd worden hieronder de schermen zo ingesteld dat ze ieder een aparte plek krijgen op het beeldscherm. Na instelling van de schermen worden ze op alle machines weergegeven.

```
Screen_SetSourceNode(ALL);
Screen_SetWindowed();
Screen_SetSize(200, 200);
Screen_SetViewingDistance(1, 3000);

int iNodeID = 0;
iNodeID = StreamingEngine_GetNodeID();

Screen_SetSourceNode(iNodeID); //screen at this node
Screen_SetPosition((iNodeID-(3*(iNodeID/3)))*200, (iNodeID/3)*200);

Screen_SetRasterAngles(15, 15);

Screen_SetRasterPlace(-1, +1, NODE00); //left upper screen
Screen_SetRasterPlace(+0, +1, NODE01);
Screen_SetRasterPlace(+1, +1, NODE02);

Screen_SetRasterPlace(-1, +0, NODE03);
Screen_SetRasterPlace(+0, +0, NODE04); //middle screen
Screen_SetRasterPlace(+1, +0, NODE05);

Screen_SetRasterPlace(-1, -1, NODE06);
Screen_SetRasterPlace(+0, -1, NODE07);
Screen_SetRasterPlace(+1, -1, NODE08); //right lower screen

Screen_SetSourceNode(ALL);
Screen_Display();
```

Na de scherminstellingen wordt er bepaald op welke machines een muis en stuurknuppel zijn aangesloten. Tijdens het onderzoek zijn applicatie programmeertalen ontwikkeld om op eenvoudige wijze waarden van de muis en stuurknuppel in te lezen. In de onderstaande code worden de muis en stuurknuppel geïnitialiseerd op de eerste machine.

```
if(StreamingEngine_HasNodeID(MASTER))
{
  Joystick_Initialise();
  Mouse_Initialise();
}
```

Na initialisatie van de muis en stuurknuppel worden de grafische objecten aangemaakt. De applicatie programmeertaal is ontwikkeld om in een bijna willekeurige volgorde de applicatie te schrijven. Vanwege de willekeurige volgorde kunnen er dynamisch objecten aangepast worden tijdens uitvoer van de applicatie. Het eerste grafische object dat wordt aangemaakt in de simulatie is de globale achtergrond mist.

```
Fog_SetFogOn();
Fog_SetCloudsOff();
Fog_SetDensity(0.6, 0.5);
Fog_SetColor(0.9, 0.9, 0.8);
```

Na instelling van de mist wordt er een zon geplaatst. De zon biedt licht in een anders vrij donkere grafische wereld. Verder worden ook de kleur, positie, oriëntatie en weerspiegelingseffecten van het licht op objecten in de grafische wereld in de onderstaande code ingesteld.

```
Light* pLight_sun = Light_NewLight();
Light_SetColor(pLight_sun, 1.0, 0.9, 0.9);
Light_SetBackgroundReflection(pLight_sun, 0.3);
Light_SetSurfaceReflection(pLight_sun, 0.7);
Light_SetShinyReflection(pLight_sun, 1.0);
Light_SetPosition(pLight_sun, 10.0, 20.0, 100.0);
Light_SetOn(pLight_sun);
```

Na instelling en het aanzetten van de zon wordt er een grafische entiteit geplaatst in de virtuele wereld met als bronmachine de eerste machine in het cluster. De bronmachine geeft aan vanuit welke machine de grafische entiteit bestuurbaar is. De entiteit wordt weergegeven door het grafisch Commanche helikopter model. Omdat het object nog geen fysische representatie heeft wordt tevens de fysische representatie van het object bepaald.

```
Entity* pEntity_entity = Entity_NewEntity();
Entity_SetSourceNode(pEntity_entity, MASTER);
Entity_SetFile(pEntity_entity, "./models/commanche.flt");
Entity_SetPosition(pEntity_entity, 800.0, 2000.0, 100.0);
Entity_SetOn(pEntity_entity);

Physics* pPhysics_physics = Entity_GetPhysics(pEntity_entity);
Physics_SetSourceNode(pPhysics_physics, MASTER);
Physics_SetDynamicsType(pPhysics_physics, SKY_PROPULSION_HELI);
```

Na bepaling van het fysische model en het plaatsen van de entiteit in het cluster met als bronmachine de eerste machine wordt de virtuele wereld ingeladen. De grafische machine kan meerdere virtuele werelden bevatten en kan deze werelden aan of uit zetten. De virtuele werelden zijn niet verplaatsbaar maar objecten kunnen zich wel in de wereld verplaatsen. In het onderstaande voorbeeld wordt een grafisch stadsmodel gebruikt.

```
World* pWorld_world = World_NewWorld();
World_SetFile(pWorld_world, "./models/town/town.pfb");
World_SetOn(pWorld_world);
```

Na bepaling van de virtuele wereld wordt er een camera ingesteld en wordt het besturingsmodel van de camera geplaatst op de eerste machine. De camera wordt zo ingesteld dat zij naar de helikopter entiteit kijkt. De camera wordt in de hierna gepresenteerde broncode verder ingesteld.

```
Camera_SetSourceNode(MASTER);
Camera_AttachToEntity(pEntity_entity);
Camera_SetRelativeDistance(200);
Camera_SetCinematicViewing();
```

Na alle simulator instellingen worden de verschillende simulator onderdelen gekoppeld en wordt de simulatie gestart. In dit geval gaat het om een eenvoudige simulatie van een helikopter die bestuurd wordt door de stuurknuppel. Verder zijn de knoppen van de stuurknuppel voor verschillende camera instellingen geprogrammeerd. De zevende knop zet de camera naar buitenzichtveld. Het buitenzichtveld is in deze simulator geprogrammeerd om met de muis de hoek van de camera te laten bepalen. Met de achtste knop bepaalt de camera zelf het zichtveld. De negende knop op de stuurknuppel plaatst de camera in het object waaraan het is gekoppeld.

```
while(dTimer < 3000)
{
  dTimer = RealtimeBarrier_GetTimeMilliseconds();

  Physics_SetInputParameters( pPhysics_physics, 0.0, 0.0, 0.0,
                                                Joystick_GetYaw(),
                                                Joystick_GetPitch(),
                                                Joystick_GetRoll(),
                                                Joystick_GetThrottle() );

  if (Joystick_GetFireClickSeven())
  {
    Camera_SetOutsideViewing();
  }
  if (Joystick_GetFireClickEight())
  {
    Camera_SetCinematicViewing();
  }
  if ( Joystick_GetFireClickNine())
  {
    Camera_SetInsideViewing();
  }

  if (Joystick_GetFireClickTen())
  {
    VisualEngine_Destroy();
    exit(0);
  }

  Camera_SetRelativeHeading( Mouse_GetPositionX(),
                             Mouse_GetPositionY(),
                                              0.0 );

  if(StreamingEngine_HasNodeID(MASTER))
  {
    Joystick_Update();
    Mouse_Update();
  }

  VisualEngine_Update();
}

VisualEngine_Destroy();
exit(0);
```

Tijdens de simulatie beweegt de camera bij elke transitie naar een nieuw zichtveld over de wereld naar de nieuw ingestelde positie zodat de gebruiker ziet waar de camera heengaat. Hierdoor krijgt de gebruiker inzicht in de objecten waar de camera naar kijkt en een beter zicht op de virtuele wereld. De tiende knop op de stuurknuppel sluit de simulatie af en beëindigt de parallelle processen in het grafisch cluster. Na drieduizend milliseconden beïndigt de simulatie zichzelf.

4 Simulatie framework

Het simulatie framework, of ASF, ontwikkeld door het TNO Fysisch Electrotechnisch Laboratorium is een applicatie programmeer omgeving bedoeld voor het generiek modelleren en programmeren van scaleerbare en configureerbare realtime defensie simulators [36]. Het framework bevat herbruikbare componenten voor gegevensverwerking, gegevens verwerking, simulator koppeling, simulatie berichtverwerking en generatie van gesimuleerde geluiden en beelden [36].

Momenteel is het framework, met in het bijzonder de grafische applicatie programmeertaal, ontwikkeld voor een geavanceerde omgeving van Silicon Graphics machines die zich bevinden in communicatie netwerken gebaseerd op de Silicon Graphics NUMAFlex [42] architectuur. Deze architectuur stelt applicatie ontwikkelaars in staat om het gedeelde geheugen van de Silicon Graphics machines als één geheugengebied te beschouwen. Verder beschouwt deze architectuur een lokaliseerbaar en classificeerbaar geheugen waarbij anders gefragmenteerd geheugen zich op dezelfde plek en onder dezelfde identificatie in een geheugen raster bevindt [42]. Civiele computersystemen beschikken over vergelijkbare geheugenmanipulatie architecturen maar daarbij is het probleem van lokalisatie niet altijd opgelost.

Een deelverzameling van het framework moet bruikbaar gemaakt worden voor clusters van machines die zijn gebaseerd op civiele netwerk architecturen waarbij de NUMAFlex geheugen omgeving niet altijd beschikbaar is. Generiekere omgevingen voor het delen van geheugen zijn echter wel altijd beschikbaar. Om het framework bruikbaar te maken voor clusters van machines die zijn gebaseerd op de civiele netwerk architecturen zijn de framework componenten opgesplitst in specifieke onderzoeksgebieden. De onderzoeksgebieden voor het ontwikkelen van een gedistribueerd grafisch cluster gerelateerd aan dit onderzoek omvatten synchronisatie-, clustering-, realtime-, communicatie- en gegevensverwerkings gebieden.

Behalve onderzoek naar het parallelliseren van de grafische applicatie programmeertaal van het framework wordt er onderzoek gedaan naar interoperabiliteit met op het framework gebaseerde applicaties voor Silicon Graphics Machines. Simulaties en applicaties die zijn ontwikkeld met de applicatie programmeertalen en de applicatie programmeertalen zelf zullen op den duur herbruikbaar moeten worden gemaakt voor gedistribueerde grafische clusters op netwerken van civiele systemen.

In dit hoofdstuk zal er een globale architectuurschets gemaakt worden van het framework en zullen de componenten in het licht van deze globale applicatie architectuur besproken worden. Ieder component wordt afzonderlijk onderzocht en besproken met een gedetailleerdere omschrijving van het grafische component. Het onderzoek uit dit hoofdstuk zal aantonen in hoeverre de componenten van het framework herbruikbaar zijn voor civiele systemen.

4.1 Framework architectuur

Het onderstaande diagram is een architectuurtekening [36,46] met de verschillende framework componenten in het framework. Ieder component is modulair opgebouwd en levert een andere functionaliteit van het framework in de vorm van een specifieke applicatie programmeertaal. De mate van herbruikbaarheid van de componenten is beperkt en aan afhankelijkheden verbonden. Grenzende gebieden geven de afhankelijkheden aan. Componenten die op zichzelf bruikbaar zouden kunnen zijn zonder afhankelijkheden zijn dik omlijnd.

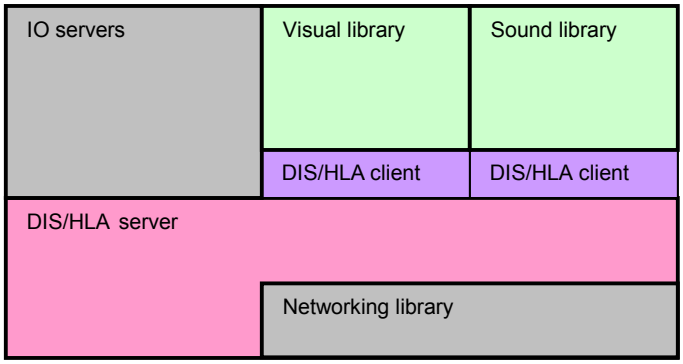


Fig1.0

4.2 Ondersteunings modules

De componenten uit dit framework worden ook wel ondersteuningsmodules genoemd en zijn specifiek ontworpen voor militaire simulaties waarbij sommige ondersteuningsmodules hergebruikt kunnen worden. Met name de geluids- en grafische componenten van het framework zijn bruikbaar voor andere doeleinden als de simulator clients daarbij weggelaten worden. De simulator clients worden gebruikt voor communicatie met de simulatie server. Tijdens communicatie wisselen de simulatie clients simulatieberichten uit met de simulatie server. Momenteel zijn de applicatie programmeertalen sterk gekoppeld aan de simulatie clients. Het resultaat hiervan is een lage samenhang tussen applicatie onderdelen. Vanwege de gemaximaliseerde koppeling, wordt hergebruik en configuratie van de afzonderlijke componenten gecompliceerder. In de volgende subparagrafen wordt een korte uitleg van ieder component gegeven.

4.2.1 Datastroom servers

De gegevensstroom servers, of IO servers, zijn componenten voor het zenden en ontvangen van gegevensstromen afkomstig uit randapparatuur, bestanden of geheugen elementen. De servers bevatten semaforen, geheugen functies, bestandsfuncties en bibliotheken voor het uitlezen van randapparaten. De semaforen zijn gemaakt voor het reserveren van systeembronnen en worden vaak in combinatie met bestands- of geheugen functies gebruikt. De bestands- en geheugenfunctie bibliotheken bevatten procedures voor het lezen en schrijven van respectievelijk schijf geheugen en gedeeld geheugen. Dit omvat functies voor het lezen en schrijven van lokale-, parallelle- en netwerkbestanden en voor de controle van NUMAFlex [36,42] gedeeld geheugen. De bibliotheken voor het uitlezen van randapparaten ondersteunen de meest voorkomende randapparaten.

4.2.2 Simulatie server

De simulatie server, of DIS/HLA server, verwerkt DIS en HLA standaarden voor het uitwisselen van simulatie berichten in een simulatie netwerk. De DIS standaard is een verouderde standaard, maar wordt nog wel gebruikt voor oudere systemen die gekoppeld zijn aan netwerken van simulators. De HLA of DIS berichten die door de server worden verwerkt zijn van oorsprong UDP berichten maar worden via de server applicatie omgezet in realtime simulatie berichten voor de ASF [36]. Het voordeel van de omschakeling naar een nieuw communicatie protocol is dat simulators direct met elkaar kunnen communiceren middels een hogere abstractielaag. Behalve het verbeteren van de communicatie wordt het ook eenvoudiger voor applicatie ontwikkelaars om simulatie clients te schrijven.

4.2.3 Simulatie client

De DIS/HLA clients ontvangen ASF berichten van de DIS/HLA server en zetten deze om in berichten voor het genereren van beelden of geluiden. Clients die zijn gekoppeld aan de DIS/HLA server communiceren middels ASF berichten met de grafische- of geluidsapplicaties die de beelden of geluiden voor simulaties genereren [36]. Hierdoor worden de clients sneller omdat ze simulatie berichten in realtime verwerken. De simulatie clients zijn momenteel nog sterk geïntegreerd met de server. De integratie stelt applicatie ontwikkelaars in staat om directe aanpassingen te maken aan de clients en de server zonder een fragmentatie van de broncode.

4.2.4 Geluids bibliotheek

De geluidsbibliotheek is een component voor het genereren van geluiden voor militaire simulaties. Omdat de simulaties militair zijn is realiteitsgehalte van absoluut belang. De bibliotheek is daarom zo ontworpen dat drie dimensionaal geluid in verschillende omgevingen geplaatst kan worden met een drie dimensionale module voor additioneel realiteitsgehalte. De geluidsbibliotheek is geschreven voor het windows platform; maar er zijn mogelijkheden om de bibliotheek uit te breiden met realtime geluidshardware om een realistischer positionele bewustwording van het geluid te kunnen beleven. De geluidsbibliotheek biedt behalve geluidsgeneratie ook mogelijkheden voor het verwerken van realtime geluidsstromen.

4.2.5 Grafische bibliotheek

De grafische applicatie programmeerbibliotheek is het centrale component van dit onderzoek. De grafische applicatie programmeerbibliotheek is een applicatie programmeertaal die alleen in een Silicon Graphics omgeving bruikbaar is. Verder is de programmeertaal niet geheel beschikbaar voor clusters van civiele computers. Tijdens het onderzoek is er een ontwerp voorgesteld dat de bibliotheek op een cluster van civiele computers bruikbaar maakt voor parallelle simulaties. Er kan met zekerheid gezegd worden dat ontwikkeling van parallelle simulaties op civiele clusters mogelijk is omdat in de praktijk is gebleken dat parallelle simulaties op clusters van civiele computers al in een ver stadium gevorderd zijn [3]. In de volgende paragraaf zal een applicatie model van de grafische applicatie programmeertaal zoals die door het TNO Fysisch Electrotechnisch Laboratorium is ontworpen worden besproken.

4.3 Simulator visual

De grafische applicatie programmeertaal is in de niet parallelle vorm bruikbaar op een netwerk van Silicon Graphics machines. De grafische processen van een grafische applicatie ontwikkeld met de applicatie programmeertaal, worden geïnstantieerd en uitgevoerd door de gespecialiseerde applicatie programmeertaal Performer van Silicon Graphics. Bij het parallelliseren van de grafische applicatie voor een grafisch cluster, moeten de grafische processen gesynchroniseerd worden op de machines in het cluster. In de volgende paragrafen bevinden zich een aantal applicatie architectuurmodellen van de applicatie programmeertaal voor grafische applicaties zoals die nu is ontworpen. Verder worden het gebruikers model en de modellen van de applicatie programmeertaal zoals ontwikkeld door het TNO Fysisch Electrotechnisch Laboratorium besproken. Later worden deze uitgebreid met parallelle extensies.

4.3.1 Gebruikers model

Voor de grafische applicatie zijn er een aantal basisfunctionaliteiten gerealiseerd die zijn weergegeven in het onderstaande gebruikersmodel. De basisfunctionaliteiten omvatten het configureren, initialiseren en deïnitialiseren van de kanalen, het afhandelen van simulatieberichten en het actualiseren en tonen van objecten in een grafische scène. De modelvorm is afkomstig uit een object georiënteerde ontwikkelomgeving, en wordt meestal gebruikt voor het generiek specificeren van het gedrag van systemen of gebruikersbehoeften.

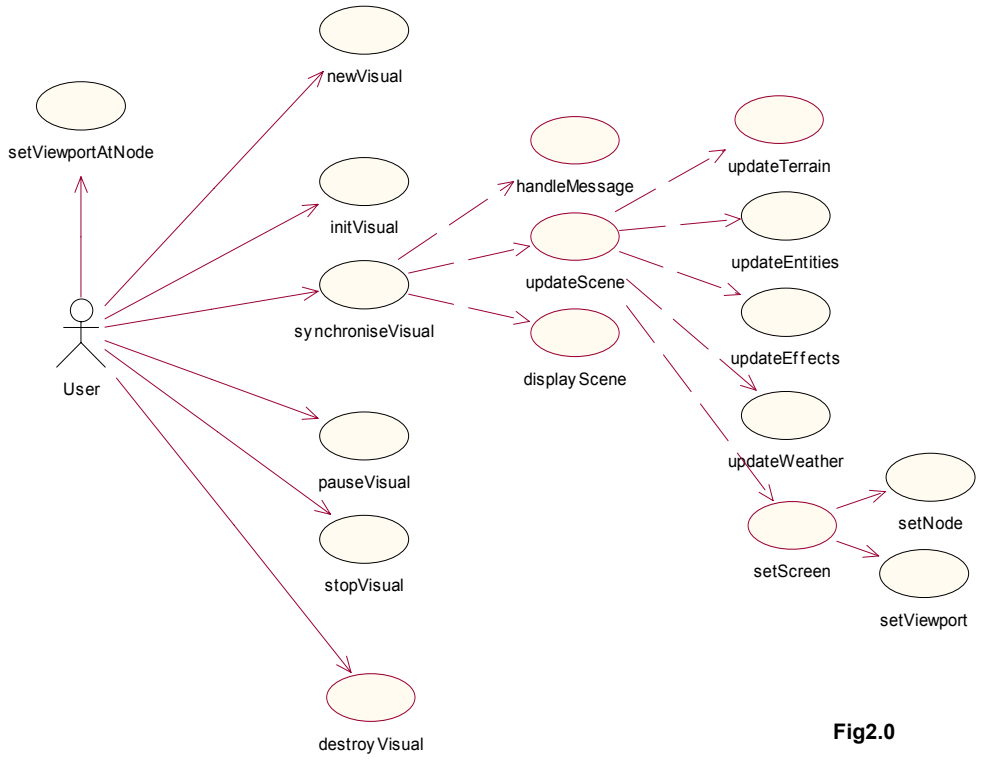


Fig2.0

4.3.2 Visual model

Het klassemodel van de applicatie programmeertaal voor grafische applicaties is hieronder weergegeven. Dit model geeft weer uit welke klassen de grafische applicatie programmeertaal bestaat. Voor het weergeven of inspecteren van een scène, worden één of meer kanalen gebruikt. Een kanaal definieert een afgebakend gebied op een beeldscherm. Op elk kanaal kan een visueel filter geplaatst worden voor het genereren van gefilterde beelden. Het TNO Fysisch Electrotechnisch Laboratorium heeft visuele filters ontwikkeld voor het genereren van infrarood- en radarbeelden. Met behulp van één of meer kanalen kan een grafische scène worden bestudeerd. Een grafische scène bevat nul, één of meer scèneobjecten en aan elk scèneobject kunnen één of meer waarnemers worden gekoppeld. Een waarnemer maakt de positie en oriëntatie van een kanaal waaraan het is gerelateerd gelijk aan de positie en oriëntatie van het scèneobject waaraan het is gekoppeld. Nadat de waarnemer is gekoppeld aan een scèneobject kan de scène worden bekeken vanuit het scèneobject of kan het scèneobject van buitenaf worden geobserveerd.

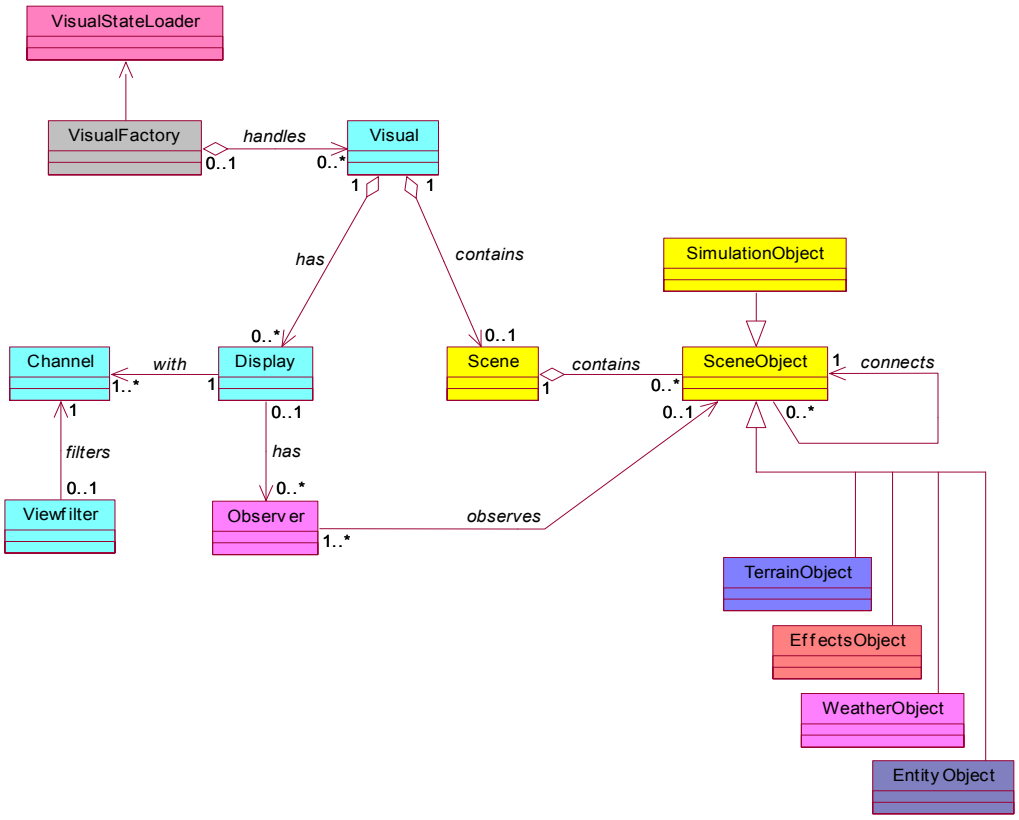


Fig2.1

4.3.3 Filesysteem model

Het klassemodel van de huidige applicatie programmeertaal voor bestanden is niet parallel over een cluster van civiele systemen. Tijdens parallellisering van de huidige applicatie programmeertaal moeten de bestaande bestandsystemen uitgebreid worden met parallelle bestandsystemen. Alle lineaire functionaliteiten blijven bestaan, maar de additionele parallelle bestandsystemen zorgen ervoor dat niet parallelle bestanden, aanwezig op een machine in een grafisch cluster, geparallelliseerd kunnen worden.

De bestandsysteem klassen zijn ontworpen voor het lezen en schrijven van specifieke lineaire bestandstypen. Instanties van de bestandsysteem klassen zijn in staat om automatisch bestandstypen te herkennen of bestandstypen in onbekende bestanden terug te vinden. De bestanden kunnen vertex-, polygoon-, positie- of bewegingsinformatie bevatten. De bestandstypen zijn op een hoger niveau een representatie van scèneobjecten, applicatie initialisaties en simulatie gebeurtenissen.

De bestandsrepresentaties zijn ingedeeld in gespecialiseerde bestandstypen die door het systeem herkend kunnen worden. Scèneobject bestandstypen, die individuele scèneobjecten kunnen representeren zijn gespecialiseerd in entiteit-, weer-, effect- en terrein bestandsklassen. Instanties van scèneobject bestandsklassen definiëren objecten die in een scène geplaatst kunnen worden. Instanties van configuratie bestandsklassen voor het configureren van grafische initialisatie componenten representeren de instellingen van een grafische applicatie. Instanties van simulatiegeschiedenis bestandstypen voor het opslaan en herhalen van simulaties representeren simulatie gebeurtenissen in de tijd.

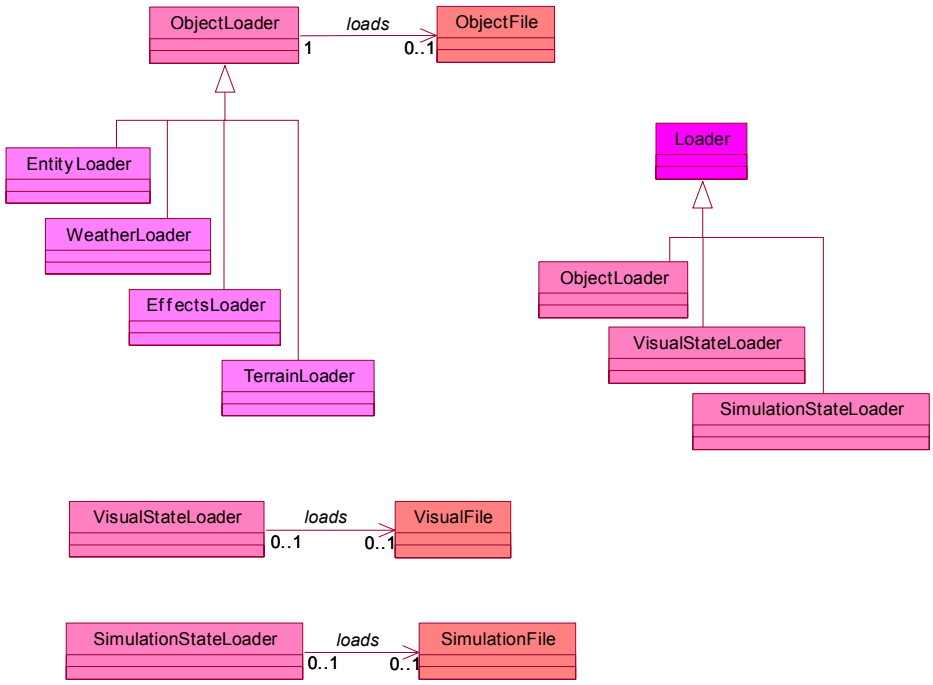


Fig2.2

4.3.4 Simulator model

Het klassemodel van de applicatie programmeertaal voor simulators is ontworpen voor het verwerken van simulatieberichten. Instanties van de klassen in het model zijn in staat om simulatieberichten te versturen en ontvangen en kunnen simulatieberichten opslaan in een simulatiebestand. Het simulatie component is een clustering van voorheen gefragmenteerde simulatiebericht klassen in de grafische applicatie. Door het introduceren van de simulatiebericht verwerking in een geconcentreerd model, wordt de nu nog maximale koppeling met het geavanceerd simulatie framework geminimaliseerd. De minimalisatie kan ook voor een snelheidswinst zorgen bij het opstarten van simulaties.

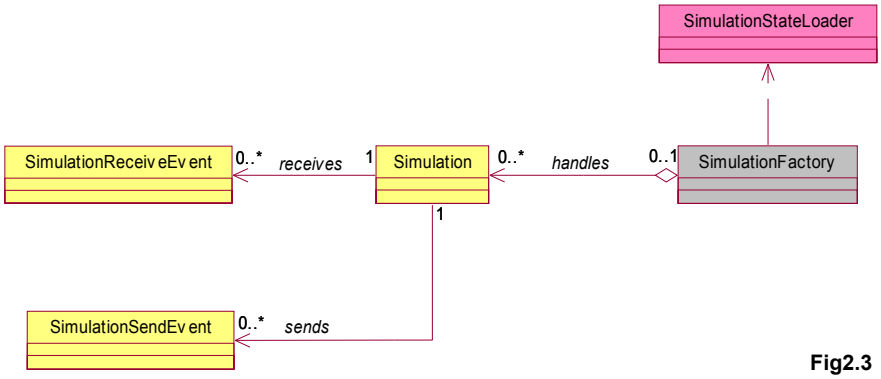


Fig2.3

4.3.5 Effecten model

Het effect klass model is ontworpen om zo realistisch doch realtime mogelijke effecten te produceren die middels een generieke applicatie programmeertaal geprogrammeerd kunnen worden. De effect klassen zijn in staat om de meest voorkomende scène effecten in de vorm van stof-, vuil-, rook-, en vuur objecten te instantiëren. Met de effect bestandsysteem klassen kunnen effecten worden ingelezen en in een scène worden geplaatst. De effect klassen geven de mogelijkheid om zo generiek mogelijk effecten te programmeren. Het onderstaand model is een generalisatie van specifieke effect klassen uit de grafische applicatie programmeertaal. Bij ontwikkeling van het gedistribueerd grafisch cluster moeten de effect klassen geparallelliseerd worden.

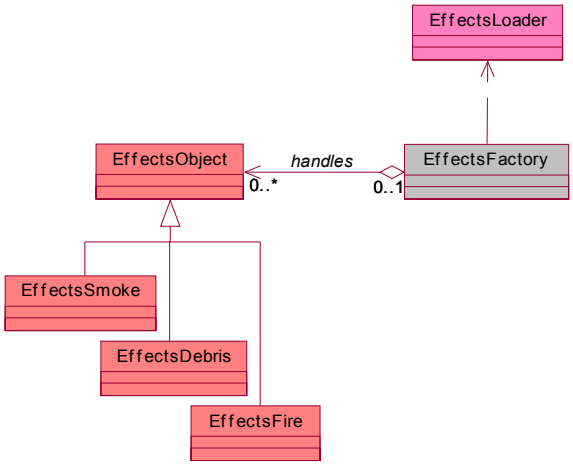


Fig2.4

4.3.6 Eniteit model

Het entiteit klassemodel bevat klassen voor het instantiëren en besturen van simulatie entiteiten. Het entiteit bestandsysteem klasse kent het simulatie entiteit type en kan de objecten waaruit het simulatie entiteit bestaat in de scène plaatsen. De entiteit typen zijn gespecialiseerd in militaire- en civiele entiteit klassen. De entiteit klasse integreert een aantal bestaande systemen die de verschillende typen entiteiten representeren. De onderstaande klassen zijn een clustering van gefragmenteerde componenten in de grafische applicatie.

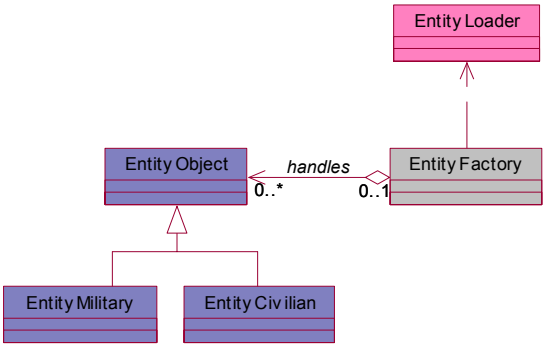


Fig2.5

4.3.7 Terrein model

Het terrein klassemodel bevat terrein klassen voor het representeren van terrein. Een terrein klasse instantie karakteriseert het terrein niet alleen op fysieke eigenschappen maar ook op terreintype. Er zijn terrein klassen voor het instantiëren en besturen van zee-, moeras- en land objecten. De terrein klassen kunnen verder worden gespecialiseerd in subterrein typen die het realiteitsgehalte van terreinsimulaties in hoge mate verbeteren. Er wordt binnen het Fysisch Electrotechnisch Laboratorium onderzoek gedaan naar de fysische eigenschappen van verschillende terreintypen en de simulatie daarvan. Het onderstaande model representeert de terreinklassen en het terrein bestandsysteem.

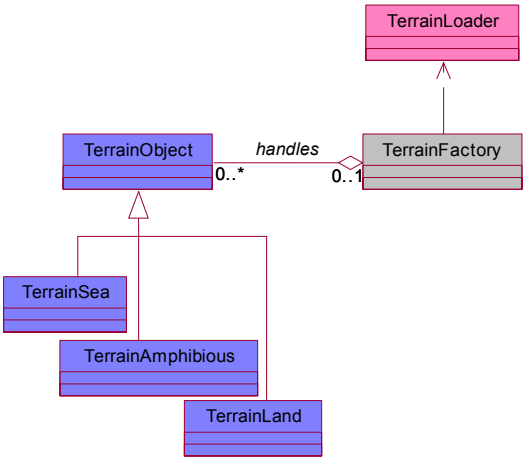


Fig2.6

4.3.8 Weer model

Het weer klassemodel is een clustering van gefragmenteerde weersimulatie functionaliteiten in de grafische applicatie programmeertaal. Door het clusteren van de gefragmenteerde informatie kunnen de klassen verder worden gespecialiseerd. Het weermodel is momenteel gespecialiseerd in klassen voor het instantiëren van mist-, wolk-, en horizon scèneobjecten. Externe klimaatveranderingen kunnen met behulp van de klassen generiek gemodelleerd en afgebeeld worden.

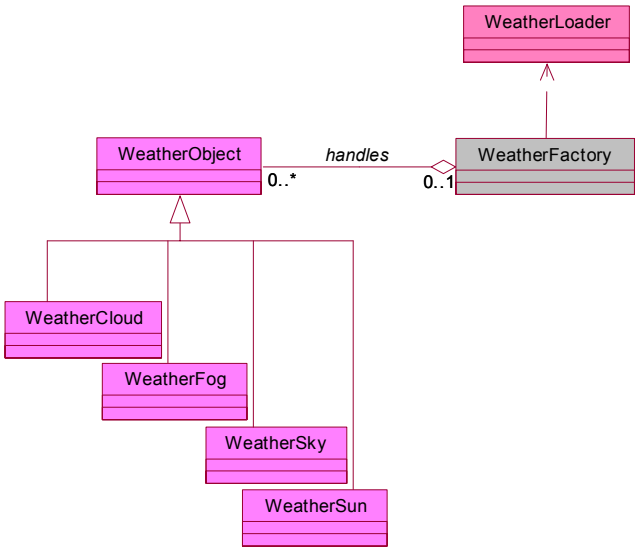


Fig2.7

4.3.9 Visual componenten

Om de componenten in de grafische applicatie programmeertaal herbruikbaar te maken is er een componenten architectuur ontworpen. De architectuur voegt klassen van de grafische applicatie programmeertaal in een bepaald component samen voor gebruik in een specifieke applicatie. Door de isolatie van klassen in gespecialiseerde componenten kunnen applicatie programmeurs probleemloos klassen hergebruiken.

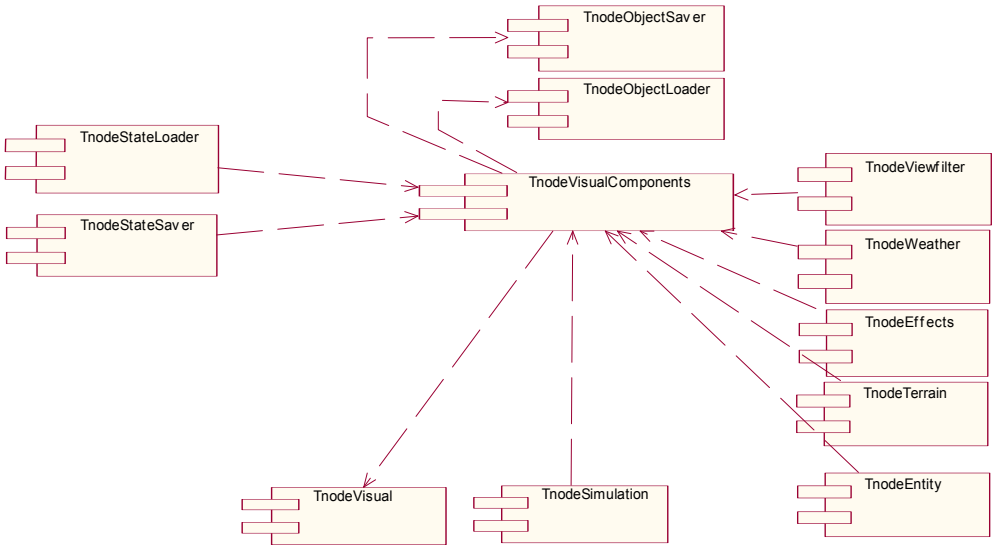


Fig2.8

5 Grafische clusters

Alvorens een gedistribueerd grafisch cluster te ontwerpen is er literatuuronderzoek gedaan naar bestaande grafische clusters, gedistribueerde grafische applicatie programmeertalen, parallelle omgevingen en grafische applicatie programmeertalen. Tijdens het onderzoek zijn de gedistribueerde grafische applicatie programmeertalen ingedeeld in verschillende typen gebaseerd op de granulariteit van de applicaties die met de talen geprogrammeerd kunnen worden.

De granulariteit van een applicatie bepaalt in hoeverre de grafische processen in de applicatie verdeeld worden over een grafisch cluster. Bij een fijne granulariteit zullen veel processen verdeeld zijn over een grafisch cluster, waardoor gespecialiseerde grafische berekeningen verdeeld zijn. Bij een grove granulariteit is er slechts een dunne laag van processen die verdeeld worden over een grafisch cluster. Een grovere granulariteit stelt gedistribueerde grafische applicaties in staat om beelden te genereren met een kortere afwijkingstijd terwijl een fijnere granulariteit gedistribueerde grafische applicaties in staat stelt om zeer gedetailleerde en gespecialiseerde beelden te genereren.

Verder onderzoek in dit hoofdstuk gaat in op de typen applicatie, de verdeling van grafische processen en gedeelde berekeningen van grafische gegevens. Verdeling van grafische processen binnen een cluster geeft applicatie programmeurs de mogelijkheid om de grafische capaciteit te gebruiken van alle machines die zich in het cluster bevinden. Bij verdeling kunnen er ook interactieve omgevingen ontstaan vanwege communicatie tussen grafische processen vanuit verschillende machines.

De gedistribueerde grafische applicatie programmeertalen worden gedetailleerd in dit hoofdstuk besproken en in het volgende hoofdstuk met elkaar vergeleken. De gedistribueerde grafische applicatie programmeertalen voor het implementeren van verschillende typen applicatie worden geanalyseerd. Een aantal implementaties van applicatie programmeertalen zijn onderzocht voor het ontwikkelen van parallelle applicaties. Ook zijn er een aantal grafische applicatie programmeertalen onderzocht die gebruikt kunnen worden voor het ontwikkelen van een grafisch cluster.

5.1 Wire opengl cluster

Het WireGL [20] grafisch cluster project was oorspronkelijk afkomstig van Silicon Graphics. Het project is tot stilstand gekomen en voortgezet onder de naam Chromium als project waarbij de broncode vrijelijk beschikbaar is. De applicatie programmeertaal is een parallellisering van de OpenGL visualisatie omgeving op functieniveau [20] en biedt OpenGL functionaliteit aan iedere machine in een cluster [21]. Op ontwerpniveau is er een geparallelliseerde OpenGL virtuele grafische machine gespecificeerd die de fijn gegranuleerde grafische processen over een cluster van machines kan verdelen.

Er is aangetoond dat, gebruik makend van zestien computers voor de berekening van grafische gegevens en zestien computers voor het visualiseren en samenvoegen van scèneobjecten in één scène, een WireGL grafisch cluster een beeld met meer dan zeventig miljoen driehoeken per seconde kan verwerken [20]. Iedere computer voor het berekenen van grafische gegevens is gespecialiseerd in gegevensverwerking, terwijl de overige computers zich specialiseren in het weergeven en samenvoegen van beelden. Er is bewezen dat een WireGL grafisch cluster scaleert over de gehele reikwijdte van beschikbare machines [20].

Door de detaillering van procesuitvoering in WireGL kunnen we spreken van een gedistribueerd grafisch cluster met zeer fijne granulariteit. De WireGL applicatie kan vanwege de fijne granulariteit de grafische processen op verschillende niveaus over de computers in een grafisch cluster verdelen. Op het hoogste procesniveau is het tile based reassembly proces dat met gespecialiseerde lightning [20] hardware of met behulp van een algemeen cluster communicatie protocol [20] beelden samenvoegt tot één beeld. Omdat tijdens het tile based reassembly proces apart berekende beelden worden samengevoegd tot één beeld is er uiteindelijk een veel hogere detaillering van grafische beelden mogelijk.

Door de verdeling van processen en het tile based reassembly proces ontstaat een gedistribueerd grafische hoofdproces dat de grafische primitieven, de processen voor manipulatie van deze primitieven en de speciale processen van de virtuele grafische machine verdeelt over een grafisch cluster [21]. De verdeelde processen worden gesynchroniseerd op de verschillende niveaus in het grafisch hoofdproces. De virtuele grafische machine voert slechts processen uit die een verandering in de grafische primitieven veroorzaken. Door deze verwerking van primitieven wordt er minder bandbreedte van het grafisch cluster gebruikt [20].

WireGL applicaties zijn vooral bedoeld voor zeer grote gegevensverzamelingen die over een cluster verdeeld zijn. Ook worden WireGL applicaties vaak gebruikt voor het specialiseren van grafische rekenprocessen en het toekennen van zulke processen aan gespecialiseerde procesgroepen in een WireGL cluster. WireGL kan gebruikt worden voor applicaties waarbij een zeer hoge resolutie gewenst is. Veel voorkomende grafische applicaties met dergelijke eisen zijn voor moleculaire visualisatie of applicaties voor het visualiseren van weersinformatie.

5.2 Silicon grafisch cluster

Het SGI project voor gedistribueerde grafische clusters omvat een verzameling van applicatie programmeertalen en configuratie systemen voor clusters van civiele systemen. De applicatie programmeertalen en configuratie systemen bestaan uit de OpenGL Performer applicatie programmeertaal voor grafische applicaties, SGI ImageSync voor het synchroniseren van beelden, SGI DataSync voor het synchroniseren van gegevens en SGI SynaptIQ voor het configureren van grafische clusters.

De OpenGL performer is een applicatie programmeertaal voor het ontwikkelen van grafische applicaties die in een realtime omgeving functioneren [22]. De SGI ImageSync is een applicatie programmeertaal voor het programmeren van kanaal synchronisatie algoritmen. Het SGI DataSync is een applicatie voor gegevenssynchronisatie. Het SGI SynaptIQ wordt gebruikt voor het gecentraliseerd configureren van grafische clusters.

De SGI ImageSync applicatie programmeertaal biedt gespecialiseerde video en frame buffer synchronisatie, gebruik makend van grafische kaarten van het merk NVIDIA [22]. De applicatie programmeertaal geeft applicatie ontwikkelaars de mogelijkheid beeldsynchronisatie op hardware niveau te bepalen. Tijdens de ontwikkeling van het grafisch cluster voor het TNO Fysisch Electrotechnisch Laboratorium zal er worden onderzocht welke synchronisatie algoritmen in de hardware voorkomen en hoe gespecialiseerde hardware kan leiden tot een verbeterde synchronisatie.

De SGI DataSync applicatie verzorgt, via een op MPI gebaseerde applicatie programmeertaal, gegevenssynchronisatie en distributie van gegevens over SGI grafische clusters [22]. Er is aangetoond dat het gebruik van op MPI gebaseerde applicatie programmeertalen de distributie en synchronisatie van gegevens voor grafische applicaties gemakkelijker maakt [3,22,28]. In de latere hoofdstukken wordt MPI nader besproken en wordt er een implementatie gepresenteerd die geschikt is voor de ontwikkeling van het grafisch cluster.

Het SGI SynaptIQ configuratie systeem is een applicatie voor het administreren van clusters onder Linux. Tijdens de ontwikkeling van het grafisch cluster is een configuratie management systeem ontwikkeld die gebruikers in staat stelt om op eenvoudige wijze belangrijke componenten van het grafisch cluster te installeren. Het configuratie management systeem kan een goed beginpunt zijn voor de ontwikkeling van een gecentraliseerde configuratie manager.

SGI grafische clusters zijn nuttig voor het visualiseren van generieke databases, zoals gesimuleerde omgevingen of interactieve virtuele werelden. Ook kunnen SGI grafische clusters bruikbaar zijn in de simulatiewereld, waar er vaak meer dan één kanaal gebruikt wordt voor het visualiseren van gesimuleerde omgevingen. Omdat de SGI graphics clusters gebruik maken van civiele systemen zouden ze ook uiterst bruikbaar kunnen zijn voor visualisatie van bedrijfsinformatie afkomstig uit geavanceerde databases.

5.3 Mantis grafisch cluster

Mantis is een client/server image generator, gespecialiseerd in militaire simulaties. Mantis is ontworpen voor uitzonderlijke synchronisatie snelheden en biedt een geheel nieuwe manier van ontwerpen, ontwikkelen en installeren van geavanceerde en realistische visuele simulaties [40]. De Mantis architectuur biedt speciale effecten zoals sensoren, weer, licht en explosies die als extensie in de Mantis architectuur zijn geïntegreerd [40]. Verder biedt de Mantis architectuur multihead data, frame synchronisatie en dynamisch geladen terrein [40].

De client/server architectuur van Mantis is volledig scaleerbaar, waardoor de verwerking van simulatiegegevens gescheiden kan worden van de bijbehorende grafische berekeningen [40]. Door de scheiding ontstaat er een verbeterde precisie en prestatie van simulaties, die in een andere omgeving een veel lagere prestatie zouden bereiken. Een Mantis gebruiker kan zich door de client/server architectuur volledig storten op het ontwikkelen van de simulatie. Mantis is, behalve client/server, uitbreidbaar met generieke Mantis extensies aangeboden door applicatie programmeertalen [40].

Een server van Mantis is verantwoordelijk voor generatie van de grafische gegevens voor de verschillende grafische kanalen. Een client kan de server gebruiken om drie dimensionale grafische beelden te vertonen op één of meer Mantis kanalen [40]. Ieder Mantis kanaal kan apart geconfigureerd worden voor een optimale prestatie van de grafische applicaties [40]. Ook kunnen communicatie poorten, bestandscaches en -extensies ingesteld worden [40].

Een Mantis client biedt grafische controle informatie aan één of meer Mantis servers. Gebruikers applicaties dienen enkel te communiceren met de Mantis client om grafische beelden te genereren. De Mantis clients bieden verder de mogelijkheid om kanalen en oogpositionering te configureren [40].

Mantis biedt een grote hoeveelheid extensies. De extensies van Mantis bieden effecten zoals rook, explosies, stof en vuur [40]. Voor het simuleren van weersomstandigheden biedt Mantis controle over de zonpositie, mist, wolken, regen, sneeuw en hagel [40]. De extensies zijn, net zoals alle andere Mantis componenten modulair en uitbreidbaar. Het type extensies maakt Mantis zeer geschikt voor militaire simulatie applicaties.

5.4 Aechelon grafisch cluster

Het grafisch cluster van Aechelon Technologies, of het CNova grafisch cluster, is de keuze van SGI voor het implementeren van efficiënte en scaleerbare simulaties in een realtime simulatie omgeving [41]. Het CNova grafisch cluster en applicatie programmeertaal is ontwikkeld door Aechelon Technologies, een bedrijf dat zich heeft gespecialiseerd in het ontwikkelen van applicaties voor de Amerikaanse defensiemarkt.

Omdat de kosten voor het ontwikkelen van defensie systemen gebaseerd op civiele technologiëen zeer hoog liggen [41], vanwege de fragmentatie van systeem onderdelen die onderzocht moeten worden, is het gebruik van een CNova systeem een gunstig alternatief. Bovendien zijn de kosten voor het ontwikkelen van additionele functionaliteiten niet van toepassing omdat deze zijn gebaseerd op de behoeften van de defensie markt [41].

De applicatie programmeertaal voor CNova is zo ontwikkeld dat de ontwikkelde applicatie vanzelf scaleert, afhankelijk van de rekencapaciteit van iedere machine in het grafisch cluster. De scalering vindt plaats door dynamische bepaling van object complexiteit [41]. Bij machines die de grafische capaciteit van het CNova systeem minder goed kunnen benutten, scaleert het CNova de grafische modellen totdat de doelfrequentie van de applicatie wordt behaald.

De CNova omgeving biedt gebruikers de mogelijkheid om stukken terrein dynamisch in te lezen en specifieke terrein gedeelten te kiezen voor gerelateerde kanalen. Het gebruik van coördinatenstelsels [41] met dubbele precisie biedt maximale nauwkeurigheid bij het uitvoeren van simulaties. De grafische applicatie programmeertaal zoals ontwikkeld tijdens dit onderzoek gebruikt, net zoals CNova, functies en coördinatenstelsels met dubbele precisie.

Ondersteuning voor een groot aantal bewegende objecten staat realistische scenario's toe. Object details zijn verder verfijnd door de toevoeging van effecten voor het actief belichten van militaire en civiele entiteiten, het weergeven van landingstelsels en het genereren van rookpluimen. Verder wordt er bij het CNova systeem een uitputtende hoeveelheid gedetailleerde objecten en terreinen geleverd bij het CNova systeem die gebruikt kunnen worden voor het simuleren van een oneindig aantal situaties.

Behalve een detaillering van individuele objecten zijn ook de speciale effecten gedetailleerd. De effecten bibliotheek omvat effecten voor het weergeven van de weersituatie, tijdsituatie, wapens en explosies. Accurate fysieke modellen van de zee [41] stellen gebruikers in staat om met CNova situaties op zee te trainen. Ook bevat CNova een bibliotheek voor het realiseren van sensor scènes [41]. Een sensor scène biedt gebruikers de mogelijkheid om het filtereffect weer te geven van bijvoorbeeld hitte sensoren.

5.5 Dive omgeving

Het Dive systeem [23] is een scaleerbare architectuur voor gedistribueerde virtuele omgevingen. Dive bevat abstracte databases van wereld elementen, of entiteiten, die middels processen benaderbaar zijn. De databases van wereld elementen zijn gedistribueerd over een netwerk of het internet. De wereld elementen zijn persistent in de database en worden pas gebruikt als een proces ze nodig heeft.

Alle objecten in een Dive omgeving kunnen worden weergegeven in een Dive klasse hiërarchie. De dive klasse hiërarchie bestaat uit entiteiten met als hoofd entiteiten de node, actor, light en view [23]. De node is verder gespecialiseerd in world, lod, switch, billboard en object. De light heeft point light, spot light en directional light als specialisatie [23]. De view bestaat uit specialisaties voor primitieven zoals boxes, spheres en multi poly's [23]. De klasse hiërarchie is sinds het ontstaan van de Dive omgeving niet meer veranderd. Door dezelfde klasse hiërarchie te gebruiken door alle versies van de omgeving heen kunnen oudere Dive applicaties communiceren met nieuwere versies.

De grafische objecten of entiteiten van de Dive omgeving zijn persistent op het netwerk aanwezig. Op het moment dat een proces de dive omgeving bezoekt, zal er een replica van het gewenste gedeelte van de database gemaakt worden en zullen de entiteiten die daarbij nodig zijn in het geheugen worden geplaatst [23]. We spreken hierbij van een actieve partiële replica van gegevens [23]. Het voordeel van het repliceren van database objecten is dat de toestand van de objecten niet centraal veranderd hoeft te worden.

Voor realtime simulaties is Dive echter niet geschikt. Globale metingen hebben aangetoond dat communicatie afwijkingstijden kunnen oplopen tot 25 ms (milliseconden) [23]. Bij realtime simulaties, waar de eisen zeer hoog liggen voor wat betreft realtime prestaties, zijn afwijkingstijden van 1-5 µs (microseconden) de toegestane limiet. Dive is wel geschikt voor generieke interactieve applicaties waarbij grote groepen met elkaar communiceren en waar scène gegevens via centrale databases benaderbaar zijn. Bij Dive toepassingen kunnen we denken aan systemen voor interactieve bedrijfscommunicatie of database systemen.

5.6 Studierstube omgeving

De studierstube omgeving is een virtual reality applicatie die gebruikers in staat stelt met meerdere machines in groepsverband samen te werken. De applicatie is zo ontworpen dat applicaties en gegevens gerepliceerd worden op het moment dat een gebruiker ze nodig heeft. Vanwege het replicatie proces worden gegevens transparant beschikbaar gesteld op de verschillende machines [4]. Omdat applicaties ook een integraal onderdeel zijn van de scènegraaf [4] kunnen gebruikers die een applicatie niet hebben tijdens een bezoek aan de studierstube de applicatie direct gaan gebruiken als ze de bijbehorende gegevens opvragen.

Vanuit technisch oogpunt is het nuttig om een gerepliceerde scènegraaf per machine in een grafisch cluster te plaatsen omdat dit inconsistenties tegengaat. Een scènegraaf is een graaf van scène elementen en representeert de scène van een simulatie. De scènegraaf kan in gerepliceerde vorm verwijzen naar parallel gedistribueerde gegevens, waardoor de maximale opslagcapaciteit van machines in een cluster wordt gebruikt. We spreken dan van een scènegraaf die de scène informatie opvraagt met behulp van een verwijzing naar de scène gegevens; er hoeft dus geen informatie te worden gerepliceerd.

De studierstube is een omgeving die uiterst geschikt bevonden wordt voor interactie met andere studierstube omgevingen. Bij de interactie hoeven de andere omgevingen niet dezelfde applicaties te bevatten, omdat die in de omgevingen die de applicatie niet bevatten worden gerepliceerd. Bij studierstube toepassingen spreekt men over hybride toepassingen waarbij verschillende systeemtypen zich aanmelden en afmelden bij de studierstube omgeving om een interactie aan te gaan met andere gebruikers in een applicatie context [4].

Hybride toepassingen komen zijn onafhankelijk van netwerk architecturen, besturingssystemen of hardware. Studierstube systemen kunnen geheel hybride zijn op besturingssysteem-, applicatie- en hardware niveaus [4]. Bij geheel hybride toepassingen spreekt men over transparante toepassingen in een heterogene omgeving. Voor militaire simulaties zijn teveel systeemsoorten niet geschikt omdat de afstemming van die systemen op elkaar de precisie van de simulaties beïnvloedt. Wel kunnen militaire simulaties op simulator niveau met elkaar gekoppeld worden.

Het concept van de gerepliceerde scènegraaf kan gebruikt worden bij militaire simulaties. Tijdens ontwikkeling van het grafisch cluster is er een gerepliceerde scènegraaf generator ontwikkeld voor de representatie van grafische objecten. Door replicatie van de scènegraaf, waarin scènegraaf elementen naar dezelfde objecten kunnen verwijzen, kan er rekencapaciteit beschikbaar worden gesteld aan applicatie gerelateerde berekeningen. De beschikbare rekencapaciteit ontstaat vanwege eliminatie van rekencapaciteit die nodig was geweest voor management van een gedistribueerde scènegraaf.

5.7 Message passing interface

De Message Passing Interface, of MPI, is een specificatie van een parallelle communicatie omgeving en is ontwikkeld in twee fasen door een open forum van parallelle systeemanalisten en applicatie ontwikkelaars [28]. De specificatie kent inmiddels vele implementaties, die ieder een stuk of het geheel van de Message Passing Interface realiseren. De Message Passing Interface wordt veel gebruikt voor het oplossen van problemen met parallelle systemen [28].

Omdat de Message Passing Interface wordt bepaald door de specificaties en niet de programmatuur, laat het de gebruikers ervan vrij in het kiezen van een eigen implementatie. De nieuwste specificatie is de MPI2 specificatie en is initieel ontwikkeld in 1997; implementaties van de standaard worden veel gebruikt voor het parallel opslaan en opvragen van gegevens en voor berichtenuitwisseling.

Tijdens ontwikkeling van het grafisch cluster voor het TNO Fysisch Electrotechnisch Laboratorium is er onderzoek gedaan naar parallelle representatie van landschapgegevens. De parallelle representatie kan gebruikt worden om middels de gerepliceerde scènegraaf landschapinformatie aan te wijzen zonder dat de gerelateerde gegevens gerepliceerd hoeven te worden. De parallelle replicatie van gegevens zorgt tevens voor een aanzienlijke versnelling van de invoer en uitvoer van deze gegevens [28].

Toekomstige versies van de specificatie ondersteunen ook procescontrole en spawning [28], spawning is een vorm van procesmanagement waarbij processen tijdens het uitvoeren van een programma verdeeld worden over een cluster. Het toevoegen van spawning en procescontrole is voordelig voor applicatie ontwikkelaars omdat MPI programmatuur met dynamische procescontrole voorheen in combinatie met andere programmeertalen voor procescontrole geschiedde. De MPI programmeertaal werd dan vaak in combinatie met programmeertalen zoals PVM of OpenMP [5] voor procescontrole gebruikt. Door het weglaten van andere programmeertalen voor procescontrole is er geen additionele configuratie management nodig om de applicatie programmeertalen en de daaruit voortvloeiende applicaties op elkaar af te stemmen.

5.8 Opengl programmeertaal

De OpenGL applicatie programmeertaal is de meest gebruikte programmeertaal voor grafische applicaties. OpenGL bestaat uit een specificatie van een virtual graphics state machine [38] en een aantal implementaties op verschillende besturingssystemen. De implementaties stellen gebruikers in staat om componenten van de state machine te besturen en vertex lists of andere primitieven te initiëren en bewerken met die componenten.

Vanwege het state transition model van de OpenGL virtual graphics state machine is er gekozen voor de programmeertaal C als applicatie programmeertaal voor controle van de graphics states [38]. De programmeertaal C is voor het grootste deel functioneel georiënteerd; waarbij het graphics states gemakkelijk aan of uit kan zetten, zonder eerst objecten te instantiëren of andere niet functionele operaties uit te voeren. De functies in C zijn vanwege hun functionele oriëntatie daarom één op één te implementeren vanuit de OpenGL virtual machine specificatie.

De OpenGL applicatie programmeertaal wordt veelal op lager niveau gebruikt voor het ontwikkelen van grafische applicatie programmeertalen zoals Performer of Mantis. De ontwikkelde talen maken gebruik van de OpenGL graphics machine die functies van de OpenGL virtual machine realiseert op een bepaald besturingssysteem. Ook worden OpenGL virtual machine functies op een lager abstractie niveau in de graphics hardware van een machine geïmplementeerd, zodat er optimaal gebruik gemaakt kan worden van de grafische versnellingsmogelijkheden die deze hardware biedt.

Toekomstige versies van de OpenGL machine in de vorm van OpenGL2.0 bieden uitbreidingen aan de machine in de vorm van onder andere dynamisch controleerbare visualisatie pijplijnen [38]. De uitbreidingen aan de OpenGL machine sluiten direct aan op oudere specificaties en bieden meer flexibiliteit. Tijdens uitbreiding van de OpenGL machine is er onderzoek gedaan naar virtuele machine elementen die statisch gerepresenteerd waren, zoals de visualisatie pijplijn. Met name functionaliteiten die een hoge mate van controle op beeldpunt niveau bieden zijn in oudere OpenGL machines niet aanwezig. Verder is er in de nieuwe OpenGL versie een taal ontwikkeld die specifiek bedoeld is voor controle van grafische elementen of processen binnen de OpenGL machine [38].

5.9 Performer programmeertaal

Performer [26] is een applicatie programmeertaal voor zeer uiteenlopende grafische applicaties. De Performer applicatie programmeertaal heeft OpenGL als basis en is ontwikkeld voor een reeks van generieke computer platforms waaronder Unix, Irix, Windows en Linux. Performer biedt een zeer breed en modulaire applicatie programmeertaal voor het snel ontwikkelen van applicaties. De applicatie programmeertaal is gerepresenteerd door een groep bibliotheekfuncties,

Performer bestaat uit de applicatie bibliotheken libpr, libpf, libpfdu, libpfutil, libpfui, libpfv, libpfmpk, libpfdb voor respectievelijk het renderen van beelden, weergeven en managen van visuele objecten, genereren van geometrische primitieven, construeren van interactieve modulaire performer applicaties, ontwikkelen van gebruikers applicaties, ontwikkelen van grafische componenten, platform onafhankelijke configuratie management en het dynamisch inladen van database objecten [26].

OpenGL Performer is generiek toepasbaar voor het ontwikkelen van verschillende soorten omgevingen. Het is een compleet database verwerkings- en rendering systeem voor realtime interactieve virtuele omgevingen [26]. De OpenGL Performer is verder bruikbaar in combinatie met andere programmeer bibliotheken of applicatie programmeertalen [26]. Zo kunnen gespecialiseerde berekeningen door gespecialiseerde systemen worden opgevangen. Een voorbeeld van een gespecialiseerd systeem is een systeem voor de generatie van sensorgegevens.

Het combineren van OpenGL Performer met andere programmeertalen op lager niveau maakt het mogelijk om systemen te ontwikkelen die OpenGL Performer hebben geïntegreerd in hun omgeving. De integratie mogelijkheden stelt gebruikers in staat om oudere systemen met minder grafische mogelijkheden uit te breiden met OpenGL Performer zonder de applicatie te hoeven veranderen. Ook kunnen er uitgebreide applicatie programmeertalen worden ontwikkeld voor simulatie effecten zoals weersimulaties en gevechtssimulaties.

Tijdens ontwikkeling van het grafisch cluster voor het TNO Fysisch Electrotechnisch Laboratorium is er gewerkt aan een gedistribueerde simulator voor generieke effecten zoals regen, mist, sneeuw en explosies, gebruik makend van elementaire grafische vormen. De elementaire grafische vormen, of geodes, zijn ontwikkeld met de Performer bibliotheek voor geometrische primitieven. De simulator voor generieke effecten wordt gebruikt bij een TNO Fysisch Electrotechnisch Laboratorium demonstratie applicatie om de flexibiliteit van het grafisch cluster aan te tonen.

6 Analyse clusters

Uit de voorgaande paragrafen is op te merken dat de meeste grafische applicaties, in clustervorm of lineair, een zeer fijne of zeer grove granulariteit hebben. De fijne granulariteit is gericht op applicaties die puur ontwikkeld zijn voor het visualiseren van grote gegevensverzamelingen of het uitvoeren van grafische processen waarbij gedetailleerde grafische berekeningen verdeeld moeten worden over een cluster (wiregl, chromium). De grove granulariteit wordt gebruikt bij applicaties die puur gericht zijn op het uitvoeren van grafische processen waarbij interactie of snelle grafische verwerking is vereist (studierstube, dive, performer, mantis, cnova).

In alle gevallen van gedistribueerde grafische applicaties wordt er een visualisatie hoofdproces geheel of gedeeltelijk geparallelliseerd, waarbij de elementen van het hoofdproces gerepliceerd of gefragmenteerd worden verdeeld over een grafisch cluster. Op het hoogste niveau worden verbindingen met grafische elementen gekoppeld aan externe of interne interactie entiteiten, respectievelijk controller apparaten (muis, stuurknuppel) of interne controllers (simulators, datastroom machines). De voorheen genoemde koppeling is de koppeling tussen het grafisch cluster en de buitenwereld, en kan een significant effect hebben op de reikwijdte van de controle over en interactie met een applicatie.

Gedistribueerde grafische applicaties en de ontwikkeling daarvan kunnen, globaal gezien, ingedeeld worden in drie groepen of stromingen. De eerste stroming omvat de zeer fijn gegranuleerde grafische clusters waarbij applicatie processen in hoge mate en reeds op een laag niveau worden verdeeld over de machines in een grafisch cluster (wiregl, chromium). De tweede stroming omvat de interactieve grafische applicaties voor interactie tussen gebruikers in een heterogeen netwerk zoals het internet of grote netwerken met verschillende besturings systemen (dive, studierstube). De derde stroming omvat grof gegranuleerde visualisatie systemen die zijn ontwikkeld voor kleinere netwerken en die gegarandeerde realtime prestaties leveren (mantis, cnova, performer, dive).

Voor de ontwikkeling van het grafisch cluster voor het TNO Fysisch Electrotechnisch Laboratorium hebben we te maken met de derde stroming. Deze stroming omvat de grof gegranuleerde grafische clusters voor gespecialiseerde visualisatie netwerken die zich meestal in een klein afgesloten en beveiligd netwerk bevinden. Om de realtime gedistribueerde grafische clusters en applicatie programmeertalen te kunnen ontwikkelen moeten we ons concentreren op deze groep en specifieke applicatie architecturen, of patronen, die daarin terug te vinden zijn.

Om een beter overzicht en scherpere controle te behouden over het grafisch cluster en de scènegraaf elementen worden scènegraaf elementen vaak geheel of gedeeltelijk gerepliceerd (dive, studierstube). Bij de voorgaande scènegraaf replicatie methoden wordt de scènegraaf op alle machines gerepliceerd of elementen van de scènegraaf pas op een machine gerepliceerd wanneer die gewenst zijn. De scenegraaf elementen kunnen daarbij verwijzen naar lokale of parallelle scèneobjecten.

Voor de ontwikkeling van een parallel grafisch cluster is het van belang te letten op de vorm van replicatie van de scènegraaf. In een realtime cluster zijn geheel gerepliceerde scènegraaf elementen effectiever omdat alle scènegraaf elementen direct beschikbaar zijn, waardoor er geen vertraging ontstaat voor het opvragen van scènegraaf elementen. Verder is de vorm van de scènegraaf van belang; als de scènegraaf bestaat uit verwijzingen naar gedistribueerde bestanden dan zal de bandbreedte en opslagcapaciteit van het grafisch cluster verdeeld worden over de machines in het cluster. Bij niet parallelle applicaties kan de scènegraaf verwijzen naar gerepliceerde lokale bestanden.

Behalve replicatie van de scènegraaf en scènegraaf elementen, is het voor gedistribueerde grafische clusters van belang om de distributie van gegevens zo dun mogelijk te houden, met alleen maar die gegevensstromen die voor controle van een applicatie werkelijk van belang zijn. De synchronisatie van gegevens is van absoluut belang voor het in realtime opereren van een grafisch cluster. Functioneel zou men kunnen denken aan gegevensstroom controle door een implementatie van een abstract gedefinieerde datastroom machine. Tijdens dit onderzoek is er een datastroom machine gedefiniëerd en geïmplementeerd voor de controle van gegevensstromen op een hoger abstractieniveau.

Voor de ontwikkeling van een gedistribueerd realtime cluster is het ook van belang controle te hebben over de zichtkanalen. Een zichtkanaal dient op een zodanige wijze gecontroleerd te worden dat men voor alle machines kan bepalen welke kanaalfuncties worden uitgevoerd. De kanaal functies omvatten, maar zijn niet gelimiteerd tot, functies voor het bepalen van de beeldpunt grootte, kanaal breedte, kanaal hoogte en kanaal afwijking ten opzichte van andere kanalen. Er is tijdens het onderzoek voor het TNO Fysisch Electrotechnisch Laboratorium een parallel kanaal ontwikkeld dat gebruikers in staat stelt om kanalen in een cluster te definiëren, te plaatsen op verschillende machines en te controleren vanuit een lokale machine.

Voor de ontwikkeling van het gedistribueerd grafisch cluster is er een geheel gerepliceerde scènegraaf ontworpen die bestaat uit verwijzingen naar parallel en lokaal opgeslagen bestanden. Een gedistribueerd realtime grafisch cluster met een geheel gerepliceerde scènegraaf zal de minste problemen veroorzaken voor scènegraaf management en realtime prestaties. Er is tijdens de ontwikkeling geconcentreerd op met name synchronisatie, communicatie en parallellisatie van gegevens, omdat er is gebleken dat dit het meeste invloed heeft op de realtime prestatie van een grafische applicatie.

Om de grafische applicatie te besturen op kanaal niveau moet er vanuit een centraal punt controle zijn over alle kanalen van grafische machines die zich bevinden in het grafisch cluster. Er is voor het TNO Fysisch Electrotechnisch Laboratorium een systeem ontwikkeld voor het op eenvoudige wijze aansturen van een kanaal. Verder zijn er koppelingsfuncties ontwikkeld om een kanaal vanuit verschillende oogpunten naar scèneobjecten in een scènegraaf te laten kijken.

7 Onderzoeks gebieden

Alvorens een applicatie programmeertaal te ontwerpen voor gedistribueerde grafische applicaties is er onderzoek gedaan naar decompositie op functioneel niveau (functionele decompositie) en op gegevens niveau (domein decompositie). Effectieve decompositie op hoog abstractieniveau leidt tot een scaleerbare en herbruikbare architectuur voor het gedistribueerd grafisch cluster.

Op functioneel niveau is er bepaald welke reikwijdte functies hebben die zich over het grafisch cluster verdelen. Omdat het gedistribueerd grafisch cluster dynamisch is zouden gedistribueerde functies moeten scaleren over de gehele reikwijdte van de machines in het grafisch cluster. De aanroep van een functie kan lokaal op één machine of op alle machines plaatsvinden. Het uitvoeren van een functie op één machine, kan een effect op een andere machine teweeg brengen. Een functie of functiedeel wordt lokaal of op alle machines gescaleerd. De methoden van aanroep, uitvoer en scalering hebben een effect op respectievelijk de bereikbaarheid, uitvoerbaarheid en reikwijdte van applicatie processen. De reikwijdte, bereikbaarheid en uitvoerbaarheid van de functies bepaalt de programmeerbaarheid van de scènegraaf elementen.

Op gegevens niveau is er bepaald welke gegevens verdeeld en gesynchroniseerd moeten worden over het netwerk. De verdeling van gegevens heeft een effect op de uiteindelijke afwijkingstijden in het grafisch cluster. De verdelingsmethode is zo ontworpen dat er dynamisch wordt bepaald hoeveel gegevens er verstuurd moeten worden afhankelijk van de hoeveelheid machines en gegevens die er in het grafisch cluster aanwezig zijn. De schaal van de applicatie wordt daarbij in twee dimensies bepaald. De mate van schaalbepaling (of scaleerbaarheid) en bestuurbaarheid bepaalt de flexibiliteit en transparantie van het grafisch cluster.

Tijdens ontwikkeling van het parallel grafisch cluster zijn er modellen gemaakt van de ontwikkelde applicatie programmeertalen. De modellen vormen een belangrijk onderdeel van de ontwikkelde applicatie programmeertalen en zijn bedoeld om broncode van de applicatie programmeertalen te beheren en gedeeltelijk te genereren. De modellen geven weer wat de relatie is tussen de methoden of functies en bijbehorende gegevens van het gedistribueerd grafisch cluster dat is ontwikkeld voor het TNO Fysisch Electrotechnisch Laboratorium.

Dit hoofdstuk bespreekt in detail de applicatie ontwikkelingen op het gebied van beeld-, frame-, en gegevenssynchronisatie. Ook wordt er een applicatie implementatie architectuur besproken en zullen de bibliotheken waarvan de applicatie afhankelijk is nader worden belicht. Er wordt in dit hoofdstuk tevens een specificatie gegeven van een communicatie machine die in het gedistribueerd grafisch cluster gebruikt wordt voor de ontwikkeling van de gedistribueerde grafische applicatie programmeertaal.

7.1 Applicatie architectuur

Voor een gedistribueerd grafisch cluster zijn er meerdere gesynchroniseerde processen nodig die samen een grafische applicatie besturen. Verder moet er vanuit een willekeurig punt in het grafisch cluster een controle mogelijkheid zijn om processen aan te sturen. De verdeling van processen en gegevensstromen over een cluster van machines, waarbij processen en gegevens gerepliceerd en verschillend mogen zijn, noemt met Multiple Instruction Multiple Data. Deze architectuurvorm is ingedeeld volgens de indeling voor parallelle systemen van Flynn [5] en beschouwt parallelle systemen op zeer hoog abstractie niveau.

Multiple Instruction Multiple Data wordt gerealiseerd door een datacommunicatie machine die gebruik maakt van een Message Passing Interface applicatie bibliotheek voor gedistribueerde berichtenuitwisseling. De datacommunicatie machine is ontworpen voor het gedistribueerd grafisch cluster en een implementatie van deze machine is gebruikt bij de ontwikkeling van de grafische applicatie programmeertaal.

Voor de ontwikkeling van een gedistribueerde grafische applicatie wordt gerepliceerde broncode gebruikt. Dit betekent dat bij gebruik van de bibliotheek, op elke computer in het grafisch cluster hetzelfde programma aanwezig zal zijn, maar dat tijdens uitvoer van dit programma de processen zich op verschillende computers zullen bevinden. Door een dergelijke benadering wordt de beheersbaarheid van het parallelle programma verbeterd en is multiprogrammering van een parallel programma vanuit lineair ongefragmenteerde programmacode mogelijk.

Hieronder is een architectuurtekening gegeven van de applicatie modules die zijn ontwikkeld voor het gedistribueerd grafisch cluster van het TNO Fysisch Electrotechnisch Laboratorium. De applicatie modules zullen in een later hoofdstuk nader aan de orde komen. Elke applicatie module is op zichzelf herbruikbaar voor andere doeleinden, maar er zijn wel afhankelijkheden. De afhankelijkheden liggen met name bij de datacommunicatie machine en het bestandsysteem. Overige afhankelijkheden liggen bij de bibliotheken die zijn gebruikt voor het implementeren van de applicatie modules.

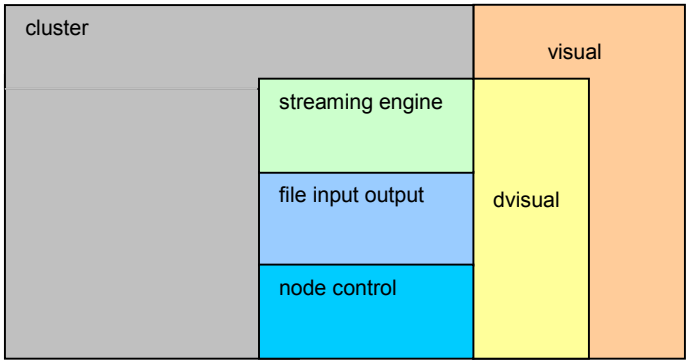


Fig3.0

7.2 Beeld synchronisatie

Beeldsynchronisatie of beeldpunt synchronisatie is van belang voor het realiseren van gelijk lopende tekenprocessen voor het plaatsen van beeldpunten. Het visuele effect van niet gesynchroniseerde beeldpunten is vrijwel niet zichtbaar maar kan wel een effect hebben op de applicatie. Zonder beeldsynchronisatie middels gespecialiseerde hardware is realtime afstemming van de tekenprocessen niet mogelijk en kunnen de realtime synchronisatie momenten van een applicatie oscilleren [44, 15]. De realtime oscillatie is het gevolg van niet of slecht afgestemde processen en van stuuralgoritmen [15] die de processen op een vaste frequentie trachten te houden. De door realtime hardware afgestemde processen garanderen dat ze binnen een bepaalde frequentie blijven [44].

Beeldsynchronisatie wordt ook wel genlocking genoemd. Genlocking is een vorm van synchronisatie waarbij een groep tekenprocessen van beelden die parallel moeten lopen op hetzelfde moment beginnen. De tijdsmomenten waarin de tekenprocessen beginnen met tekenen mogen met een bepaalde tijdsafwijking verschillen, dit verschil noemt men de genlock latency. Bij genlocking middels hardware geeft de genlocking hardware pulses af die door de ontvangers gebruikt kunnen worden voor synchronisatie van de tekenprocessen.

Om genlocking te realiseren is gekozen voor een hardware oplossing met realtime hardware in de vorm van RedHawk machines met afgeschermde processoren [44] voor het genereren van realtime pulses. Deze hardware is bij het TNO Fysisch Electrotechnisch Laboratorium onderzocht op genlocking en realtime prestaties [43]. Uit de conclusies van het onderzoek is gebleken dat de realtime prestaties van de RedHawk in het ergste geval binnen een afwijking van 11-30 microseconden [43] constant blijven. De afwijking met RedHawk hardware is aanzienlijk minder dan de afwijking bij synchronisatie voor civiele computers die binnen een afwijking van 100-300 milliseconden [44] constant blijft.

De realtime hardware van RedHawk biedt gestandaardiseerde POSIX oplossingen voor realtime synchronisatie van communicatie processen. Voor het ontwikkelen van een genlock is een voorstel gedaan van een RedHawk hardware architectuur die genlocking op hardware niveau kan realiseren. In de hardware architectuur wordt beeldsynchronisatie bepaald door een apart synchronisatie netwerk. Voor de implementatie van deze oplossing heeft RedHawk hardware componenten geïntroduceerd die in realtime opereren.

De volgende paragrafen zullen in verder detail ingaan op een barrier, de architectuur en de broncode van de synchronisatie systemen. In elke paragraaf zullen de synchronisatie systemen in verschillende vormen worden besproken. Ook zal er voor elk systeem een architectuurschets gemaakt worden van de hardware waar de synchronisatie op gerealiseerd zal worden. Verder zullen er synchronisatie broncode fragmenten worden gepresenteerd zoals die voorkomen in de gedistribueerde grafische applicatie programmeertaal.

7.2.1 Synchronisatie barrier

Het onderstaande figuur illustreert de werking van een synchronisatie barrier, of synchronisatie grens, tijdens genlocking. Het figuur is een beeld van een hypothetisch verloop van synchronisatie punten. Elk element in het figuur heeft een andere betekenis. Het figuur illustreert een groep van drie synchroniserende tekenprocessen, over een cluster van drie machines. Twee van de drie tekenprocessen duren te lang en veroorzaken een vertraging. Door de vertraging worden de tekenprocessen in de volgende synchronisatiestap gesynchroniseerd in het derde tijdsslot. Zodoende wordt er door de vertraagde tekenprocessen een genlock signaal overgeslagen en beginnen de tekenprocessen pas weer te tekenen bij het volgende genlock signaal. De volgende groep tekenprocessen valt binnen de genlock tijdspanne en veroorzaken geen vertraging. Bij de derde genlock veroorzaakt het eerste tekenproces een vertraging en zullen de volgende tekenprocessen een genlock signaal overslaan.

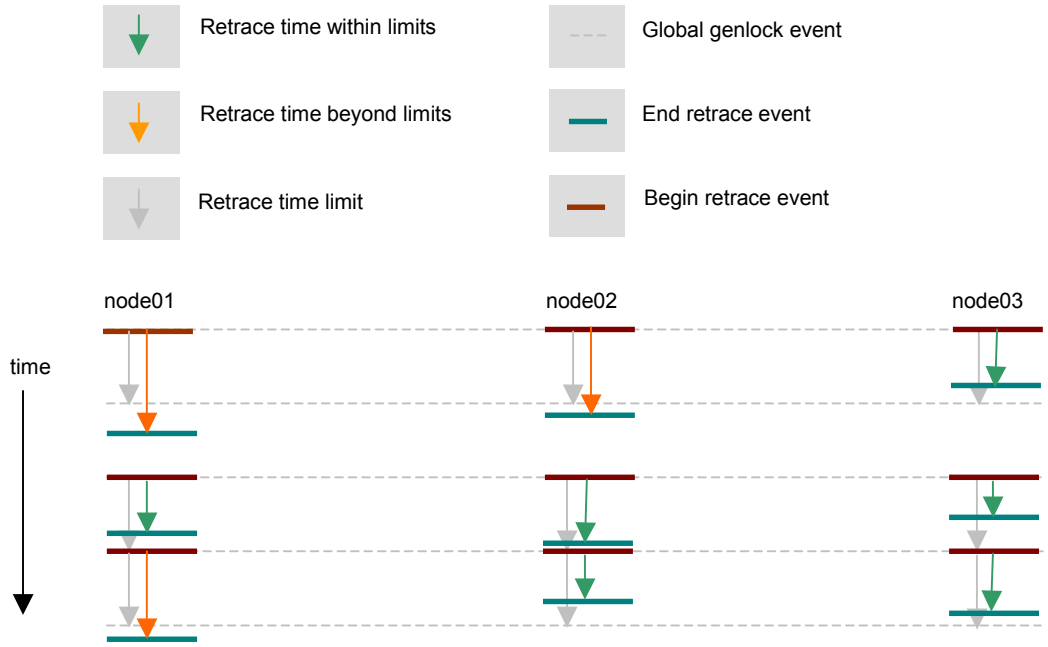


Fig4.0

7.2.2 Synchronisatie architectuur

De onderstaande architectuur geeft twee mogelijkheden voor een genlocked hardware model. Het eerste model is een genlock generator die op een gespecialiseerde genlock machine aanwezig is. De pulse generator genereert een realtime genlock signaal dat door de grafische kaarten wordt afgevangen, een eis is hierbij dat de eerste grafische kaart een genlock signaal uitgang en de overige grafische kaarten een genlock ingang moet hebben. Het tweede model is een gedistribueerd genlock signaal. Bij het tweede model is er geen centrale genlock generator maar wachten alle grafische processen op elkaar alvorens een nieuwe groep beeldpunten te tekenen, hierbij hebben alle grafische kaarten een genlock uitgang, een doorgang en een ingang. Tijdens afstemming van het realtime proces wordt er pas een nieuw genlock signaal gegenereerd als alle grafische processen klaar zijn. De tweede groep is van toepassing voor het gedistribueerd cluster. Zowel de tweede als de eerste genlock architectuur in het model kunnen op hardware- of applicatie niveau worden geïmplementeerd. Tijdens ontwikkeling van het grafisch cluster voor het TNO Fysisch Electrotechnisch Laboratorium is er een genlock algoritme voor hardware genlocking voorgesteld.

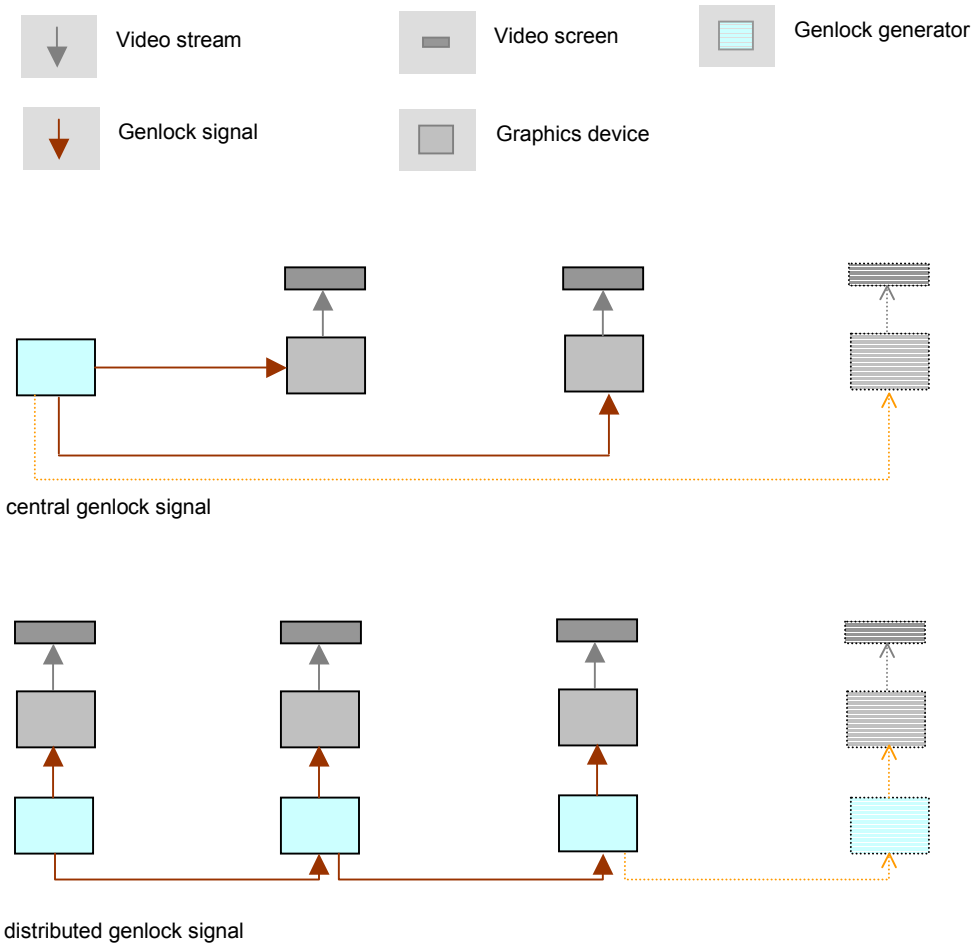


Fig4.1

7.3 Frame synchronisatie

Bij frame synchronisatie, of framelocking, worden de frames van de beelden die getekend zijn gesynchroniseerd. Framelocking wordt doorgaans gerealiseerd door het plaatsen van realtime barriers tussen de swapbuffer-, clearscreen- en drawscreen operaties van grafische bibliotheken. Zonder framelocking zal er bij een ongesynchroniseerde applicatie een visuele afwijking zijn in de momenten waarop het tekenen van de verschillende frames beginnen. Bij frame synchronisatie worden de beginpunten van de frame tekenprocessen voor elk frame op elke machine gelijk gehouden.

Tijdens het onderzoek zijn bepaalde technieken voor proces synchronisatie bestudeerd en is er berekend of deze een rigide realtime prestatie konden garanderen. Er is ontdekt dat interruptie en afbreking van synchronisatie momenten [13] kan zorgen voor gegarandeerde realtime prestaties. Deze techniek bleek slechts effectief voor het implementeren van systemen waarbij gegevens weggelaten mochten worden [13]. De techniek zou bijvoorbeeld niet gebruikt kunnen worden voor realtime synchronisatie van beelden, omdat het dan afkappingen zou moeten doen. De methode zou echter wel gebruikt kunnen worden bij het verdelen en berekenen van gegevensverzamelingen voor bijvoorbeeld regen of mist, waar het gedeeltelijk wegvallen van gegevens geen zichtbaar effect heeft op de visualisatie.

Om realtime prestatie eisen van het grafisch cluster te behalen en behouden zonder realtime hardware, is er een integrerende en gedistribueerde regelaar ontwikkeld. Deze regelaar is niet strikt realtime maar zal een gemiddelde hebben dat op den duur convergeert naar een constante realtime waarde. Omdat het convergent gedrag op den duur een simulatie binnen dezelfde tijdspanne zal houden als een realtime regelaar, zal de simulatie op niet realtime hardware een betrouwbare weergave zijn van de werkelijkheid. De regelaar bevat behalve een integrerende capaciteit, ook een voorspellende capaciteit die generiek de framesnelheid voorspelt en de regelaar daar op afstemt. Omdat de voorspellende capaciteit gedeeltelijk chaotische beeldovergangen voorspelt, heeft het voorspellende gedeelte in de regelaar slechts een klein relatief effect op het algehele synchronisatie proces.

De realtime regelaar wordt gesynchroniseerd over een netwerk middels een barrier die gebruik maakt van een implementatie van het Message Passing Interface. Het effect van de barrier is een frame synchronisatie op het moment dat alle parallel lopende tekenprocessen over een gedistribueerd grafisch cluster klaar zijn met het tekenen van een enkel beeld. Bij de meeste synchronisatie technieken zal het langzaamste tekenproces bepalen met welke snelheid de synchronisatie plaatsvindt. Andere synchronisatie technieken, gedeeltelijk gebaseerd op proces interruptie technieken [13], kunnen leiden tot een vervaagd beeld [47].

De volgende paragrafen zullen verder ingaan op de barrier en architectuur voor synchronisatie. De synchronisatie barrier zal in verschillende vormen worden besproken. De synchronisatie architectuur geeft een beeld van de hardware architectuur voor synchronisatie.

7.3.1 Synchronisatie barrier

Voor frame frequentie begrenzing moet er een algoritme ontworpen worden dat een gemiddeld frame tekenproces frequentie aanhoudt. Het frame tekenproces moet ook bij overloop van een tijdslimiet naar een centrale frequentie convergeren. Een algoritme, ontwikkeld in samenwerking met het TNO Fysisch Electrotechnisch Laboratorium, is voorgesteld in de vorm van een regelaar. Hieronder worden twee voorbeelden gegeven van synchronisatie processen voor respectievelijk locked synchronisatie en reactieve synchronisatie. Bij locked synchronisatie worden de frame tekenprocessen in een vast tijdsslot geplaatst. Reactieve synchronisatie wordt toegepast als het langzaamste tekenproces de synchronisatiesnelheid bepaalt. De ontwikkelde regelaar, die later tot in detail besproken zal worden, realiseert een hybride vorm van locked en reactieve synchronisatie. De tijdsberekeningen in de implementatie van de regelaar zijn gebaseerd op de Simple Direct Medialayer welke direct beschikbaar is in het gebruikte besturingssysteem.

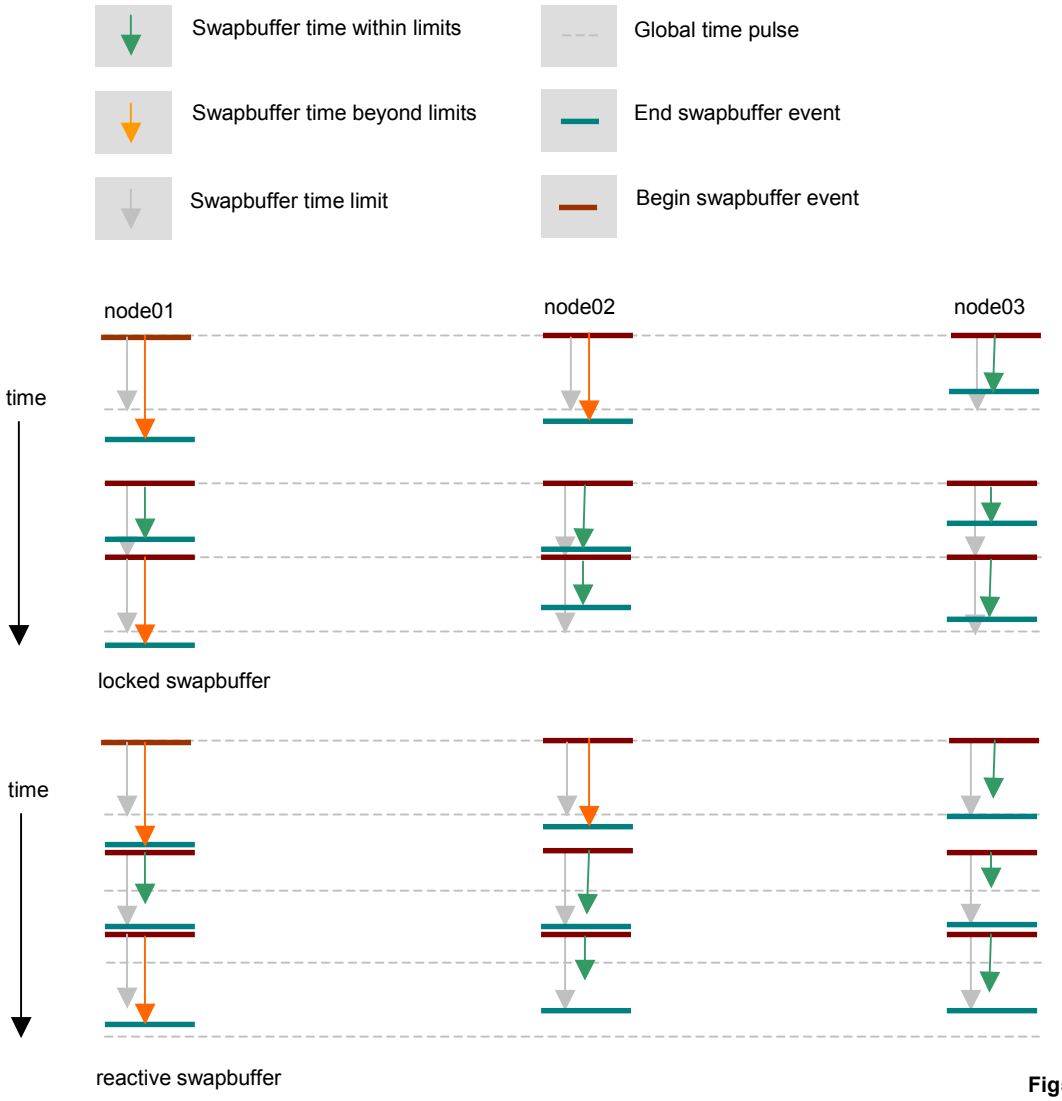


Fig5.0

7.3.2 Synchronisatie architectuur

Voor de synchronisatie van frame tekenprocessen wordt vaak een netwerk gebruikt. Hieronder zijn twee hardware architecturen voor synchronisatie netwerken gegeven. De eerste architectuur bestaat uit een netwerk gekoppeld middels op Ethernet gebaseerde netwerkkaarten. De tweede architectuur maakt gebruik van een gespecialiseerd netwerk voor synchronisatie van realtime processen. De architectuur in het grafisch cluster zoals ontwikkeld tijdens dit onderzoek is voorbereid op een gespecialiseerd RedHawk synchronisatienetwerk. Een RedHawk machine heeft, zoals eerder aangegeven, in het ergste geval een synchronisatie afwijking van 11-30 µs (microseconden) per machine [43]. Voor eenvoud van implementatie is er gebruik gemaakt van de eerste op Ethernet gebaseerde architectuur. De synchronisatie vindt hierbij plaats over een switched Ethernet netwerk. De onderstaande modellen zijn een representatie van respectievelijk op Ethernet netwerk gebaseerde synchronisaties en op realtime hardware gebaseerde synchronisaties. Voor implementatie van op hardware gebaseerde synchronisatie voldoet de software aan realtime POSIX standaarden bij de vervanging van twee op SDL gebaseerde functies door POSIX functies voor het opvragen van de absolute tijd en het wachten van een bepaalde tijd.

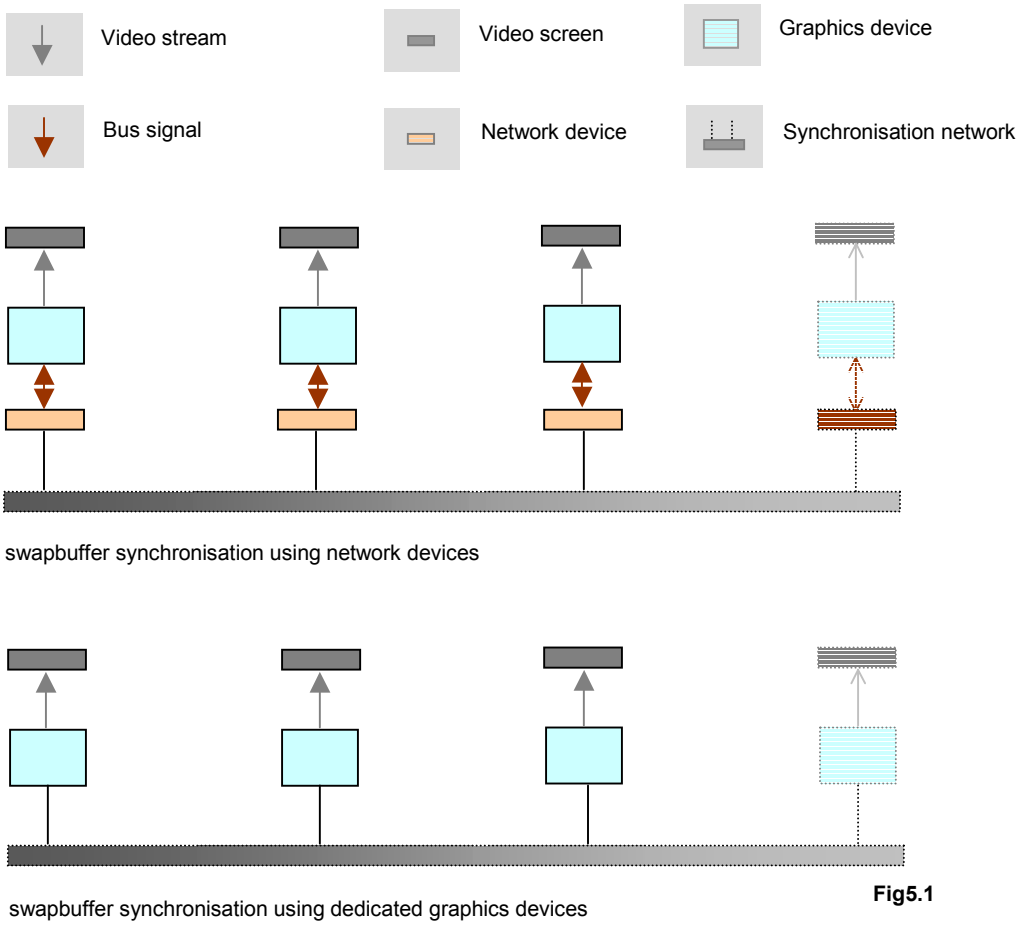


Fig5.1

7.4 Data synchronisatie

Een significant gedeelte van de ontwikkeling van de gedistribueerde grafische applicatie programmeertaal is het onderzoeken en ontwikkelen van synchronisatie algoritmen voor gegevens geweest. De algoritmen zijn voor een groot deel geïmplementeerd, met uitzondering van de schrijf- en lees operaties voor parallelle bestanden. Verder zijn er voor een effectieve gegevenssynchronisatie, verschillende onderdelen zoals interpolatie van communicatie processen onderzocht en geïmplementeerd. Ook zijn er algoritmen ontworpen voor realtime controle van processen en zijn deze algoritmen geïntegreerd met de ontwikkelde systemen.

Realtime gegevenssynchronisatie is een vorm van synchronisatie waarbij gegevens in realtime gesynchroniseerd worden tussen andere realtime processen door. Bij het synchroniseren van gegevens worden lees- of schrijf operaties op die gegevens in realtime gesynchroniseerd. Verder wordt bij gegevenssynchronisatie de interpolatie van de lees- of schrijf operaties geregeld door een interpolatie algoritme.

Tijdens het onderzoek is er een datacommunicatie machine ontwikkeld in de vorm van een applicatie programmeertaal. Het ontwerp van de datacommunicatie machine moest rekening houden met controle van gegevensstromen vanuit een willekeurige machine in een netwerk. Verder was het van belang dat de datacommunicatie machine dynamisch aanpaste afhankelijk van de grootte van de gegevens en de grootte van het netwerk. De machine moest het ook mogelijk maken om grafische processen te interpoleren met communicatie processen voor de realisatie van realtime frequentie controle.

De datacommunicatie machine is na gedetailleerd onderzoek van de Message Passing Interface op zodanige wijze ontwikkeld dat het scaleert en interpoleert afhankelijk van de grootte van het netwerk, het aantal actieve gegevensstromen en hoeveelheid processen. De interpolatie mogelijkheid van de datacommunicatie machine omvat het interpoleren van applicatie processen met communicatie-, beeld- en gegevens- synchronisatie processen. Verdere ontwikkeling van de datacommunicatie machine zal verfijndere wevingen van proces interpolaties mogelijk moeten maken. Een eis voor een communicatie bij interpolatie van grafische processen tussen communicatie processen is dat elk grafisch proces binnen zijn interne frequentie moet blijven zonder dat de communicatie de frequentie van het proces verstoort of vertraagt. De datacommunicatie machine scaleert op gegevensstroom- en netwerk niveau.

Scalering op gegevensstroom niveau, wat verticale scaleerbaarheid is genoemd, betekent dat indien het aantal gegevensstromen groeit, de datacommunicatie machine meer berichten over het netwerk zal sturen voor synchronisatie. Scalering op netwerk niveau, wat horizontale scaleerbaarheid is genoemd, stelt de machine in staat om afhankelijk van het aantal machines of applicatie processen, te groeien of te krimpen. De volgende paragrafen zullen in verder detail ingaan op de synchronisatie onderdelen en het scaleren. Een globaal model van de datacommunicatie machine voor gegevenssynchronisatie zal worden voorgesteld, en er zal een voorbeeld gegeven worden van de programma broncode die gebruikt wordt om communicatie te realiseren.

7.4.1 Datacommunicatie machine

Het onderstaande model is een architectuur model van de datacommunicatie machine zoals die is ontwikkeld voor de grafische applicatie programmeertaal. Een eis voor de machine was dat het de gegevensstromen moest synchroniseren zonder dat het direct zou communiceren bij aanroep van een communicatie functie. Er was dus een regelschema nodig voor het regelen van de gegevensstromen.

De datacommunicatie machine is, globaal gezien, opgedeeld in drie processen die gedurende het passeren van een programma hoofdproces worden uitgevoerd. De eerste twee processen bestaan uit het opvragen of instellen van informatiestroom elementen voor distributie over een netwerk en het laatste proces bestaat uit het distribueren van de informatie. Uit metingen is gebleken dat een instellings- of opvragings proces ongeveer 1-100 nanoseconden in beslag neemt. Het instellen of opvragen van de informatiestroom elementen wordt op een willekeurig moment in een computer programma gedaan. De gebruiker van de datacommunicatie machine zal slechts met het instellen of opvragen van informatiestroom elementen worden geconfronteerd. Persistente gegevensstructuren zorgen er in de implementatie voor dat een opvraag proces op alle machines synchroon loopt met de ingestelde waarde van de informatiestroom.

Het is duidelijk dat in de machine architectuur het opvragen en instellen van de informatiestroom waarden in het hoofdproces van een programma gedaan wordt. Opvragen en instellen van de gegevensstroom waarden kan hier geheel arbitrair en ongeordend plaatsvinden. Tijdens het machine hoofdproces, dat slechts één keer en meestal later plaatsvindt, worden alle ingestelde gegevensstromen verzonden en ontvangen over het netwerk. Deze ingestelde gegevensstromen kunnen tijdens het verzenden en ontvangen geïnterpoleerd worden met andere systeemprocessen om het totale tijdsverloop van een applicatie beter te regelen.

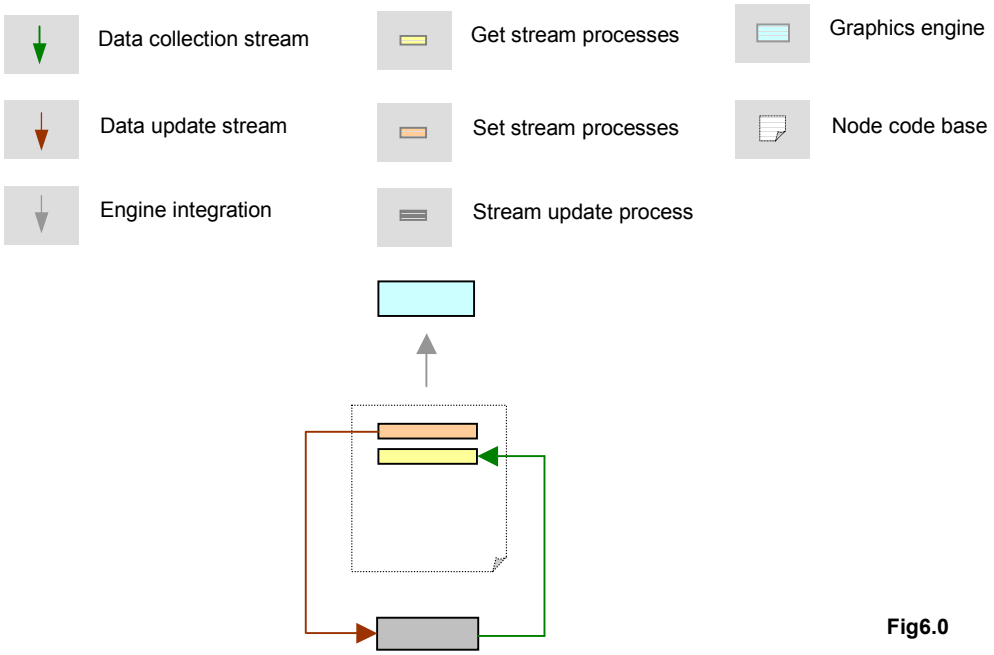


Fig6.0

7.4.2 Verticale scaleerbaarheid

De machine, zoals die is ontworpen voor de gedistribueerde simulatie demonstratie, is scaleerbaar over het aantal gegevensstromen. Verder is de machine scaleerbaar over het aantal machines in het netwerk. De horizontale scaleerbaarheid is de scaleerbaarheid over het aantal machines in het netwerk. De verticale scaleerbaarheid is de scaleerbaarheid over het aantal gegevensstromen, dat er van een bepaald type aanwezig is.

Ieder gegevensstroom type wordt in pakketten van een vaste grootte, of batchgewijs, over het netwerk verstuurd ook als er geen gegevensstroom aan verbonden is. Dit betekent dat als er nog geen actief bruikbare informatie in een gegevensstroom zit, zij toch zal worden verstuurd. Het voordeel van deze techniek is dat de machine pas na een maximale overschrijding van gegevensstromen zich opnieuw zal scaleren. Hierdoor wordt het eenvoudiger om generieke interpolatie algoritmen te introduceren en is het effect van het toevoegen van gegevens niet merkbaar, omdat ongebruikte gegevensstromen geheel opgebruikt moeten zijn voordat de machine zichzelf opnieuw scaleert. Indien een batchgewijze versturing van gegevensstroom typen ongewenst is, is het mogelijk om de batch grootte op één te zetten. Bij de voorgaande grootte scaleert de machine bij elke nieuwe gegevensstroom die over het netwerk verstuurd wordt.

Het onderstaande model geeft weer hoe de machine scaleert bij een maximale overschrijding van de hoeveelheid gegevensstromen met een batchgrootte van vier. De eerste scalering is een initialisatie van de eerste batch gegevensstromen. Tijdens de eerste scalering, aangegeven door een pijl met het getal één, worden er drie gegevensstromen over het netwerk verstuurd. Bij de tweede scalering, komt er een gegevensstroom bij en wordt de tweede batch geïnitialiseerd en verstuurd over het netwerk. Bij de derde scalering wordt het eerste slot in de tweede batch gebruikt door een nieuwe gegevensstroom. Indien een gegevensstroom wordt verwijderd, zal de machine zichzelf terug scaleren tot de vorige hoeveelheid verstuurde batches.

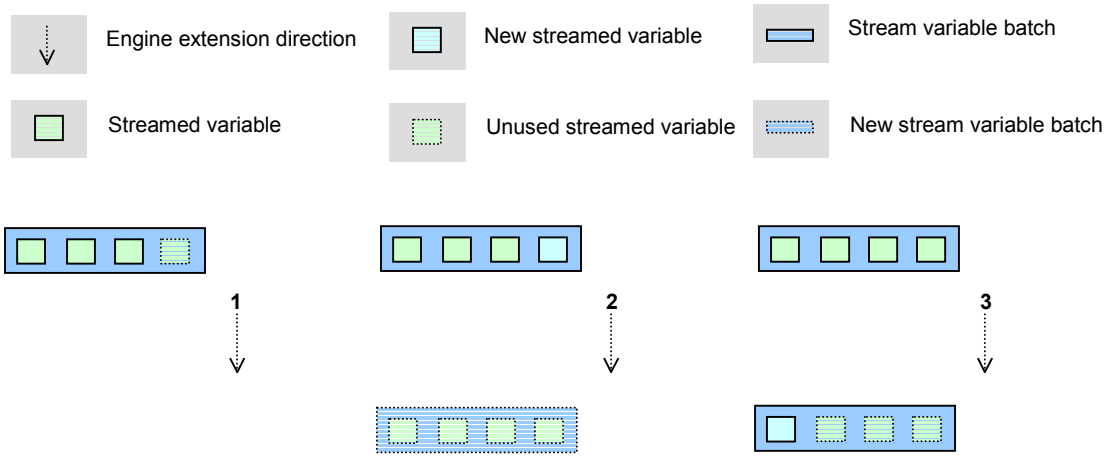


Fig6.1

7.4.3 Horizontale scaleerbaarheid

De horizontale scaleerbaarheid van de machine is totaal afhankelijk van het aantal machines of virtuele machines in een netwerk. Bij een toename van het aantal machines zal het aantal mogelijke broadcasts per batch facultatief toenemen. De toeneming van mogelijke broadcasts is vanwege een synchronisatie techniek die individuele verzend- en ontvangst processen overbodig maakt en gegevens op een enkele machine zichtbaar maakt op alle machines. Het gebruiken van collectieve broadcast functies maakt broncode management van verzend- en ontvangst processen overbodig [27]. Verder kunnen machine specifieke implementaties van collectieve functies de prestaties van programmatuur ten opzichte van verzend- en ontvangst functies optimaliseren zonder de programmatuur zelf te hoeven veranderen [27].

Het onderstaande model geeft een voorbeeld van een horizontale scalering van twee machines naar drie en van drie machines naar vier. Bij iedere scalering nemen het aantal mogelijke broadcast gegevensstroom kanalen facultatief toe. De eerste scalering komt voor bij twee broadcast machines en is aangeduid door een pijl met het getal één. Bij de eerste scalering bestaat er een enkel gegevensstroom kanaal voor een broadcast. De tweede scalering komt voor bij een toevoeging van een derde broadcast machine en is aangeduid door een pijl met het getal twéé. Bij de tweede scalering komen er twee gegevensstromen bij. De derde scalering voegt aan de vierde broadcast machine drie gegevensstroom kanalen toe.

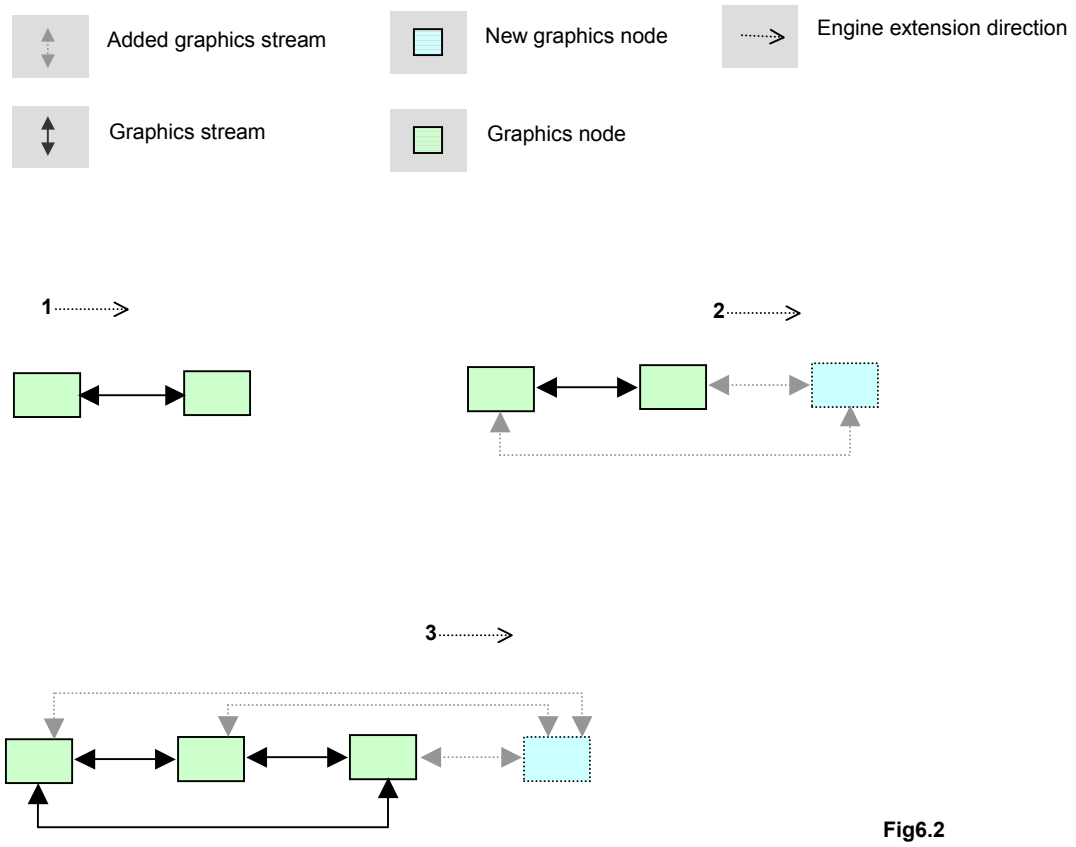


Fig6.2

8 Cluster omgeving

Het model voor het cluster is modulair en op componenten gebaseerd. Dit betekent dat verschillende componenten of cluster functionaliteiten aan het cluster toegevoegd of van het cluster verwijderd kunnen worden. In de volgende paragraaf bevindt zich een model dat het op componenten gebaseerd cluster [25,28,38] verder verduidelijkt. Verder zullen de paragrafen daarna een configuratie manager bespreken die is ontwikkeld voor configuratie en installatie van de componenten in het cluster. De cluster omgeving bestaat uit een netwerk van Linux machines die via de MPICH messaging omgeving met elkaar communiceren.

De MPICH omgeving werd geselecteerd vanwege de mogelijkheid tot uitbreiding naar een fast interconnect SCI-MPICH [29] of 1000MBps netwerk, dat een communicatiesnelheid van respectievelijk 235MBps of 1000MBps en een latency lager dan repectievelijk 5 of 1 µs (microseconden) bereikt bij maximale interconnectiviteit. Bij gebruik van dit cluster kan een fast SCI of 1000MBps interconnect tussen verschillende computers in combinatie met RedHawk realtime hardware betrouwbare realtime simulaties realiseren. Toekomstige versies van de RedHawk realtime omgeving bieden verder een realtime Linux Myrinet MPI/RT [46] implementatie waarmee deterministische, of gegarandeerde, realtime berekeningen gemaakt kunnen worden.

Behalve SCI interconnectiviteiten kunnen eenvoudiger implementeerbare GM-MPICH netwerken van civiele systemen snelheden behalen die vergelijkbaar zijn met, en in sommige gevallen zelf sneller [14] zijn dan SGI Origin 2000 systemen. Een civiel GM-MPICH Myrinet M3E32 cluster [14] bereikt communicatiesnelheden tot 1.8 Gbits/s (gigabits per seconde) en latencies van 9.3 µs (microseconden). In vergelijking met de CrayT3E supercomputer (34.71 µs) heeft een eenvoudig pentium-III beowulf Myrinet 2000 cluster [14] een aanzienlijke latency voorsprong.

Een andere reden voor het gebruiken van de MPICH implementatie is dat er op het gebied van platform onafhankelijkheid onderzoek en ontwikkeling gedaan wordt naar een meta clustering omgeving. De meta clustering omgeving, MT-MPICH genaamd, kan gebruikt worden voor het ontwikkelen van hybride clusters [34], waarin verschillende parallelle netwerken middels hetzelfde protocol met elkaar communiceren. Deze transparante omgeving is zo opgesteld dat heterogene parallelle computer netwerken in een grafisch cluster niet alleen collectief rekenkracht en opslagcapaciteit kunnen leveren [35], maar ook middels hetzelfde protocol met elkaar kunnen communiceren. De collectieve opslagcapaciteit en rekenkracht wordt, in abstracte zin, geboden door een metacomputer voor de communicatie van berichten [35].

Verder blijkt uit experimenten met onder andere Macintosh [2], Windows [3,6] en Linux [5,6,14] platforms dat realtime gedistribueerde grafische clusters op civiele systemen een prestatievermogen, interconnectiviteit en transparantie hebben bereikt wat voorheen zelfs niet met supercomputers mogelijk was. Dit voorgaande maakt het voor commerciële bedrijven steeds aantrekkelijker om dergelijke clusters te gebruiken voor de ontwikkeling van parallelle applicaties.

8.1 Configuratie architectuur

Hieronder bevindt zich een architectuurmodel van de applicatie zoals die is ontwikkeld voor het TNO Fysisch Electrotechnisch Laboratorium. De applicatie bestaat uit een configuratie manager, realtime synchronisatie barrier, datacommunicatie machine, machine controllers, parallelle bestandsystemen, randapparaat bibliotheken. De applicatie is afhankelijk van externe bibliotheken. De externe applicatie afhankelijkheden zijn weergegeven door blauwe pijlen die aangeven van welke bibliotheken het gedistribueerd grafisch cluster afhankelijk is. Zonder deze bibliotheken zal het grafisch cluster niet functioneren. In de volgende hoofdstukken zullen de componenten in de architectuur nader worden belicht.

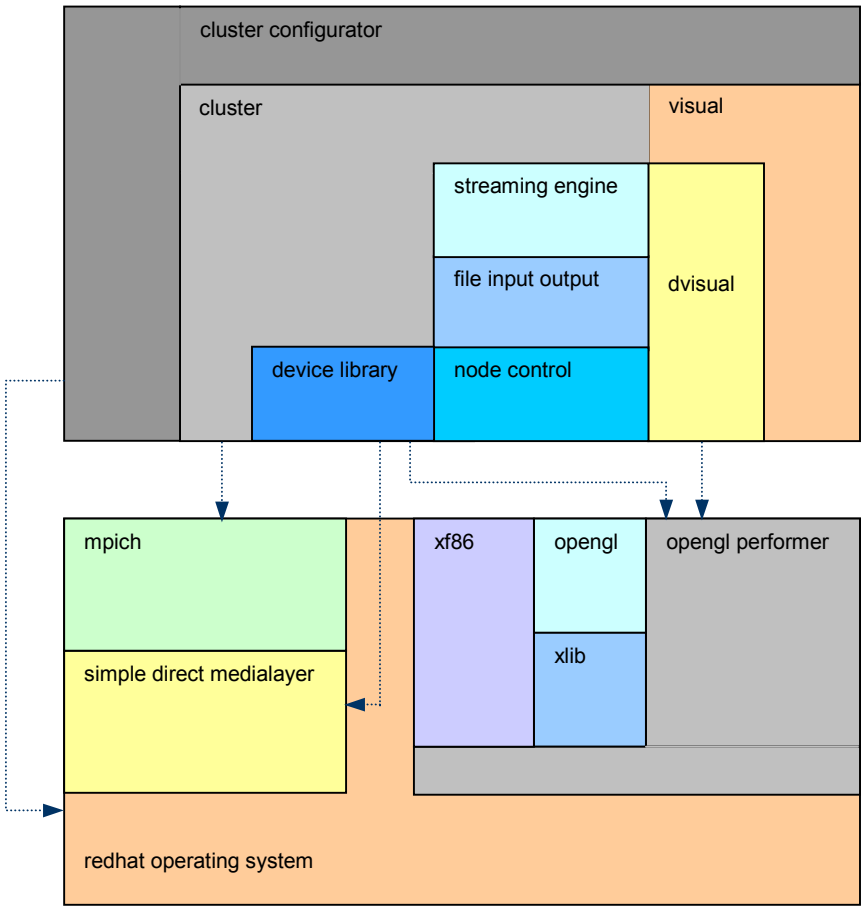


Fig7.0

8.2 Configuratie manager

Het gedistribueerd grafisch cluster is geprogrammeerd voor een clustering omgeving die gebruik maakt van de Message Passing Interface, Performer, Simple Direct Medialayer en RedHat Linux. De Message Passing Interface en Performer bibliotheek zijn in de voorgaande hoofdstukken besproken. De Simple Direct Medialayer is een applicatie programmeertaal voor Directe toegang tot systeembronnen. Om de gedistribueerde applicatie programmeertaal en clustering omgeving te kunnen gebruiken is er een configuratie manager ontwikkeld die de net genoemde onderdelen installeert en configureert. Deze configuratie manager stelt gebruikers in staat om op eenvoudige wijze clusters te configureren en cluster componenten te installeren.

De configuratie manager is een initiële aanzet om op een hoger niveau de clustering omgeving beter te controleren. Dit betekent dat bij doorontwikkeling van de configuratie manager, verscheidene nu nog statische instellingen dynamisch gemaakt moeten worden door bijvoorbeeld een netwerk applicatie voor het configureren van het cluster te ontwikkelen. Het voordeel van een configuratie manager voor het configureren van het cluster is dat op een willekeurige RedHat Linux machine de configuratie van een applicatie direct is georganiseerd en eventueel vernieuwd.

Wegens de directe organisatie en installatie van benodigde cluster en applicatie onderdelen middels een configuratie manager kunnen de cluster en applicatie onderdelen meegeleverd worden op een schijf waardoor er voor een gebruiker geen configuratie management meer nodig is. Deze benadering wordt veelvuldig toegepast in de commerciële wereld. Hier worden compact disk roms en de bijbehorende afhankelijkheden direct met een applicatie meegeïnstalleerd. In de open source wereld is een dergelijke installatie benadering nog niet van toepassing maar stijgt de behoefte voor configuratie management aanzienlijk.

De configuratie manager is ontwikkeld als een gebruikersvriendelijke menugestuurde omgeving die gebruikers in staat stelt het configuratie management uit te besteden aan een programma dat speciaal is ontwikkeld voor dit soort doeleinden. Het voordeel van een manager is dat bij uitlevering van een applicatie aan externe instanties, de programmatuur direct kan worden geïnstalleerd en geconfigureerd zonder additionele werkzaamheden. De volgende paragrafen geven een model van de configuratie manager zoals die momenteel is geïmplementeerd in het cluster management programma.

8.3 Configuratie stroomdiagram

Het onderstaande stroomdiagram is een weergave van de menustructuur en systeem acties als gevolg van menu keuzen. Iedere menu keuze leidt tot een andere menustructuur of het uitvoeren van een systeem configuratie of configuratie test. De omgeving is een gebruikersvriendelijk menu gestuurde applicatie die configuratie management van de volledige simulatie cluster in drie stappen mogelijk maakt. De eerste stap omvat het opstarten van de installer, de tweede stap het kiezen van de install all functionaliteit en de laatste stap het testen van de omgeving.

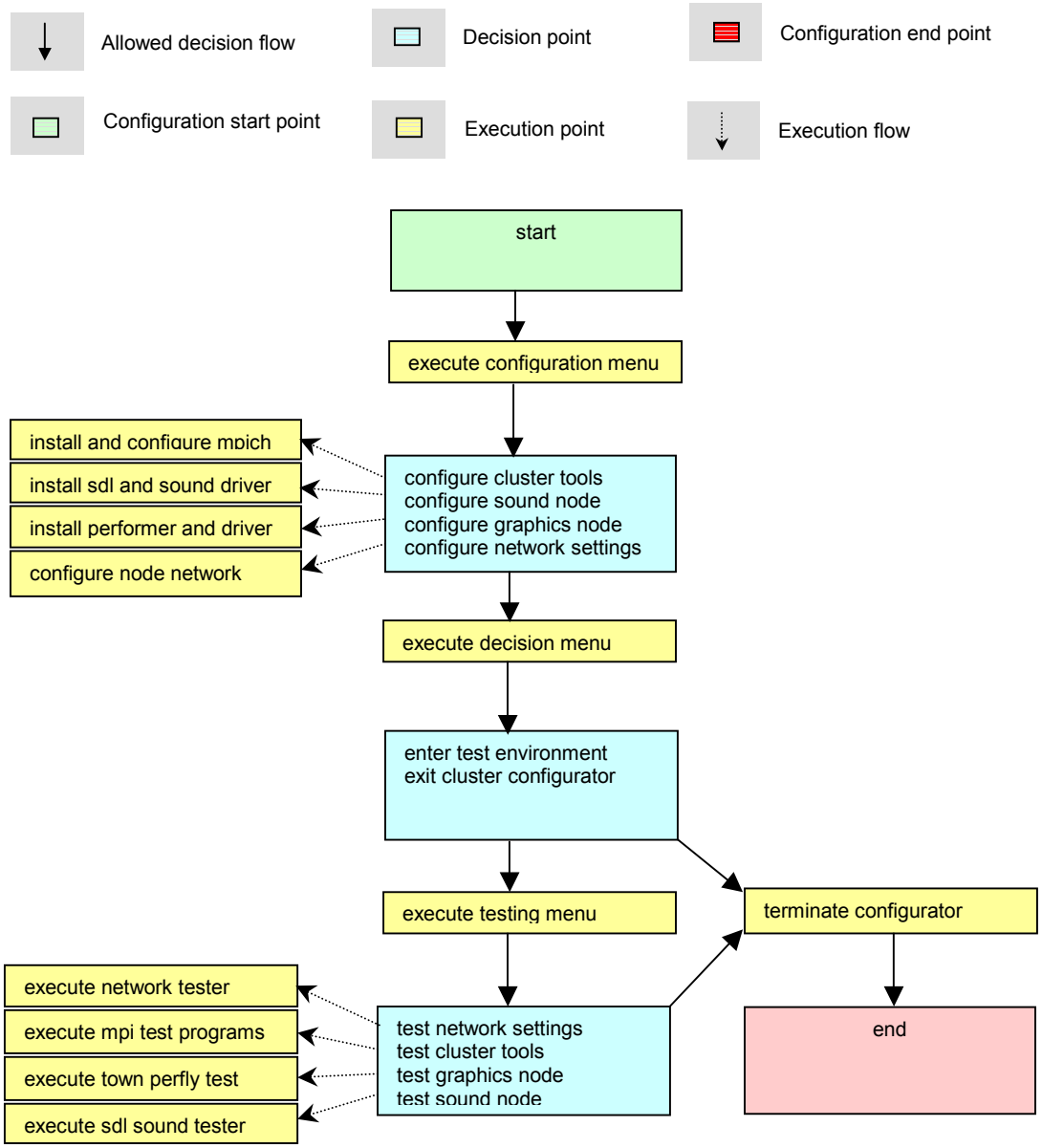


Fig8.0

8.4 Schermvoorbeeld manager

Hieronder is een schermvoorbeeld van de cluster manager weergegeven. De manager is momenteel in een ver beginstadium en is er weinig aandacht geschonken aan de ontwikkeling van de manager. De manager bestaat uit een eenvoudige interface die gebruikers in staat stelt om snel een cluster te installeren en te laten configureren. De manager start op met de eenvoudige vraag om een configuratie CDRom in de CDRom schijf te plaatsen. Na plaatsing van de CDRom kunnen verschillende machine componenten voor beeld, geluid, realtime en clustering geïnstalleerd worden door middel van een aantal menukeuzes die de gebruiker voorgelegd krijgt.

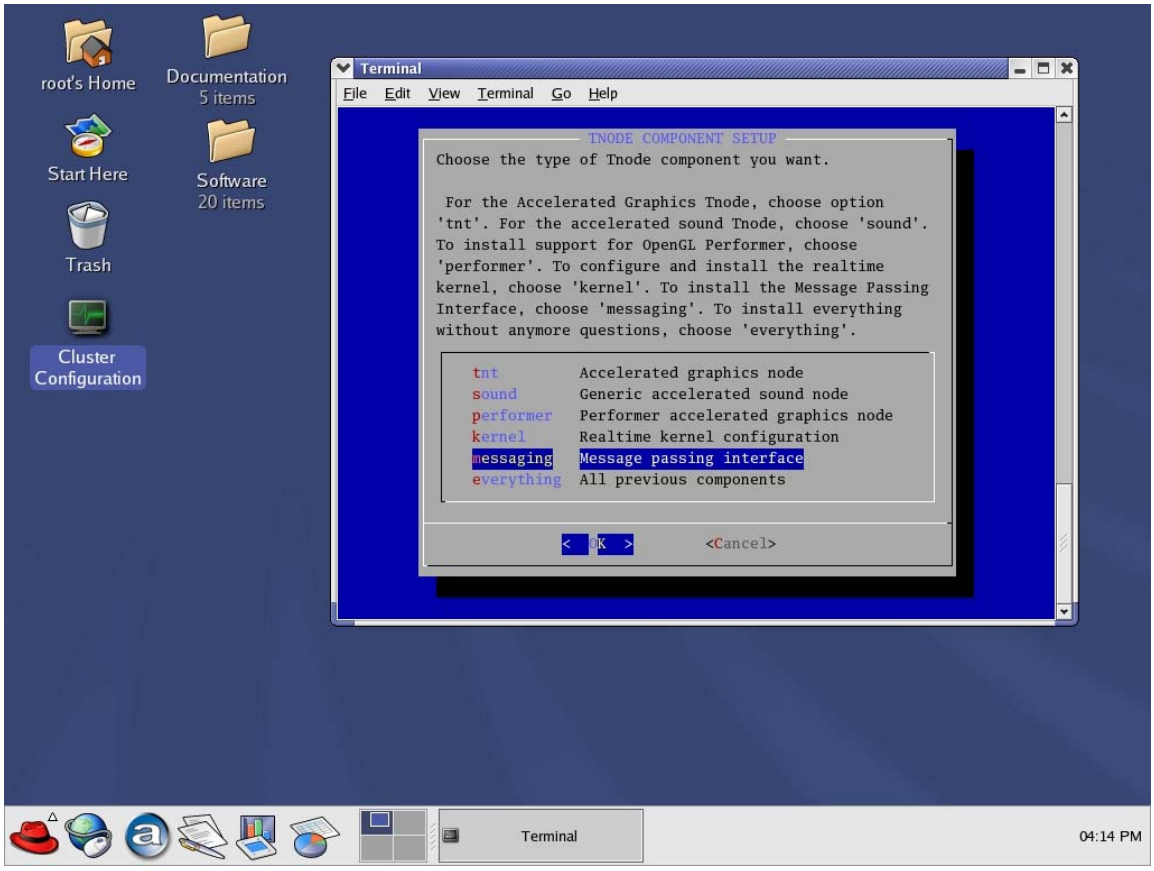


Fig8.1

De configuratie manager kan wellicht in een later stadium doorontwikkeld worden. Zoals die nu bestaat is de manager reeds bruikbaar voor een snelle cluster installatie en configuratie. Voor verfijndere installatie zou een manager ontwikkeld moeten worden die alle parameters van een cluster vanaf een centraal punt kan configureren. Deze configuratie zou middels een web interface het beste effect bereiken. Met name omdat een web interface platform onafhankelijkheid biedt. Ontwikkeling van een dergelijke cluster controller zou het grafisch cluster nog beter configureerbaar maken.

9 Lineaire visual

Om de lineaire grafische applicatie bruikbaar te maken voor een cluster van civiele systemen onder Linux is er een architectuur vereenvoudiging gemaakt van de grafische applicatie en zijn deze vereenvoudigde componenten geparallelliseerd. De methode van parallellisatie van deze grafische applicatie wordt in een later hoofdstuk toegepast om een architectuur voorstel te geven voor de bestaande grafische applicatie. Het grafische component in dit hoofdstuk is een grafische applicatie speciaal ontworpen voor eenvoudige parallellisatie van grafische simulatie componenten. Het geparallelliseerd grafisch cluster zal in applicatievorm dienen als demonstratie applicatie voor het uiteindelijk te parallelliseren systeem. De applicatie programmeertaal voor grafische clusters bevat een regelaar. Deze regelaar, in parallelle en niet parallelle vorm, is ontwikkeld voor commerciële doeleinden en is in een militaire simulatie omgeving niet van toepassing.

De reden dat de regelaar niet bruikbaar is voor militaire doeleinden is omdat de tijdscontroles zijn gebaseerd op functieaanroepen die niet tegemoetkomen aan de standaarden voor POSIX realtime. Voor realtime gedistribueerde simulaties is het van belang dat de tijdsafwijkingen gegarandeerd binnen een vooraf gestelde afwijkingstijd blijven [45]; bij het ontwikkelde systeem kunnen de maximum en minimum tijdsafwijkingen buiten een gegarandeerde tijdsafwijking vallen. De tijdsynchronisatie voldoet aan de eisen voor de realtime applicatie, maar het besturingssysteem voldoet niet aan de POSIX standaarden voor realtime. Tijdens de ontwikkeling van de applicatie is er een voorstel gedaan voor het vervangen van de nu nog niet op POSIX gebaseerde wacht- en tijdopvraag functies door functies die dat wel zijn. Verder is er een hardware architectuur voorgesteld die de synchronisatie van het grafisch cluster in een RedHawk omgeving realiseert. Indien de hardware- en applicatie standaarden voor POSIX worden opgevolgd zal de applicatie op het gebied van grafische synchronisatie en prestaties voldoen aan de op POSIX gebaseerde standaarden.

De volgende paragrafen geven een model van de regelaar en applicatie componenten van de lineaire grafische applicatie programmeertaal die gebruikt wordt voor de implementatie van de parallelle grafische applicatie. Het model is ontwikkeld in samenwerking met het TNO Fysisch Electrotechnisch Laboratorium. De regelaar is tijdens de ontwikkeling van het gedistribueerd grafisch cluster verder verfijnd met synchronisatie extrapolaties en geïnterpoleerde stuuralgoritmen. Bij het interpoleren van de stuuralgoritmen, waarbij één algoritme voor de frequentie van de applicatie en een ander voor de beeldfrequentie is geïmplementeerd, is verkennend onderzoek gedaan naar het oscillerend effect van gekoppelde systemen om de oscillatie te elimineren. De resulterende algoritmen die uit dit verkennend onderzoek zijn voortgekomen zijn in de lineaire- en parallelle- grafisch applicatie programmeertaal geïmplementeerd.

9.1 Realtime regelaar

Voor het realiseren van een correct functionerende realtime synchronisatie is er een algoritme ontworpen dat gebruikt kan worden in realtime clusters. Bij een reactie op de omgeving zal het synchronisatie algoritme het systeem vertragen of versnellen en convergeren naar een centrale doelfrequentie. Het onderstaande model is een weergave van de realtime regelaar die wordt gebruikt voor het regelen van de frame tekenproces synchronisaties. Het regelen van de applicatie synchronisaties wordt gedaan door een vergelijkbaar algoritme zonder de integrerende capaciteit.

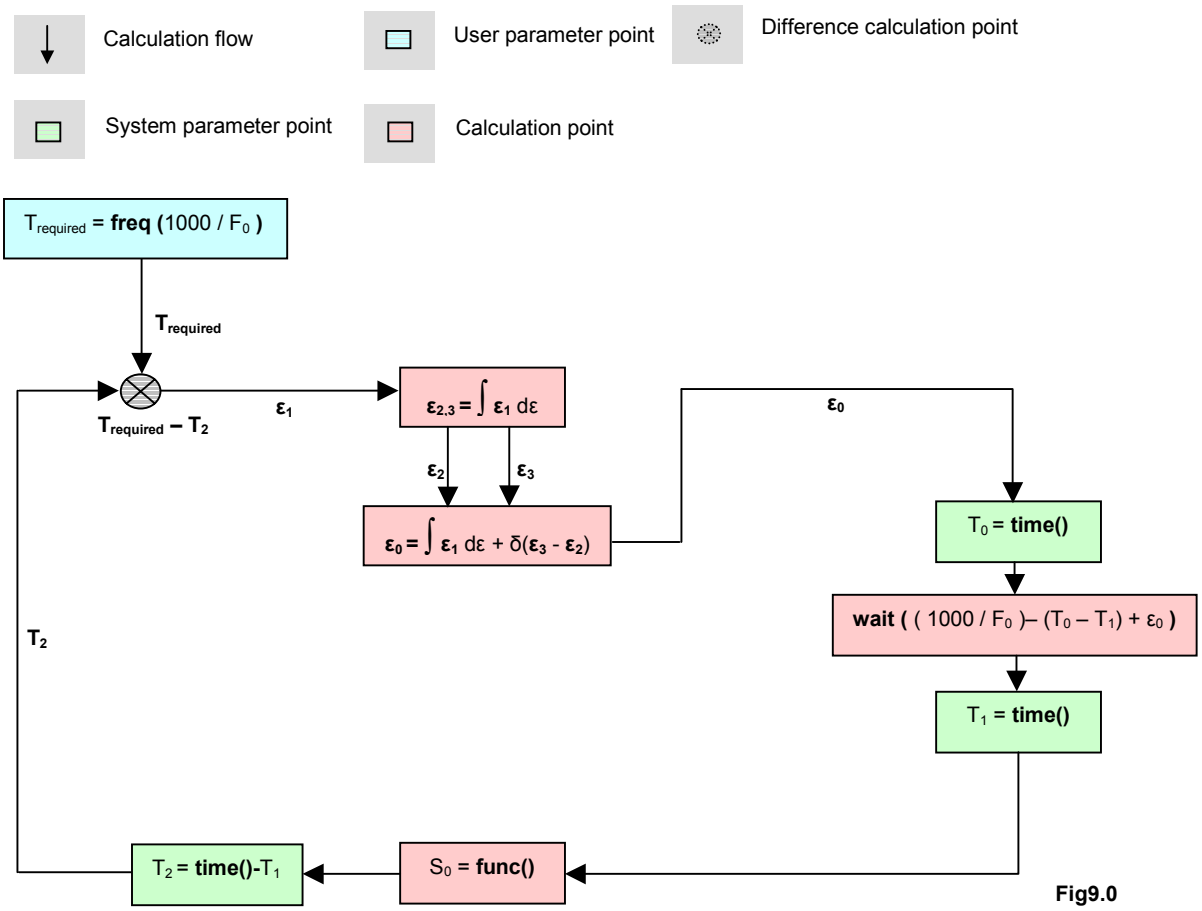


Fig9.0

In de bovenstaande regelaar wordt een doelfrequentie (**F_0**) omgezet naar een gewenste tijd (in milliseconden) en werkt de regelaar de processen naar de gewenste tijd toe (**$T_{required}$**) door het integreren van de afwijking (**$T_{required} - T_2$**) tussen de gehele cyclus tijd en de gewenste tijd. De geïntegreerde afwijkingstijd (**ε_1**) wordt gebruikt in het voorspellen van de toekomstige afwijkingstijd (δ(**ε_3** - **ε_2**)). De toekomstige afwijkingstijd wordt gemeten als het verschil tussen de afwijkingstijd vòòr de vorige afwijkingstijd (**ε_2**) en de vorige afwijkingstijd (**ε_3**) waarna het wordt vermenigvuldigd met een zeer klein getal (**δ**) om het effect van de voorspelling te minimaliseren. De geïntegreerde waarde van de afwijkingstijd (**ε_1**) en de voorspelde afwijkingstijd (δ(**ε_3** - **ε_2**)) worden gebruikt in het wacht proces (**wait**) om de swapbuffer en grafische applicatie te regelen. Voor het regelen van de applicatie processen is de integrerende capaciteit verwijderd om oscillatie te voorkomen.

9.2 Kanaal model

De realtime regelaar wordt gebruikt voor het synchroniseren van de frame tekenprocessen van het onderstaande kanaal. De frame tekenprocessen worden op een enkele machine in niet parallelle vorm uitgevoerd. Het synchronisatie algoritme wordt als lineair proces aangeroepen en uitgevoerd. In de parallelle vorm wordt het synchronisatie algoritme parallel uitgevoerd en zal het meerdere kanalen in een netwerk synchroniseren. Het kanaal is een vereenvoudiging van het bestaande kanaal uit de grafische applicatie programmeertaal zoals die is ontwikkeld door het TNO Fysisch Electrotechnisch Laboratorium. In de latere hoofdstukken zullen de parallelle vormen van het kanaal het synchronisatie algoritme nader aan de orde komen.

De lineaire grafische applicatie bevat een kanaal dat is gekoppeld aan een scènegraaf. Het kanaal is een representatie van de schermgrenzen en schermdiepte. In een grafische applicatie wordt het kanaal vaak een window of scherm genoemd. Het kanaal is lokaal geprogrammeerd en gekoppeld aan de hoofdklasse van de grafische applicatie. Bij activering van het kanaal zal een scherm zichtbaar worden gemaakt waar de bijbehorende scènegraaf op wordt gevisualiseerd. Het onderstaande klassemodel is een applicatie programmeertaal omschrijving voor het programmeren van het kanaal van de lineaire grafische applicatie. Het kanaal is ontwikkeld voor eenvoudige uitbreiding met gedistribueerde grafische clusters. Dit betekent dat de implementatie van het kanaal functies bevat die bij parallelle uitvoer het kanaal parallelliseren, plaatsen en synchroniseren in een netwerk met andere kanalen middels het gedistribueerd synchronisatie algoritme en de datastroom machine.

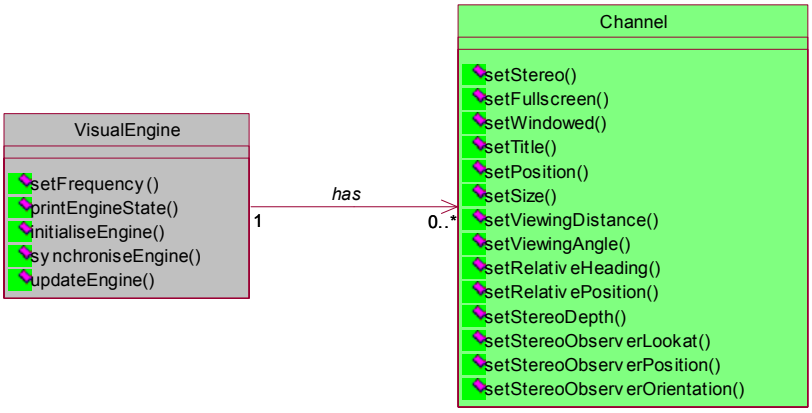


Fig10.0

9.3 Observant model

De lineaire grafische applicatie bevat een waarnemer, of camera, voor het observeren van individuele scènegraaf objecten. Het onderstaande klassemodel is een applicatie programmeertaal omschrijving van de waarnemer. De waarnemer kan zichzelf gemakkelijk koppelen aan en ontkoppelen van scène entiteiten of scèneobjecten. De koppeling en ontkoppeling worden grafisch weergegeven door een dynamisch geprogrammeerde camera die gebruikers duidelijk ruimtelijk inzicht geeft in de richting en oriëntatie van het object waaraan het is gekoppeld. De waarnemer is gelijk aan een waarnemer in de stealth [45] van het TNO Fysisch Electrotechnisch Laboratorium met het enige verschil dat het voor het Linux besturingssysteem is ontwikkeld en er verschillende zichtvelden ingesteld kunnen worden. Ook is een waarnemer in de implementatie van het grafisch cluster slechts koppelbaar aan een enkel entiteit object. Andere scèneobjecten zoals licht- of wereldobjecten zijn niet koppelbaar aan de waarnemer omdat er tijdens het ontwikkeltraject is besloten dat deze koppelingen vrijwel nooit gebruikt zullen worden.

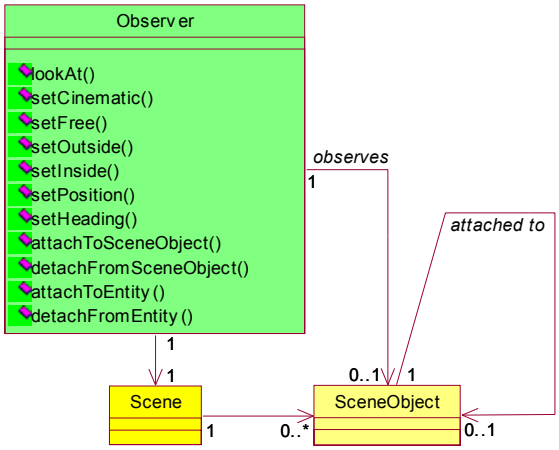


Fig10.1

9.4 Scène model

Het onderstaande klassemodel geeft weer welke scèneobjecten er voor de lineaire grafische applicatie zijn ontwikkeld. De scène is zo ontwikkeld dat het kan worden gedistribueerd. Bij het instantiëren van de scènegraaf herkent de scène klasse of het zich in een cluster bevindt. Tijdens compilatie van een parallelle applicatie zullen er parallelle broncode elementen worden toegevoegd. Bij compilatie van een lineaire grafische applicatie worden de parallelle componenten niet meegecompileerd. De scènegraaf werkt in zowel parallelle als lineaire vorm hetzelfde met uitzondering dat, op een cluster van machines, de aparte instanties van de scènegraaf zich in parallelle vorm synchroniseren. De scèneobject applicatie programmeertaal bestaat uit een groep klassen en eenvoudige methoden die manipulatie van de scèneobjecten toestaan. De methoden zijn zo geprogrammeerd dat ze op bijna ieder punt in het visualisatie proces aangeroepen kunnen worden. Mede hierdoor wordt het mogelijk gemaakt om dynamisch gedrag van objecten te bepalen en in realtime uit te voeren zonder een vooraf bepaalde volgorde.

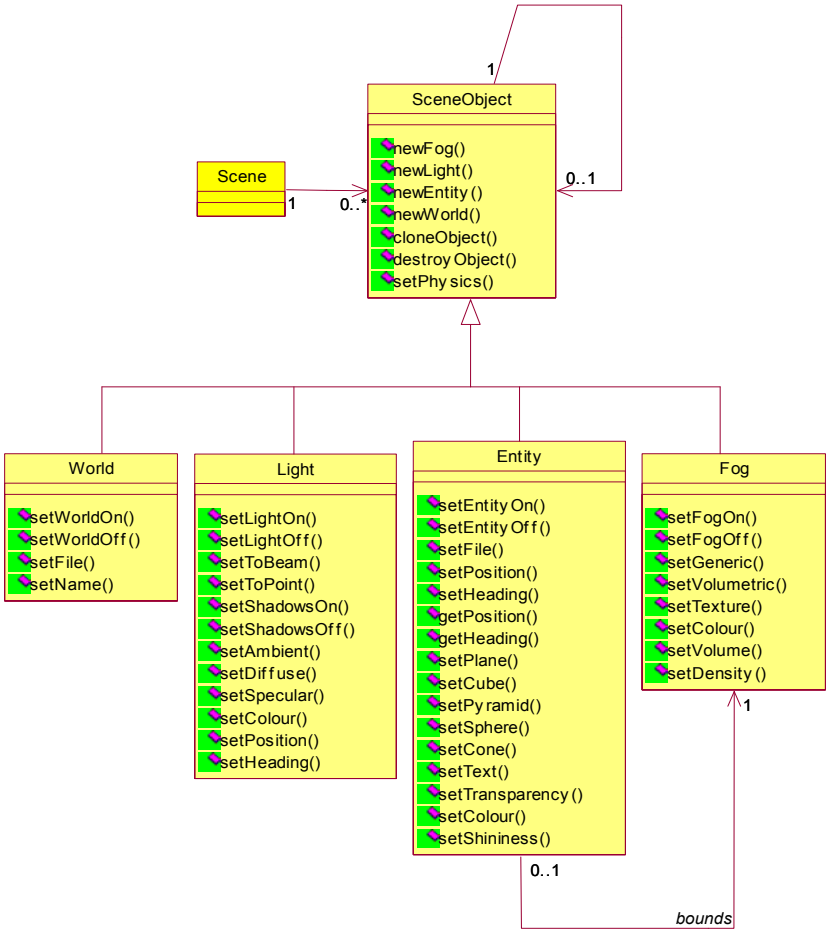


Fig10.2

9.5 Fysisch model

Het onderstaande fysische model is een model voor het definiëren van de positie, oriëntatie, snelheid en acceleratie van een scèneobject en de krachten die werken op een scèneobject. Dit fysische model biedt gebruikers de mogelijkheid om fysische eigenschappen van scèneobjecten te definiëren. Het fysische model biedt ook de mogelijkheid om scèneobjecten uit te breiden met additionele fysische eigenschappen. Het model wordt gerealiseerd in een fysisch object. De gedistribueerde versie van het fysische model kan worden gebruikt bij distributie van scèneobject attributen zoals positie en oriëntatie.

Het fysische model bestaat uit fysische functies die het dynamische gedrag van een object kunnen manipuleren. Deze functies zijn uitvoerbaar met behulp van methoden voor het bepalen van rotatie- of translatie krachten op objecten en methoden voor het bepalen van globalere fysische eigenschappen zoals het type voortstuwings systeem. Het voordeel van het loskoppelen van fysische eigenschappen van de objecten is dat alle operaties die de positie van objecten in de scène beïnvloeden door externe simulators of gespecialiseerde applicaties gerealiseerd kunnen worden zonder dat de objecten daarvoor zelf hoeven te veranderen.

Behalve het bepalen van fysische eigenschappen zoals massa, biedt het fysische object gebruikers de mogelijkheid om elementaire dynamische modellen te definiëren voor bijvoorbeeld helicopter-, fastjet-, tank- of schipsimulaties. Bij het koppelen van een fysisch model aan een scèneobject, zal het scèneobject zich afhankelijk van de ingestelde invoervariabelen volgens het bijbehorende fysische object gedragen. Uitbreiding van het fysische model zal met name op het gebied van fysische gedragseigenschappen en gedetailleerdere translatie- en rotatie berekeningen bevorderend zijn voor de precisie van het dynamische gedrag van het fysische object.

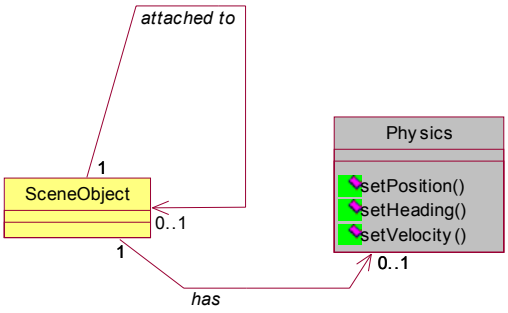


Fig10.3

10 Gedistribueerd visual

Tijdens het onderzoek is er bij alle onderzoekspunten geconcentreerd op de ontwikkeling van een gedistribueerd grafisch cluster. Het gedistribueerd grafisch cluster omvat de lineaire grafisch applicatie programmeertaal voor een enkele machine en een gedistribueerde grafische applicatie programmeertaal voor een cluster van machines. Alle voorheen besproken clustering componenten voor het communiceren van grafische gegevens en synchroniseren van informatiestromen worden met success toegepast in de applicatie programmeertaal voor gedistribueerde grafische applicaties.

De gedistribueerde grafische applicatie programmeertaal is volledig los van de lineaire grafische applicatie programmeertaal te gebruiken. Dit heeft als voordeel dat gebruikers met civiele- computers of netwerken, gedistribueerde of niet gedistribueerde applicaties kunnen ontwikkelen. De gedistribueerde grafische applicatie programmeertaal is ontworpen met fast prototyping als uitgangspunt om applicaties in zeer hoog tempo te ontwikkelen. De applicatie programmeertaal stelt iedere gebruiker in staat om met minder dan vijftig regels broncode een simulatie demonstratie te schrijven voor een cluster van machines.

De lineaire grafische applicatie is een volledige grafische applicatie programmeertaal voor het programmeren van drie dimensionale grafische applicaties. De applicatie programmeertaal omvat generieke functies voor het genereren van scène elementen en bevat verder uitbreidingsmogelijkheden voor gekoppelde fysische componenten. Het bijgevoegde fysisch component bevat direct beschikbare dynamische modellen om de dynamiek van helicopter-, tank-, en fastjet simulaties na te bootsen. Het fysische component en alle andere componenten zijn zo ontwikkeld dat ze zichzelf aanpassen in een parallelle omgeving.

Distributie van onderdelen van het fysische object is één van de parallellisaties van de bestaande lineaire grafische applicatie. Tijdens het parallelliseren van de lineaire applicatie is tot in detail onderzocht welke gegevens en welke functies geparallelliseerd moesten worden. Het resultaat van het onderzoek is een datacommunicatie machine, een synchronisatie algoritme en een interpolatie algoritme voor respectievelijk het synchroniseren van gegevensstromen en het synchroniseren en interpoleren van processen. Om de prestaties zo hoog mogelijk te houden is er door een selecte groep wetenschappers van het TNO Fysisch Electrotechnisch Laboratorium een keuze gemaakt uit gegevenselementen die op het hoogste niveau een minimaal effect hebben op de realtime prestaties van objecten in de gedistribueerde scènegraaf.

In dit hoofdstuk worden de ontwikkelde componenten gedetailleerder omschreven. Verder wordt de ontwikkeling van ieder applicatie component behandeld, en worden de keuzen die zijn gemaakt verhelderd. Ook worden er in dit hoofdstuk voorstellen gemaakt van verbeterde algoritmen die een nog verdere ontwikkeling en verfijning van de gedistribueerde grafische applicatie mogelijk maken. De technische details die betrekking hebben op de applicatie zullen worden verklaard.

10.1 Datacommunicatie machine

Voor het parallelliseren van de verschillende grafische processen is een realtime datacommunicatie machine ontworpen, deze machine is in de voorgaande hoofdstukken globaal aan de orde gekomen. De machine bevat componenten die gebruikers in staat stelt om met herbruikbare functies gedistribueerde grafische applicaties te programmeren. De datacommunicatie machine bevat componenten die gebruikt kunnen worden voor het ontwikkelen van verschillende soorten cluster applicaties.

De MPICH functies die zijn gebruikt bestaan voor het grootste deel uit collectieve operaties. Collectieve operaties verminderen niet alleen de kans op applicatiefouten [27] maar verminderen ook de complexiteit en hoeveelheid broncode waaruit een programma bestaat [27]. Voor realtime prestaties van het systeem is gekozen voor een civiel 100MBps switched netwerk zonder additionele realtime hardware componenten. Het civiele netwerk kan worden uitgebreid met 1000MBps netwerk apparatuur voor een aanzienlijke prestatieverbetering. De datacommunicatie machine is niet realtime vanwege het feit dat de MPICH functies nog geen MPI/RT implementaties kent.

Uit onderzoek naar realtime systemen, is gebleken dat de global virtual realtime in realtime netwerken sterk afneemt naarmate er meer machines worden toegevoegd aan een cluster [15]. Ook is gebleken dat in Linux realtime omgevingen, netwerkkaart besturings systemen moeten worden aangepast zodat de pakket buffers op nul komen te staan [15] om zo snel mogelijk realtime pakketten te versturen en ontvangen. Verdere ontwikkelingen op het gebied van MPI/RT [46] zullen realtime netwerken mogelijk maken. De ontwikkelde machine functies zijn ontwikkeld voor een niet realtime op MPI gebaseerde netwerkomgeving en maken gebruik van lokaal gemeten tijdsverschillen die over het netwerk gesynchroniseerd worden. Het is echter van belang dat de communicatie machine wordt vernieuwd zodra er MPI/RT implementaties van MPICH beschikbaar zijn.

In de volgende hoofdstukken komen de machine architectuur, het machine applicatiemodel en de machine applicatie programmeertaal naar voren. De machine architectuur is een weergave van de machine componenten. Het applicatie model is een object georiënteerd model voor de machine applicatie programmeertaal. De architectuur en applicatie componenten worden gebruikt in de parallel gedistribueerde grafische applicatie zoals die is ontwikkeld voor het TNO Fysisch Electrotechnisch Laboratorium. Metingen in het laatste hoofdstuk tonen aan dat de architectuur binnen de eisen voor realtime van de grafische applicatie blijven.

10.1.1 Machine architectuur

De machine architectuur geeft op een hoog niveau weer welke gegevensstroom typen er zijn en welke additionele functionaliteit de datacommunicatie machine biedt. De datacommunicatie machine biedt vijf basis gegevensstroom typen en één samengesteld gegevensstroom type. De basis gegevensstroom typen bestaan uit float, double, integer, long en string gegevensstromen. Het samengesteld gegevensstroom type bevat alle basisgegevensstromen. Ieder gegevensstroom type scaleert verticaal op basis van de gegevensstroom batch grootte. De bron van het gegevensstroom type kan worden bepaald of in realtime worden veranderd door de applicatie programmeur. Behalve de mogelijkheid tot het communiceren van gegevens bevat de machine ook elementaire functionaliteiten voor bijvoorbeeld het controleren van machine identificatie nummers. De datacommunicatie machine is op zich afhankelijk van een deelverzameling van de functies afkomstig uit de MPICH bibliotheek. De deelverzameling bestaat voor het grootste deel uit collectieve functies.

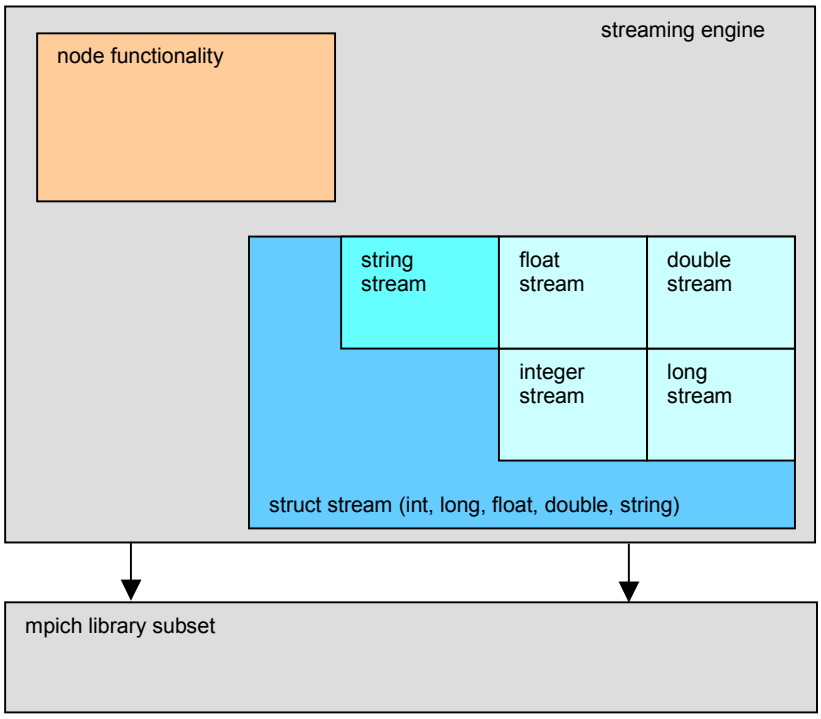


Fig13.1

10.1.2 Applicatie model

Het applicatie model van de datacommunicatie machine geeft weer welke applicatie programmeerklassen en methoden er zijn om de machine te besturen. De applicatie componenten zijn zo eenvoudig mogelijk ontworpen om direct gebruik van de datacommunicatie machine mogelijk te maken. Ook is de applicatie programmeertaal zo ontwikkeld dat het toegankelijk wordt voor een grote groep applicatie programmeurs. Gebruik van de datacommunicatie machine zal voornamelijk bestaat uit het initialiseren, opvragen en instellen van de gegevensstromen. De hoofdprocessen van de datacommunicatie machine zijn met andere processen interpoleerbaar op broadcast niveau. Een applicatie programmeur kan met de interpoleerbare functies een vorm van interpolatie introduceren waardoor andere processen gebruik kunnen maken van het gedistribueerd systeem tussen communicatie processen door.

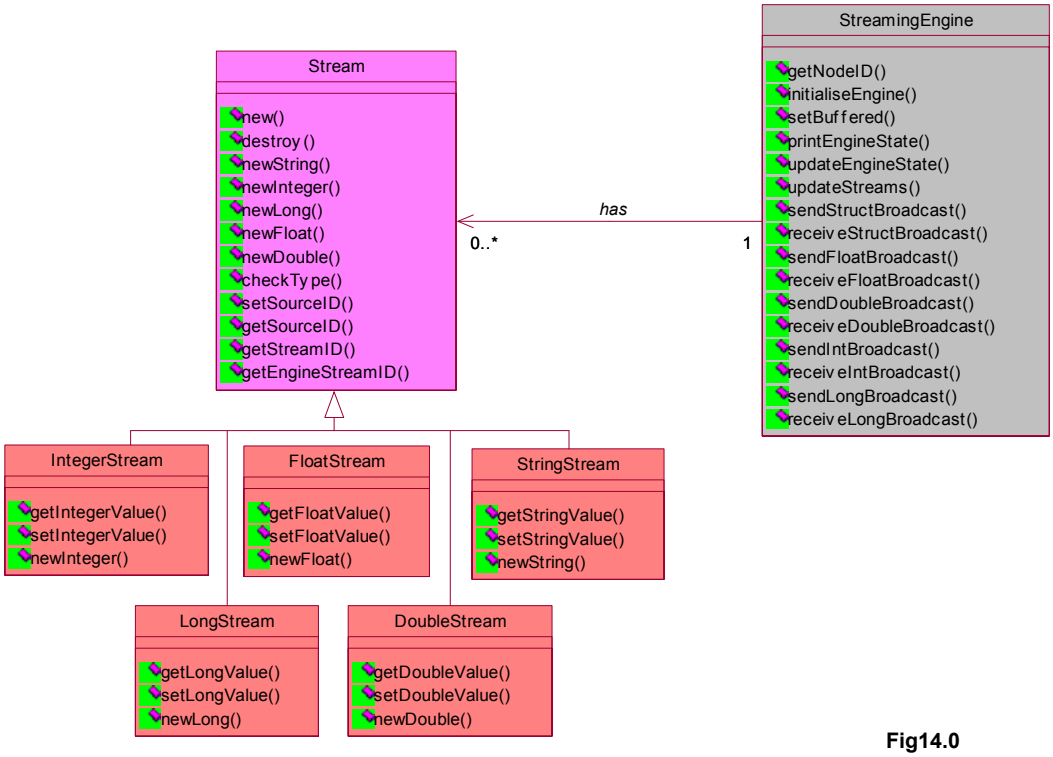


Fig14.0

10.1.3 Machine broncode

Hieronder is een codefragment gegeven van de datacommunicatie machine zoals die is ontwikkeld voor het gedistribueerd grafisch cluster. Opmerkelijk is de eenvoud van de broncode en de toepassing met verschillende typen gegevensstromen. De onderstaande broncode is een voorbeeld van een toepassing ontwikkeld met een functioneel georiënteerde programmeertaal. Voor de uiteindelijke applicatie programmeertalen zijn beiden functionele, in de vorm van C, en object georiënteerde, in de vorm van C++, programmeertalen gerepresenteerd. De onderstaande broncode is een voorbeeld van een werkende applicatie ontwikkeld met de C programmeertaal.

```
StreamingEngine_Initialise(iArgumentCount, pacArgumentContents);

iSelfNodeID = StreamingEngine_GetNodeID();

Stream* pStream_stream0 = Stream_NewString(NODE00);
Stream* pStream_stream1 = Stream_NewLong(NODE01);
Stream* pStream_stream2 = Stream_NewDouble(NODE02);

char acValue[20] = NULL;
long lValue = 0;
double dValue = 0;

while(iCounter < 10)
{
  Stream_SetStringValue(pStream_stream0, "NORMAL STRING");
  Stream_GetStringValue(pStream_stream0, &acValue);
  Stream_SetLongValue(pStream_stream1, (long) iCounter);
  Stream_GetLongValue(pStream_stream1, &lValue);
  Stream_SetDoubleValue(pStream_stream2, (double) iCounter/10);
  Stream_GetDoubleValue(pStream_stream2, &dValue);

  if (iCounter == 5)
  {
    Stream_SetSourceID(pStream_stream0, NODE01);
    Stream_SetSourceID(pStream_stream1, NODE00);
  }

  StreamingEngine_Update();
  iCounter = iCounter + 1;
}

Stream_Destroy(Stream_stream0);
Stream_Destroy(Stream_stream1);
StreamingEngine_PrintState(NODE00);

StreamingEngine_Destroy();
```

De gebruikte MPI functies van de machine bestaan voor het grootste deel uit generiek collectieve functies. Een collectieve functie is een functie waarmee een gedeelde operatie gerealiseerd kan worden. Een gedeelde operatie bestaat uit een verzameling operaties die samen een enkel doel realiseren. De collectieve functies zijn gebruikt om de eenvoud en efficiëntie van de applicatie programmeertaal te vergroten en de kans op fouten van de applicatie te elimineren [27]. Ook neemt de hoeveelheid broncode dankzij het gebruik van collectieve functies sterk af en wordt de inhoud leesbaarder gemaakt [27]. Ten opzichte van programmacode die dezelfde communicatieprocessen in versturende- en ontvangende functies realiseert is de collectieve benadering bevorderend voor de generieke hanteerbaarheid van gegevensstromen.

10.2 File systeem

Het systeem voor bestandsmanipulatie omvat beiden parallelle en lineaire functionaliteiten voor het lezen en schrijven van bestanden. Ook is men met het bestandsysteem in staat om berichten te ontvangen en te versturen, wat gebeurt met behulp van functies die communicatie processen direct aanroepen. Bij de datacommunicatie machine gebeurt dit anders en worden de communicatie functies eerst gerangschikt en daarna pas aangeroepen in het hoofdproces van de machine.

Het bestandsysteem bevat functie definities voor het inlezen en wegschrijven van bestanden. De nieuwe MPI2 standaard stelt gebruikers in staat om parallel opgeslagen gegevens aanzienlijk sneller dan lineair opgeslagen gegevens weg te schrijven en in te lezen [28] over een netwerk. Behalve het parallel schrijven en lezen van gegevens stelt de standaard gebruikers in staat om lokaal geheugenruimten of bestandwijzers te gebruiken voor het lezen van geselecteerde stukken gegevens uit bestanden [28].

Deze benadering kan gebruikt worden voor het benaderen van zeer grote bestanden zoals virtuele landschappen. Het is duidelijk dat het bestandsysteem gebruikers in staat moet stellen om zeer grote gegevensverzamelingen in stukken te benaderen. De stukken van de bestanden worden bepaald door middel van geheugendefinities. Een geheugendefinitie geeft aan welk gedeelte van een bestand in het geheugen aanwezig moet zijn.

Voor de implementatie zouden single independent noncontiguous requests [28] gebruikt moeten worden voor het inlezen van verschillende stukken gegevens die per machine verschillen maar uit hetzelfde bestand afkomstig zijn. Het voorgaande request type stelt processen in staat om ieder afhankelijk van elkaar gegevens in te lezen in lokale geheugengebieden. De lokale gegevens zouden stukken landschap of virtuele werelden kunnen representeren.

In de volgende paragraaf bevindt zich een architectuurmodel van gedistribueerde gegevens in twee vormen, gerepliceerd en parallel opgeslagen. De twee soorten gegevens moeten benaderbaar gemaakt worden via de applicatie programmeertaal voor filesystemen. Bij het opslaan en opvragen van parallel of niet parallel opgeslagen gegevens is conversie impliciet. Dit betekent dat indien er een niet parallel bestand bestaat en dat parallel wordt opgeslagen, het bestand automatisch geparallelliseerd wordt. Hetzelfde geldt voor het inlezen van parallelle bestanden en het wegschrijven als lokaal bestand.

10.2.1 Filesysteem architectuur

De onderstaande modellen geven weer hoe gegevensstromen een schakeling maken van een linear naar een parallel bestand en van een parallel naar een lineair bestand en beschrijven hoe bestandsfragmenten middels een vooraf gedefinieerde geheugenruimte in lokale machines worden ingelezen. De onderstaande modellen geven het dynamische gedrag weer van de architectuur van het bestandsysteem op een cluster van civiele computers.

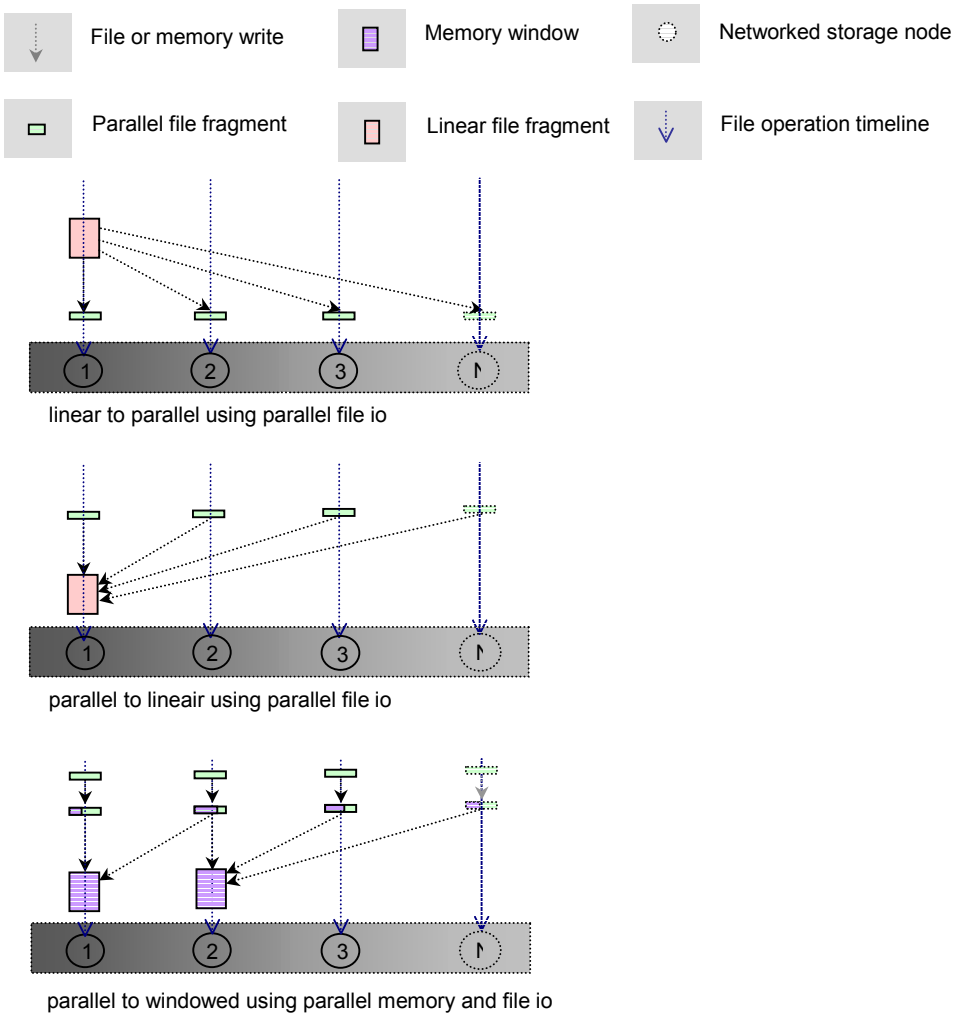


Fig15.0

10.2.2 Applicatie model

Het bestandsapplicatie model biedt een applicatie programmeertaal in de vorm van klassen en methoden voor parallelle opslag en opvraging van bestanden en het parallelliseren of lineariseren van bestanden. Deze applicatie programmeertaal is ontwikkeld om geheugenruimtes van alle machines in een cluster van civiele systemen te kunnen gebruiken. De applicatie programmeertaal zal uiteindelijk gebruikt worden voor het parallel lezen en schrijven van zeer grote bestanden in het gedistribueerd grafisch cluster. Behalve bestandsfunctionaliteit, biedt de applicatie programmeertaal een aantal basisfunctionaliteiten voor communicatie en identificatie. Voor implementatie van het bestandsysteem moet worden gelet op het type gegeven dat het bestandsysteem inleest en wegschrijft. Het ideale model stelt gebruikers in staat om bestandsdelen van willekeurige bestandstypen lineair of parallel in te lezen of weg te schrijven en stukken daarvan te plaatsen in lokaal geheugen.

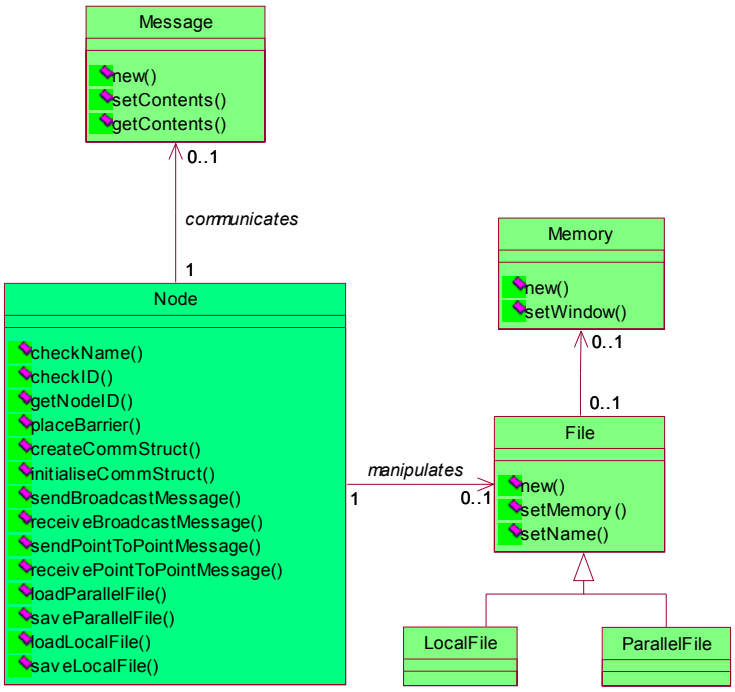


Fig16.0

10.3 Gedistribueerd visual

Het grafisch cluster bevat controle- en gegevensstroom componenten, in de vorm van controle- en communicatie functies. De controle groep is ontworpen om vanuit individuele machines of vanuit alle machines controle te kunnen krijgen over de functionaliteiten van scènegraaf elementen of deze elementen te instantiëren of vernietigen. De gegevensstroom groep gebruikt de datacommunicatie machine om verschillende randapparaten of simulators die niet gedistribueerd of gedeeltelijk gedistribueerd zijn te kunnen laten communiceren met scènegraaf elementen.

Bij de controle groep zijn controle functies in broncode vorm aanwezig op alle machines in het cluster. Bij de gegevensstroom groep spreken we van gegevens die in de eerste instantie slechts zichtbaar zijn op de machine waar ze zijn gegenereerd. De elementaire gegevenstransformatie die gedaan wordt voor het benaderen van deze gegevens vanuit de machines waar de gegevens niet zichtbaar zijn, is het globaliseren of broadcasten van de gegevens over alle machines.

Het gedistibueerd grafisch cluster bestaat technisch gezien uit extensies van de niet gedistribueerde grafische applicatie. De extensies gebruiken gegevensstromen die zijn ontwikkeld met de applicatie programmeertaal van de datacommunicatie machine. De gegevensstromen zijn representaties van voornamelijk scèneobjecten uit de grafische applicatie. De scèneobjecten worden initieel geïnstantieerd tijdens generatie van objecten; deze objecten zijn dan overal zichtbaar in de gerepliceerde scènegraaf van de grafische applicatie. Nadat een scèneobject is gegenereerd, worden er gegevensstromen geïnstantieerd die de objecten vanuit een gegevensbron controleren.

De geïnstantieerde gegevensstromen zijn uniek voor ieder objecttype en representeren het object met gegevensstromen uit de datacommunicatie machine. Tijdens uitvoer van een hoofdproces wordt ook het hoofdproces van de datacommunicatie machine aangeroepen of worden processen geïnterpoleerd met communicatie processen afhankelijk van de instelling. De verantwoordelijkheid voor het regelen van gegevensstromen ligt hierdoor geheel bij de datacommunicatie machine, waardoor scèneobjecten apart benaderd kunnen worden zonder herprogrammering van de gegevensstromen. Door het apart verwerken van de gegevensstromen op lager communicatie niveau kunnen er veranderingen worden aangebracht aan de datacommunicatie machine zonder dat het gedistribueerd grafisch cluster veranderd hoeft te worden. Dit maakt hergebruik en modulariteit van het gedistribueerd grafisch systeem aanzienlijk eenvoudiger.

De volgende paragrafen geven een architectuur van de realtime regelaars en gedistribueerde componenten van de grafische applicatie, waaronder het gegevensstroom-, kanaal-, waarnemer-, scène-, en fysische model. De modellen geven een beeld van de programmeerklassen voor de verschillende gedistribueerde grafische componenten.

10.3.1 Proces synchronisatie

Om realtime synchronisatie over een netwerk van civiele machines mogelijk te maken is een regelaar ontwikkeld die over een cluster van machines realtime synchronisatie toepast, deze regelaar is in eerdere hoofdstukken gedetailleerd besproken. De parallelle versie van de realtime regelaar is een uitbreiding van de realtime regelaar voor de lineaire grafische applicatie. Het gedistribueerd grafisch cluster maakt gebruik van de parallelle realtime regelaar om de frame tekenprocessen en applicatie berekeningsprocessen in het grafisch cluster parallel te laten lopen.

De realtime regelaar wordt gedistribueerd over een netwerk van civiele machines en parallel uitgevoerd. Het model horende bij de distributie is hieronder weergegeven. De gedistribueerde realtime regelaar bestaat uit replicaties van de regelaar uit de vorige hoofdstukken. De gedistribueerde regelaar wordt over een synchronisatie netwerk apart gesynchroniseerd en voorspelt het verschil tussen de synchronisatie momenten op dusdanige wijze dat er tijdens uitvoering van het parallelle regelalgoritme vrijwel geen zichtbaar verschil tussen de gesynchroniseerde processen is te ontdekken. Dit wordt duidelijk in de meetresultaten die in het laatste hoofdstuk naar voren komen.

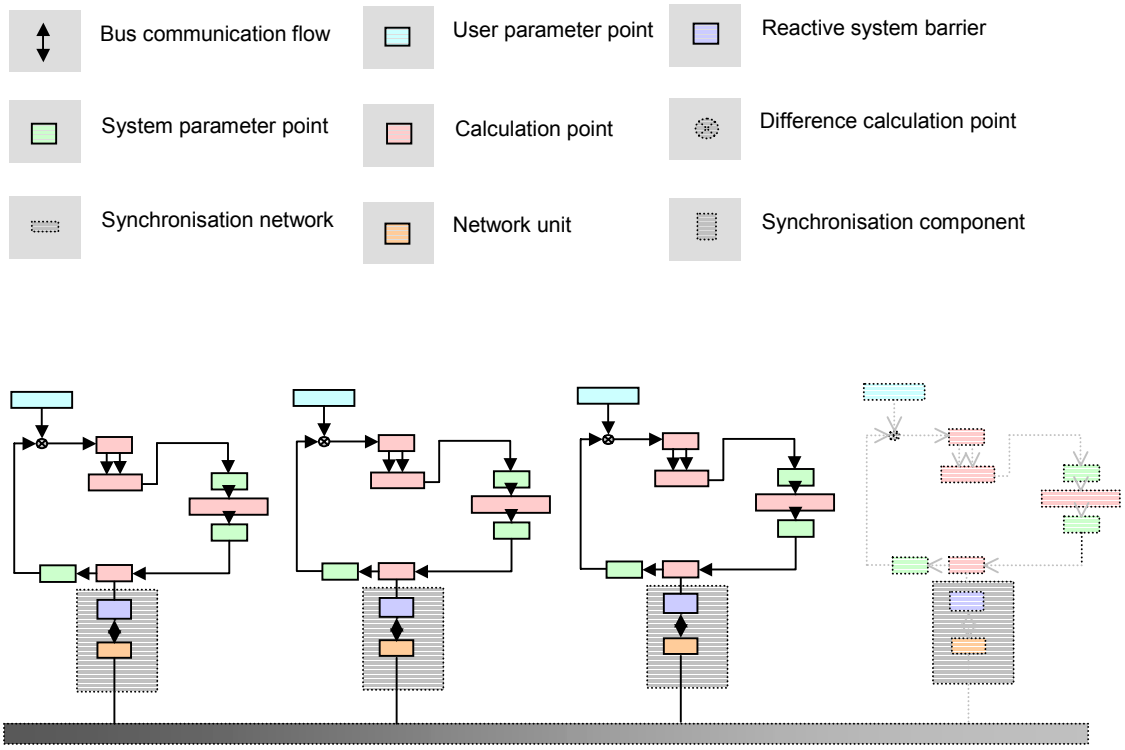


Fig17.1

10.3.2 Data communicatie

Behalve het regelen van synchronisatiemomenten voor met name de frame tekenprocessen, is er ook een gegevensstroom architectuur ontworpen voor gedistribueerde communicatie. De gegevensstroom architectuur is een weergave van de datacommunicatie machine over een cluster van civiele computers. De datacommunicatie machine is een parallel uitvoerbaar programma en is in de voorgaande hoofdstukken uitgebreid aan bod gekomen. De architectuur hieronder geeft weer op welk moment tijdens de uitvoer van de programmacode, gegevenssynchronisatie van de datacommunicatie machine zal plaatsvinden. De gegevenssynchronisatie kan worden geïnterpoleerd met andere processen om de algehele applicatiesnelheid en controle op proces frequenties te verbeteren.

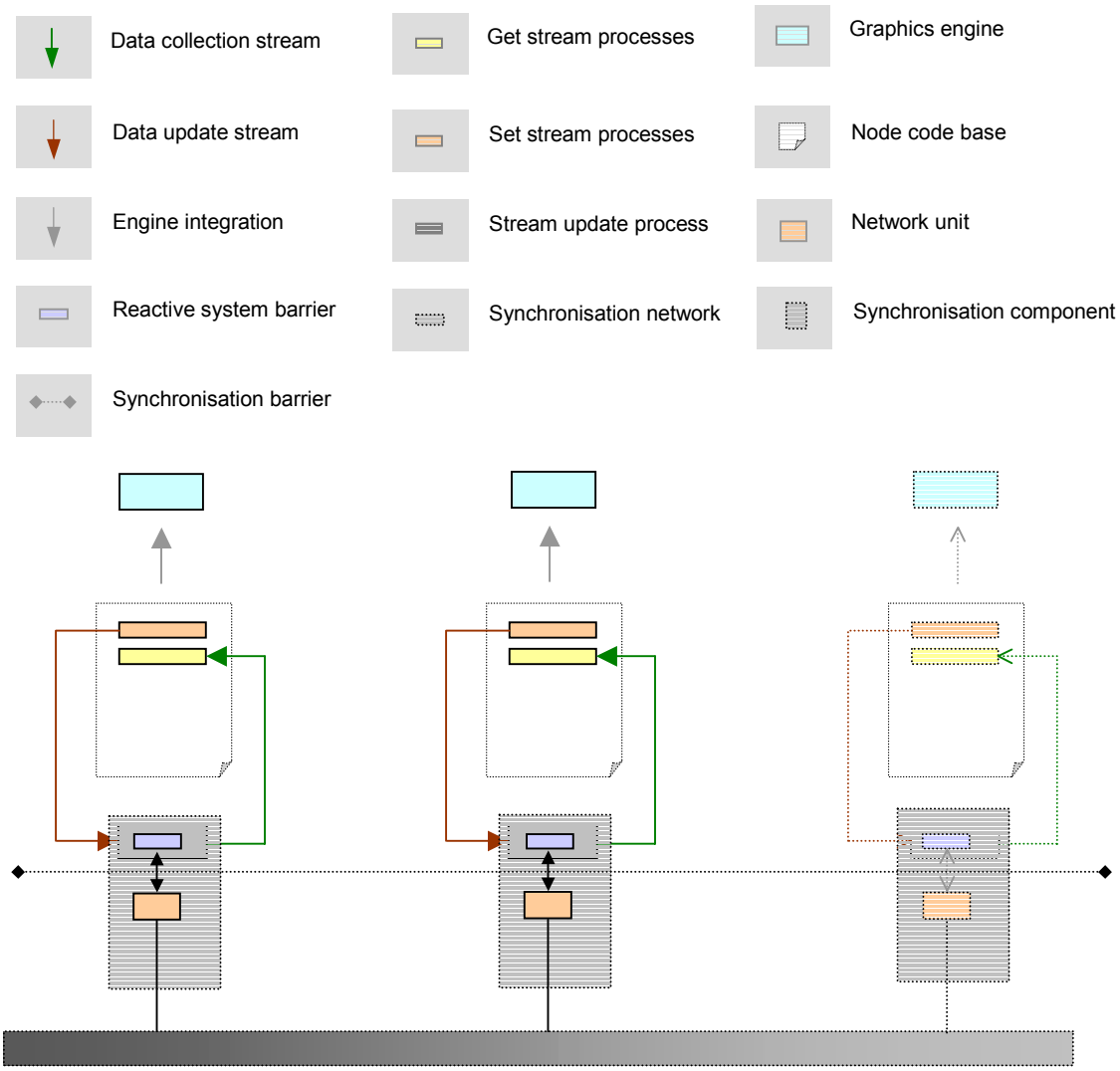


Fig18.0

10.3.3 Datastroom model

Het datastroom model, of gegevensstroom model, is een uitbreiding van de grafische applicatie met parallelle gegevensstroom elementen. Deze elementen maken gebruik van de datacommunicatie machine en zijn een representatie van de informatiestromen die een relatie hebben met de grafische objecten waaraan ze zijn gekoppeld. De gegevensstroom typen zijn representaties van de informatie van gerelateerde scèneobjecten in een gedistribueerde grafische applicatie. Tijdens het ontwikkelen van de parallelle applicatie programmeertalen zijn er gegevensstromen gedefinieerd die het gedrag van de parallelle simulators weergeeft op een hoger niveau middels de gedistribueerde grafische objecten. Met het gegevensstroom model kunnen de simulator componenten op eenvoudige wijze gedistribueerd worden over een cluster van systemen. De gegevensbron wordt door het gegevensstroom object bepaald en is gekoppeld aan een bepaalde machine. Hieronder is het applicatie model van de gegevensstroom klassen voor de geparallelliseerde versie van de lineaire grafische applicatie.

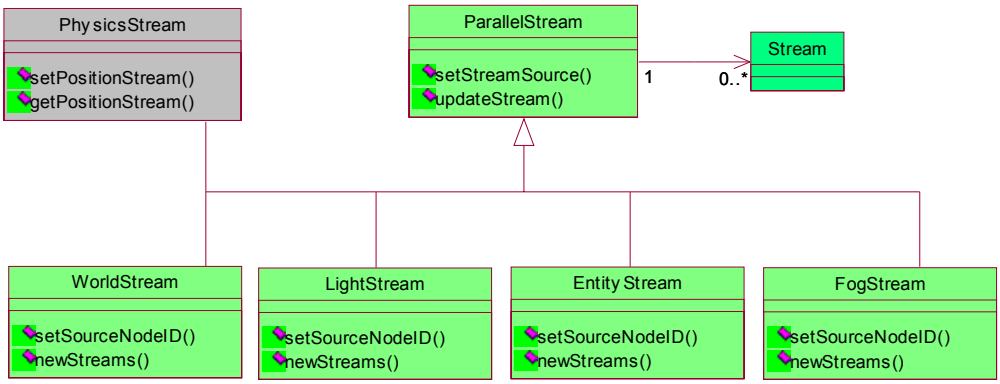


Fig19.0

10.3.4 Kanaal model

Het kanaal model is een applicatie model van de klassen voor het gedistribueerde kanaal en de gedistribueerde waarnemer van de applicatie programmeertaal voor het grafisch cluster. Het gedistribueerde kanaal wordt parallel uitgevoerd en geeft controle over het gedrag van het kanaal op een willekeurige machine. Het kanaal wordt gesynchroniseerd door middel van de realtime regelaar. Iedere machine kan een andere kanaal instelling hebben en bij ieder kanaal kan een aparte waarnemer horen. Een gedistribueerde waarnemer, of gedistribueerde camera, kan globaal zijn, waardoor alle kanalen met hetzelfde cameramodel werken, of lokaal, waardoor kanalen ieder met een eigen camera model werken. Het camera model wordt ingesteld op een specifieke machine en kan als globaal of lokaal camera model dienen. Bij activatie van het globale camera model bevinden camera eigenschappen zich op een lokale machine en worden deze gedistribueerd over het cluster. Bij deactivatie van het globale camera model zullen de kanalen ieder hun eigen camera model gebruiken. Hieronder bevinden zich de applicatie programmeerklassen voor het kanaal, de waarnemer en de grafische hoofdapplicatie.

Fig19.1

10.3.5 Observant model

Het model voor de gedistribueerde waarnemer en de bijbehorende relaties zijn hieronder gedetailleerder weergegeven en gekoppeld aan scèneobjecten. Het verschil met de waarnemer voor lineaire grafische applicaties is dat een gedistribueerde waarnemer gekoppeld kan worden aan gedistribueerde scèneobjecten. Bij een koppeling zal een gedistribueerde waarnemer gekoppeld worden aan een gedistribueerd scène entiteit. Bij koppeling aan een gedistribueerd scène entiteit wordt het camera model ingesteld om het scène entiteit te observeren volgens het actieve camera model. De positie en oriëntatie informatie worden hierbij verdeeld en gesynchroniseerd over alle machines in het grafisch cluster. Hieronder bevinden zich de applicatie programmeerklassen van de gedistribueerde waarnemer zoals die is ontwikkeld voor het gedistribueerd grafisch cluster.

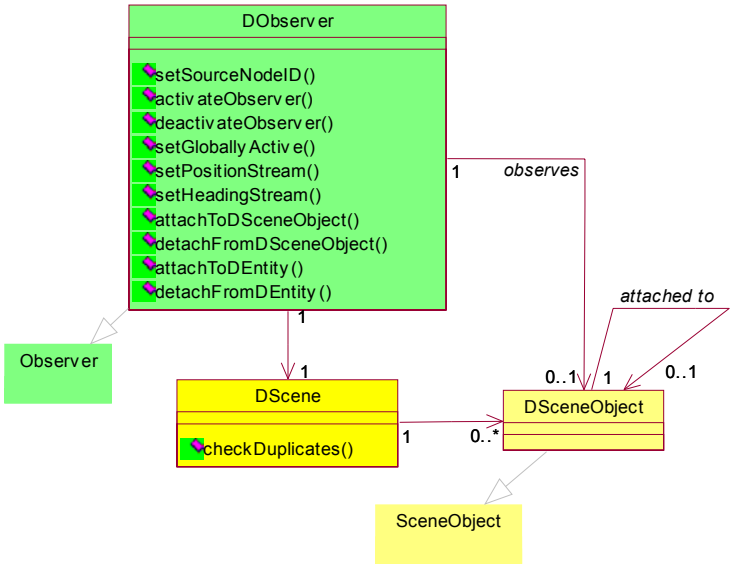


Fig19.2

10.3.6 Scènegraaf model

Het scènegraaf model is een weergave van de klassen voor gedistribueerde scèneobjecten. De gedistribueerde scènegraaf is een gerepliceerde scènegraaf die vanuit verschillende machines naar alle objecten in het grafisch cluster verwijst. Ieder scènegraaf element is apart gekoppeld aan een gegevensstroom die een grafisch object representeert. De object gegevensstromen stellen gebruikers in staat om de scèneobjecten van de grafische applicatie gedistribueerd te programmeren. De scènegraaf zal in zijn geheel gerepliceerd worden over alle machines in het gedistribueerd grafisch cluster. Door replicatie wordt scènegraaf management overbodig, en kunnen scèneobjecten vanuit elke machine benaderd worden.

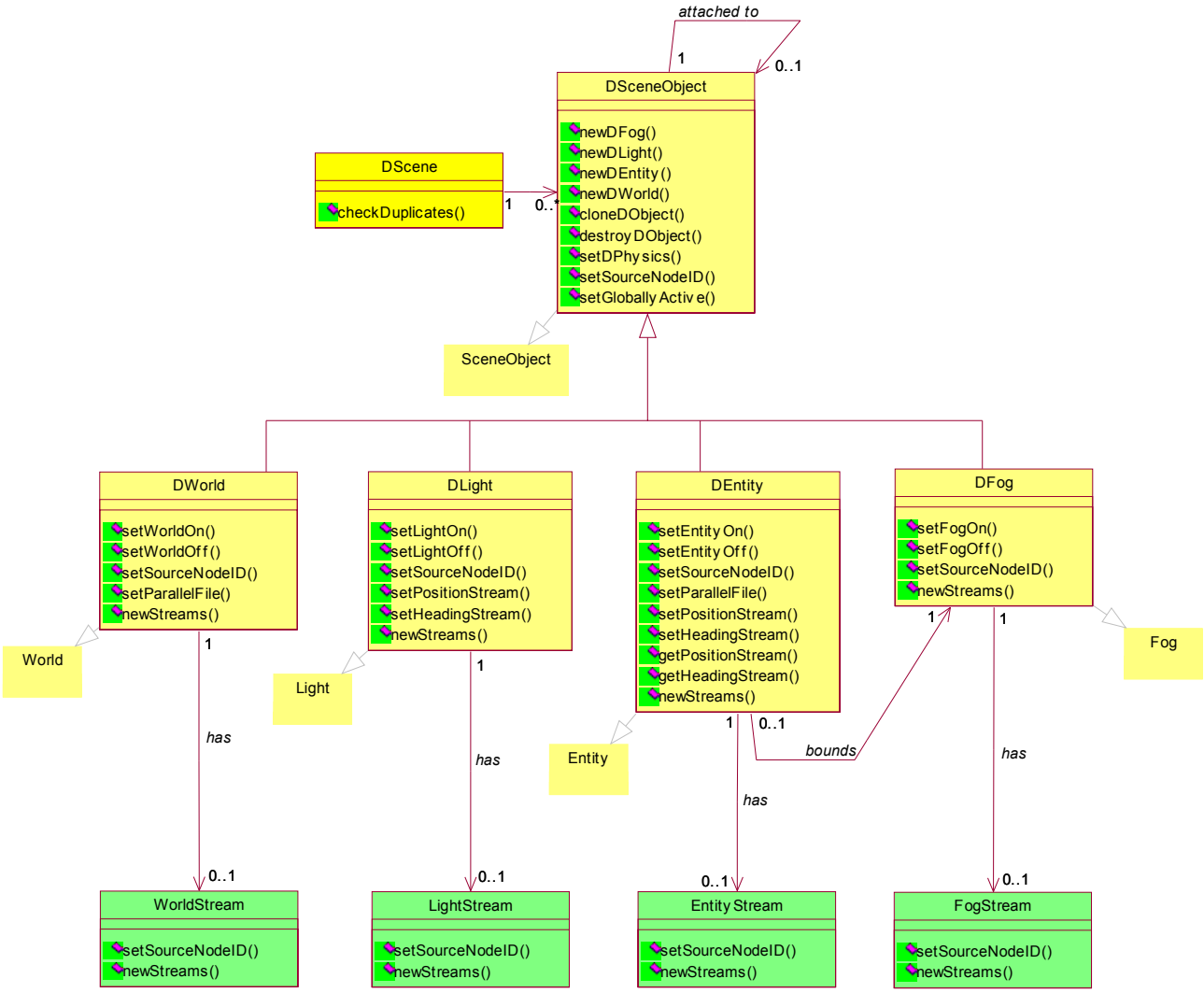


Fig19.3

10.3.7 Fysisch model

Het gedistribueerd fysische model omschrijft een applicatie programmeertaal die gebruikt kan worden om het gedistribueerde gedeelte van het fysische object te manipuleren. De fysische gegevensstroom klassen bieden een groep methoden voor het gedistribueerd programmeren van positie en oriëntatie informatie. Distributie van slechts de positie en oriëntatie van een object betekent dat alle fysische gegevens behalve positie en oriëntatie lokaal op een door de gebruiker ingestelde machine worden berekend. De resultaten van deze lokale berekeningen worden als positie en oriëntatie informatie verzonden over het netwerk. De positie en oriëntatie worden gesynchroniseerd met behulp van de datacommunicatie machine. Het bereik van de gegevensstromen van het fysische object is gelimiteerd tot positie en oriëntatie informatie om de doorvoer van fysische gegevens te minimaliseren. Uitbreiding van de positie en oriëntatie informatie gegevensstroom typen met andere gegevensstroom typen kan gedistribueerde simulators mogelijk maken waarbij ook informatie als krachten op objecten en massa worden doorgegeven voor gebruik in gedistribueerde simulator componenten. In het volgende diagram is een klasse representatie van het fysische model gegeven.

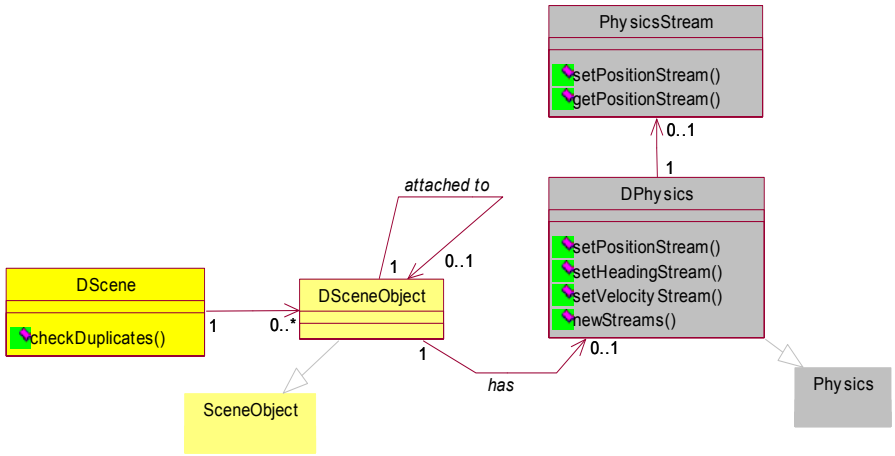


Fig19.4

11 Visual integratie

Het gedistribueerd grafisch cluster is ontwikkeld in een commerciëel kader, en zal daarom ook slechts bruikbaar zijn voor commerciële doeleinden. Om de bestaande grafische applicatie zoals die is ontwikkeld door het TNO Fysisch Electrotechnisch Laboratorium te integreren met de ontwikkelde en ontworpen componenten, moet de grafische applicatie met gespecialiseerde en gedistribueerde klassen worden uitgebreid.

Voor integratie zijn een drietal modellen ontwikkeld die ieder een apart domein van de grafische applicatie parallelliseren. Het eerste model omvat een specialisatie van de grafische hoofdklassen zelf. Het tweede model is een parallellisatie van de nu nog lineaire bestandsystemen voor drie dimensionale objecten. Het laatste model omvat een benadering van realtime gegevensstromen zoals die ook in de ontwikkelde applicatie zijn gedemonstreerd.

De grafische applicatie moet verder worden uitgebreid met realtime componenten voor frame- en gegevenssynchronisatie. Deze realtime componenten zullen slechts in de vorm van hardware een goede uitkomst bieden voor het correct functioneren van de grafische applicatie in een militaire simulatie omgeving. Een van de eisen van de hardware is dat zij POSIX realtime functies biedt aan de applicatie ontwikkelaar. Het parallel grafisch cluster demonstreert enkele realtime componenten die voor de specialisatie van de grafische applicatie van TNO Fysisch Electrotechnisch Laboratorium hergebruikt kunnen worden. De belangrijkste componenten zijn hierbij de datacommunicatie machine en de realtime barrier.

Voor beeldsynchronisatie biedt gespecialiseerde synchronisatie hardware een goede oplossing. Deze hardware bestaat uit kaarten die een apart synchronisatie signaal genereren. Voor commerciële en militaire toepassingen kan de RedHawk architectuur worden gebruikt. Behalve de RedHawk is de Silicon Graphics Onyx4 architectuur ook bruikbaar voor dergelijke toepassingen. Door middel van hardware verbindingen wordt het synchronisatie signaal gebruikt om de actieve tekenprocessen in het cluster gesynchroniseerd te laten lopen.

De volgende paragrafen stellen specialisaties van de klassen van de bestaande grafische applicatie voor, waar de gespecialiseerde klassen de grafische applicatie programmeertaal uitbreiden met parallelle componenten. Verder zullen er een parallel bestandsysteem en communicatiemodel worden besproken die de grafische applicatie uitbreiden met componenten die het mogelijk maken om grafische applicaties te ontwikkelen voor een gedistribueerd cluster.

11.1 Distributie model

Her onderstaande model is een weergave van specialisaties van klassen die nodig zijn om de huidige grafische applicatie te distribueren. De specialisaties van klassen bestaan uit uitbreidingen voor de objecten, de waarnemer, het kanaal en de hoofdklassen van de grafische applicatie. De specialisaties bieden synchronisatie mogelijkheden aan en kennen gegevensstromen toe aan scèneobjecten om gedistribueerd programmeren van de scèneobjecten mogelijk te maken.

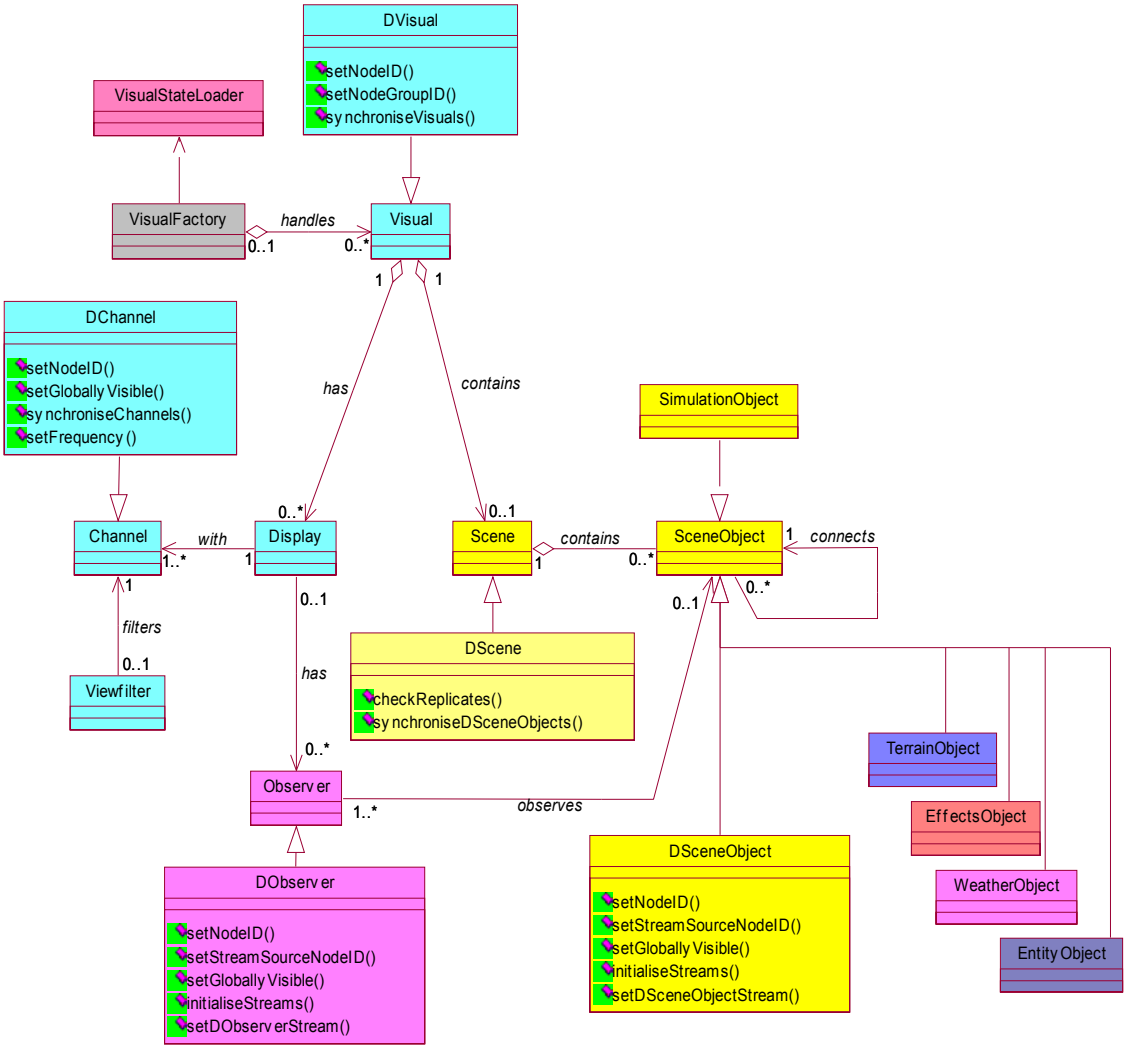


Fig20.0

11.2 Filesysteem model

Het parallelle bestandsysteem breidt de bestaande bestandsystemen, die bestaan uit NFS methoden, uit met methoden voor het parallel lezen en schrijven van bestanden. De uitbreiding is een implementatie van klassen en methoden die gebruik kunnen worden voor het lezen en schrijven van bestanden gebaseerd op de Message Passing Interface. Door uitbreiding van het grafisch cluster met klassen en methoden voor parallelle bestandsmanipulatie kan de opslagcapaciteit van het gehele grafische cluster gebruikt worden. Hieronder bevinden zich de klassen voor manipulatie van het parallelle bestandsysteem en de specialisaties die de klassen uitbreiden met lokale en parallelle bestanden.

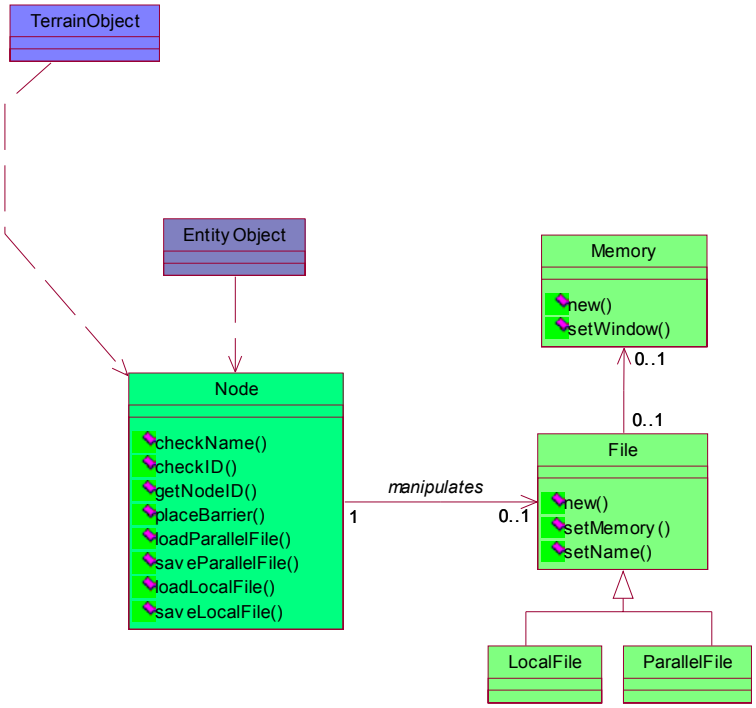


Fig20.1

11.3 Datacommunicatie model

Het datacommunicatie model breidt de bestaande grafische applicatie uit met methoden en klassen voor gedistribueerde communicatie. Parallelle gegevensstroom typen worden toegevoegd aan gedistribueerde componenten en aan aparte onderdelen van de scèneobjecten en scène waarnemers. Bij het instantiëren van een gedistribueerd component worden de gegevensstromen bruikbaar gemaakt voor gedistribueerd programmeren van grafische componenten. De scène klasse en de hoofdklassen van de grafische applicatie worden in het onderstaande model afhankelijk van functies van de datacommunicatie machine voor gegevenssynchronisatie.

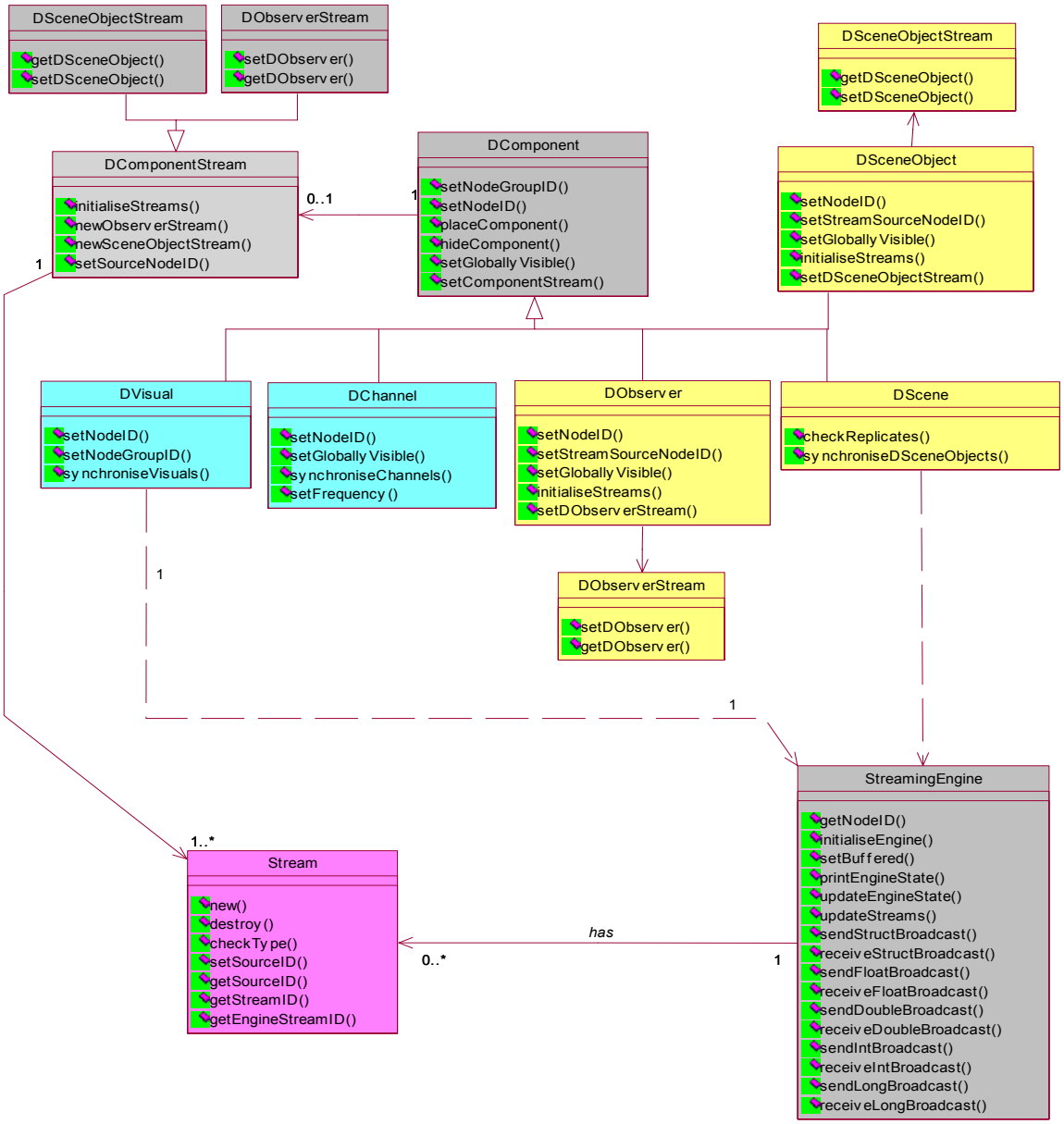


Fig20.2

12 Evaluatie

Voor het evalueren van de ontwikkelde applicatie programmeertalen is een simulatie applicatie geschreven. De synchronisatie- en frequentie tijden en zijn opgenomen in bestanden en gevisualiseerd met de GNUPLOT applicatie. Het bereik van de tijden en frequenties is zo bepaald dat de meetresultaten gemakkelijk met elkaar vergeleken kunnen worden. De kleuren in de visualisaties van de meetresultaten geven aan welke machines de resultaten hebben gegenereerd.

Om het grafisch cluster te representeren is de parallelle MPICH omgeving van een enkele machine gebruikt. De omgeving representeert hierbij een cluster van machines door meerdere parallelle processen op de machine toe te laten. De metingen zijn verricht op een computer met 512MB DDR RAM geheugen, een ABIT KR7A KT266A DDR moederbord, een processor van het type AMD XP 1700+ 266MHz en een grafische kaart van het type ABIT GFORCE 4MX met 64MB DDR intern geheugen.

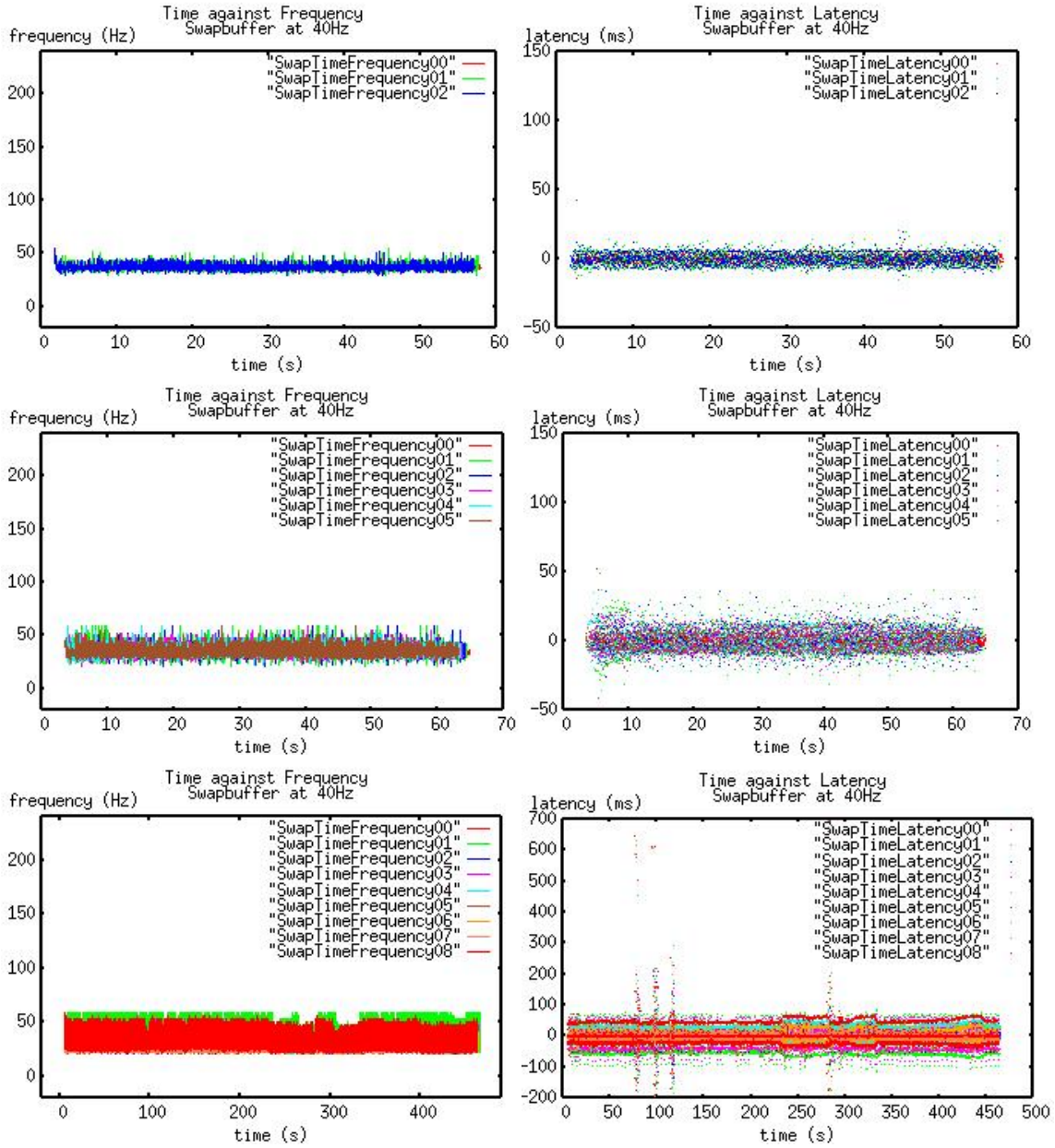
Elk parallel programma neemt ongeveer 64MB RAM in beslag en gebruikt een processor capaciteit van ruim 60%. Bij een capaciteitsbenutting van 9 machines is de processor capaciteit van elke machine ongeveer 130 MHz met een intern geheugen van 50MB RAM. De grafische capaciteit is bij 9 machines gelijkwaardig aan de capaciteit van een grafische kaart met 8MB RAM intern geheugen. De netwerk capaciteit neemt sterk af en is vergelijkbaar met de capaciteit van een 0.1MBps Ethernet netwerk.

De computer representeert tijdens uitvoer van de parallelle programma's een cluster van respectievelijk 3, 6, en 9 machines. Tijdens uitvoer van de simulatie worden hetzelfde aantal parallelle applicaties geïnstantieerd. De metingen zijn verricht op de machine zelf en worden niet geëxtrapoleerd naar een cluster van gelijkwaardige machines. Tijdens het meten zijn de frequenties door dynamische en statische algoritmen bepaald. Dynamische algoritmen scaleren de frequenties naar de hoogste waarden. Statische algoritmen houden de frequenties op een vaste waarde.

De statische algoritmen zijn een implementatie van de in de voorgaande hoofdstukken beschreven algoritmen. De dynamische algoritmen doen een voorspelling van de maximaal toegestane frequenties voor ieder parallel programma en zijn ontstaan tijdens verkennend onderzoek naar dynamische frequentiebepalings algoritmen. De statische metingen hebben vooraf bepaalde frequenties. De dynamische metingen hebben frequenties die bepaald worden door de dynamische frequentiebepalings algoritmen. De statische frequenties zijn zo bepaald dat de grenswaarden voor swap- en gegevenssynchronisatie worden bereikt bij een capaciteitsbenutting van 9 machines.

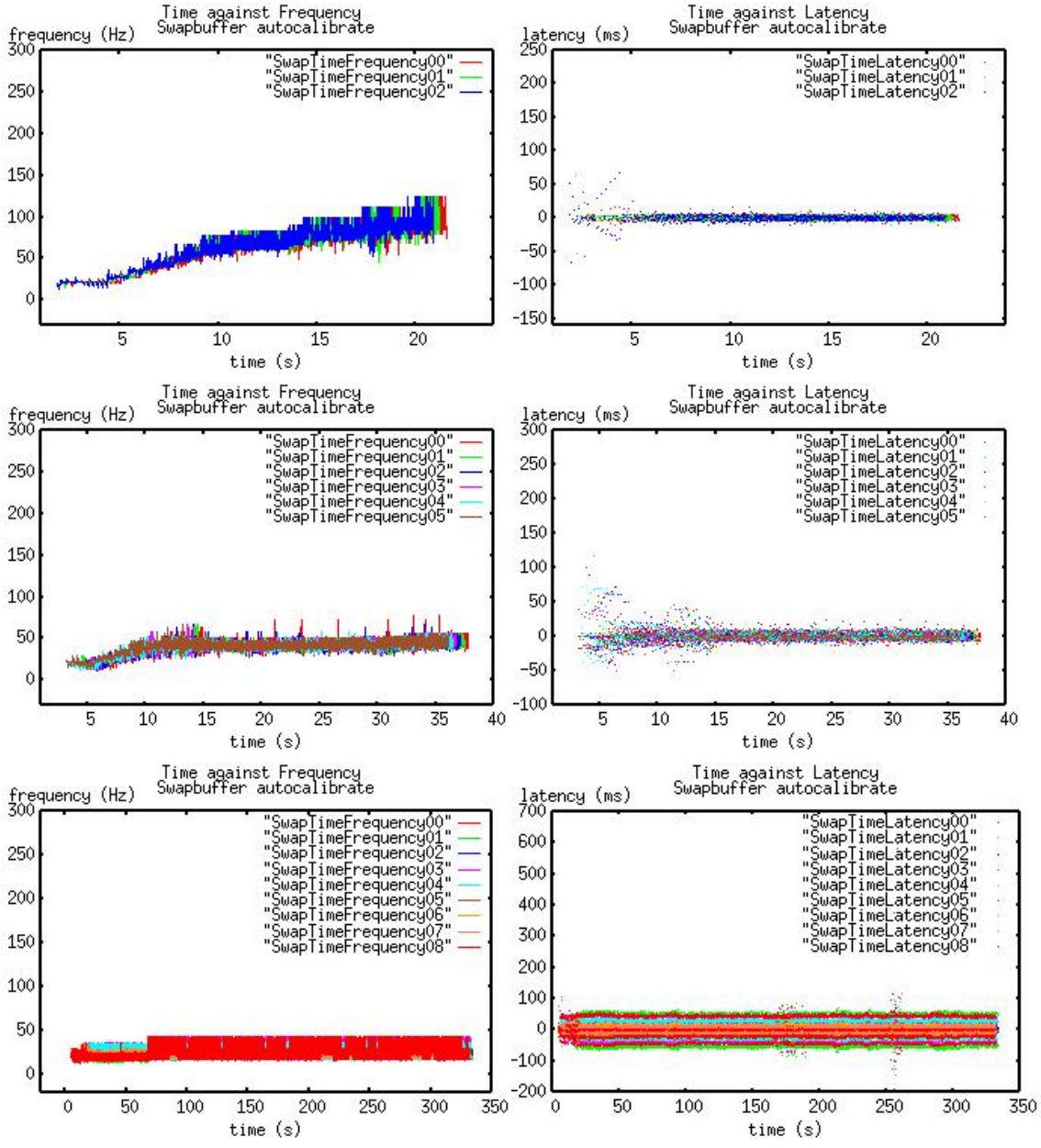
12.1 Frame synchronisatie statisch

De onderstaande grafieken zijn een weergave van de metingen verricht van de statisch bepaalde frame synchronisatie frequenties en frame synchronisatie afwijkingstijden op respectievelijk 3, 6 en 9 machines. Tijdens uitvoer van de simulatie is de swapbuffer frequentie op een vaste frequentie van 40Hz gezet.



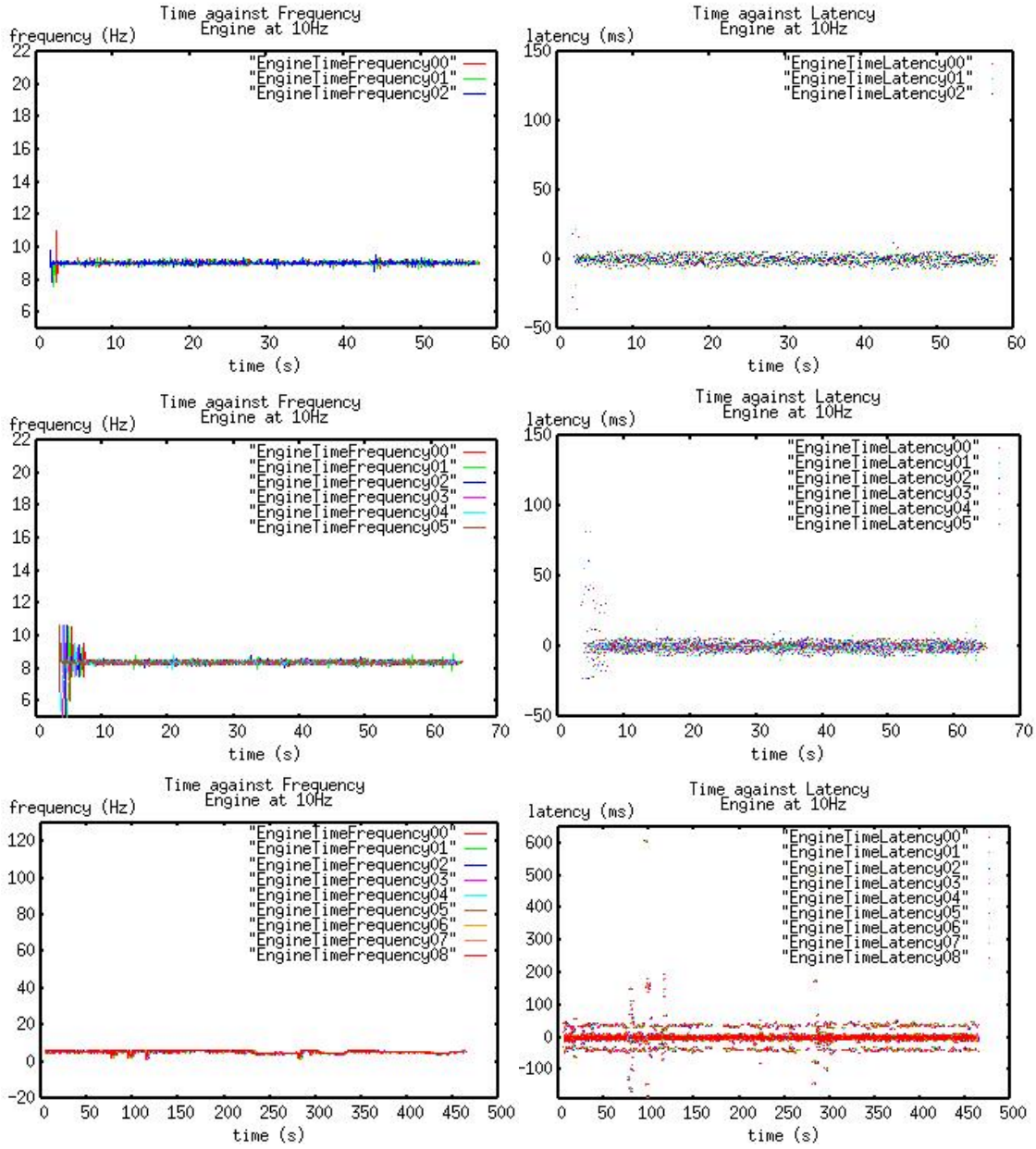
12.2 Frame synchronisatie dynamisch

De onderstaande grafieken zijn een weergave van de metingen verricht van de dynamisch bepaalde swapbuffer synchronisatie frequenties en swapbuffer synchronisatie afwijkingstijden op respectievelijk 3, 6 en 9 machines. Tijdens uitvoer van de simulatie wordt de maximale swapbuffer frequentie door het dynamisch frequentie algoritme bepaald.



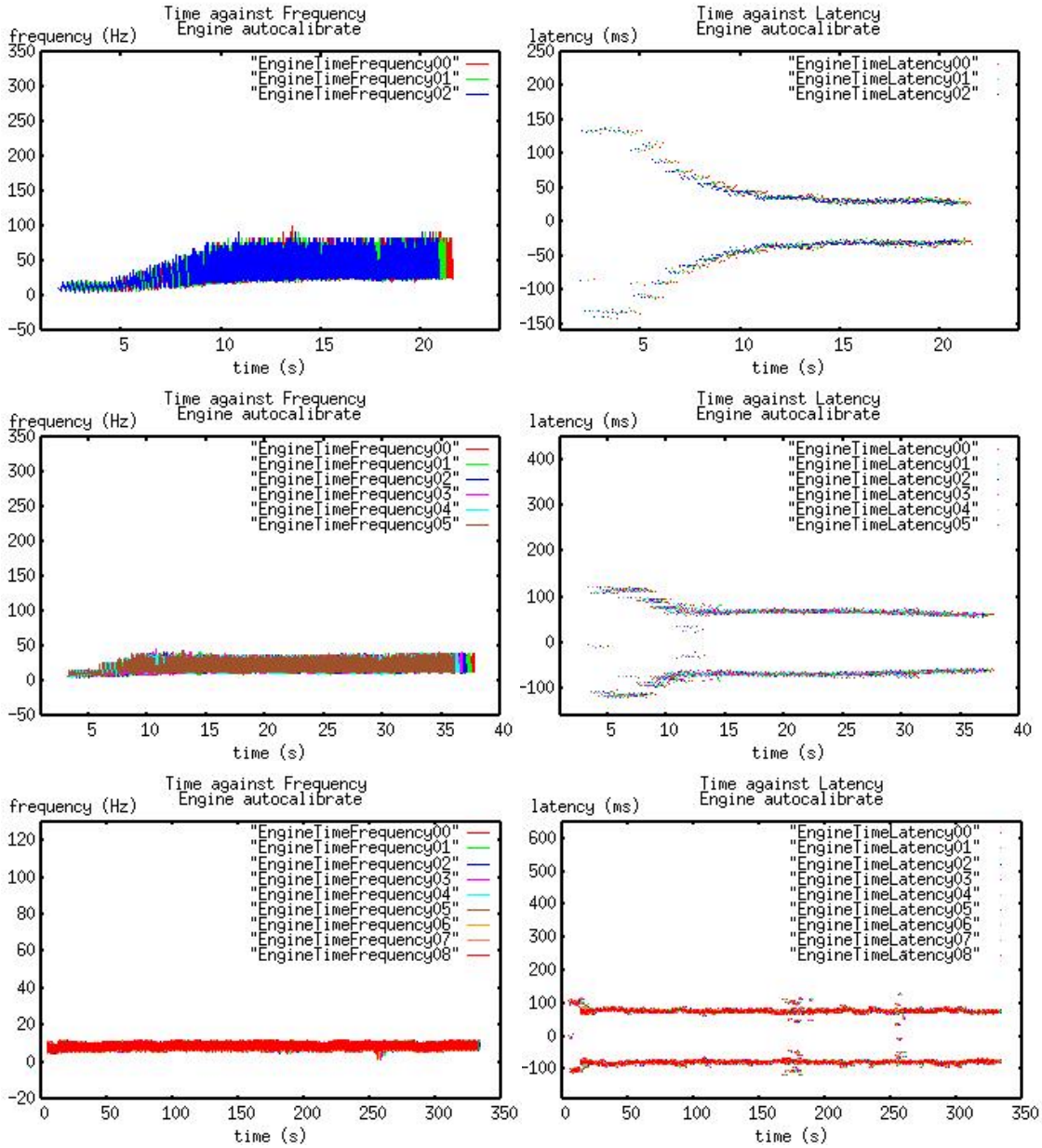
12.3 Data synchronisatie statisch

De onderstaande grafieken zijn een weergave van de metingen verricht van de statisch bepaalde synchronisatie frequenties en synchronisatie afwijkingstijden op respectievelijk 3, 6 en 9 machines. Tijdens uitvoer van de simulatie is de datacommunicatie machine frequentie op een vaste frequentie van 10Hz gezet.



12.4 Data synchronisatie dynamisch

De onderstaande grafieken zijn een weergave van de metingen verricht van de dynamische bepaalde synchronisatie frequenties en synchronisatie afwijkingstijden op respectievelijk 3, 6 en 9 machines. Tijdens uitvoer van de simulatie wordt de maximale datacommunicatie machine frequentie door het dynamisch frequentie algoritme bepaald.



12.5 Evaluatie meetresultaten

Uit de metingen is gebleken dat de swapbuffer bij 3 machines boven een gewenste swapbuffer van 60Hz uitstijgt met ruim 100Hz. De afwijkingstijden van de swapbuffer liggen ongeveer tussen de 4 milliseconden en 3 microseconden. De swapbuffer afwijkingstijden liggen bij 3 machines in het beste geval binnen de grenswaarden van 3-5 microseconden. Bij 3 machines blijft de gegevenssynchronisatie frequentie in de buurt van de ingestelde waarde van 10Hz en bereikt bij dynamische frequentiebepaling een frequentie van zelfs 60Hz. De afwijkingstijden hebben een maximum waarde van ongeveer 6 milliseconden en een minimum van 3 microseconden.

Tijdens een simulatie met 6 machines dalen de waarden naar ruim 60Hz, en blijft de ingestelde swapbuffer op 40Hz. De dynamisch bepaalde swapbuffer bereikt een frequentie van ruim 50Hz. De swapbuffer vertraging vindt plaats vanwege een verdeling van de systeem processorkracht. De afwijkingstijden liggen bij 6 machines ongeveer hetzelfde als bij 3 machines. De gegevenssynchronisatie frequenties blijven met een statische frequentie instelling voor 6 machines ongeveer hetzelfde. Bij een cluster van machines zullen de frequenties en afwijkingstijden van de statische instellingen ongeveer hetzelfde gedrag vertonen als in de statische meetresultaten naar voren is gekomen.

Gedurende een simulatie van 9 machines dalen de swapbuffer waarden naar 30Hz. De gegevenssynchronisatiewaarden blijven ongeveer hetzelfde. De daling van de swapbuffer is het gevolg van een verdeling van systeem processorkracht en geheugen over 9 parallelle processen. De machine die gebruikt is om de metingen te verrichten bereikt bij 9 parallelle grafische processen een grenswaarde waarbij de processen nog wel gesynchroniseerd lopen, maar over langere tijd en bij variërende processen minder betrouwbare resultaten zullen genereren.

Bij visuele observatie van de simulatie met respectievelijk 3, 6, en 9 machines genereren de beelden bij de eerste twee groepen van 3 en 6 machines een zeer gunstig visueel effect. De simulatie blijft bestuurbaar en er vinden geen vertragingen plaats. Bij 9 gesimuleerde machines blijft de simulatie goed bestuurbaar en is bij een statische instelling de simulatie betrouwbaar genoeg voor demonstratie applicaties.

Uit de metingen en observaties is duidelijk dat de synchronisatie algoritmen de gewenste doelstellingen bereiken voor een cluster van 6 machines. De uitslagen voor de ontwikkelde algoritmen zijn daarom ook gunstig te noemen. Omdat de grenswaarden in hoge mate worden bepaald door de machines die zich in het grafisch cluster bevinden kunnen er sterk verbeterde resultaten ontstaan bij de verandering van de cluster configuratie.

13 Conclusies

Uit het onderzoek is gebleken dat een cluster van civiele machines het mogelijk maakt om een gedistribueerde grafische applicatie programmeertaal te ontwikkelen in een korte tijd en in een relatief geïsoleerde omgeving. De eenvoud van de programmeertalen bepaalt in hoge mate de programmeerbaarheid van parallelle systemen. Verder bieden gespecialiseerde algoritmen voor proces interpolatie en tijdcontrole een oplossing voor de synchronisatieproblematiek. Ook elimineren dynamische scalering en bepaling van respectievelijk communicatie- en proces frequentie in hoge mate de beheersproblematiek.

De ontwikkelde applicatie programmeertalen stellen gebruikers in staat om op eenvoudige wijze een omringende virtuele omgeving te creëren. Door de generieke opbouw van de applicatie programmeertalen wordt het programmeerveld zeer breed en kunnen er verschillende typen applicaties ontwikkeld worden. Doelgericht onderzoek naar generieke applicatie programmeertalen voor het ontwikkelen van realtime applicaties in een grafisch cluster van machines is daarom ook bepalend voor het success van de resultaten.

Vanwege het clusteren van de grafische systemen bereikt de grafische applicatie een sterk verbeterde grafische precisie. Ontwikkeling en onderzoek naar vereenvoudigde applicatie programmeertalen heeft aangetoond dat een software ontwikkelaar in staat is om in korte tijd een gedistribueerde grafische applicatie te ontwikkelen. Parallelle applicatie programmeertalen die het mogelijk maken om met één codebasis een parallelle applicatie te ontwikkelen vergroten verder de beheersbaarheid van parallelle systemen.

Experimenten met interpolatie van processen hebben aangetoond dat variabele frequenties voor verschillende processen mogelijk worden gemaakt zonder opsplitsing van de processen in verschillende applicaties. Dit vermindert de communicatie tussen processen op lokaal niveau. Verbetering van en onderzoek naar algoritmen voor procesinterpolatie zal uiteindelijk een mogelijkheid creëren om processen in realtime op verschillende frequenties te laten functioneren.

Verder onderzoek naar dynamische frequentiebepaling van hoofdprocessen heeft aangetoond dat subprocessen op de gewenste frequenties kunnen worden uitgevoerd met interpolatie algoritmen. Met de ontwikkelde algoritmen voor frequentiebepaling zijn gunstige prestaties bereikt. Dynamische scalering van communicatieprocessen en gebruik van collectieve functies biedt verder een mogelijkheid om parallelle programma's te ontwikkelen en te definiëren met één enkele codebasis.

Bij gebruik van civiele apparatuur voor het ontwikkelen van simulatie applicaties in een niet realtime omgeving zijn de realtime prestaties gelimiteerd. Er is gebleken dat civiele apparatuur niet de gewenste precisie bereikt die nodig is om een gegarandeerd realtime simulator te ontwikkelen. Zonder het gebruik van realtime machines zullen applicaties daarom ook niet het gewenste realiteitsgehalte bereiken dat gewenst is in een realtime simulatie omgeving.

Referenties

[1] Gregory R. Andrews, Foundations of Multithreaded, Parallel, and Distributed Programming, *Pearson Addison Wesley*, USA, 1999.
[2] P. Neteczon, Supercomputing for the masses: A Parallel Macintosh Cluster, *Parallel processing and Applied Mathematics*, Poland, 2001.
[3] N. Ghost, Have PC-IG's lived up to expectations? Lessons learned, *AMS Integrated systems*, Donibristle, Dunfermline, Scotland, UK, 2002.
[4] D. Schmalstieg, G. Hesina, Distributed Applications for Collaborative Augmented Reality, *Vienna University of Technology Publications*, Austria, 2001.
[5] J. Salmon, T.L. Sterling, D.J. Becker, D.F. Savarese, How to build a Beowulf, *MIT Press*, Cambridge, MA, USA, 1999.
[6] A. Ahm, Clustering: How to build Cluster Computers, *Devbuilder Online Publications*, USA, 2001.
[7] R. David, Modeling of hybrid systems using continuous and hybrid Petri Nets, *IEEE Symposium*, 1997.
[8] P.T. Eugster, R. Gueraovi, C.H. Damm, On objects and events, *ACM Transactions on Parallel Computer Systems*, 2001.
[9] D. Rayside, G.T. Campbell, An Arstotelian understanding of object oriented programming, *ACM Transactions on Object Oriented Programming*, Minneapolis, USA, 2000.
[10] D. Baraff, Non penetrating Rigid Body simulation, *Eurographics State of the Art Reports*, Barcelona, Spain, 1993.
[11] W.H. Press, B.P. Flannery, S.A. Teukolsky, W.T. Vetterling, Numerical Recipes in C, *Cambridge University Press*, pages 97-115, Cambridge, NY, USA, 1996.
[12] R. Sedgewick, Algorithms in C Parts 1-4, *Addison Wesley Publishing Company*, pages 303-333, Jamestown, Rhode Island, USA, 1997.
[13] J. Borkowski, Interrupt and Cancellation as Synchronisation Methods, *Parallel processing and Applied Mathematics*, Poland, 2001.
[14] C.D. Norton, T.A. Wick, Parallel Unstructured AMR and Gigabit Networking for Beowulf-Class Clusters, *Parallel processing and Applied Mathematics*, Poland, 2001.
[15] M. Reke, Entwicklung & Implementierung eines echtzeitfahigen netzwerk protokolls, *RWTH scalable computing*, Aachen, Germany, 2001.
[16] P.A. Wilsey, X. Fan, L.M D'Souza, pGVT: An algorithm for Accurate GVT Estimation, *Center for Digital Systems Engineering*, Cincinnati, Ohio, USA, 1997.
[17] A.A.J. Langenkamp, P.M. Elgershuizen, P.L.J. van Lieshout, W. Huiskamp, Transputers: Towards realtime visual systems, *IEEE Symposium on Computer Architecture and Real Time Graphics*, USA, 1989.
[18] W. Huiskamp, R. Kiel, M.H. Smit, High Performance Computing takes the bus, *TNO Physics and Electronics Laboratory*, The Hague, The Netherlands, 1993.
[20] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, P. Hanrahan, WireGL: A Scalable Graphics System for Clusters, *Stanford University Press*, USA, 2001.
[21] I. Buck, G. Humphreys, P. Hanrahan, WireGL: Tracking Graphics State For Networked Rendering, *Stanford University Press*, USA, 2000.

[22] C. Curtis, D. Young, D. Friesen, C. Vane, R. Sanders, J. Slye, M. Schwenden, SGI Graphics Cluster Guide and Datasheet, *Silicon Graphics Publishing*, USA, 2001.
[23] E. Frecon, M. Stenius, DIVE: A Scaleable Network Architecture for Distributed Virtual Environments, *Swedish Institute of Computer Science*, Kista, Sweden, 1998.
[24] Silicon Graphics, Performer Programmers Documentation, *Silicon Graphics Inc*, CA, USA, 2002.
[25] G. Eckel, K. Jones, OpenGL Performer Programming Guide, *Silicon Graphics Inc*, CA, USA, 2002.
[26] G. Eckel, K. Jones, T. Domeier, OpenGL Performer Getting Started Guide, *Silicon Graphics Inc*, CA, USA, 2002.
[27] S. Gorlatch, Send Recv considered harmful: Myths and truths about Parallel Programming, *Springer Verlag*, Berlin, Germany, 2001.
[28] W. Gropp, E. Lusk, R. Thakur, Using MPI-2 Advanced features of the Message Passing Interface, *MIT Press*, Cambridge, MA, USA, 1999.
[29] W. Gropp, E. Lusk, N. Doss, A. Skjellum, A high performance, portable implementation of the MPI Message Passing Interface Standard, *Parallel Computing*, Aachen, Germany, 1996.
[30] W. Gropp, E. Lusk, Users Guide for MPICH, A portable implementation of MPI, *Argonne National Laboratory*, Argonne, Illinois, USA, 2000.
[31] W. Gropp, E. Lusk, MPI Programmers Guide, *Argonne National Laboratory*, Argonne, Illinois, USA, 2000.
[32] J. Worringen, K. Scholtyssik, MP MPICH User documentation and technical notes, *RWTH scalable computing*, Aachen, Germany, 2002.
[33] W. Gropp, E. Lusk, MPICH Model MPI Implementation reference manual, *Argonne National Laboratory*, Argonne, Illinois, USA, 2002.
[34] M. Poppe, MetaMPICH: an Extension of MPICH for heterogenous Metacomputers, *RWTH Scalable Computing Aachen*, Germany, 1999.
[35] T. Bemmerl, MetaMPICH: Flexible coupling of Heterogenous MPI Systems, *RWTH Scalable Computing Aachen,* Aachen, Germany, 1999.
[36] W. Huiskamp, R.J. Elias, Platform Framework: Software Architecture for Reconfigurable Simulators, *ITEC*, Lausanne, Switzerland, 1998.
[37] RedHat, RedHat Linux 8.0 Customization Guide, Administration Solutions for Intermediate Users, *RedHat Incorporated*, NC, USA, 2002.
[38] M. Segal, K. Akeley, The OpenGL Graphics System: A Specification, *Silicon Graphics Inc*, CA, USA, 1999.
[39] M. Raynal, J.M. Helary, Synchronisation and Control of Distributed Systems and Programs, *Wiley series in Parallel Computing*, Rennes, France, 1996.
[40] Quantum, Redefining Image Generation with Mantis, *Quantum3D Online Publications*, CA, USA, 2002.
[41] Aechelon Technologies, Cnova System Specifications, *Aechelon Technologies Inc*, San Carlos, CA, USA, 2003.
[42] T. Schultz, S. Wilkening, C. Danzer, G. Traefald, P. Donlin, Memory Management Control Programmer's Manual, *SGI Technical Publications*, pages 8-17, Mountain View, CA, USA, 2002.
[43] W. Huiskamp, J. van Geest, R.J. Elias, RedHawk/iHawk System Evaluation, *TNO Physics Electronics Laboratory*, The Hague, The Netherlands, 2004.

[44] S. Brosky, S. Rotolo, Shielded Processors: Guaranteeing Sub-millisecond Response in Standard Linux, *Concurrent Computer Corporation*, Pompano Beach, FL, USA, 2002.
[45] R.J. Elias, W. Huiskamp, Advanced Simulation Framework: A Generic Approach to Distributed Simulation, *TNO Physics Electronics Laboratory*, The Hague, The Netherlands, 1999.
[46] M. Apte, S. Chakravarthi, J. Padmanabhan, A. Skjellum, A Synchronised Realtime Linux based Myrinet Cluster for Deterministic High Performance Computing and MPI/RT, *High Performance Computing Laboratory, Dept. of Computer Science*, Mississippi State University, Mississippi, USA, 2001.
[47] J. Boomgaardt, F. Kuiper, Softgenlock: Non deterministic Genlocking in Realtime Linux, *TNO Physics Electronics Laboratory*, The Hague, The Netherlands, 2003.

Begrippenlijst

applicatie programmeertaal	programmeertaal voor het ontwikkelen van applicaties
architectuur	generiek model van samenhangende applicaties
barrier	begrenzing voordat parallelle processen synchroniseren
beeldsynchronisatie	synchronisatie van bij elkaar horende beeldpunten
bibliotheek	groep programmeertalen of programmeertaal
civiel systeem	systeem dat verkrijgbaar is op de civiele markt
civiele computer	computer die verkrijgbaar is op de civiele markt
cluster	een groep verbonden computers of machines
data synchronisatie	synchronisatie van bij elkaar horende gegevens
datacommunicatie machine	programma voor het realiseren van communicatie
frame synchronisatie	synchronisatie van bij elkaar horende beelden in de tijd
framelock latency	afwijkingstijd tussen momenten waarin beelden synchroniseren
framelocking	dezelfde betekenis als frame synchronisatie
gedistribueerde grafische applicatie	grafische applicatie die geprogrammeerd is voor een cluster
generalisatie	veralgemenalisering van een klasse in een generieke klasse
genlock latency	afwijkingstijd tussen beeldpunt synchronisatie momenten
genlocking	dezelfde betekenis als beeld synchronisatie
grafisch cluster	gedistribueerde grafische machines of programma's
grafische pijp	kanaal dat gedistribueerde grafische processen realiseert
horizontale scaleerbaarheid	scaleerbaarheid van communicatie over aantal machines
interpolatie	tussenvoeging van procesgroepen in de tijd
kanaal	uitvoerkanaal dat grafische beelden presenteert
klasse	klasse component van een object georiënteerd model
klassemodel	model gemodelleerd volgens de uml schema techniek
lineair bestand	bestand dat opgeslagen is op een enkele machine
lineaire grafische applicatie	grafische applicatie die gebruik maakt van een enkele machine
lokale machine	machine die aangestuurd wordt door eigen interne processen
machine	machine of computer al dan niet in een cluster of netwerk
parallel	synchroon bestaand of op hetzelfde moment beginnend
parallel bestand	bestand dat op meer dan een enkele machine is opgeslagen
parallellisatie	parallelliseren van een lineair bestand of proces
POSIX	overdraagbaar systeem interface voor computer omgevingen
proces	functie of proces dat zich bevindt op een enkele machine
realtime	begrip waarin bepaalde vaste tijdsafstanden bestaan
replicatie	verwijzing naar een systeemcomponent dat hetzelfde is
scène	gebied dat één of meer grafische objecten beschrijft
scènegraaf	graaf die een scène en de objecten daarin beschrijft
spawning	verdelen van processen middels een programma
specialisatie	specialisatie van een entiteit of klasse in een specifieke klasse
stuuralgoritme	algoritme dat een systeem of programmadeel bestuurt
synchronisatie	gelijktrekking van processen, gegevens of stromingen
synchronisatie algoritme	algoritme dat een systeem of applicatie synchroniseert
verticale scaleerbaarheid	scaleerbaarheid van gegevensstromen in een cluster

INHOUDSOPGAVE

I. C Cluster Component

```
#define CLUSTER 1

#define MAX_NODES 9
#define MAX_STREAMS 30
#define MAX_STREAMSLOTS 5
#define MAX_STRING_SIZE 25

enum {
      ALL    = -1,
      MASTER = 0,
      NODE00 = 0,
      NODE01 = 1,
      NODE02 = 2,
      NODE03 = 3,
      NODE04 = 4,
      NODE05 = 5,
      NODE06 = 6,
      NODE07 = 7,
      NODE08 = 8,
      NODE09 = 9,
      NODE10 = 10
    };

enum {
      STREAM_STRING,
      STREAM_INTEGER,
      STREAM_LONG,
      STREAM_FLOAT,
      STREAM_DOUBLE,
      MAX_TYPES
    };

typedef struct Stream {
  struct Stream* pStream_prev;
  struct Stream* pStream_next;
  int iIdentification;
  int iEngineIndex;
  int iIndex;

  short bIsInteger;
  short bIsLong;
  short bIsFloat;
  short bIsDouble;
  short bIsString;

  int iSourceID;
  int iDestinationID;
} Stream;
```

```
typedef struct StreamingEngine {
  int iNodes;
  int iSelfNodeID;

  struct Stream* pStream_first;
  struct Stream* pStream_last;
  struct Stream* pStream_current;

  int iMaxCount;

  int iIntegerCount[MAX_NODES];
  int iLongCount[MAX_NODES];
  int iFloatCount[MAX_NODES];
  int iDoubleCount[MAX_NODES];
  int iStringCount[MAX_NODES];

  int iMaxStreamCount;

  int iFloatStreamCount;
  int iDoubleStreamCount;
  int iIntegerStreamCount;
  int iLongStreamCount;
  int iStringStreamCount;

  short bIsInitialised;
  short bIsDirect;
  short bIsUpdated;
} StreamingEngine;

typedef struct MessageStream {
  long     alSequence[MAX_STREAMSLOTS];
  int      aiSequence[MAX_STREAMSLOTS];
  double adSequence[MAX_STREAMSLOTS];
  float    afSequence[MAX_STREAMSLOTS];

  char     acSequence[MAX_STRING_SIZE*MAX_STREAMSLOTS];
} MessageStream;

typedef struct StringStream {
  char acSequence[MAX_STRING_SIZE*MAX_STREAMSLOTS];
} StringStream;

typedef struct LongStream {
  long   alSequence[MAX_STREAMSLOTS];
} LongStream;

typedef struct IntegerStream {
  int    aiSequence[MAX_STREAMSLOTS];
} IntegerStream;

typedef struct FloatStream {
  float  afSequence[MAX_STREAMSLOTS];
} FloatStream;

typedef struct DoubleStream {
  double adSequence[MAX_STREAMSLOTS];
} DoubleStream;
```

```
MessageStream MessageStream_SendReceiveNode[MAX_NODES][MAX_STREAMS];
FloatStream FloatStream_SendReceiveNode[MAX_NODES][MAX_STREAMS];
DoubleStream DoubleStream_SendReceiveNode[MAX_NODES][MAX_STREAMS];
IntegerStream IntegerStream_SendReceiveNode[MAX_NODES][MAX_STREAMS];
LongStream LongStream_SendReceiveNode[MAX_NODES][MAX_STREAMS];
StringStream StringStream_SendReceiveNode[MAX_NODES][MAX_STREAMS];

MessageStream MessageStream_BroadcastNode[MAX_NODES][MAX_STREAMS];
FloatStream FloatStream_BroadcastNode[MAX_NODES][MAX_STREAMS];
DoubleStream DoubleStream_BroadcastNode[MAX_NODES][MAX_STREAMS];
IntegerStream IntegerStream_BroadcastNode[MAX_NODES][MAX_STREAMS];
LongStream LongStream_BroadcastNode[MAX_NODES][MAX_STREAMS];
StringStream StringStream_BroadcastNode[MAX_NODES][MAX_STREAMS];

MessageStream MessageStream_PersistentBroadcastNode[MAX_NODES][MAX_STREAMS];
FloatStream FloatStream_PersistentBroadcastNode[MAX_NODES][MAX_STREAMS];
DoubleStream DoubleStream_PersistentBroadcastNode[MAX_NODES][MAX_STREAMS];
IntegerStream IntegerStream_PersistentBroadcastNode[MAX_NODES][MAX_STREAMS];
LongStream LongStream_PersistentBroadcastNode[MAX_NODES][MAX_STREAMS];
StringStream StringStream_PersistentBroadcastNode[MAX_NODES][MAX_STREAMS];

MPI_Datatype MPI_Datatype_MessageStream;
MPI_Datatype MPI_Datatype_FloatStream;
MPI_Datatype MPI_Datatype_DoubleStream;
MPI_Datatype MPI_Datatype_IntegerStream;
MPI_Datatype MPI_Datatype_LongStream;
MPI_Datatype MPI_Datatype_StringStream;

static struct StreamingEngine StreamingEngine_engine;

void StreamingEngine_UpdateState();
short StreamingEngine_HasNodeID(int iNodeID);
short StreamingEngine_HasNodeName(char* pcNodeName);

int StreamingEngine_GetNodeID();
char* StreamingEngine_GetNodeName();

void StreamingEngine_SetUpdated();
short StreamingEngine_IsUpdated();

void StreamingEngine_Initialise(int iArgumentCount, char* pacArgumentContents[]);
void StreamingEngine_PrintState(int iNodeID);
void StreamingEngine_Update();

int StreamingEngine_LimitMaximumStreamCount();
int StreamingEngine_GetNumberOfNodes();
int StreamingEngine_GetNumberOfFloatStreams();
int StreamingEngine_GetNumberOfFloats(int iNodeID);
int StreamingEngine_GetNumberOfDoubleStreams();
int StreamingEngine_GetNumberOfDoubles(int iNodeID);
int StreamingEngine_GetNumberOfIntegerStreams();
int StreamingEngine_GetNumberOfIntegers(int iNodeID);
int StreamingEngine_GetNumberOfLongStreams();
int StreamingEngine_GetNumberOfLongs(int iNodeID);
int StreamingEngine_GetNumberOfStringStreams();
int StreamingEngine_GetNumberOfStrings(int iNodeID);

void StreamingEngine_SendMessageBroadcast(int iEngineStreamID, int iSourceNodeID);
void StreamingEngine_ReceiveMessageBroadcast(int iEngineStreamID, int iSourceNodeID);
void StreamingEngine_SendFloatBroadcast(int iEngineStreamID, int iSourceNodeID);
void StreamingEngine_ReceiveFloatBroadcast(int iEngineStreamID, int iSourceNodeID);
void StreamingEngine_SendDoubleBroadcast(int iEngineStreamID, int iSourceNodeID);
void StreamingEngine_ReceiveDoubleBroadcast(int iEngineStreamID, int iSourceNodeID);
void StreamingEngine_SendIntBroadcast(int iEngineStreamID, int iSourceNodeID);
void StreamingEngine_ReceiveIntBroadcast(int iEngineStreamID, int iSourceNodeID);
void StreamingEngine_SendLongBroadcast(int iEngineStreamID, int iSourceNodeID);
void StreamingEngine_ReceiveLongBroadcast(int iEngineStreamID, int iSourceNodeID);
void StreamingEngine_SendStringBroadcast(int iEngineStreamID, int iSourceNodeID);
void StreamingEngine_ReceiveStringBroadcast(int iEngineStreamID, int iSourceNodeID);
```

```
Stream* Stream_New(int iTypeID, int iNodeID);
Stream* Stream_NewString(int iNodeID);
Stream* Stream_NewInteger(int iNodeID);
Stream* Stream_NewLong(int iNodeID);
Stream* Stream_NewFloat(int iNodeID);
Stream* Stream_NewDouble(int iNodeID);

short Stream_IsOfType(Stream* pStream_stream, int iTypeID);
void Stream_Destroy(Stream* pStream_stream);

void Stream_SetSourceID(Stream* pStream_stream, int iSourceID);
void Stream_GetSourceID(Stream* pStream_stream, int* piSourceID);
void Stream_GetStreamID(Stream* pStream_stream, int* piStreamID);
void Stream_GetEngineStreamID(Stream* pStream_stream, int* piStreamID);

void Stream_SetStringValue(Stream* pStream_stream, const char* pcString);
void Stream_GetStringValue(Stream* pStream_stream, char* pcString);
void Stream_GetIntegerValue(Stream* pStream_stream, int* piValue);
void Stream_SetIntegerValue(Stream* pStream_stream, int iValue);
void Stream_GetLongValue(Stream* pStream_stream, long* plValue);
void Stream_SetLongValue(Stream* pStream_stream, long lValue);
void Stream_GetFloatValue(Stream* pStream_stream, float* pfValue);
void Stream_SetFloatValue(Stream* pStream_stream, float fValue);
void Stream_GetDoubleValue(Stream* pStream_stream, double* pdValue);
void Stream_SetDoubleValue(Stream* pStream_stream, double dValue);
```

II. C Synchronisation Component

```
typedef struct RealtimeBarrier
{
  short bIsLogged;              //the barrier logs all the time data in a barrier log file

  short bIsReactive;            //the barrier pulses within predicted latency boundaries
  short bIsNormal;              //the barrier pulses within the set latency boundaries
  short bIsDirect;              //the barrier pulses at the next possible interrupt
  short bIsGlobal;              //the barrier is set to be global across all processes

  short bWasTriggered;          //the indicator for a barrier trigger
  short bTriggerReleased;       //the indicator for a trigger release

  int iBarrierID;               //the barrier identification number
  int iMissedFrames;            //the missed frames of the barrier

  double dCurrentTrigger;       //the current realtime triggers
  double dCurrentInterrupts;    //the current interrupt frequency
  double adPastTriggers[100]; //the past realtime triggers

  double dCurrentTime;          //the current realtime hit
  double dPrevTime;             //the previously realtime barrier time
  double dStepTime;             //the realtime barrier step time

  double dMinLatency;           //the minimum barrier latency
  double dMaxLatency;           //the maximum barrier latency

  double dStartTime;            //the required barrier start time
  double dEndTime;              //the required barrier end time

  double dUnitsPerSecond;       //the number of time units per second
} RealtimeBarrier;

typedef struct RealtimeClock
{
  double dClockStartTime;
} RealtimeClock;

RealtimeBarrier aRealtimeBarrier_realtime[50];
```

```
double Clock_GetMicroseconds();
double Clock_GetMilliseconds();
double Clock_GetSeconds();
double Clock_Reset();

void Clock_SleepMicroseconds(double dTime);
void Clock_SleepMilliseconds(double dTime);
void Clock_SleepSeconds(double dTime);

void RealtimeBarrier_PlaceLocal();
void RealtimeBarrier_PlaceGlobal();

double RealtimeBarrier_GetTimeMicroseconds();
double RealtimeBarrier_GetTimeMilliseconds();
double RealtimeBarrier_GetTimeSeconds();

void RealtimeBarrier_SetTimeUnitsAsMicroseconds(int iBarrierID);
void RealtimeBarrier_SetTimeUnitsAsMilliseconds(int iBarrierID);
void RealtimeBarrier_SetTimeUnitsAsSeconds(int iBarrierID);

void RealtimeBarrier_SetFrequency(double dFrequency, int iBarrierID);
void RealtimeBarrier_SetLatencyPercentage(double dMinPercentage, double dMaxPercentage, int iBarrierID);

void RealtimeBarrier_SetStartTime(double dStartTime, int iBarrierID);
void RealtimeBarrier_SetEndTime(double dEndTime, int iBarrierID);

double RealtimeBarrier_GetCurrentTime(int iBarrierID);

void RealtimeBarrier_SetDirect(int iBarrierID);
void RealtimeBarrier_SetNormal(int iBarrierID);
void RealtimeBarrier_SetReactive(int iBarrierID);
void RealtimeBarrier_SetGlobal(int iBarrierID);
void RealtimeBarrier_SetLocal(int iBarrierID);

void RealtimeBarrier_MicrosecondsNew(double dStartTime,  double dFrequency,
                                     double dMinLatencyPercentage, double dMaxLatencyPercentage, int iBarrierID);

void RealtimeBarrier_Reset(int iBarrierID);
void RealtimeBarrier_Place(int iBarrierID);

void RealtimeBarrier_StartPulseGenerator(int iBarrierID);
```

III. C Visual Component

```
#define VISUAL 1
#define MAX_FORCES 2
#define MAX_ENTITY_STATES 10
#define MAX_ENTITY_STRINGSIZE 15

enum {
  BASIC_PROPULSION_VELOCITY,          //VELOCITY BASED
  BASIC_PROPULSION_FRICTION,          //FRICTION BASED
  BASIC_PROPULSION_FRICTIONLESS,      //FRICTIONLESS BASED
  SKY_PROPULSION_JET,                 //SKY BASED JET
  SKY_PROPULSION_HELI,                //SKY BASED HELI
  SKY_PROPULSION_ROCKET,              //SKY BASED ROCKET
  SKY_PROPULSION_BOMB,                //SKY BASED BOMB
  GROUND_PROPULSION_TANK,             //GROUND BASED TANK
  GROUND_PROPULSION_TRUCK,            //GROUND BASED TRUCK
  GROUND_PROPULSION_CAR,              //GROUND BASED CAR
  GROUND_PROPULSION_SOLDIER,          //GROUND BASED ENTITY
  WATER_PROPULSION_SHIP,              //WATER BASED SHIP
  WATER_PROPULSION_BOAT,              //WATER BASED BOAT
  WATER_PROPULSION_DIVER,             //WATER BASED DIVER
};

enum {
  DEACTIVATED_DAMPING,        //DAMP NOBODY
  IMMEDIATE_DAMPING,          //DAMP IMMEDIATELY
  LINEAR_DAMPING,             //DAMP LINEARLY
  EXPONENTIAL_DAMPING         //DAMP EXPONENTIALLY
};

typedef struct VisualEngine {
  short  bIsInitialised;
  short  bIsParallel;

  short  bHasDynamicInterleave;
  short  bHasDynamicFrequency;

  double dFrequency;
  double dStepTime;
  double dSwapbufferFrequency;
  double dSwapbufferStepTime;

  double dAverageEngineStepTime;
  double dAverageSwapbufferStepTime;
  double dAverageEngineFrequency;
  double dAverageSwapbufferFrequency;

  double dFrequencyRatio;

  double dCurrentTime;
  long    lCurrentFrame;

  //parallel extensions
  #if CLUSTER
    short bUpdatedStreams;
  #endif

} VisualEngine;
```

```
typedef struct Channel {
  pfVec3        pfVec3_RelativePosition;
  pfVec3        pfVec3_RelativeHeading;
  pfVec3        pfVec3_RelativeObserverPosition;
  pfVec3        pfVec3_RelativeObserverHeading;
  pfVec3        pfVec3_StereoObserverLookat;
  pfVec3        pfVec3_StereoObserverPosition;
  pfVec3        pfVec3_StereoObserverHeading;

  pfPipeWindow*      pfPipeWindow_Window;
  pfChannel*         pfChannel_ChannelDefault;
  pfChannel*         pfChannel_ChannelLeftEye;
  pfChannel*         pfChannel_ChannelRightEye;
  pfPipe*            pfPipe_Pipe;

  double        dStereoAngle;

  char*      pcWindowTitle;
  int         iPositionX;
  int         iPositionY;
  int         iSizeX;
  int         iSizeY;

  double        dNearestPoint;
  double        dFarthestPoint;
  double        dHorizontalAngle;
  double        dVerticalAngle;

  short      bHasStereoBuffers;
  short      bIsFullscreen;
  short      bIsWindowed;
  short      bIsStereo;

  short      bIsOn;

  //parallel extensions
  #if CLUSTER
    int iSourceNodeID;
    struct Stream* pStream_bIsOn;
  #endif

} Channel;

typedef struct Camera {
  short bIsCinematic;
  short bIsLocked;
  short bIsInside;
  short bIsOutside;
  short bIsFree;

  short bChangedView;

  pfCoordd           pfCoordd_offset;
  pfCoordd           pfCoordd_camera;
  struct Entity*     pEntity_attachment;

  //parallel extensions
  #if CLUSTER
    int iSourceNodeID;
    int iSourceCameraID;

    struct Stream*     pStream_cameraX;
    struct Stream*     pStream_cameraY;
    struct Stream*     pStream_cameraZ;
    struct Stream*     pStream_cameraH;
    struct Stream*     pStream_cameraP;
    struct Stream*     pStream_cameraR;
  #endif

} Camera;
```

```
typedef struct Scene {
  int         iNumberOfLights;
  int         iNumberOfEntities;
  int         iNumberOfWorlds;
  int         iNumberOfWeathers;

  struct Light*         pLight_first;
  struct World*         pWorld_first;
  struct Weather*       pWeather_first;
  struct Entity*        pEntity_first;

  struct Light*         pLight_current;
  struct World*         pWorld_current;
  struct Weather*       pWeather_current;
  struct Entity*        pEntity_current;

  struct Light*         pLight_last;
  struct World*         pWorld_last;
  struct Weather*       pWeather_last;
  struct Entity*        pEntity_last;

  struct Light*         pLight_select;
  struct World*         pWorld_select;
  struct Weather*       pWeather_select;
  struct Entity*        pEntity_select;

  pfScene*    pfScene_scene;

  pfVec3      pfVec3_FogDensity;
  pfVec3      pfVec3_FogColor;
  pfVec3      pfVec3_CloudHeight;

  pfEarthSky*           pfEarthSky_clear;
  pfVolFog*             pfVolFog_clouds;
  pfFog*                pfFog_fog;

  short       bIsInitialised;
  short       bCloudsOn;
  short       bFogOn;

  //parallel extensions
  #if CLUSTER
    struct Light*       pLight_active;
    struct World*       pWorld_active;
    struct Weather*     pWeather_active;
    struct Entity*      pEntity_active;
  #endif

} Scene;

typedef struct PhysicsScene {
  struct PhysicsScene*          pPhysicsScene_parent;
  struct PhysicsScene*          pPhysicsScene_next;
  struct Physics*               pPhysics_physics;
} PhysicsScene;
```

```
typedef struct Physics {
  double dDifferentialPositionVector[MAX_FORCES][6];
  double dFilteredPositionVector[6];
  double dPositionVector[6];
  double dSurfaceVector[6];
  double dFrictionVector[6];

  double dAbsoluteForceVector[MAX_FORCES][6];
  double dAbsoluteVelocityVector[MAX_FORCES][3];
  double dAbsoluteAccelerationVector[MAX_FORCES][3];
  double dAbsoluteAngularVelocityVector[MAX_FORCES][3];
  double dAbsoluteAngularAccelerationVector[MAX_FORCES][3];
  double dRelativeForceVector[MAX_FORCES][6];
  double dRelativeVelocityVector[MAX_FORCES][3];
  double dRelativeAccelerationVector[MAX_FORCES][3];
  double dRelativeAngularVelocityVector[MAX_FORCES][3];
  double dRelativeAngularAccelerationVector[MAX_FORCES][3];
  double dDifferentialForceVector[MAX_FORCES][6];
  double dDifferentialVelocityVector[MAX_FORCES][3];
  double dDifferentialAccelerationVector[MAX_FORCES][3];
  double dDifferentialAngularVelocityVector[MAX_FORCES][3];
  double dDifferentialAngularAccelerationVector[MAX_FORCES][3];

  double dInputParameters[7];
  struct PhysicsScene* pPhysicsScene_child;
  struct Physics* pPhysics_parent;

  int iComponentIndex;

  int iDynamicsType;
  int iHeadingThresholdType;
  int iPositionThresholdType;

  double dPitchSign;
  double dBodyMass;
  double dBodyFrictionConstant;
  double dBodyFrictionUpperBound;
  double dBodyAccelerationThreshold;
  double dBodyVelocityThreshold;

  double dMinYaw;
  double dMaxYaw;
  double dMinPitch;
  double dMaxPitch;
  double dMinRoll;
  double dMaxRoll;

  double dMinPositionX;
  double dMaxPositionX;
  double dMinPositionY;
  double dMaxPositionY;
  double dMinPositionZ;
  double dMaxPositionZ;

  pfDoubleDCS* pfDoubleDCS_PositionVector;

  //parallel extensions
  #if CLUSTER
    int iSourceNodeID;

    struct Stream* pStream_PositionVectorX;
    struct Stream* pStream_PositionVectorY;
    struct Stream* pStream_PositionVectorZ;

    struct Stream* pStream_PositionVectorH;
    struct Stream* pStream_PositionVectorP;
    struct Stream* pStream_PositionVectorR;
  #endif

} Physics;
```

```
typedef struct Light {
  struct Light* pLight_next;
  struct Light* pLight_prev;

  char*                pcFileName;
  char*                pcName;
  int                  iIdentification;

  double       dAmbientRed;
  double       dAmbientGreen;
  double       dAmbientBlue;
  double       dDiffuseRed;
  double       dDiffuseGreen;
  double       dDiffuseBlue;
  double       dSpecularRed;
  double       dSpecularGreen;
  double       dSpecularBlue;

  double       dAmbient;
  double       dDiffuse;
  double       dSpecular;

  pfSwitch*            pfSwitch_switch;
  struct Physics*      pPhysics_model;
  struct Entity*       pEntity_parent;

  pfLightSource*       pfLightSource_light;
  pfTexture*           pfTexture_texture;
  pfFrustum*           pfFrustum_frustum;
  pfFog*               pfFog_fog;
  pfNode*               pfNode_model;

  short bIsOn;

  //parallel extensions
  #if CLUSTER
    int iSourceNodeID;
    struct Stream* pStream_bIsOn;
  #endif

} Light;
```

```
typedef struct Entity {
  struct Entity*        pEntity_next;
  struct Entity*        pEntity_prev;
  int                   iIdentification;
  char*                 pcName;

  struct Physics*       pPhysics_model;
  struct Entity*        pEntity_parent;
  struct Camera*        pCamera_attachment;

  char*                 pcFileName;
  pfSwitch*             pfSwitch_switch;
  pfSphere*             pfSphere_bounds;
  pfBox*                pfBox_bounds;
  pfNode*               pfNode_model;

  pfGeoSet*             pfGeoSet_geometry;
  pfGeode*              pfGeode_primitive;
  pfGeoState*           pfGeoState_attributes;
  pfMaterial*           pfMaterial_material;

  pfText*       pfText_text;
  pfFont*       pfFont_font;
  pfString*     pfString_string;

  int       iStringStateCounter;
  int       iNumberStateCounter;

  char      aacStateStringName[MAX_ENTITY_STATES][MAX_ENTITY_STRINGSIZE];
  char      aacStateStringValue[MAX_ENTITY_STATES][MAX_ENTITY_STRINGSIZE];
  char      aacStateNumberName[MAX_ENTITY_STATES][MAX_ENTITY_STRINGSIZE];
  double    adStateNumberValue[MAX_ENTITY_STATES];

  short bIsSelected;
  short bIsDragged;

  short bIsOn;

  //parallel extensions
  #if CLUSTER
    int iSourceNodeID;
    struct Stream* pStream_bIsSelected;
    struct Stream* pStream_bIsDragged;
    struct Stream* pStream_bIsOn;
  #endif

} Entity;

typedef struct World {
  struct World* pWorld_next;
  struct World* pWorld_prev;
  int       iIdentification;
  char*     pcName;

  Physics*      pPhysics_model;

  pfSwitch*             pfSwitch_switch;
  pfSphere*             pfSphere_bounds;
  pfNode*               pfNode_world;
  char*                 pcFileName;

  short bIsOn;

  //parallel extensions
  #if CLUSTER
    int iSourceNodeID;
    struct Stream* pStream_bIsOn;
  #endif

} World;
```

```
static struct VisualEngine      VisualEngine_engine;
static struct Camera            Camera_camera;
static struct Channel           Channel_channel;
static struct Scene             Scene_scene;

void VisualEngine_Initialise(int iArgumentCount, char *pacArgumentContents[]);
void VisualEngine_SetEngineFrequency(double dFrequency);
void VisualEngine_SetSwapbufferFrequency(double dFrequency);

void VisualEngine_SetDynamicInterleaveOn();
void VisualEngine_SetDynamicFrequencyOn();
void VisualEngine_SetDynamicInterleaveOff();
void VisualEngine_SetDynamicFrequencyOff();
void VisualEngine_PrintState();
void VisualEngine_Update();
void VisualEngine_Destroy();

void Screen_SetRelativePosition(double dPositionX, double dPositionY, double dPositionZ);
void Screen_SetRelativeHeading(double dYaw, double dPitch, double dRoll);
void Screen_SetViewingAngle(double dHorizontalAngle, double dVerticalAngle);
void Screen_SetViewingDistance(double dNearest, double dFarthest);
void Screen_SetSize(int iWidth, int iHeight);
void Screen_SetPosition(int iPositionX, int iPositionY);
void Screen_SetTitle(char* pcTitle);

void Screen_SetFullscreen();
void Screen_SetWindowed();

void Screen_SetStereo();
void Screen_SetStereoDepth(double dStereo);
void Screen_SetStereoObserverLookat(double dPositionX, double dPositionY, double dPositionZ);
void Screen_SetStereoObserverPosition(double dPositionX, double dPositionY, double dPositionZ);
void Screen_SetStereoObserverHeading(double dYaw, double dPitch, double dRoll);

void Screen_SetSourceNode(int iNodeID);
void Screen_SetRasterAngles(double dHorizontalAngle, double dVerticalAngle);
void Screen_SetRasterPlace(int iHorizontal, int iVertical, int iNodeID);
void Screen_Display();

void Camera_AttachToEntity(Entity* pEntity_entity);
void Camera_DetachFromEntity();

void Camera_SetFreeViewing();
void Camera_SetCinematicViewing();
void Camera_SetOutsideViewing();
void Camera_SetInsideViewing();
void Camera_SetLockedViewing();
void Camera_SetRelativeHeading(double dYaw, double dPitch, double dRoll);
void Camera_SetRelativeDistance(double dDistance);
void Camera_SetPosition(double dPositionX, double dPositionY, double dPositionZ);
void Camera_SetHeading(double dYaw, double dPitch, double dRoll);
void Camera_SetSourceNode(int iNodeID);
```

```
void* Fog_NewFog();
void* Fog_LastFog();
void Fog_Destroy(void* pFog_fog);

void Fog_SetOn(void* pFog_fog);
void Fog_SetOff(void* pFog_fog);
void Fog_AttachToEntity(void* pFog_fog, Entity* pEntity_entity);
void Fog_DetachFromEntity(void* pFog_fog);
void Fog_SetFogArea(void* pFog_fog, Entity* pEntity_entity);
void Fog_SetFogColor(void* pFog_fog, double dRed, double dGreen, double dBlue);
void Fog_SetFogDensity(void* pFog_fog, double dFogDensity);
void Fog_SetFogTexture(void* pFog_fog, char* pcTextureFileName);

Physics* Fog_GetPhysics();

void Fog_SetFogOn();
void Fog_SetFogOff();
void Fog_SetCloudsOn();
void Fog_SetCloudsOff();
void Fog_SetDensity(double dFogDensity, double dCloudDensity);
void Fog_SetColor(double dRed, double dGreen, double dBlue);
void Fog_SetClouds(double dHeightGround, double dHeightSky);
void Fog_SetSourceNode(void* pFog_fog, int iNodeID);

Light* Light_NewLight();
Light* Light_LastLight();
void Light_Destroy(Light* pLight_light);

void Light_SetOn(Light* pLight_light);
void Light_SetOff(Light* pLight_light);
void Light_AttachToEntity(Light* pLight_light, Entity* pEntity_entity);
void Light_DetachFromEntity(Light* pLight_light);
void Light_SetToBeam(Light* pLight_light, double dSourceCone, double dTargetCone, double dLength, double dSpread);
void Light_SetToPoint(Light* pLight_light, double dRadius, double dSpread);
void Light_SetDistance(Light* pLight_light, double dDistanceAttenuation);
void Light_SetAmbient(Light* pLight_light, double dAmbient);
void Light_SetDiffuse(Light* pLight_light, double dDiffuse);
void Light_SetSpecular(Light* pLight_light, double dSpecular);
void Light_SetAmbientColor(Light* pLight_light, double dAmbientRed, double dAmbientGreen, double dAmbientBlue);
void Light_SetDiffuseColor(Light* pLight_light, double dDiffuseRed, double dDiffuseGreen, double dDiffuseBlue);
void Light_SetSpecularColor(Light* pLight_light, double dSpecularRed, double dSpecularGreen, double dSpecularBlue);
void Light_SetBackgroundReflection(Light* pLight_light, double dAmbient);
void Light_SetSurfaceReflection(Light* pLight_light, double dDiffuse);
void Light_SetShinyReflection(Light* pLight_light, double dSpecular);
void Light_SetBackgroundColor(Light* pLight_light, double dRed, double dGreen, double dBlue);
void Light_SetSurfaceColor(Light* pLight_light, double dRed, double dGreen, double dBlue);
void Light_SetShinyColor(Light* pLight_light, double dRed, double dGreen, double dBlue);
void Light_SetColor(Light* pLight_light, double dRed, double dGreen, double dBlue);
void Light_SetPosition(Light* pLight_light, double dPositionX, double dPositionY, double dPositionZ);
void Light_SetHeading(Light* pLight_light, double dYaw, double dPitch, double dRoll);

Physics* Light_GetPhysics();
```

```
World* World_NewWorld();
World* World_LastWorld();
void World_Destroy();
void World_SetOn(World* pWorld_world);
void World_SetOff(World* pWorld_world);
void World_SetFile(World* pWorld_world, char* pcFileName);

Entity* Entity_NewEntity();
Entity* Entity_LastEntity();
void Entity_Destroy(Entity* pEntity_entity);

void Entity_SetOn(Entity* pEntity_entity);
void Entity_SetOff(Entity* pEntity_entity);
void Entity_SetFile(Entity* pEntity_entity, char* pcFileName);
void Entity_SetClonedEntity(Entity* pEntity_entity, Entity* pEntity_use);
void Entity_AttachToEntity(Entity* pEntity_entity, Entity* pEntity_parent);
void Entity_DetachFromEntity(Entity* pEntity_entity);
void Entity_SetText(Entity* pEntity_entity, char* pcText);
void Entity_SetPlane(Entity* pEntity_entity, double dHeight, double dWidth);
void Entity_SetCube(Entity* pEntity_entity, double dHeight, double dWidth, double dLength);
void Entity_SetPyramid(Entity* pEntity_entity, double dHeight, double dBaseWidth, double dBaseLength);
void Entity_SetSphere(Entity* pEntity_entity, double dSphereRadius);
void Entity_SetCone(Entity* pEntity_entity, double dConeHeight, double dBottomCircleRadius, double dTopCircleRadius);
void Entity_SetLine(Entity* pEntity_entity, double d, double dX, double dY, double dZ, double dX, double dY, double dZ);
void Entity_SetPosition(Entity* pEntity_entity, double dPositionX, double dPositionY, double dPositionZ);
void Entity_SetHeading(Entity* pEntity_entity, double dYaw, double dPitch, double dRoll);
void Entity_SetSize(Entity* pEntity_entity, double dHeight, double dWidth, double dLength);
void Entity_SetTransparency(Entity* pEntity_entity, double dTransparency);
void Entity_SetColor(Entity* pEntity_entity, double dRed, double dGreen, double dBlue);
void Entity_SetShininess(Entity* pEntity_entity, double dShininess);
short Entity_CollisionWithEntity(Entity* pEntity_entityThis, Entity* pEntity_entityOther);
short Entity_CollisionWithWorld(Entity* pEntity_entity, World* pWorld_world);
short Entity_DistanceFromEntity(Entity* pEntity_entityThis, Entity* pEntity_entityOther, double* pdDistance);
short Entity_DistanceFromWorld(Entity* pEntity_entity, World* pWorld_world, int iPlaneDirection, double* pdDistance);
short Entity_AngleFromWorld(Entity* pEntity_e, World* pWorld_w, int iPlaneDir, double* pdY, double* pdP, double* pdR);

Physics* Entity_GetPhysics(Entity* pEntity_entity);

void Entity_SetStateString(Entity* pEntity_entity, const char* pcStateName, const char* pcStateString);
short Entity_GetStateStringEquals(Entity* pEntity_entity, const char* pcStateName, const char* pcStateString);
void Entity_GetStateString(Entity* pEntity_entity, const char* pcStateName, char* pcStateString);
void Entity_SetStateNumber(Entity* pEntity_entity, char* pcStateName, double dStateNumber);
short Entity_GetStateNumberEquals(Entity* pEntity_entity, char* pcStateName, double dStateNumber);
short Entity_GetStateNumberBetween(Entity* pEntity_entity, char* pcStateName, double dStateMin, double dStateMax);
void Entity_GetStateNumber(Entity* pEntity_entity, char* pcStateName, double* pdStateNumber);

void Entity_SetSourceNode(Entity* pEntity_entity, int iNodeID);
```

```
Physics* Physics_NewPhysics();
Physics* Physics_LastPhysics();
void Physics_Destroy(Physics* pPhysics_body);

void Physics_SetDynamicsType(Physics* pPhysics_body, int iDynamicsType);
void Physics_SetBaseFrequency(Physics* pPhysics_body, double dFrequency);
void Physics_SetComponentIndex(Physics* pPhysics_body, int iComponentIndex);

void Physics_SetPosition(Physics* pPhysics_body, double dPositionX, double dPositionY, double dPositionZ);
void Physics_SetHeading(Physics* pPhysics_body, double dYaw, double dPitch, double dRoll);
void Physics_GetPosition(Physics* pPhysics_body, double* pdPositionX, double* pdPositionY, double* pdPositionZ);
void Physics_GetHeading(Physics* pPhysics_body, double* pdYaw, double* pdPitch, double* pdRoll);

void Physics_SetAbsoluteVelocity(Physics* pPhysics_body, double dVelocityX, double dVelocityY, double dVelocityZ);
void Physics_SetAbsoluteAcceleration(Physics* pPhysics_body, double dAccelX, double dAccelY, double dAccelZ);
void Physics_SetAbsoluteAngularVelocity(Physics* pPhysics_body, double dVelYaw, double dVelPitch, double dVelRoll);
void Physics_SetAbsoluteAngularAcceleration(Physics* pPhysics_body, double dAYaw, double dAPitch, double dARoll);
void Physics_SetAbsoluteForce(Physics* pPhysics_body, double dForceX, double dForceY, double dForceZ);
void Physics_SetAbsoluteAngularForce(Physics* pPhysics_body, double dFYaw, double dFPitch, double dFRoll);

void Physics_SetRelativeVelocity(Physics* pPhysics_body, double dVelocityX, double dVelocityY, double dVelocityZ);
void Physics_SetRelativeAcceleration(Physics* pPhysics_body, double dAccelX, double dAccelY, double dAccelZ);
void Physics_SetRelativeAngularVelocity(Physics* pPhysics_body, double dVYaw, double dVPitch, double dVRoll);
void Physics_SetRelativeAngularAcceleration(Physics* pPhysics_body, double dAYaw, double dAPitch, double dARoll);
void Physics_SetRelativeForce(Physics* pPhysics_body, double dForceX, double dForceY, double dForceZ);
void Physics_SetRelativeAngularForce(Physics* pPhysics_body, double dFYaw, double dFPitch, double dFRoll);

void Physics_SetDifferentialPosition(Physics* pPhysics_body, double dPositionX, double dPositionY, double dPositionZ);
void Physics_SetDifferentialHeading(Physics* pPhysics_body, double dYaw, double dPitch, double dRoll);
void Physics_SetDifferentialVelocity(Physics* pPhysics_body, double dVelocityX, double dVelocityY, double dVelocityZ);
void Physics_SetDifferentialAcceleration(Physics* pPhysics_body, double dAX, double dAY, double dAZ);
void Physics_SetDifferentialAngularVelocity(Physics* pPhysics_body, double dVYaw, double dVPitch, double dVRoll);
void Physics_SetDifferentialAngularAcceleration(Physics* pPhysics_body, double dYaw, double dPitch, double dRoll);
void Physics_SetDifferentialForce(Physics* pPhysics_body, double dForceX, double dForceY, double dForceZ);
void Physics_SetDifferentialAngularForce(Physics* pPhysics_body, double dFYaw, double dFPitch, double dFRoll);

void Physics_SetMass(Physics* pPhysics_body, double dMass);
void Physics_SetFriction(Physics* pPhysics_body, double dFrictionConstant, double dFrictionUpperBound);
void Physics_SetAccelerationThreshold(Physics* pPhysics_body, double dBodyAcceleration);
void Physics_SetVelocityThreshold(Physics* pPhysics_body, double dBodyVelocity);

void Physics_SetInputParameters(Physics* pPhysics_body, double dParamX, double dParamY, double dParamZ,
                                double dParamYaw, double dParamPitch, double dParamRoll, double dParamThrust);

void Physics_SetSurfaceDistance(Physics* pPhysics_body, int iPlaneDirection, double dDistance);
void Physics_SetSurfaceAngle(Physics* pPhysics_body, int iPlaneDirection, double dPitch, double dYaw, double dRoll);

void Physics_AttachToPhysics(Physics* pPhysics_physics, Physics* pPhysics_parent);
void Physics_DetachFromPhysics(Physics* pPhysics_physics);

void Physics_SetSourceNode(Physics* pPhysics_body, int iNodeID);
```

IV. C Sound Component

```
#define MAX_SOUNDS 500
#define FADER_TIME 60
#define ENGINE_TIME 15
#define POSITIONER_TIME 30

typedef struct SoundPhysics {
  double dPositionVector[6];
  double dVelocityVector[4];
} SoundPhysics;

typedef struct Obstruction {
  double dWidth;
  double dHeight;
  double dLength;

  double dPositionX;
  double dPositionY;
  double dPositionZ;
} Obstruction;

typedef struct Sound {
  Obstruction*   pObstruction_obstruction;
  struct Sound* pSound_next;
  struct Sound* pSound_prev;

  int      iIdentification;
  char*  pcFileName;
  char*  pcName;

  struct SoundPhysics* pPhysics_model;
  struct Entity*             pEntity_attachment;

  int             iChannel;
  double        dVolume;
  double        dFader;
  double        dPitch;

  short         bIsRelativeToObserver;
  short         bIsPreparing;
  short         bIsStopped;
  short         bIsPaused;
  short         bIsPlaying;
  short         bIsLooping;

  short         bIsInFader;
  short         bIsOutFader;
  short         bIsInOutFader;

  Mix_Chunk*     pMixChunk_sound;

  short bIsOn;
} Sound;

typedef struct SoundScene {
  struct Sound*           pSound_first;
  struct Sound*           pSound_current;
  struct Sound*           pSound_last;
  struct Sound*           pSound_select;
  struct SoundPhysics* pPhysics_observer;

  int iEngineStepTime;
} SoundScene;
```

```
typedef struct StorageDevice {
  double  dPollFrequency;
  int        iIdentification;
  char*    pcName;

  SDL_CD* pSDL_CD_cdrom;

  short   bIsPlaying;
  short   bIsPaused;
  short   bIsEjected;

  short   bTrackChanged;

  int     iTrackAmount;
  int     iTrackNumber;

  double  dTrackLength;
  double  dTrackMinutes;
  double  dTrackSeconds;
  double  dSizeMegabytes;
} StorageDevice;

static struct SoundScene SoundScene_scene;
static struct StorageDevice StorageDevice_cdrom;

void CompactDisk_SetVolume(double dVolume);
void CompactDisk_SetTrack(int iTrackNumber);
void CompactDisk_PlayTrack(int iTrackNumber);

void CompactDisk_Play();
void CompactDisk_Stop();
void CompactDisk_Pause();

Sound* Sound_NewSound();

void Sound_SetObserverOff(Sound* pSound_sound);
void Sound_SetObserverOn(Sound* pSound_sound);

void Sound_SetFile(Sound* pSound_source, char* pcFileName);
void Sound_SetFadeIn(Sound* pSound_source);
void Sound_SetFadeOut(Sound* pSound_source);
void Sound_SetFadeInOut(Sound* pSound_source);
void Sound_SetFadeOff(Sound* pSound_source);
void Sound_SetLoopedOn(Sound* pSound_source);
void Sound_SetLoopedOff(Sound* pSound_source);
void Sound_SetVolume(Sound* pSound_source, double dVolume);
void Sound_SetPosition(Sound* pSound_source, double dPosX, double dPosY, double dPosZ);
void Sound_SetVelocity(Sound* pSound_source, double dMagnitude, double dVelX, double dVelY, double dVelZ);
void Sound_SetPitch(Sound* pSound_source, double dPitch);
void Sound_UpdatePosition(Sound* pSound_source);

int  Sound_UpdateChannelID(Sound* pSound_source);

void Sound_Play(Sound* pSound_source);
void Sound_Pause(Sound* pSound_source);
void Sound_Stop(Sound* pSound_source);
void Sound_Destroy(Sound* pSound_source);

void SoundEngine_SetObserver(double dPosX, double dPosY, double dPosZ,
                             double dYaw,  double dPitch,  double dRoll);

void SoundEngine_SetFrequency(double dFrequency);
void SoundEngine_Initialise();
void SoundEngine_Destroy();

void SoundEngine_Update();
```

V. C Device Component

```
#define JOYSTICK_THRESHOLD 0.02
#define JOYSTICK_DAMPING 6.0
#define JOYSTICK_MAX_THRESHOLD 0.65
#define JOYSTICK_CALIBRATION_POINT 0.85

typedef struct DeviceChannel {
  pfPipeWindow* pfPipeWindow_Window;
  pfPipe*       pfPipe_Pipe;

  Display*      pDisplay_screen;
  Window        Window_root;

  char*         pcWindowTitle;

  int           iPositionX;
  int           iPositionY;
  int           iSizeX;
  int           iSizeY;

  int           iScreenX;
  int           iScreenY;
} DeviceChannel;
```

```
typedef struct InputDevice {
  double dPollFrequency;
  int       iIdentification;
  char*   pcName;

  pfuMouse           pfuMouse_mouse;           //performer mouse
  pfuEventStream  pfuEventStream_stream;   //performer events
  SDL_Event        SDL_Event_event;           //sdl events
  SDL_Joystick*    pSDL_Joystick_joy;          //sdl joystick

  double dOrigin[5];
  double dOriginX;
  double dOriginY;
  double dOriginZ;
  double dOriginP;
  double dOriginQ;

  double dPosition[5];
  double dPositionX;
  double dPositionY;
  double dPositionZ;
  double dPositionP;
  double dPositionQ;

  double dVelocity[5];
  double dVelocityX;
  double dVelocityY;
  double dVelocityZ;
  double dVelocityP;
  double dVelocityQ;

  double dClickPositionX[10];
  double dClickPositionY[10];
  double dClickPositionZ[10];
  double dClickPositionP[10];
  double dClickPositionQ[10];

  double dReleasePositionX[10];
  double dReleasePositionY[10];
  double dReleasePositionZ[10];
  double dReleasePositionP[10];
  double dReleasePositionQ[10];

  short  bButtonDoubleClick[15];
  short  bButtonClick[15];
  short  bButtonDown[15];
  short  bButtonDrag[15];

  char    cKeyPressed;
  int       iKeyPressed;
  short  bHasState[10];

  int iNumberOfAxes;
  int iNumberOfHats;
  int iNumberOfButtons;
  int iNumberOfBalls;

  //parallel extensions
  #if CLUSTER
    int iSourceNodeID;
    Stream* pStream_dOrigin[MAX_NODES][5];
  #endif
} InputDevice;
```

```
static struct DeviceChannel    DeviceChannel_input;
static struct InputDevice      InputDevice_mouse;
static struct InputDevice      InputDevice_keyboard;
static struct InputDevice      InputDevice_joystick;

void Devices_SetInput(pfPipeWindow* pfPipeWindow_window);

void Devices_Initialise();
void Devices_Update();
void Devices_Destroy();

void Joystick_Initialise();
void Mouse_Initialise();
void Keyboard_Initialise();

void Mouse_Update();
void Joystick_Update();
void Keyboard_Update();

void Joystick_Destroy();
void Mouse_Destroy();
void Keyboard_Destroy();

void Mouse_SetLocalPointer();
void Mouse_SetGlobalPointer();
void Mouse_SetPointerOn();
void Mouse_SetPointerOff();

double Mouse_GetVelocityX();
double Mouse_GetVelocityY();
double Mouse_GetPositionX();
double Mouse_GetPositionY();
double Mouse_GetClickPositionX();
double Mouse_GetClickPositionY();
double Mouse_GetReleasePositionX();
double Mouse_GetReleasePositionY();

short Mouse_GetButtonDownLeft();
short Mouse_GetButtonDownMiddle();
short Mouse_GetButtonDownRight();
short Mouse_GetButtonDragLeft();
short Mouse_GetButtonDragMiddle();
short Mouse_GetButtonDragRight();
short Mouse_GetButtonClickLeft();
short Mouse_GetButtonClickMiddle();
short Mouse_GetButtonClickRight();
short Mouse_GetButtonDoubleClickLeft();
short Mouse_GetButtonDoubleClickMiddle();
short Mouse_GetButtonDoubleClickRight();

double Joystick_GetPitch();
double Joystick_GetYaw();
double Joystick_GetRoll();
double Joystick_GetPositionX();
double Joystick_GetPositionY();
double Joystick_GetPositionZ();
double Joystick_GetThrottle();

short Joystick_GetHatDownUp();
short Joystick_GetHatDownDown();
short Joystick_GetHatDownLeft();
short Joystick_GetHatDownRight();
```

```
short Joystick_GetFireDown(int iFireButton);
short Joystick_GetFireDownOne();
short Joystick_GetFireDownTwo();
short Joystick_GetFireDownThree();
short Joystick_GetFireDownFour();
short Joystick_GetFireDownFive();
short Joystick_GetFireDownSix();
short Joystick_GetFireDownSeven();
short Joystick_GetFireDownEight();
short Joystick_GetFireDownNine();
short Joystick_GetFireDownTen();

short Joystick_GetFireClick(int iFireButton);
short Joystick_GetFireClickOne();
short Joystick_GetFireClickTwo();
short Joystick_GetFireClickThree();
short Joystick_GetFireClickFour();
short Joystick_GetFireClickFive();
short Joystick_GetFireClickSix();
short Joystick_GetFireClickSeven();
short Joystick_GetFireClickEight();
short Joystick_GetFireClickNine();
short Joystick_GetFireClickTen();

short Joystick_GetHatClickUp();
short Joystick_GetHatClickDown();
short Joystick_GetHatClickLeft();
short Joystick_GetHatClickRight();

short Keyboard_GetRightArrowKey();
short Keyboard_GetLeftArrowKey();
short Keyboard_GetUpArrowKey();
short Keyboard_GetDownArrowKey();
short Keyboard_GetSpaceKey();

char Keyboard_GetNormalKey();

short Keyboard_GetFunctionKey();
short Keyboard_GetEscKey();
short Keyboard_GetTabKey();
short Keyboard_GetEnterKey();

short Keyboard_GetLeftControlKey();
short Keyboard_GetRightControlKey();
short Keyboard_GetLeftShiftKey();
short Keyboard_GetRightShiftKey();
short Keyboard_GetLeftShiftControlKey();
short Keyboard_GetRightShiftControlKey();
short Keyboard_GetLeftAltKey();
short Keyboard_GetRightAltKey();
```

VI. C++ Cluster Component

```
#ifndef STREAMINGENGINE_H_HEADER_INCLUDED_BFECFEEC
#define STREAMINGENGINE_H_HEADER_INCLUDED_BFECFEEC
class Stream;

class StreamingEngine
{
  public:

  void updateState();

  short hasNodeID(int iNodeID);
  short hasNodeName(char* pcNodeName);
  int getNodeID();
  char* getNodeName();

  void setUpdated();
  short isUpdated();

  void initialise(int iArgumentCount, char* pacArgumentContents[]);

  void printState(int iNodeID);
  void update();

  int limitMaximumStreamCount();
  int getNumberOfNodes();
  int getNumberOfFloatStreams();
  int getNumberOfDoubleStreams();
  int getNumberOfIntegerStreams();
  int getNumberOfLongStreams();
  int getNumberOfStringStreams();
  int getNumberOfFloats(int iNodeID);
  int getNumberOfDoubles(int iNodeID);
  int getNumberOfIntegers(int iNodeID);
  int getNumberOfLongs(int iNodeID);
  int getNumberOfStrings(int iNodeID);

  void sendMessageBroadcast(int iEngineStreamID, int iSourceNodeID);
  void receiveMessageBroadcast(int iEngineStreamID, int iSourceNodeID);
  void sendFloatBroadcast(int iEngineStreamID, int iSourceNodeID);
  void receiveFloatBroadcast(int iEngineStreamID, int iSourceNodeID);
  void sendDoubleBroadcast(int iEngineStreamID, int iSourceNodeID);
  void receiveDoubleBroadcast(int iEngineStreamID, int iSourceNodeID);
  void sendIntBroadcast(int iEngineStreamID, int iSourceNodeID);
  void receiveIntBroadcast(int iEngineStreamID, int iSourceNodeID);
  void sendLongBroadcast(int iEngineStreamID, int iSourceNodeID);
  void receiveLongBroadcast(int iEngineStreamID, int iSourceNodeID);
  void sendStringBroadcast(int iEngineStreamID, int iSourceNodeID);
  void receiveStringBroadcast(int iEngineStreamID, int iSourceNodeID);


  private:

  int iNodes;
  int iSelfNodeID;

  Stream* pStream_first;
  Stream* pStream_last;
  Stream* pStream_current;

  int iMaxCount;
  int iMaxStreamCount;
  int iFloatStreamCount;
  int iDoubleStreamCount;
  int iIntegerStreamCount;
  int iLongStreamCount;
  int iStringStreamCount;
```

```
  short bIsInitialised;
  short bIsDirect;
  short bIsUpdated;

  int iIntegerCount[MAX_NODES];
  int iLongCount[MAX_NODES];
  int iFloatCount[MAX_NODES];
  int iDoubleCount[MAX_NODES];
  int iStringCount[MAX_NODES];
};


#endif /* STREAMINGENGINE_H_HEADER_INCLUDED_BFECFEEC */


#ifndef STREAM_H_HEADER_INCLUDED_BFECB61E
#define STREAM_H_HEADER_INCLUDED_BFECB61E

class Stream
{
  public:

  void new(int iTypeID, int iNodeID);
  short isOfType(int iTypeID);
  void destroy();

  void setSourceID(int iSourceID);
  void getSourceID(int* piSourceID);
  void getStreamID(int* piStreamID);
  void getEngineStreamID(int* piStreamID);


  private:

  void newString(int iNodeID);
  void newInteger(int iNodeID);
  void newLong(int iNodeID);
  void newFloat(int iNodeID);
  void newDouble(int iNodeID);

  Stream* pStream_prev;
  Stream* pStream_next;

  int iIdentification;
  int iEngineIndex;
  int iIndex;

  short bIsInteger;
  short bIsLong;
  short bIsFloat;
  short bIsDouble;
  short bIsString;

  int iSourceID;
  int iDestinationID;
};


#endif /* STREAM_H_HEADER_INCLUDED_BFECB61E */
```

```
#ifndef MESSAGESTREAM_H_HEADER_INCLUDED_BFEC9E55
#define MESSAGESTREAM_H_HEADER_INCLUDED_BFEC9E55
#include "Stream.h"

class MessageStream : public Stream
{
  public:

  void setLongValue(long lValue);
  void getLongValue(long* plValue);
  void setDoubleValue(double dValue);
  void getDoubleValue(double* pdValue);
  void setIntValue(int iValue);
  void getIntValue(int* piValue);
  void setFloatValue(float fValue);
  void getFloatValue(float* pfValue);
  void setStringValue(const char* pcString);
  void getStringValue(char* pcString);


  private:

  long alSequence[MAX_STREAMSLOTS];
  int aiSequence[MAX_STREAMSLOTS];
  double adSequence[MAX_STREAMSLOTS];
  float afSequence[MAX_STREAMSLOTS];
  char acSequence[MAX_STREAMSLOTS*MAX_STRING_SIZE];
};



#endif /* MESSAGESTREAM_H_HEADER_INCLUDED_BFEC9E55 */



#ifndef STRINGSTREAM_H_HEADER_INCLUDED_BFECDCCC
#define STRINGSTREAM_H_HEADER_INCLUDED_BFECDCCC

class StringStream : public Stream
{
  public:

  void setStringValue(const char* pcString);
  void getStringValue(char* pcString);


  private:

  char acSequence[MAX_STREAMSLOTS*MAX_STRING_SIZE];
};



#endif /* STRINGSTREAM_H_HEADER_INCLUDED_BFECDCCC */
```

```
#ifndef DOUBLESTREAM_H_HEADER_INCLUDED_BFECAF99
#define DOUBLESTREAM_H_HEADER_INCLUDED_BFECAF99
#include "Stream.h"

class DoubleStream : public Stream
{
  public:

  void setDoubleValue(double dValue);
  void getDoubleValue(double* pdValue);


  private:

  double adSequence[MAX_STREAMSLOTS];
};



#endif /* DOUBLESTREAM_H_HEADER_INCLUDED_BFECAF99 */



#ifndef FLOATSTREAM_H_HEADER_INCLUDED_BFECF704
#define FLOATSTREAM_H_HEADER_INCLUDED_BFECF704
#include "Stream.h"

class FloatStream : public Stream
{
  public:

  void setFloatValue(float fValue);
  void getFloatValue(float* pfValue);


  private:

  float afSequence[MAX_STREAMSLOTS];
};



#endif /* FLOATSTREAM_H_HEADER_INCLUDED_BFECF704 */
```

```
#ifndef LONGSTREAM_H_HEADER_INCLUDED_BFECEF47
#define LONGSTREAM_H_HEADER_INCLUDED_BFECEF47
#include "Stream.h"

class LongStream : public Stream
{
  public:

  void setLongValue(long lValue);
  void getLongValue(long* plValue);


  private:

  long alSequence[MAX_STREAMSLOTS];
};



#endif /* LONGSTREAM_H_HEADER_INCLUDED_BFECEF47 */



#ifndef INTEGERSTREAM_H_HEADER_INCLUDED_BFECB2B2
#define INTEGERSTREAM_H_HEADER_INCLUDED_BFECB2B2
#include "Stream.h"

class IntegerStream : public Stream
{
  public:

  void setIntegerValue(int iValue);
  void getIntegerValue(int* piValue);


  private:

  int aiSequence[MAX_STREAMSLOTS];
};



#endif /* INTEGERSTREAM_H_HEADER_INCLUDED_BFECB2B2 */
```

VII. C++ Synchronisation Component

```
#ifndef REALTIMEBARRIER_H_HEADER_INCLUDED_BFECB8DE
#define REALTIMEBARRIER_H_HEADER_INCLUDED_BFECB8DE
class RealtimeClock;

class RealtimeBarrier
{
  public:

  void placeGlobal();
  void placeLocal();

  double getTimeMicroseconds();
  double getTimeMilliseconds();
  double getTimeSeconds();

  void setTimeUnitAsMicroseconds();
  void setTimeUnitAsMilliseconds();
  void setTimeUnitAsSeconds();
  void setFrequency(double dFrequency);
  void setLatencyPercentage(double dMinPercentage, double dMaxPercentage);
  void setStartTime(double dStartTime);
  void setEndTime(double dEndTime);

  double getCurrentTime();

  void setDirect();
  void setNormal();
  void setReactive();
  void setGlobal();
  void setLocal();
  void reset();
  void place();
  void startPulseGenerator();


  private:

  short bIsLogged;
  short bIsReactive;
  short bIsNormal;
  short bIsDirect;
  short bIsGlobal;
  short bWasTriggered;
  short bTriggerReleased;

  int iBarrierID;
  int iMissedFrames;

  double dCurrentTrigger;
  double dCurrentInterrupts;
  double dCurrentTime;
  double dPrevTime;
  double dStepTime;
  double dMinLatency;
  double dMaxLatency;
  double dStartTime;
  double dEndTime;
  double dUnitsPerSecond;
  double adPastTriggers[100];

  RealtimeClock* pRealtimeClock_clock;
};


#endif /* REALTIMEBARRIER_H_HEADER_INCLUDED_BFECB8DE */
```

```
#ifndef REALTIMECLOCK_H_HEADER_INCLUDED_BFEC89E4
#define REALTIMECLOCK_H_HEADER_INCLUDED_BFEC89E4
class RealtimeBarrier;

class RealtimeClock
{
  public:

  double getMicroseconds();
  double getMilliseconds();
  double getSeconds();
  double reset();

  void sleepMicroseconds(double dTime);
  void sleepMilliseconds(double dTime);
  void sleepSeconds(double dTime);

  double getStartTimeMicroseconds();
  double getStartTimeMilliseconds();
  double getStartTimeSeconds();


  private:

  double dClockStartTime;
  RealtimeBarrier* pRealtimeBarrier_barrier;
};



#endif /* REALTIMECLOCK_H_HEADER_INCLUDED_BFEC89E4 */
```

VIII. C++ Visual Component

```
#ifndef VISUALENGINE_H_HEADER_INCLUDED_BFECE08A
#define VISUALENGINE_H_HEADER_INCLUDED_BFECE08A

class VisualEngine
{
  public:

  void initialise(int iArgumentCount, char* pacArgumentContents[]);
  void setEngineFrequency(double dFrequency);
  void setSwapbufferFrequency(double dFrequency);
  void setDynamicInterleaveOn();
  void setDynamicInterleaveOff();
  void setDynamicFrequencyOn();
  void setDynamicFrequencyOff();

  void printState(int iNodeID);

  void update();
  void destroy();

  private:

  short bIsInitialised;
  short bIsParallel;
  short bHasDynamicInterleave;
  short bHasDynamicFrequency;

  double dFrequency;
  double dStepTime;
  double dSwapBufferFrequency;
  double dSwapBufferStepTime;
  double dAverageStepTime;
  double dAverageSwapbufferStepTime;
  double dAverageEngineFrequency;
  double dAverageSwapbufferFrequency;
  double dFrequencyRatio;

  double dCurrentTime;
  long lCurrentFrame;

  short bUpdatedStreams;
};


#endif /* VISUALENGINE_H_HEADER_INCLUDED_BFECE08A */
```

```
#ifndef CHANNEL_H_HEADER_INCLUDED_BFECAE8A
#define CHANNEL_H_HEADER_INCLUDED_BFECAE8A

class Channel
{
  public:

  void setRelativePosition(double dPositionX, double dPositionY, double dPositionZ);
  void setRelativeHeading(double dYaw, double dPitch, double dRoll);
  void setViewingAngle(double dHorizontalAngle, double dVerticalAngle);
  void setViewingDistance(double dNearest, double dFarthest);
  void setPosition(int iPositionX, int iPositionY);
  void setSize(int iWidth, int iHeight);
  void setTitle(char* pcTitle);
  void setFullscreen();
  void setWindowed();
  void setStereo();

  void setStereoDepth(double dStereo);
  void setStereoObserverLookat(double dPositionX, double dPositionY, double dPositionZ);
  void setStereoObserverPosition(double dPositionX, double dPositionY, double dPositionZ);
  void setStereoObserverHeading(double dYaw, double dPitch, double dRoll);

  void setSourceNode(int iNodeId);
  void setRasterAngles(double dHorizontalAngle, double dVerticalAngle);
  void setRasterPlace(int iHorizontal, int iVertical, int iNodeID);
  void display();


  private:

  pfVec3 pfVec3_RelativePosition;
  pfVec3 pfVec3_RelativeHeading;
  pfVec3 pfVec3_RelativeObserverPosition;
  pfVec3 pfVec3_RelativeObserverHeading;
  pfVec3 pfVec3_StereoObserverLookat;
  pfVec3 pfVec3_StereoObserverPosition;
  pfVec3 pfVec3_StereoObserverHeading;

  pfPipeWindow* pfPipeWindow_Window;
  pfChannel* pfChannel_ChannelDefault;
  pfChannel* pfChannel_ChannelLeftEye;
  pfChannel* pfChannel_ChannelRightEye;
  pfPipe* pfPipe_Pipe;

  double dStereoAngle;
  char* pcWindowTitle;
  int iPositionX;
  int iPositionY;
  int iSizeX;
  int iSizeY;

  double dNearestPoint;
  double dFarthestPoint;
  double dHorizontalAngle;
  double dVerticalAngle;

  short bHasStereoBuffers;
  short bIsFullscreen;
  short bIsWindowed;
  short bIsStereo;
  short bIsOn;

  int iSourceNodeID;
  Stream* pStream_bIsOn;
};


#endif /* CHANNEL_H_HEADER_INCLUDED_BFECAE8A */
```

```
#ifndef CAMERA_H_HEADER_INCLUDED_BFECD3A3
#define CAMERA_H_HEADER_INCLUDED_BFECD3A3
class Entity;

class Camera
{
  public:

  void attachToEntity(Entity* pEntity_entity);
  void detachFromEntity();

  void setFreeViewing();
  void setCinematicViewing();
  void setOutsideViewing();
  void setInsideViewing();
  void setLockedViewing();

  void setRelativeHeading(double dYaw, double dPitch, double dRoll);
  void setRelativeDistance(double dDistance);
  void setPosition(double dPositionX, double dPositionY, double dPositionZ);
  void setHeading(double dYaw, double dPitch, double dRoll);

  void setSourceNode(int iNodeID);


  private:

  short bIsCinematic;
  short bIsLocked;
  short bIsInside;
  short bIsOutside;
  short bIsFree;
  short bChangedView;

  pfCoordd pfCoordd_offset;
  pfCoordd pfCoordd_camera;

  Entity* pEntity_attachment;

  int iSourceNodeID;
  int iSourceCameraID;

  Stream* pStream_cameraX;
  Stream* pStream_cameraY;
  Stream* pStream_cameraZ;
  Stream* pStream_cameraH;
  Stream* pStream_cameraP;
  Stream* pStream_cameraR;
};


#endif /* CAMERA_H_HEADER_INCLUDED_BFECD3A3 */
```

```
#ifndef SCENE_H_HEADER_INCLUDED_BFECE03B
#define SCENE_H_HEADER_INCLUDED_BFECE03B
class Light;
class Entity;
class Fog;

class Scene
{
  public:

  void update();
  void checkDuplicates();

  void setFogOn();
  void setFogOff();
  void setCloudsOn();
  void setCloudsOff();

  void setGlobalFogColor(double dRed, double dGreen, double dBlue);
  void setGlobalFogDensity(double dFogDensity);

  Light* lastLight();
  Entity* lastEntity();
  World* lastWorld();
  Fog* lastFog();


  private:

  int iNumberOfLights;
  int iNumberOfEntities;
  int iNumberOfWorlds;
  int iNumberOfFogs;

  Light* pLight_first;
  World* pWorld_first;
  Fog* pFog_first;
  Entity* pEntity_first;

  Light* pLight_current;
  World* pWorld_current;
  Fog* pFog_current;
  Entity* pEntity_current;

  Light* pLight_last;
  World* pWorld_last;
  Fog* pFog_last;
  Entity* pEntity_last;

  Light* pLight_active;
  World* pWorld_active;
  Entity* pEntity_active;
  Fog* pFog_active;

  pfScene* pfScene_scene;

  pfVec3 pfVec3_FogDensity;
  pfVec3 pfVec3_FogColor;
  pfVec3 pfVec3_CloudHeight;

  pfEarthSky* pfEarthSky_clear;
  pfVolFog* pfVolFog_clouds;
  pfFog* pfFog_fog;

  short bIsInitialised;
  short bCloudsOn;
  short bFogOn;
};

#endif /* SCENE_H_HEADER_INCLUDED_BFECE03B */
```

```
#ifndef SCENEOBJECT_H_HEADER_INCLUDED_BFEC8726
#define SCENEOBJECT_H_HEADER_INCLUDED_BFEC8726

class SceneObject
{
};



#endif /* SCENEOBJECT_H_HEADER_INCLUDED_BFEC8726 */



#ifndef WORLD_H_HEADER_INCLUDED_BFEC899A
#define WORLD_H_HEADER_INCLUDED_BFEC899A
class Physics;
class Camera;

class World : public SceneObject
{
  public:

  void newWorld();
  World* lastWorld();

  void destroy();
  void setOn();
  void setOff();

  void setFile(char* pcFileName);
  void setViewingDistance(double dDistance);
  void setPagingDistance(double dDistance);
  void setParallelFile(char* pcFileName);
  void setPagingRatio(double dValue);


  private:

  World* pWorld_next;
  World* pWorld_prev;

  Camera* pCamera_camera;

  int iIdentification;
  char* pcName;

  Physics* pPhysics_model;
  pfSwitch* pfSwitch_switch;
  pfSphere* pfSphere_bounds;
  pfNode* pfNode_world;

  char* pcFileName;
  short bIsOn;
  int iSourceNodeID;
  short pStream_bIsOn;
};


#endif /* WORLD_H_HEADER_INCLUDED_BFEC899A */
```

```
#ifndef FOG_H_HEADER_INCLUDED_BFEC971A
#define FOG_H_HEADER_INCLUDED_BFEC971A
class Physics;
class Entity;

class Fog : public SceneObject
{
  public:

  void newFog();
  Fog* lastFog();
  void destroy();

  void setOn();
  void setOff();

  void attachToEntity(Entity* pEntity_entity);
  void detachFromEntity();

  void setFogArea(Entity* pEntity_entity);
  void setFogColor(double dRed, double dGreen, double dBlue);
  void setFogDensity(double dDensity);
  void setFogTexture(char* pcTextureFileName);

  Physics* getPhysics();

  void setSourceNode(int iNodeID);
  void setPosition(double dPositionX, double dPositionY, double dPositionZ);
  void setHeading(double dYaw, double dPitch, double dRoll);


  private:

  Fog* pFog_next;
  Fog* pFog_prev;

  int iIdentification;

  Fog* pfFog_fog;
  pfVolFog* pfVolFog_fog;
  pfVec3 pfVec3_FogDensity;
  pfVec3 pfVec3_FogColor;

  short bFogOn;
  int iSourceNodeID;

  Physics* pPhysics_model;
  Entity* pEntity_entity;
  Stream* pStream_bIsOn;
  Entity* pEntity_area;
};


#endif /* FOG_H_HEADER_INCLUDED_BFEC971A */
```

```
#ifndef ENTITY_H_HEADER_INCLUDED_BFEC81D7
#define ENTITY_H_HEADER_INCLUDED_BFEC81D7

class Entity : public SceneObject
{
  public:

  void newEntity();
  Entity* lastEntity();

  void destroy();
  void setOn();
  void setOff();

  void setFile(char* pcFileName);
  void setClonedEntity(Entity* pEntity_use);
  void attachToEntity(Entity* pEntity_parent);
  void detachFromEntity();

  void setText(char* pcText);

  void setPlane(double dHeight, double dWidth);
  void setCube(double dHeight, double dWidth, double dLength);
  void setPyramid(double dHeight, double dBaseWidth, double dBaseLenght);
  void setSphere(double dSphereRadius);
  void setCone(double dConeHeight, double dBottomCircleRadius, double dTopCircleRadius);
  void setLine(double dWidth, double dStartX, double dStartY, double dStartZ,
               double dEndX, double dEndY, double dEndZ);

  void setPosition(double dPositionX, double dPositionY, double dPositionZ);
  void setHeading(double dYaw, double dPitch, double dRoll);
  void setSize(double dHeight, double dWidth, double dLength);

  void setTransparency(double dTransparency);
  void setColor(double dRed, double dGreen, double dBlue);
  void setShininess(double dShininess);

  short collisionWithEntity(Entity* pEntity_other);
  short collisionWithWorld(World* pWorld_world);

  void distanceFromEntity(Entity* pEntity_other, double* pdDistance);
  void distanceFromWorld(World* pWorld_world, int iPlaneDirection, double* pdDistance);
  void angleFromWorld(World* pWorld_world, int iPlaneDirection, double* pdYaw, double* pdPitch, double* pdRoll);

  Physics* getPhysics();

  void setStateString(char* pcStateName, char* pcStateString);
  void getStateString(char* pcStateName, char* pcStateString);
  void setStateNumber(char* pcStateName, double dStateNumber);
  void getStateNumber(char* pcStateName, double* pdStateNumber);

  void setSourceNode(int iNodeID);


  private:

  Entity* pEntity_next;
  Entity* pEntity_prev;

  int iIdentification;
  char* pcName;

  Physics* pPhysics_model;
  Entity* pEntity_parent;
  Camera* pCamera_attachment;
```

```
  char* pcFileName;
  pfSwitch* pfSwitch_switch;
  pfSphere* pfSphere_bounds;
  pfBox* pfBox_bounds;
  pfNode* pfNode_model;
  pfGeoSet* pfGeoSet_geometry;
  pfGeode* pfGeode_primitive;
  pfGeoState* pfGeoState_attributes;
  pfMaterial* pfMaterial;
  pfText* pfText_text;
  pfFont* pfFont_font;
  pfString* pfString_string;

  int iStringStateCounter;
  int iNumberStateCounter;

  char aacStateStringName[MAX_ENTITY_STATES][MAX_ENTITY_STRINGSIZE];
  char aacStateStringValue[MAX_ENTITY_STATES][MAX_ENTITY_STRINGSIZE];
  char aacStateNumberName[MAX_ENTITY_STATES][MAX_ENTITY_STRINGSIZE];
  char adStateNumberValue[MAX_ENTITY_STATES];

  short bIsSelected;
  short bIsDragged;
  short bIsOn;

  int iSourceNodeID;

  Stream* pStream_bIsSelected;
  Stream* pStream_bIsDragged;
  Stream* pStream_bIsOn;
};


#endif /* ENTITY_H_HEADER_INCLUDED_BFEC81D7 */
```

```
#ifndef LIGHT_H_HEADER_INCLUDED_BFEC94F7
#define LIGHT_H_HEADER_INCLUDED_BFEC94F7
class Entity;

class Light : public SceneObject
{
  public:

  void newLight();
  Light* lastLight();
  void destroy();

  void setOn();
  void setOff();
  void attachToEntity(Entity* pEntity);
  void detachFromEntity();

  void setToBeam(double dSourceCone, double dTargetCone, double dLength, double dSpread);
  void setToPoint(double dRadius, double dSpread);
  void setDistance(double dDistanceAttenuation);

  void setAmbient(double dAmbient);
  void setDiffuse(double dDiffuse);
  void setSpecular(double dSpecular);

  void setAmbientColor(double dAmbientRed, double dAmbientGreen, double dAmbientBlue);
  void setDiffuseColor(double dDiffuseRed, double dDiffuseGreen, double dDiffuseBlue);
  void setSpecularColor(double dSpecularRed, double dSpecularGreen, double dSpecularBlue);

  void setBackgroundReflection(double dAmbient);
  void setSurfaceReflection(double dDiffuse);
  void setShinyReflection(double dSpecular);

  void setBackgroundColor(double dRed, double dGreen, double dBlue);
  void setSurfaceColor(double dRed, double dGreen, double dBlue);
  void setShinyColor(double dRed, double dGreen, double dBlue);
  void setColor(double dRed, double dGreen, double dBlue);

  void setPosition(double dPositionX, double dPositionY, double dPositionZ);
  void setHeading(double dYaw, double dPitch, double dRoll);

  Physics* getPhysics();


  private:

  Light* pLight_next;
  Light* pLight_prev;

  char* pcFileName;
  char* pcName;

  int iIndentification;

  double dAmbientRed;
  double dAmbientGreen;
  double dAmbientBlue;

  double dDiffuseRed;
  double dDiffuseGreen;
  double dDiffuseBlue;

  double dSpecularRed;
  double dSpecularGreen;
  double dSpecularBlue;

  double dAmbient;
  double dDiffuse;
  double dSpecular;
```

```
  pfSwitch* pfSwitch_switch;
  Physics* pPhysics_model;
  Entity* pEntity_parent;
  pfLightSource* pfLightSource_light;
  pfTexture* pfTexture_texture;
  pfFrustum* pfFrustum_frustum;
  pfFog* pfFog_fog;
  pfNode* pfNode_model;

  short bIsOn;

  int iSourceNodeID;
  Stream* pStream_bIsOn;
};


#endif /* LIGHT_H_HEADER_INCLUDED_BFEC94F7 */

#ifndef PHYSICSSCENE_H_HEADER_INCLUDED_BFECB33D
#define PHYSICSSCENE_H_HEADER_INCLUDED_BFECB33D

class PhysicsScene
{
  public:

  void updateChildren();
  void updateScene();
  void clearScene();

  void setFrequency(double dFrequency);


  private:

  PhysicsScene* pPhysicsScene_parent;
  PhysicsScene* pPhysicsScene_next;
  Physics* pPhysics_physics;
};


#endif /* PHYSICSSCENE_H_HEADER_INCLUDED_BFECB33D */
```

```
#ifndef PHYSICS_H_HEADER_INCLUDED_BFECBC50
#define PHYSICS_H_HEADER_INCLUDED_BFECBC50
class PhysicsScene;

class Physics
{
  public:

  void update();
  void setBaseFrequency(double dFrequency);
  void newPhysics();

  Physics* lastPhysics();

  void destroy();

  void setDynamicsType(int iDynamicsType);
  void setComponentIndex(int iComponentIndex);
  void setPosition(double dPositionX, double dPositionY, double dPositionZ);
  void setHeading(double dPitch, double dYaw);
  void getPosition(double* pdPositionX, double* pdPositionY, double* pdPositionZ);
  void getHeading(double* pdYaw, double* pdPitch, double* pdRoll);

  void setAbsoluteVelocity(double dVelocityX, double dVelocityY, double dVelocityZ);
  void setAbsoluteAcceleration(double dAccelerationX, double dAccelerationY, double dAccelerationZ);
  void setAbsoluteAngularVelocity(double dVelocityYaw, double dVelocityPitch, double dVelocityRoll);
  void setAbsoluteAngularAcceleration(double dAccelerationYaw, double dAccelerationPitch, double dAccelerationRoll);
  void setAbsoluteForce(double dForceX, double dForceY, double dForceZ);
  void setAbsoluteAngularForce(double dForceYaw, double dForcePitch, double dForceRoll);

  void setRelativeVelocity(double dVelocityX, double dVelocityY, double dVelocityZ);
  void setRelativeAcceleration(double dAccelerationX, double dAccelerationY, double dAccelerationZ);
  void setRelativeAngularVelocity(double dVelocityYaw, double dVelocityPitch, double dVelocityRoll);
  void setRelativeAngularAcceleration(double dAccelerationYaw, double dAccelerationPitch, double dAccelerationRoll);
  void setRelativeForce(double dForceX, double dForceY, double dForceZ);
  void setRelativeAngularForce(double dForceYaw, double dForcePitch, double dForceRoll);

  void setDifferentialPosition(double dPositionX, double dPositionY, double dPositionZ);
  void setDifferentialVelocity(double dVelocityX, double dVelocityY, double dVelocityZ);
  void setDifferentialAcceleration(double dAccelerationX, double dAccelerationY, double dAccelerationZ);
  void setDifferentialAngularVelocity(double dVelocityYaw, double dVelocityPitch, double dVelocityRoll);
  void setDifferentialAngularAcceleration(double dAccelerationYaw, double dAccelerationPitch, double dAccelerationRoll);
  void setDifferentialForce(double dForceX, double dForceY, double dForceZ);
  void setDifferentialAngularForce(double dForceYaw, double dForcePitch, double dForceRoll);

  void setMass(double dMass);
  void setFriction(double dFrictionConstant, double dFrictionUpperBound);
  void setAccelerationThreshold(double dBodyAcceleration);
  void setVelocityThreshold(double dBodyVelocity);

  void setInputParameters(double dParameterX, double dParameterY, double dParameterZ, double dParameterYaw,
                          double dParameterPitch, double dParameterRoll, double dParameterThrust);

  void setSurfaceDistance(int iPlaneDirection, double dPitch, double dYaw, double dRoll);
  void setSurfaceAngle(int iPlaneDirection, double dPitch, double dYaw, double dRoll);

  void attachToPhysics(Physics* pPhysics_parent);
  void detachFromPhysics();

  void setSourceNode(int iNodeID);
```

```
  private:

  double dFrequency;
  PhysicsScene* pPhysicsScene_child;
  Physics* pPhysics_parent;

  int iComponentIndex;
  int iDynamicsType;
  int iHeadingThresholdType;
  int iPositionThresholdType;
  int iSourceNodeID;

  double dPitchSign;
  double dBodyMass;
  double dBodyFrictionConstant;
  double dBodyFrictionUpperBound;
  double dBodyAccelerationThreshold;
  double dBodyVelocityThreshold;

  double dMinYaw;
  double dMaxYaw;
  double dMinPitch;
  double dMaxPitch;
  double dMinRoll;
  double dMaxRoll;
  double dMinPositionX;
  double dMaxPositionX;
  double dMinPositionY;
  double dMaxPositionY;
  double dMinPositionZ;
  double dMaxPositionZ;

  pfDoubleDCS* pfDoubleDCS_PositionVector;

  double dDifferentialPositionVector[MAX_FORCES][6];
  double dFilteredPositionVector[6];
  double dPositionVector[6];
  double dSurfaceVector[6];
  double dFrictionvector[6];
  double dInputParameters[7];

  double dAbsoluteForceVector[MAX_FORCES][6];
  double dAbsoluteVelocityVector[MAX_FORCES][3];
  double dAbsoluteAccelerationVector[MAX_FORCES][3];
  double dAbsoluteAngularVelocityVector[MAX_FORCES][3];
  double dAbsoluteAngularAccelerationVector[MAX_FORCES][3];

  double dRelativeForceVector[MAX_FORCES][6];
  double dRelativeVelocityVector[MAX_FORCES][3];
  double dRelativeAccelerationVector[MAX_FORCES][3];
  double dRelativeAngularVelocityVector[MAX_FORCES][3];
  double dRelativeAngularAccelerationVector[MAX_FORCES][3];

  double dDifferentialForceVector[MAX_FORCES][6];
  double dDifferentialVelocityVector[MAX_FORCES][3];
  double dDifferentialAccelerationVector[MAX_FORCES[3];
  double dDifferentialAngularVelocityVector[MAX_FORCES][3];
  double dDifferentialAngularAccelerationVector[MAX_FORCES][3];

  Stream* pStream_PositionVectorX;
  Stream* pStream_PositionVectorY;
  Stream* pStream_PositionVectorZ;
  Stream* pStream_PositionVectorH;
  Stream* pStreamPositionVectorP;
  Stream* pStreamPositionVectorR;
};


#endif /* PHYSICS_H_HEADER_INCLUDED_BFECBC50 */
```

IX. C++ Sound Component

```
#ifndef SOUNDENGINE_H_HEADER_INCLUDED_BFEA2303
#define SOUNDENGINE_H_HEADER_INCLUDED_BFEA2303

class SoundEngine
{
  public:

  void setObserver(double dPosX, double dPosY, double dPosZ, double dYaw, double dPitch, double dRoll);
  void setFrequency(double dFrequency);

  void initialise();
  void destroy();
  void update();


  private:

  int iEngineStepTime;
  double dEngineFrequency;
};



#endif /* SOUNDENGINE_H_HEADER_INCLUDED_BFEA2303 */



#ifndef SOUNDSCENE_H_HEADER_INCLUDED_BFEA7027
#define SOUNDSCENE_H_HEADER_INCLUDED_BFEA7027
class SoundObstruction;
class Sound;
class SoundPhysics;

class SoundScene
{
  public:

  void synchronise();
  void initialise();
  void destroy();


  private:

  Sound* pSound_first;
  Sound* pSound_current;
  Sound* pSound_last;
  Sound* pSound_select;

  SoundObstruction* pSoundObstruction_first;
  SoundObstruction* pSoundObstruction_current;
  SoundObstruction* pSoundObstruction_last;

  SoundPhysics* pPhysics_observer;
};



#endif /* SOUNDSCENE_H_HEADER_INCLUDED_BFEA7027 */
```

```
#ifndef SOUND_H_HEADER_INCLUDED_BFEA6DF1
#define SOUND_H_HEADER_INCLUDED_BFEA6DF1

class Sound
{
  public:

  void newSound();
  void setObserverOn();
  void setObserverOff();
  void setObstructedOn();
  void setObstructedOff();

  void setFile(char* pcFileName);
  void setFadeIn();
  void setFadeOut();
  void setFadeInOut();
  void setFadeOff();
  void setLoopedOn();
  void setLoopedOff();

  void setVolume(double dVolume);
  void setPosition(double dPosX, double dPosY, double dPosZ);
  void setVelocity(double dVelocityX, double dVelocityY, double dVelocityZ);
  void setPitch(double dPitch);
  void updatePosition();
  void updateChannelID();

  void play();
  void pause();
  void stop();
  void destroy();


  private:

  Sound* pSound_next;
  Sound* pSound_prev;

  int iIdentification;
  char* pcFileName;
  char* pcName;

  SoundPhysics* pPhysics_model;

  int iChannel;
  double dVolume;
  double dFader;
  double dPitch;

  short bIsRelativeToObserver;
  short bIsObstructed;
  short bIsPreparing;
  short bIsStopped;
  short bIsPaused;
  short bIsPlaying;
  short bIsLooping;
  short bIsInFader;
  short bIsOutFader;
  short bIsInOutFader;

  Mix_Chunk* pMixChunk_sound;

  short bIsOn;
};


#endif /* SOUND_H_HEADER_INCLUDED_BFEA6DF1 */
```

```
#ifndef SOUNDOBSTRUCTION_H_HEADER_INCLUDED_BFEA50DA
#define SOUNDOBSTRUCTION_H_HEADER_INCLUDED_BFEA50DA

class SoundObstruction
{
  public:

  void newObstruction();
  void setArea(double dWidth, double dHeight);
  void setPosition(double dPositionX, double dPositionY, double dPositionZ);
  void setReflectance(double dReflectance);


  private:

  double dWidth;
  double dHeight;
  double dPositionX;
  double dPositionY;
  double dPositionZ;
  double dReflectance;

  Obstruction* pObstruction_next;
  Obstruction* pObstruction_prev;
};



#endif /* SOUNDOBSTRUCTION_H_HEADER_INCLUDED_BFEA50DA */



#ifndef SOUNDPHYSICS_H_HEADER_INCLUDED_BFEA524D
#define SOUNDPHYSICS_H_HEADER_INCLUDED_BFEA524D

class SoundPhysics
{
  public:

  void setPosition(double dPosX, double dPosY, double dPosZ);
  void setHeading(double dYaw, double dPitch, double dRoll);
  void setVelocity(double dVelX, double dVelY, double dVelZ);


  private:

  double dPositionVector[6];
  double dVelocityVector[4];
};



#endif /* SOUNDPHYSICS_H_HEADER_INCLUDED_BFEA524D */
```

```
#ifndef STORAGEDEVICE_H_HEADER_INCLUDED_BFEA39E0
#define STORAGEDEVICE_H_HEADER_INCLUDED_BFEA39E0

class StorageDevice
{
  public:

  void initialise();
  void destroy();


  private:

  double dPollFrequency;
  int iIdentification;
  char* pcName;

  SDL_CD* pSDL_CD_cdrom;

  short bIsPlaying;
  short bIsPaused;
  short bIsEjected;
  short bTrackChanged;

  int iTrackAmount;
  int iTrackNumber;
  double dTrackLength;
  double dTrackMinutes;
  double dTrackSeconds;
  double dSizeMegabytes;
};


#endif /* STORAGEDEVICE_H_HEADER_INCLUDED_BFEA39E0 */


#ifndef COMPACTDISK_H_HEADER_INCLUDED_BFEA32F9
#define COMPACTDISK_H_HEADER_INCLUDED_BFEA32F9
#include "StorageDevice.h"

class CompactDisk : public StorageDevice
{
  public:

  void setVolume(double dVolume);
  void setTrack(int iTrackNumber);
  void playTrack(int iTrackNumber);

  void play();
  void stop();
  void pause();
};


#endif /* COMPACTDISK_H_HEADER_INCLUDED_BFEA32F9 */
```

```
#ifndef MEMORYSTICK_H_HEADER_INCLUDED_BFEA3D25
#define MEMORYSTICK_H_HEADER_INCLUDED_BFEA3D25
#include "StorageDevice.h"

class MemoryStick : public StorageDevice
{
  public:

  void setSound(int iSoundNumber);
  void playSound(int iSoundNumber);

  void play();
  void stop();
  void pause();

  void setVolume(double dVolume);
};


#endif /* MEMORYSTICK_H_HEADER_INCLUDED_BFEA3D25 */
```

X. C++ Device Component

```
#ifndef DEVICEENGINE_H_HEADER_INCLUDED_BFEA46E2
#define DEVICEENGINE_H_HEADER_INCLUDED_BFEA46E2
class InputDevice;

class DeviceEngine
{
  public:

  void initialise();
  void update();
  void destroy();


  private:

  InputDevice* apInputDevice_device[10];
};



#endif /* DEVICEENGINE_H_HEADER_INCLUDED_BFEA46E2 */



#ifndef INPUTDEVICE_H_HEADER_INCLUDED_BFEA3138
#define INPUTDEVICE_H_HEADER_INCLUDED_BFEA3138

class InputDevice
{
  public:


  private:

  double dPollFrequency;
  int iIdentification;
  char* pcName;

  pfuMouse pfuMouse_mouse;
  pfuEventStream pfuEventStream_stream;
  SDL_Event SDL_Event_event;
  SDL_Joystick* pSDL_Joystick_joy;

  double dOriginX;
  double dOriginY;
  double dOriginZ;
  double dOriginP;
  double dOriginQ;
  double dOriginR;

  double dPositionX;
  double dPositionY;
  double dPositionZ;
  double dPositionP;
  double dPositionQ;
  double dPositionR;

  double dVelocityX;
  double dVelocityY;
  double dVelocityZ;
  double dVelocityP;
  double dVelocityQ;
  double dVelocityR;

  char cKeyPressed;
  int iKeyPressed;
```

```
  int iNumberOfAxes;
  int iNumberOfHats;
  int iNumberOfButtons;
  int iNumberOfBalls;
  int iSourceNodeID;
  short bIsOfType;

  double dOrigin[6];
  double dPosition[6];
  double dVelocity[6];

  double dClickPositionX[10];
  double dClickPositionY[10];
  double dClickPositionZ[10];
  double dClickPositionP[10];
  double dClickPositionQ[10];
  double dClickPositionR[10];

  double dReleasePositionX[10];
  double dReleasePositionY[10];
  double dReleasePositionZ[10];
  double dReleasePositionP[10];
  double dReleasePositionQ[10];
  double dReleasePositionR[10];

  short bButtonDoubleClick[15];
  short bButtonClick[15];
  short bButtonDown[15];
  short bButtonDrag[15];
  short bHasState[10];

  Stream* pStream_dOrigin[MAX_NODES][6];
};


#endif /* INPUTDEVICE_H_HEADER_INCLUDED_BFEA3138 */


#ifndef DEVICECHANNEL_H_HEADER_INCLUDED_BFEA3240
#define DEVICECHANNEL_H_HEADER_INCLUDED_BFEA3240

class DeviceChannel
{
  pfPipeWindow* pfPipeWindow_Window;
  pfPipe* pfPipe_Pipe;
  Display* pDisplay_screen;

  Window Window_root;

  char* pcWindowTitle;
  int iPositionX;
  int iPositionY;
  int iSizeX;
  int iSizeY;
  int iScreenX;
  int iScreenY;
};


#endif /* DEVICECHANNEL_H_HEADER_INCLUDED_BFEA3240 */
```

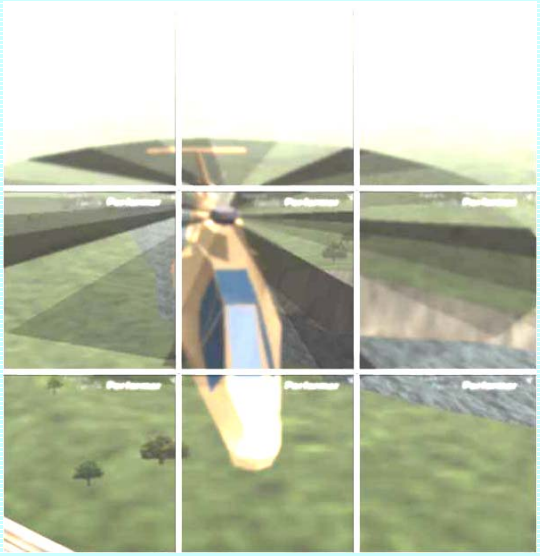
```
#ifndef DEVICEMOUSE_H_HEADER_INCLUDED_BFEA22D6
#define DEVICEMOUSE_H_HEADER_INCLUDED_BFEA22D6
#include "InputDevice.h"

class DeviceMouse : public InputDevice
{
  public:

  void initialise();
  void update();
  void destroy();

  void setLocalPointer();
  void setGlobalPointer();
  void setPointerOn();
  void setPointerOff();

  double getVelocityX();
  double getVelocityY();
  double getPositionX();
  double getPositionY();

  double getClickPositionX();
  double getClickPositionY();
  double getReleasePositionX();
  double getReleasePositionY();

  double getButtonDownLeft();
  double getButtonDownMiddle();
  double getButtonDownRight();
  double getButtonDragLeft();
  double getButtonDragMiddle();
  double getButtonDragRight();
  double getButtonClickLeft();
  double getButtonClickMiddle();
  double getButtonClickRight();
  double getButtonDoubleClickLeft();
  double getButtonDoubleClickMiddle();
  double getButtonDoubleClickRight();

  void setInput(pfPipeWindow* pfPipeWindow_window);
};


#endif /* DEVICEMOUSE_H_HEADER_INCLUDED_BFEA22D6 */
```

```
#ifndef DEVICEKEYBOARD_H_HEADER_INCLUDED_BFEA5E38
#define DEVICEKEYBOARD_H_HEADER_INCLUDED_BFEA5E38
#include "InputDevice.h"

class DeviceKeyboard : public InputDevice
{
  public:

  void initialise();
  void update();
  void destroy();

  short getRightArrowKey();
  short getLeftArrowKey();
  short getUpArrowKey();
  short getDownArrowKey();

  short getSpaceKey();
  short getNormalKey();
  short getFunctionKey();
  short getEscKey();
  short getTabKey();
  short getEnterKey();

  short getLeftControlKey();
  short getRightControlKey();
  short getLeftShiftKey();
  short getRightShiftKey();
  short getLeftShiftControlKey();
  short getRightShiftControlKey();
  short getLeftAltKey();
  short getRightAltKey();
};


#endif /* DEVICEKEYBOARD_H_HEADER_INCLUDED_BFEA5E38 */
```

```
#ifndef DEVICEJOYSTICK_H_HEADER_INCLUDED_BFEA19A9
#define DEVICEJOYSTICK_H_HEADER_INCLUDED_BFEA19A9
#include "InputDevice.h"

class DeviceJoystick : public InputDevice
{
  public:

  void initialise();
  void update();
  void destroy();

  double getPitch();
  double getYaw();
  double getRoll();

  double getPositionX();
  double getPositionY();
  double getPositionZ();
  double getThrottle();

  short getHatDownUp();
  short getHatDownDown();
  short getHatDownLeft();
  short getHatDownRight();

  short getFireDown(int iFireButton);
  short getFireClick(int iFireButton);

  short getHatClickUp();
  short getHatClickDown();
  short getHatClickLeft();
  short getHatClickRight();
};


#endif /* DEVICEJOYSTICK_H_HEADER_INCLUDED_BFEA19A9 */
```

Realtime Distributed Visualisation for Commercial Off The Shelf Clusters

BIJLAGE MODELS



Versie 3.01 1/7/04

Edgar Herbie Antonius van Tetering

TNO FEL

Nederlandse Organisatie voor Toegepast Natuurwetenschappelijk Onderzoek
TNO Fysisch Electrotechnisch Laboratorium

TNO-rapport
RDV.1.0.3

Realtime Distributed Visualisation

TNO Fysisch en Elektronisch Laboratorium

Oude Waalsdorperweg 63
Postbus 96864
2509 JG 's-Gravenhage

Telefoon 070 374 00 00
Fax 070 328 09 61

Datum
29 januari 2005

Auteur
E.H.A. van Tetering

Opdrachtnummer	001
Opdrachtgever	FEL
Organisatieonderdeel	0204
Projectbegeleider	H. Jense
Organisatieonderdeel	Command, Control & Simulation

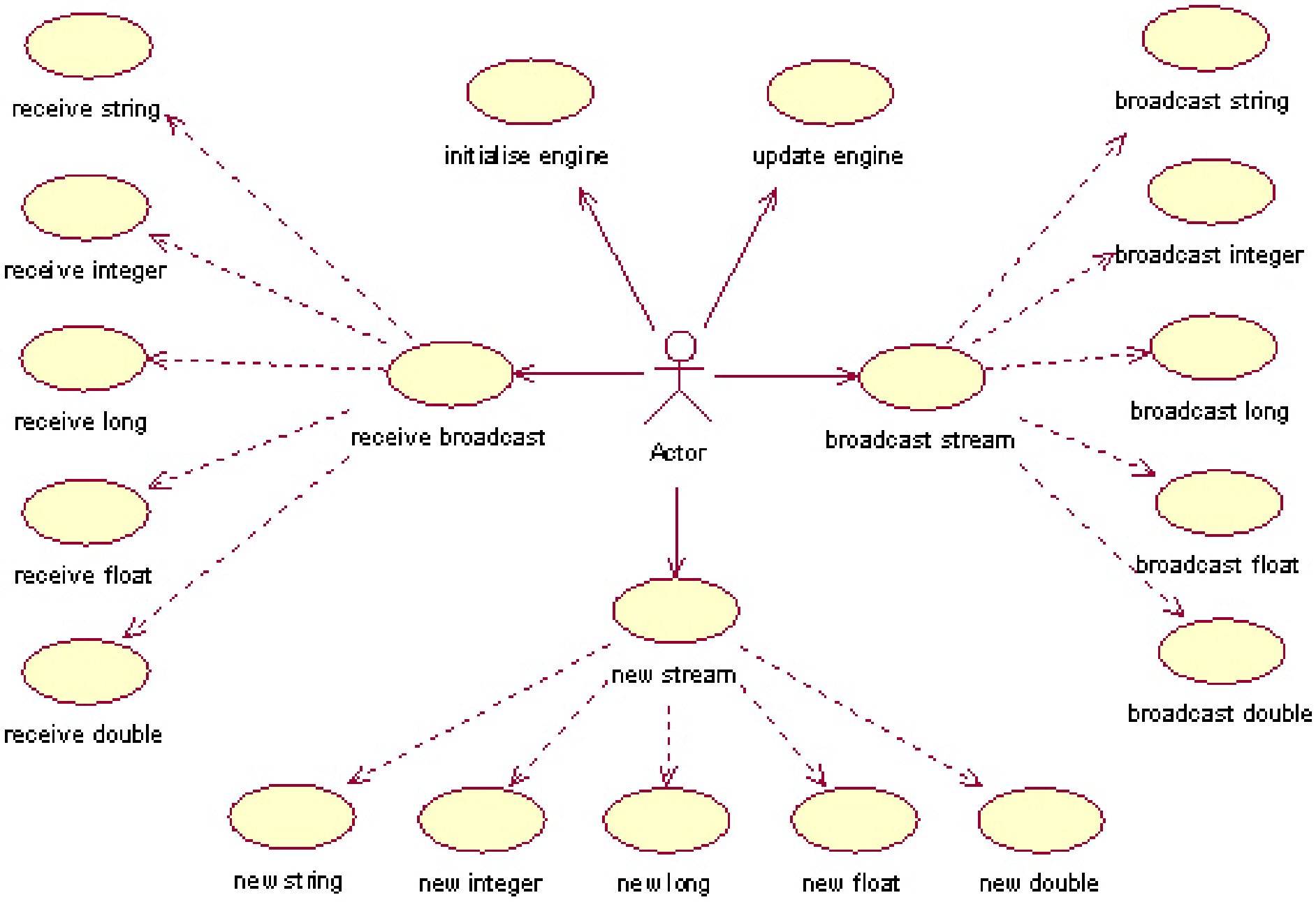
Rubricering	Ongeclassificeerd
Titel	Realtime Distributed Visualisation
Managementuittreksel	Niet van toepassing
Conclusies	Niet van toepassing
Rapporttekst	1-10
Bijlagen	Models, Headers, Manuals
Vastgesteld door	H. Jense
Vastgesteld d.d.	R. Krijnen

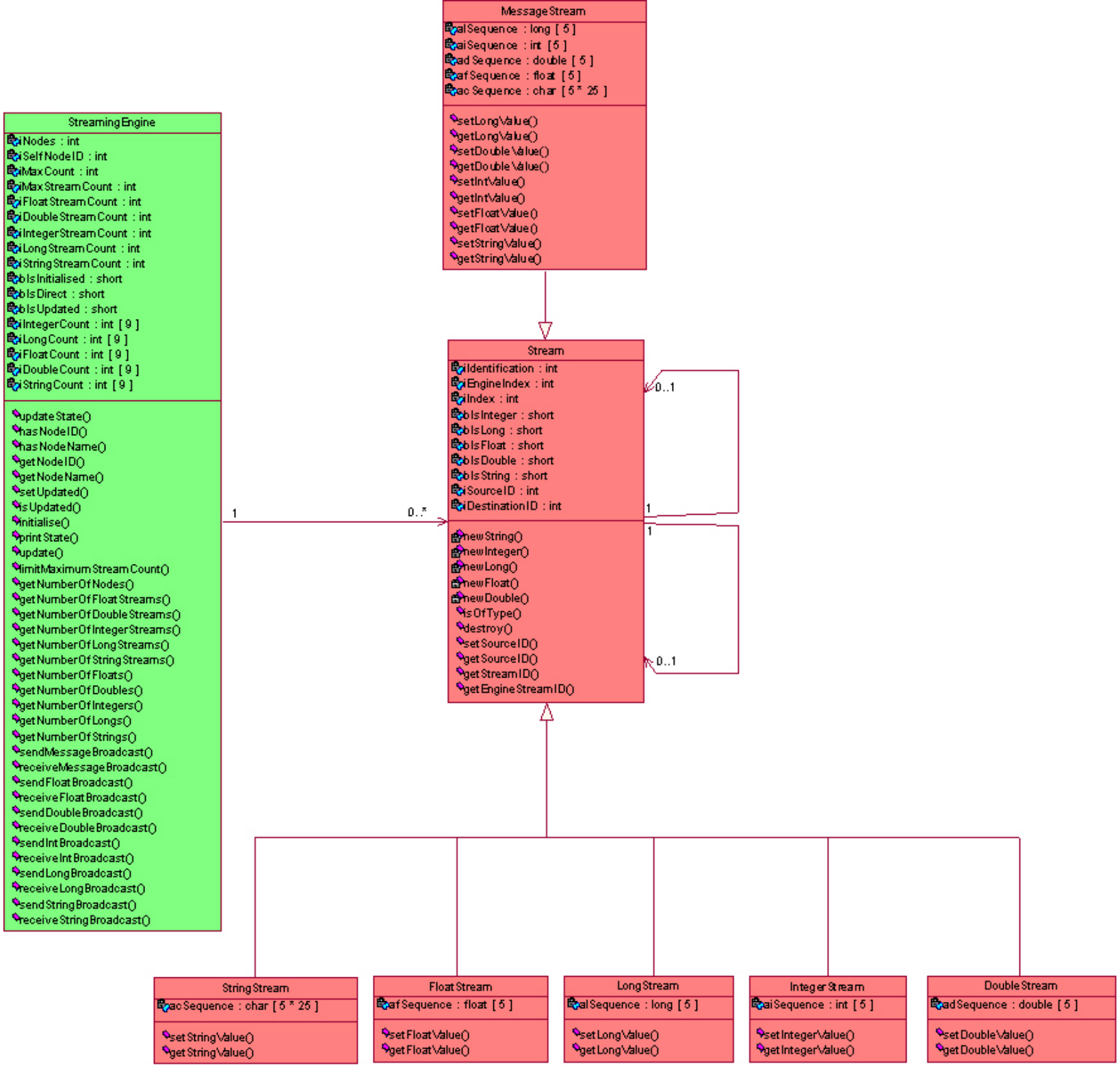
Exemplaarnr.	Models
Oplage	4
Aantal pagina's	91
Aantal bijlagen	3

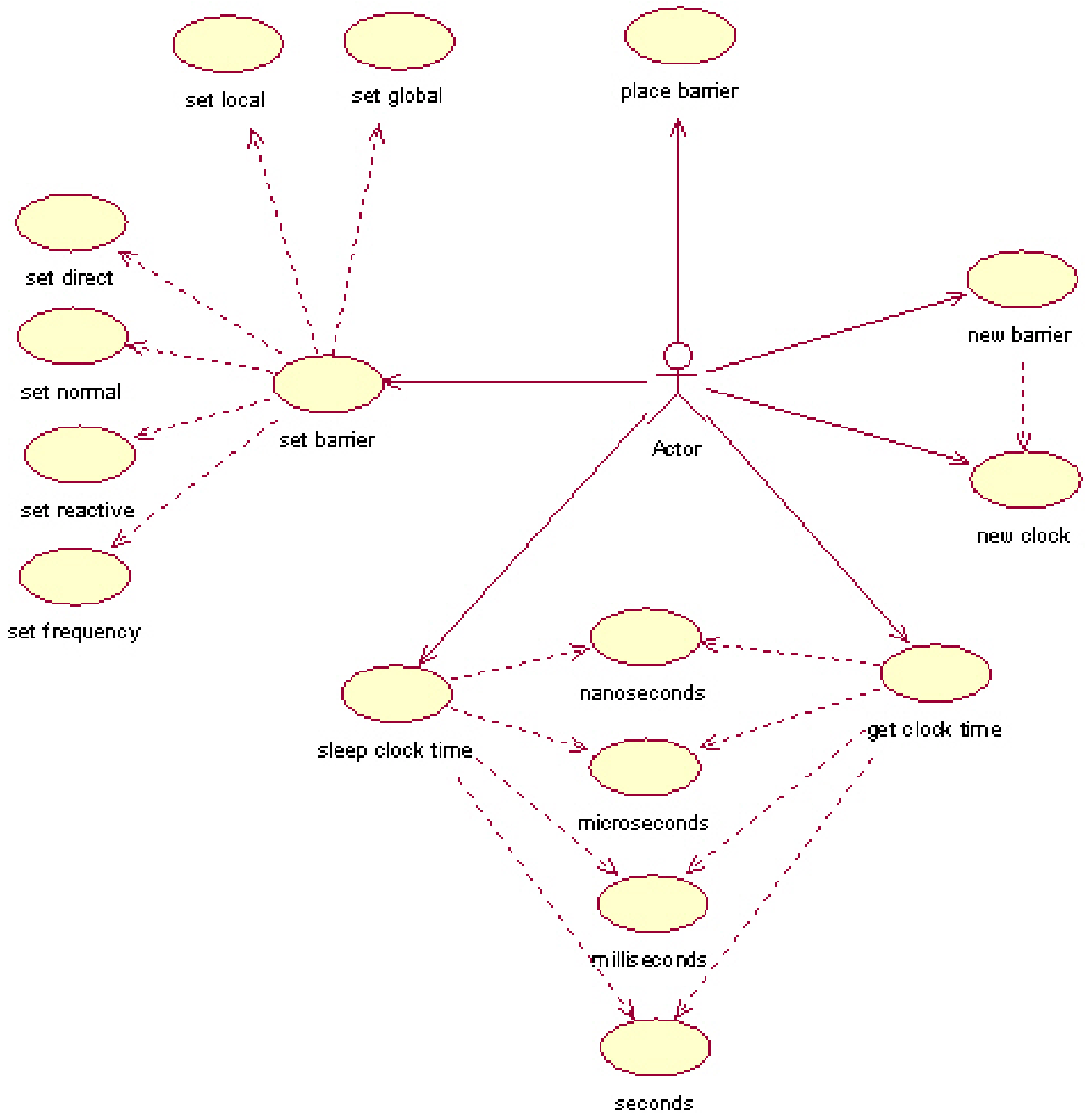
TNO Fysisch en Elektronisch Laboratorium is onderdeel
van TNO Defensieonderzoek waartoe verder behoren:

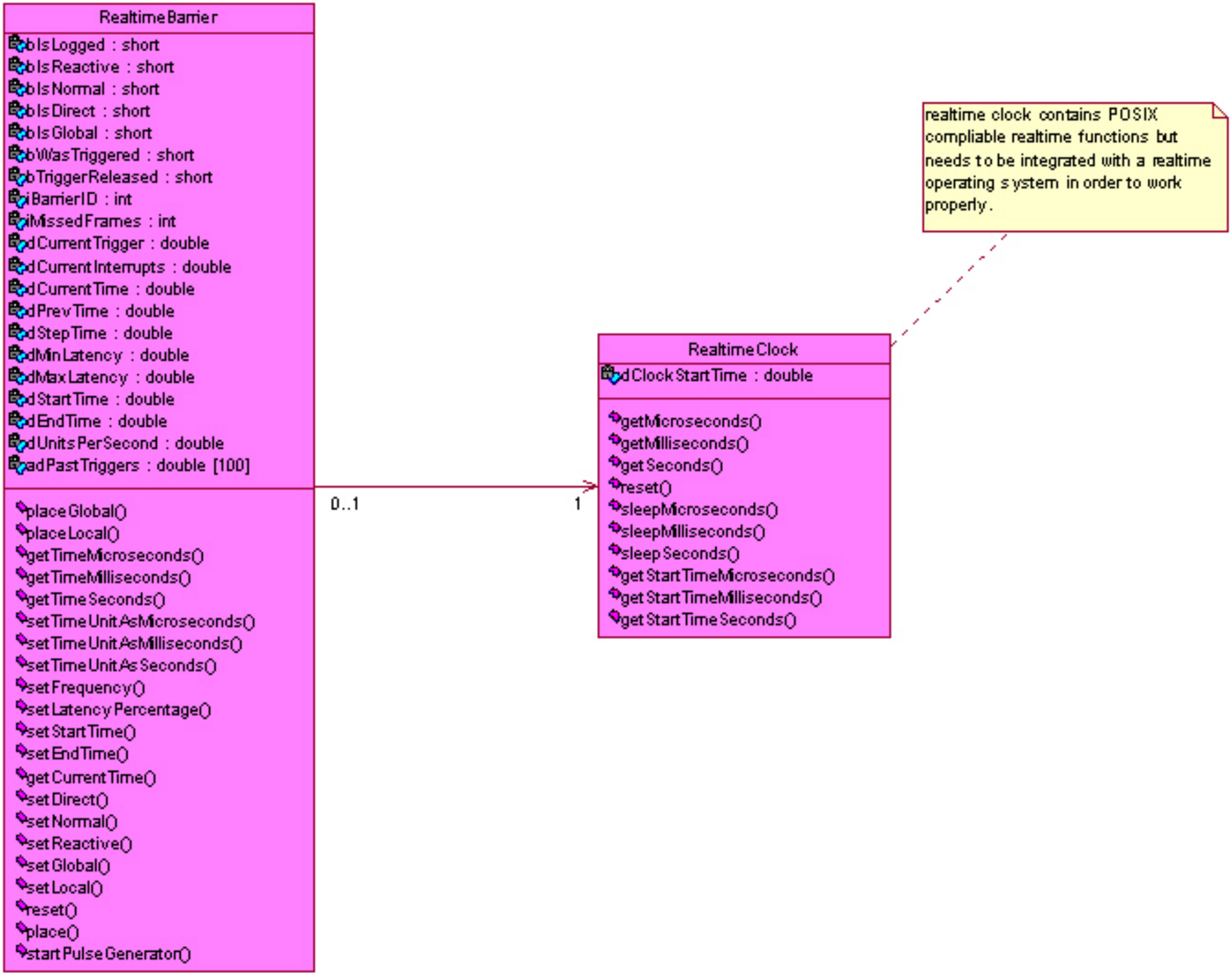
TNO Prins Maurits Laboratorium
TNO Technische Menskunde

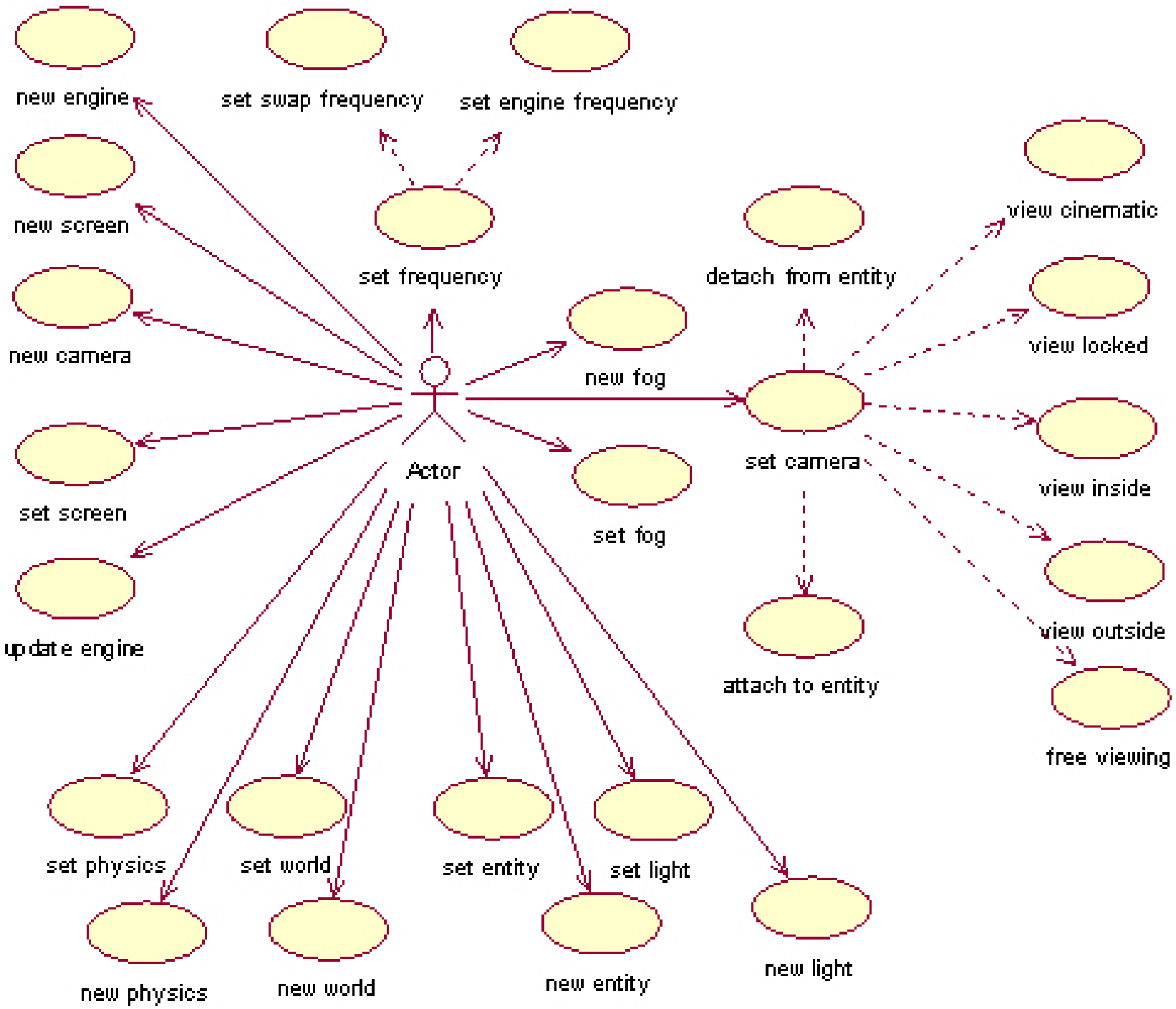
Nederlandse Organisatie voor toegepast-natuurwetenschappelijk onderzoek TNO

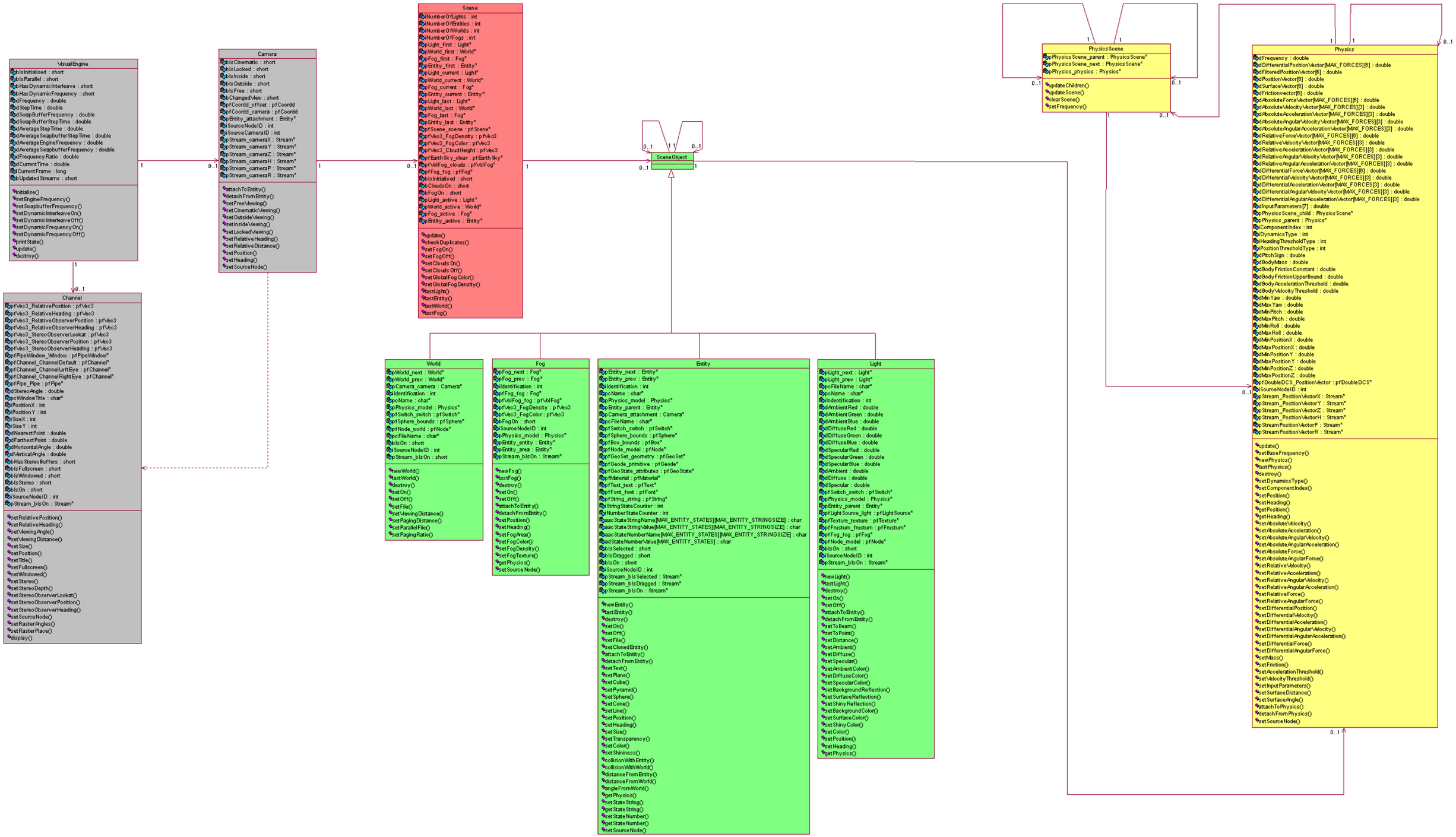

receive string
initialise engine
update engine
broadcast string
receive integer
broadcast integer
receive long
receive broadcast
Actor
broadcast stream
broadcast long
receive float
broadcast float
new stream
receive double
broadcast double
new string
new integer
new long
new float
new double

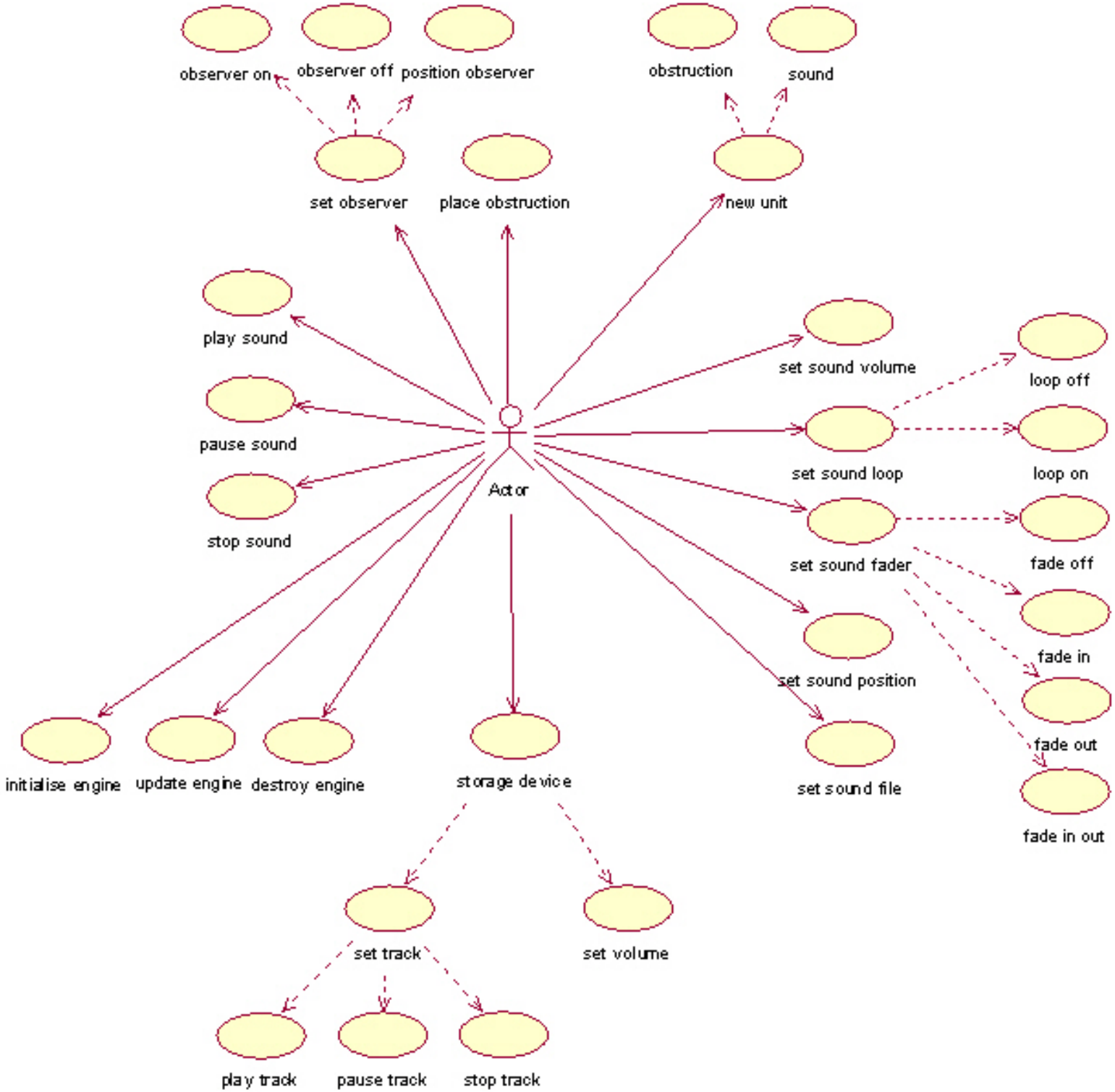

MessageStream
alSequence : long [5]
aiSequence : int [5]
adSequence : double [5]
afSequence : float [5]
acSequence : char [5 * 25]
setLongValue()
getLongValue()
setDoubleValue()
getDoubleValue()
setIntValue()
getIntValue()
setFloatValue()
getFloatValue()
setStringValue()
getStringValue()
StreamingEngine
iNodes : int
iSelfNodeID : int
iMaxCount : int
iMaxStreamCount : int
iFloatStreamCount : int
iDoubleStreamCount : int
iIntegerStreamCount : int
iLongStreamCount : int
iStringStreamCount : int
bIsInitialised : short
bIsDirect : short
bIsUpdated : short
iIntegerCount : int [9]
iLongCount : int [9]
iFloatCount : int [9]
iDoubleCount : int [9]
iStringCount : int [9]
updateState()
hasNodeID()
hasNodeName()
getNodeID()
getNodeName()
setUpdated()
isUpdated()
initialise()
printState()
update()
limitMaximumStreamCount()
getNumberOfNodes()
getNumberOfFloatStreams()
getNumberOfDoubleStreams()
getNumberOfIntegerStreams()
getNumberOfLongStreams()
getNumberOfStringStreams()
getNumberOfFloats()
getNumberOfDoubles()
getNumberOfIntegers()
getNumberOfLongs()
getNumberOfStrings()
sendMessageBroadcast()
receiveMessageBroadcast()
sendFloatBroadcast()
receiveFloatBroadcast()
sendDoubleBroadcast()
receiveDoubleBroadcast()
sendIntBroadcast()
receiveIntBroadcast()
sendLongBroadcast()
receiveLongBroadcast()
sendStringBroadcast()
receiveStringBroadcast()
1
0..*
Stream
iIdentification : int
iEngineIndex : int
iIndex : int
bIsInteger : short
bIsLong : short
bIsFloat : short
bIsDouble : short
bIsString : short
iSourceID : int
iDestinationID : int
newString()
newInteger()
newLong()
newFloat()
newDouble()
isOfType()
destroy()
setSourceID()
getSourceID()
getStreamID()
getEngineStreamID()
0..1
1
1
0..1
StringStream
acSequence : char [5 * 25]
setStringValue()
getStringValue()
FloatStream
afSequence : float [5]
setFloatValue()
getFloatValue()
LongStream
alSequence : long [5]
setLongValue()
getLongValue()
IntegerStream
aiSequence : int [5]
setIntegerValue()
getIntegerValue()
DoubleStream
adSequence : double [5]
setDoubleValue()
getDoubleValue()

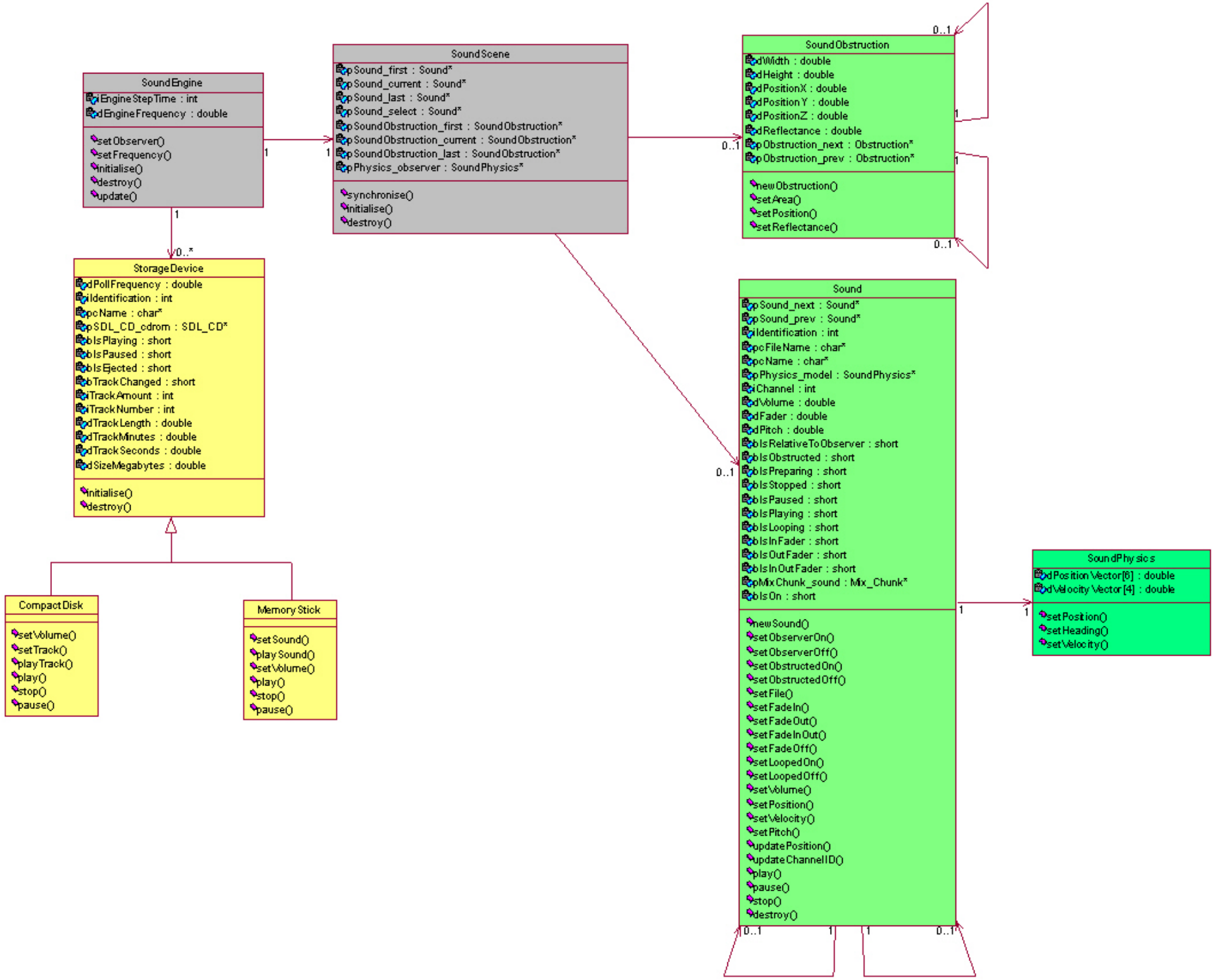

set local
set global
place barrier
set direct
set normal
set barrier
set reactive
set frequency
Actor
new barrier
new clock
nanoseconds
sleep clock time
get clock time
microseconds
milliseconds
seconds

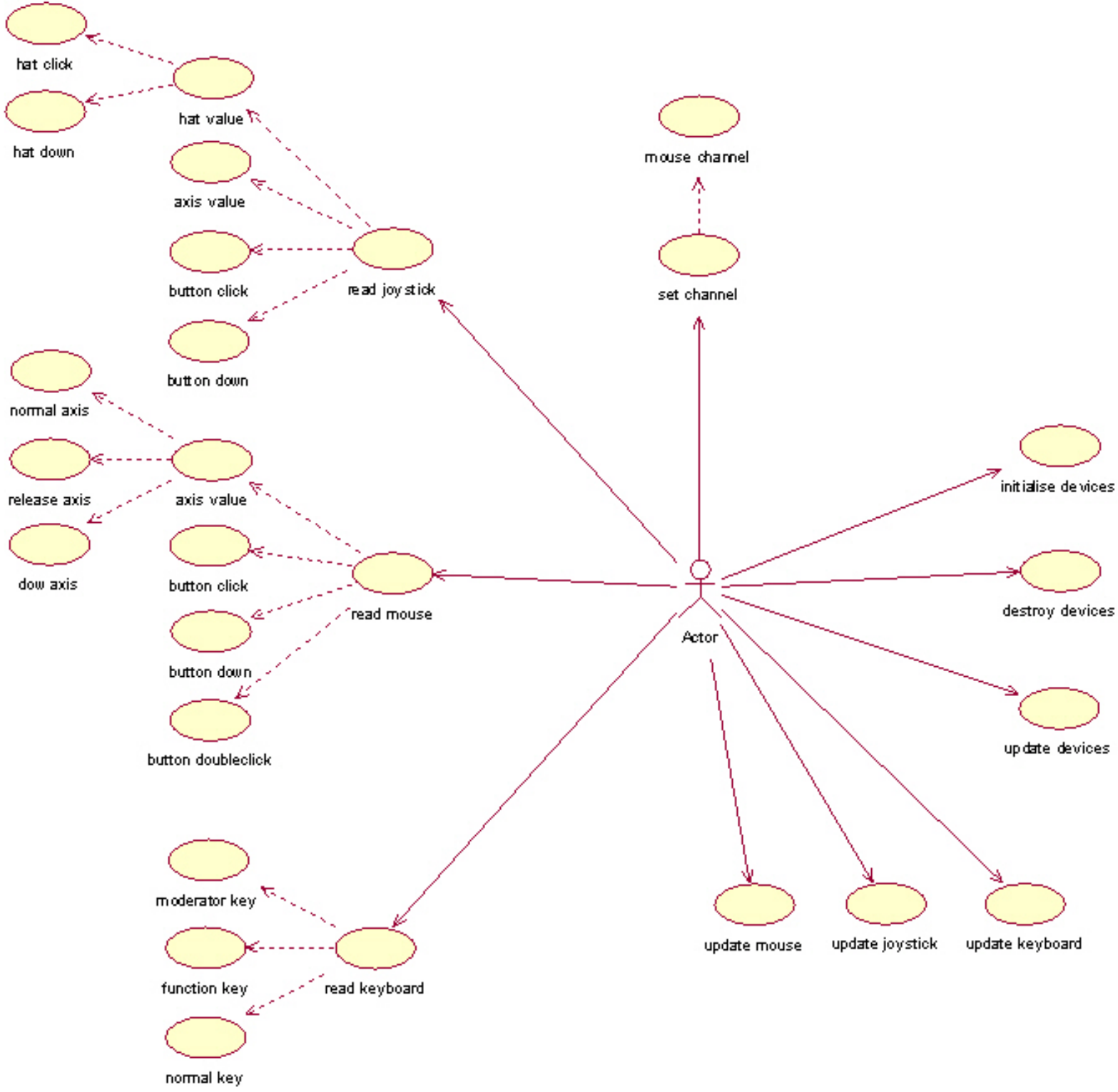

RealtimeBarrier
bIsLogged : short
bIsReactive : short
bIsNormal : short
bIsDirect : short
bIsGlobal : short
bWasTriggered : short
bTriggerReleased : short
iBarrierID : int
iMissedFrames : int
dCurrentTrigger : double
dCurrentInterrupts : double
dCurrentTime : double
dPrevTime : double
dStepTime : double
dMinLatency : double
dMaxLatency : double
dStartTime : double
dEndTime : double
dUnitsPerSecond : double
adPastTriggers : double [100]
placeGlobal()
placeLocal()
getTimeMicroseconds()
getTimeMilliseconds()
getTimeSeconds()
setTimeUnitAsMicroseconds()
setTimeUnitAsMilliseconds()
setTimeUnitAsSeconds()
setFrequency()
setLatencyPercentage()
setStartTime()
setEndTime()
getCurrentTime()
setDirect()
setNormal()
setReactive()
setGlobal()
setLocal()
reset()
place()
startPulseGenerator()
0..1
1
RealtimeClock
dClockStartTime : double
getMicroseconds()
getMilliseconds()
getSeconds()
reset()
sleepMicroseconds()
sleepMilliseconds()
sleepSeconds()
getStartTimeMicroseconds()
getStartTimeMilliseconds()
getStartTimeSeconds()
realtime clock contains POSIX compliable realtime functions but needs to be integrated with a realtime operating system in order to work properly.


new engine
set swap frequency
set engine frequency
new screen
set frequency
detach from entity
view cinematic
new camera
new fog
view locked
Actor
set camera
set screen
set fog
view inside
update engine
view outside
attach to entity
free viewing
set physics
set world
set entity
set light
new physics
new world
new entity
new light


VisualEngine
bIsInitialised : short
bIsParallel : short
bHasDynamicInterleave : short
bHasDynamicFrequency : short
dFrequency : double
dStepTime : double
dSwapBufferFrequency : double
dSwapBufferStepTime : double
dAverageStepTime : double
dAverageSwapbufferStepTime : double
dAverageEngineFrequency : double
dAverageSwapbufferFrequency : double
dFrequencyRatio : double
dCurrentTime : double
iCurrentFrame : long
iUpdatedStreams : short
initialise()
setEngineFrequency()
setSwapbufferFrequency()
setDynamicInterleaveOn()
setDynamicInterleaveOff()
setDynamicFrequencyOn()
setDynamicFrequencyOff()
printState()
update()
destroy()
Camera
bIsCinematic : short
bIsLocked : short
bIsInside : short
bIsOutside : short
bIsFree : short
bChangedView : short
pfCoordd_offset : pfCoordd
pfCoordd_camera : pfCoordd
pEntity_attachment : Entity*
iSourceNodeID : int
iSourceCameraID : int
pStream_cameraX : Stream*
pStream_cameraY : Stream*
pStream_cameraZ : Stream*
pStream_cameraH : Stream*
pStream_cameraP : Stream*
pStream_cameraR : Stream*
attachToEntity()
detachFromEntity()
setFreeViewing()
setCinematicViewing()
setOutsideViewing()
setInsideViewing()
setLockedViewing()
setRelativeHeading()
setRelativeDistance()
setPosition()
setHeading()
setSourceNode()
Channel
pfVec3_RelativePosition : pfVec3
pfVec3_RelativeHeading : pfVec3
pfVec3_RelativeObserverPosition : pfVec3
pfVec3_RelativeObserverHeading : pfVec3
pfVec3_StereoObserverLookat : pfVec3
pfVec3_StereoObserverPosition : pfVec3
pfVec3_StereoObserverHeading : pfVec3
pfPipeWindow_Window : pfPipeWindow*
pfChannel_ChannelDefault : pfChannel*
pfChannel_ChannelLeftEye : pfChannel*
pfChannel_ChannelRightEye : pfChannel*
pfPipe_Pipe : pfPipe*
dStereoAngle : double
pcWindowTitle : char*
iPositionX : int
iPositionY : int
iSizeX : int
iSizeY : int
dNearestPoint : double
dFarthestPoint : double
dHorizontalAngle : double
dVerticalAngle : double
bHasStereoBuffers : short
bIsFullscreen : short
bIsWindowed : short
bIsStereo : short
bIsOn : short
iSourceNodeID : int
pStream_bIsOn : Stream*
setRelativePosition()
setRelativeHeading()
setViewingAngle()
setViewingDistance()
setSize()
setPosition()
setTitle()
setFullscreen()
setWindowed()
setStereo()
setStereoDepth()
setStereoObserverLookat()
setStereoObserverPosition()
setStereoObserverHeading()
setSourceNode()
setRasterAngles()
setRasterPlace()
display()
Scene
iNumberOfLights : int
iNumberOfEntities : int
iNumberOfWorlds : int
iNumberOfFogs : int
pLight_first : Light*
pWorld_first : World*
pFog_first : Fog*
pEntity_first : Entity*
pLight_current : Light*
pWorld_current : World*
pFog_current : Fog*
pEntity_current : Entity*
pLight_last : Light*
pWorld_last : World*
pFog_last : Fog*
pEntity_last : Entity*
pfScene_scene : pfScene*
pfVec3_FogDensity : pfVec3
pfVec3_FogColor : pfVec3
pfVec3_CloudHeight : pfVec3
pfEarthSky_clear : pfEarthSky*
pfVolFog_clouds : pfVolFog*
pfFog_fog : pfFog*
bIsInitialised : short
bCloudsOn : short
bFogOn : short
pLight_active : Light*
pWorld_active : World*
pFog_active : Fog*
pEntity_active : Entity*
update()
checkDuplicates()
setFogOn()
setFogOff()
setCloudsOn()
setCloudsOff()
setGlobalFogColor()
setGlobalFogDensity()
lastLight()
lastEntity()
lastWorld()
lastFog()
SceneObject
World
pWorld_next : World*
pWorld_prev : World*
pCamera_camera : Camera*
iIdentification : int
pcName : char*
pPhysics_model : Physics*
pfSwitch_switch : pfSwitch*
pfSphere_bounds : pfSphere*
pfNode_world : pfNode*
pcFileName : char*
bIsOn : short
iSourceNodeID : int
pStream_bIsOn : short
newWorld()
lastWorld()
destroy()
setOn()
setOff()
setFile()
setViewingDistance()
setPagingDistance()
setParallelFile()
setPagingRatio()
Fog
pFog_next : Fog*
pFog_prev : Fog*
iIdentification : int
pfFog_fog : Fog*
pfVolFog_fog : pfVolFog*
pfVec3_FogDensity : pfVec3
pfVec3_FogColor : pfVec3
bFogOn : short
iSourceNodeID : int
pPhysics_model : Physics*
pEntity_entity : Entity*
pEntity_area : Entity*
pStream_bIsOn : Stream*
newFog()
lastFog()
destroy()
setOn()
setOff()
attachToEntity()
detachFromEntity()
setPosition()
setHeading()
setFogArea()
setFogColor()
setFogDensity()
setFogTexture()
getPhysics()
setSourceNode()
Entity
pEntity_next : Entity*
pEntity_prev : Entity*
iIdentification : int
pcName : char*
pPhysics_model : Physics*
pEntity_parent : Entity*
pCamera_attachment : Camera*
pcFileName : char*
pfSwitch_switch : pfSwitch*
pfSphere_bounds : pfSphere*
pfBox_bounds : pfBox*
pfNode_model : pfNode*
pfGeoSet_geometry : pfGeoSet*
pfGeode_primitive : pfGeode*
pfGeoState_attributes : pfGeoState*
pfMaterial : pfMaterial*
pfText_text : pfText*
pfFont_font : pfFont*
pfString_string : pfString*
iStringStateCounter : int
iNumberStateCounter : int
aaacStateStringName[MAX_ENTITY_STATES][MAX_ENTITY_STRINGSIZE] : char
aaacStateStringValue[MAX_ENTITY_STATES][MAX_ENTITY_STRINGSIZE] : char
aaacStateNumberName[MAX_ENTITY_STATES][MAX_ENTITY_STRINGSIZE] : char
aadStateNumberValue[MAX_ENTITY_STATES] : char
bIsSelected : short
bIsDragged : short
bIsOn : short
iSourceNodeID : int
pStream_bIsSelected : Stream*
pStream_bIsDragged : Stream*
pStream_bIsOn : Stream*
newEntity()
lastEntity()
destroy()
setOn()
setOff()
setFile()
setClonedEntity()
attachToEntity()
detachFromEntity()
setText()
setPlane()
setCube()
setPyramid()
setSphere()
setCone()
setLine()
setPosition()
setHeading()
setSize()
setTransparency()
setColor()
setShininess()
collisionWithEntity()
collisionWithWorld()
distanceFromEntity()
distanceFromWorld()
angleFromWorld()
getPhysics()
setStateString()
getStateString()
setStateNumber()
getStateNumber()
setSourceNode()
Light
pLight_next : Light*
pLight_prev : Light*
pcFileName : char*
pcName : char*
iIndentification : int
dAmbientRed : double
dAmbientGreen : double
dAmbientBlue : double
dDiffuseRed : double
dDiffuseGreen : double
dDiffuseBlue : double
dSpecularRed : double
dSpecularGreen : double
dSpecularBlue : double
dAmbient : double
dDiffuse : double
dSpecular : double
pfSwitch_switch : pfSwitch*
pPhysics_model : Physics*
pEntity_parent : Entity*
pfLightSource_light : pfLightSource*
pfTexture_texture : pfTexture*
pfFrustum_frustum : pfFrustum*
pfFog_fog : pfFog*
pfNode_model : pfNode*
bIsOn : short
iSourceNodeID : int
pStream_bIsOn : Stream*
newLight()
lastLight()
destroy()
setOn()
setOff()
attachToEntity()
detachFromEntity()
setToBeam()
setToPoint()
setDistance()
setAmbient()
setDiffuse()
setSpecular()
setAmbientColor()
setDiffuseColor()
setSpecularColor()
setBackgroundReflection()
setSurfaceReflection()
setShinyReflection()
setBackgroundColor()
setSurfaceColor()
setShinyColor()
setColor()
setPosition()
setHeading()
getPhysics()
PhysicsScene
pPhysicsScene_parent : PhysicsScene*
pPhysicsScene_next : PhysicsScene*
pPhysics_physics : Physics*
updateChildren()
updateScene()
clearScene()
setFrequency()
Physics
dFrequency : double
dDifferentialPosition\Vector[MAX_FORCES][6] : double
dFilteredPosition\Vector[6] : double
dPosition\Vector[6] : double
dSurface\Vector[6] : double
dFrictionvector[6] : double
dAbsoluteForce\Vector[MAX_FORCES][6] : double
dAbsoluteVelocity\Vector[MAX_FORCES][3] : double
dAbsoluteAcceleration\Vector[MAX_FORCES][3] : double
dAbsoluteAngularVelocity\Vector[MAX_FORCES][3] : double
dAbsoluteAngularAcceleration\Vector[MAX_FORCES][3] : double
dRelativeForce\Vector[MAX_FORCES][6] : double
dRelativeVelocity\Vector[MAX_FORCES][3] : double
dRelativeAcceleration\Vector[MAX_FORCES][3] : double
dRelativeAngularVelocity\Vector[MAX_FORCES][3] : double
dRelativeAngularAcceleration\Vector[MAX_FORCES][3] : double
dDifferentialForce\Vector[MAX_FORCES][6] : double
dDifferentialVelocity\Vector[MAX_FORCES][3] : double
dDifferentialAcceleration\Vector[MAX_FORCES][3] : double
dDifferentialAngularVelocity\Vector[MAX_FORCES][3] : double
dDifferentialAngularAcceleration\Vector[MAX_FORCES][3] : double
dInputParameters[7] : double
pPhysicsScene_child : PhysicsScene*
pPhysics_parent : Physics*
iComponentIndex : int
iDynamicsType : int
iHeadingThresholdType : int
iPositionThresholdType : int
dPitchSign : double
dBodyMass : double
dBodyFrictionConstant : double
dBodyFrictionUpperBound : double
dBodyAccelerationThreshold : double
dBodyVelocityThreshold : double
dMinYaw : double
dMaxYaw : double
dMinPitch : double
dMaxPitch : double
dMinRoll : double
dMaxRoll : double
dMinPositionX : double
dMaxPositionX : double
dMinPositionY : double
dMaxPositionY : double
dMinPositionZ : double
dMaxPositionZ : double
pfDoubleDCS_Position\Vector : pfDoubleDCS*
iSourceNodeID : int
pStream_Position\VectorX : Stream*
pStream_Position\VectorY : Stream*
pStream_Position\VectorZ : Stream*
pStream_Position\VectorH : Stream*
pStreamPosition\VectorP : Stream*
pStreamPosition\VectorR : Stream*
update()
setBaseFrequency()
newPhysics()
startPhysics()
destroy()
setDynamicsType()
setComponentIndex()
setPosition()
setHeading()
setPosition()
setHeading()
setAbsoluteVelocity()
setAbsoluteAcceleration()
setAbsoluteAngularVelocity()
setAbsoluteAngularAcceleration()
setAbsoluteForce()
setAbsoluteAngularForce()
setRelativeVelocity()
setRelativeAcceleration()
setRelativeAngularVelocity()
setRelativeAngularAcceleration()
setRelativeForce()
setRelativeAngularForce()
setDifferentialPosition()
setDifferentialVelocity()
setDifferentialAcceleration()
setDifferentialAngularVelocity()
setDifferentialAngularAcceleration()
setDifferentialForce()
setDifferentialAngularForce()
setMass()
setFriction()
setAccelerationThreshold()
setVelocityThreshold()
setInputParameters()
setSurfaceDistance()
setSurfaceAngle()
attachToPhysics()
detachFromPhysics()
setSourceNode()


observer on
observer off
position observer
obstruction
sound
set observer
place obstruction
new unit
play sound
pause sound
stop sound
Actor
set sound volume
loop off
set sound loop
loop on
set sound fader
fade off
fade in
fade out
fade in out
set sound position
set sound file
initialise engine
update engine
destroy engine
storage device
set track
set volume
play track
pause track
stop track


SoundEngine
iEngineStepTime : int
dEngineFrequency : double
setObserver()
setFrequency()
initialise()
destroy()
update()
SoundScene
pSound_first : Sound*
pSound_current : Sound*
pSound_last : Sound*
pSound_select : Sound*
pSoundObstruction_first : SoundObstruction*
pSoundObstruction_current : SoundObstruction*
pSoundObstruction_last : SoundObstruction*
pPhysics_observer : SoundPhysics*
synchronise()
initialise()
destroy()
SoundObstruction
dWidth : double
dHeight : double
dPositionX : double
dPositionY : double
dPositionZ : double
dReflectance : double
pObstruction_next : Obstruction*
pObstruction_prev : Obstruction*
newObstruction()
setArea()
setPosition()
setReflectance()
StorageDevice
dPollFrequency : double
iIdentification : int
pcName : char*
pSDL_CD_cdrom : SDL_CD*
bIsPlaying : short
bIsPaused : short
bIsEjected : short
bTrackChanged : short
iTrackAmount : int
iTrackNumber : int
dTrackLength : double
dTrackMinutes : double
dTrackSeconds : double
dSizeMegabytes : double
initialise()
destroy()
CompactDisk
setVolume()
setTrack()
playTrack()
play()
stop()
pause()
MemoryStick
setSound()
playSound()
setVolume()
play()
stop()
pause()
Sound
pSound_next : Sound*
pSound_prev : Sound*
iIdentification : int
pcFileName : char*
pcName : char*
pPhysics_model : SoundPhysics*
iChannel : int
dVolume : double
dFader : double
dPitch : double
bIsRelativeToObserver : short
bIsObstructed : short
bIsPreparing : short
bIsStopped : short
bIsPaused : short
bIsPlaying : short
bIsLooping : short
bIsInFader : short
bIsOutFader : short
bIsInOutFader : short
pMixChunk_sound : Mix_Chunk*
bIsOn : short
newSound()
setObserverOn()
setObserverOff()
setObstructedOn()
setObstructedOff()
setFile()
setFadeIn()
setFadeOut()
setFadeInOut()
setFadeOff()
setLoopedOn()
setLoopedOff()
setVolume()
setPosition()
setVelocity()
setPitch()
updatePosition()
updateChannelID()
play()
pause()
stop()
destroy()
SoundPhysics
dPositionVector[6] : double
dVelocityVector[4] : double
setPosition()
setHeading()
setVelocity()
1
0..*
0..1
1
0..1
1
0..1
1
0..1
1
1
0..1
1
1
0..1


hat click
hat value
hat down
axis value
button click
read joystick
button down
mouse channel
set channel
normal axis
release axis
axis value
dow axis
button click
read mouse
button down
button doubleclick
initialise devices
destroy devices
update devices
Actor
moderator key
function key
read keyboard
normal key
update mouse
update joystick
update keyboard


DeviceEngine
apInputDevice_device[10] : InputDevice*
initialise()
update()
destroy()
1
0..10
InputDevice
dPollFrequency : double
iIdentification : int
pcName : char*
pfuMouse_mouse : pfuMouse
pfuEventStream_stream
SDL_Event_event : SDL_Event
pSDL_Joystick_joy : SDL_Joystick*
dOrigin[6] : double
dOriginX : double
dOriginY : double
dOriginZ : double
dOriginP : double
dOriginQ : double
dOriginR : double
dPosition[6] : double
dPositionX : double
dPositionY : double
dPositionZ : double
dPositionP : double
dPositionQ : double
dPositionR : double
dVelocity[6] : double
dVelocityX : double
dVelocityY : double
dVelocityZ : double
dVelocityP : double
dVelocityQ : double
dVelocityR : double
dClickPositionX[10] : double
dClickPositionY[10] : double
dClickPositionZ[10] : double
dClickPositionP[10] : double
dClickPositionQ[10] : double
dClickPositionR[10] : double
dReleasePositionX[10] : double
dReleasePositionY[10] : double
dReleasePositionZ[10] : double
dReleasePositionP[10] : double
dReleasePositionQ[10] : double
dReleasePositionR[10] : double
bButtonDoubleClick[15] : short
bButtonClick[15] : short
bButtonDown[15] : short
bButtonDrag[15] : short
cKeyPressed : char
iKeyPressed : int
bHasState[10] : short
iNumberOfAxes : int
iNumberOfHats : int
iNumberOfButtons : int
iNumberOfBalls : int
iSourceNodeID : int
pStream_dOrigin[MAX_NODES][6] : Stream*
bIsOfType : short
DeviceMouse
initialise()
update()
destroy()
setLocalPointer()
setGlobalPointer()
setPointerOn()
setPointerOff()
getVelocityX()
getVelocityY()
getPositionX()
getPositionY()
getClickPositionX()
getClickPositionY()
getReleasePositionX()
getReleasePositionY()
getButtonDownLeft()
getButtonDownMiddle()
getButtonDownRight()
getButtonDragLeft()
getButtonDragMiddle()
getButtonDragRight()
getButtonClickLeft()
getButtonClickMiddle()
getButtonClickRight()
getButtonDoubleClickLeft()
getButtonDoubleClickMiddle()
getButtonDoubleClickRight()
setInput()
0..1
1
DeviceJoystick
initialise()
update()
destroy()
getPitch()
getYaw()
getRoll()
getPositionX()
getPositionY()
getPositionZ()
getThrottle()
getHatDownUp()
getHatDownDown()
getHatDownLeft()
getHatDownRight()
getFireDown()
getFireClick()
getHatClickUp()
getHatClickDown()
getHatClickLeft()
getHatClickRight()
DeviceKeyboard
initialise()
update()
destroy()
getRightArrowKey()
getLeftArrowKey()
getUpArrowKey()
getDownArrowKey()
getSpaceKey()
getNormalKey()
getFunctionKey()
getEscKey()
getTabKey()
getEnterKey()
getLeftControlKey()
getRightControlKey()
getLeftShiftKey()
getRightShiftKey()
getLeftShiftControlKey()
getRightShiftControlKey()
getLeftAltKey()
getRightAltKey()
0..1
DeviceChannel
pfPipeWindow_Window : pfPipeWindow*
pfPipe_Pipe : pfPipe*
pDisplay_screen : Display*
pWindow_root : Window
pcWindowTitle : char*
iPositionX : int
iPositionY : int
iSizeX : int
iSizeY : int
iScreenX : int
iScreenY : int
1

Kennis voor zaken

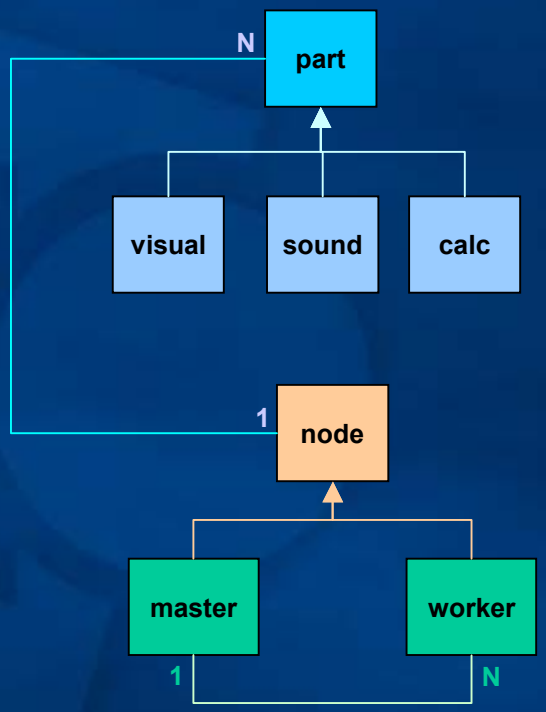
TNO (realtime) cluster

- Disposable cluster components
- Scrap cluster concept

- Cluster analysis tools
- Cluster install scripts
- Generic interfaces

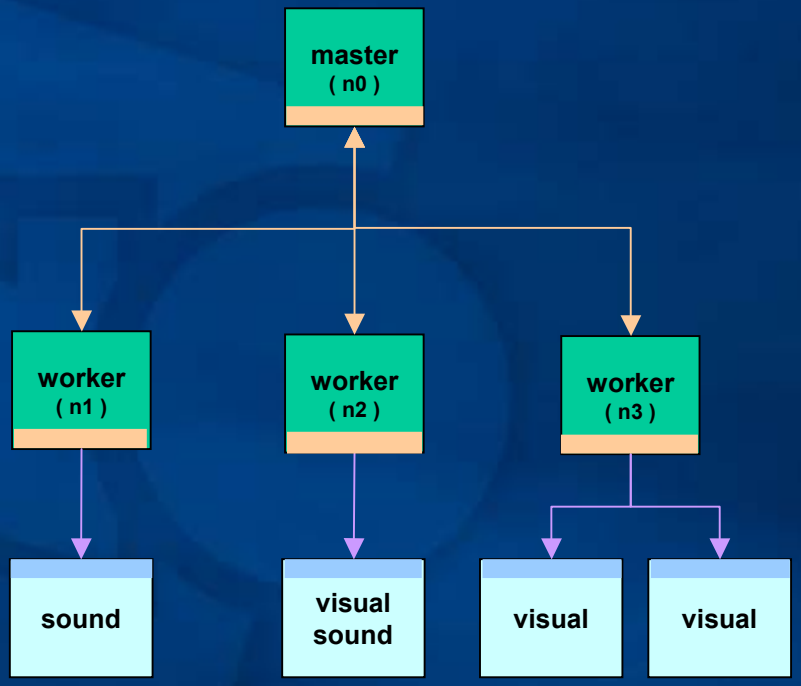
Node architecture

- Generic node model
- snd, calc, vis (parts)
- master, worker (nodes)
- scripted installation



Network architecture

- Single depth topology
- n0 (master)
- n1..nM (workers)
- scripted installation



Software environment

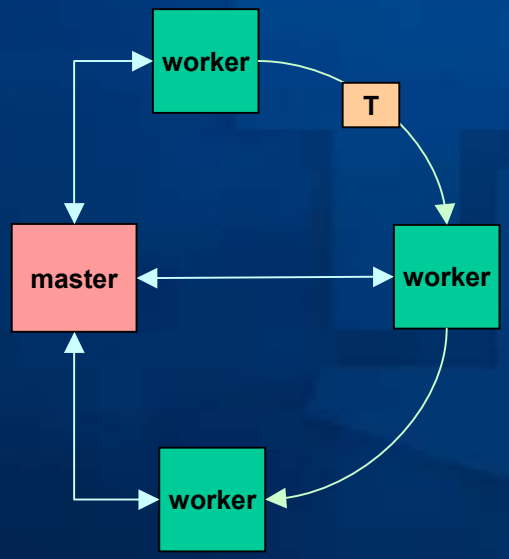
- Opengl Performer API
- Message Passing API
- Realtime Interface API

- Linux Source
- Windows Binary

- **Accepted international standards (POSIX/IEEE/ISO)**
- **Generic tools for installation, measurement, control.**

Global realtime networking

- Timeslot control
- Master controller
- global realtime clock



PRO's

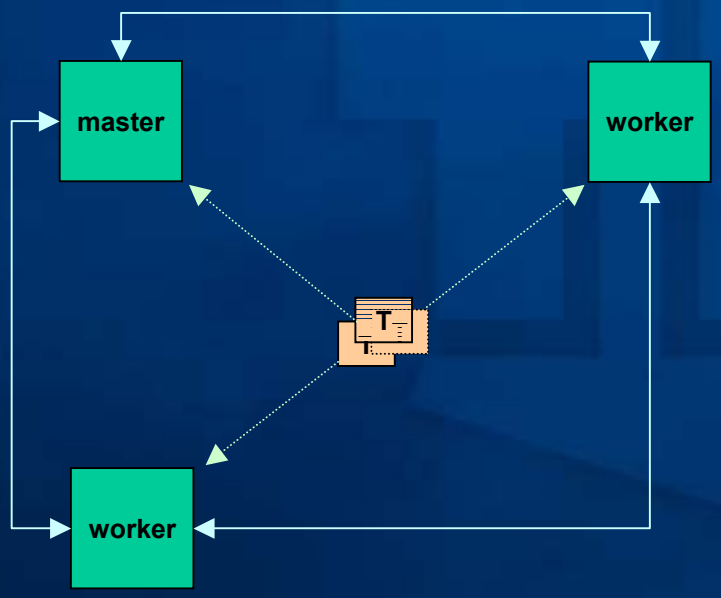
- instantaneous message transmissions
- realtime slots available for each node
- good for two to three unique nodes
- all system internals are open source

CON's

- network size ∝ (significant) realtime speed
- causes major jitter from corrections
- requires infinite master node uptime
- lacks widely scoped software genericity

Bounded realtime networking

- Global fuzzy realtime
- Reactive control
- Process synchronised



PRO's

- network size independent realtime speed
- transparent across network structure
- does not require infinite master uptime
- allows for maximum message throughput

CON's

- delayed processes delay realtime clock
- generates fuzzy global realtime clock
- dependent on message passing interface
- coarse grained realtime clock precision

Future (realtime) cluster targets

- Generalisation of cluster concept
- Reuse of installed cluster toolkits

- Development of barrier tools
- Development of messaging tools
- Refinement of cluster installers

Future (realtime) visual targets

- Modularisation of visual components
- Roundtrip engineering using rational

- Development of synchronisation tools
- Development of visual clustering tools
- Refinement of models using rational

Kennis voor zaken

Realtime Distributed Simulation

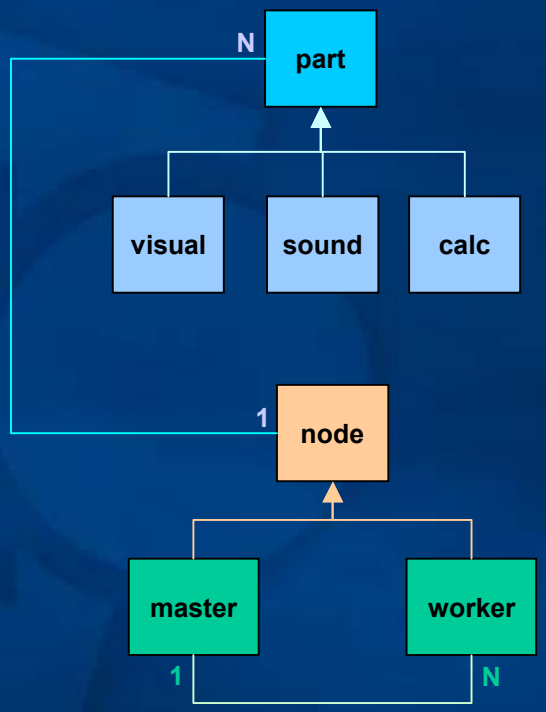
- RedHat cluster based simulation
- Shared parallel data management
- Parallel stream management
- Realtime distributed visualisation

- Out of the box installation

RedHat Cluster

- Flexible Node components
- Message Passing Interface
- Simple Direct Media API
- Performer OpenGL API

- Node configuration tool



Realtime Synchronisation

- **Data synchronisation (Datasynch)**

  Synchronisation of all datastreams such that all corresponding datastream input and output processes start simultaneously on each node.

- **Screen synchronisation (Swapbuffersynch)**

  Synchronisation of all swapbuffers such that all required and corresponding draw processes take place simultaneously on each node.

- **Pixel synchronisation (Genlocking)**

  Synchronisation of the draw processes of a group of pixels such that all corresponding pixel groups are drawn simultaneously on each node.

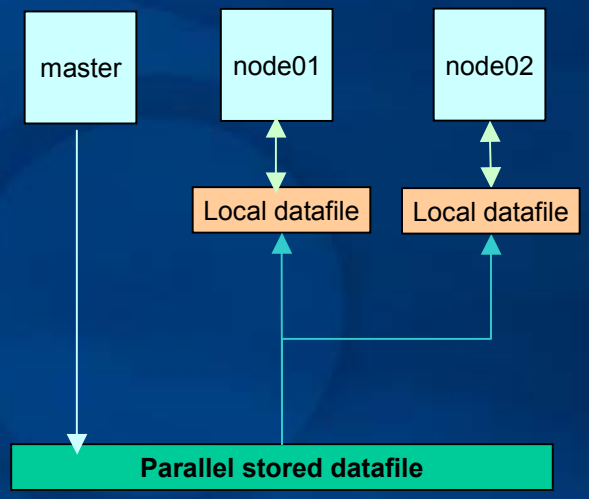
Synchronised Visual Prototype

Distributed OpenGL Visual

Parallel Data Management

- Shared parallel datastreams
- Parallel loaders and savers
- Message passing interface

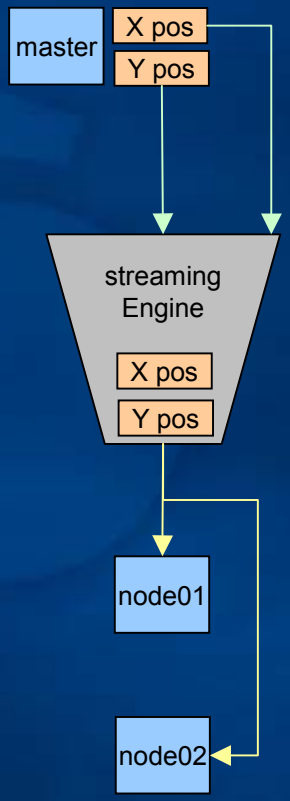
- Simple programming interface



Parallel Stream Management

- Parameter level streaming
- Dynamic streaming engine
- Message passing interface

- Simple programming interface



Realtime Distributed Visualisation

- Simple viewport interface
- Parallel programmable nodes
- Parallel input and output
- Parallel streaming engine
- Coarse grained parallelisation
- Realtime system performance

Synchronised Visual Prototype

Distributed Performer Visual

Prototype System Specification

- RedHawk Specification

Processor:	Dual Processors
Memory:	Redhawk Specific
Graphics:	TNT Graphics
Network:	Redhawk Specific
Inputs:	Universal Serial Bus

- TNode Specification

Processor:	Pentium 350 MHz
Memory:	128Mb DDR Ram
Graphics:	TNT Graphics
Network:	100Mbps Switch Ethernet
Inputs:	Universal Serial Bus

Commercial Applications

- Urban mission planning and intelligence
- Situation awareness and digital observation
- Business datavisualisation and datamining
- Military simulation and training prototypes
- Visualisation systems for very large datasets
- Instantaneously mobile military simulations

Afstudeerstage

- **Naam**

  Edgar Herbie Antonius van Tetering.

- **Geboren**

  4 Januari 1973 te Den Haag.

- **Afstudeerplek**

  TNO Fysisch Electrotechnisch Laboratorium te Den Haag.

- **Afstudeeronderwerp**

  Commercial Off The Shelf Realtime Distributed Visualisation.

Introductie

Tijdens de afstudeerstage is er geconcentreerd op het ontwikkelen van applicatie programmeertalen voor realtime gedistribueerde simulators op clusters van civiele machines.

- Waarom Civiel?

  Civiele hardware heeft in het verleden meerdere malen bewezen geschikt te zijn voor realtime gedistribueerde applicaties.

- Waarom Realtime?

  Realtime garandeert een prestatie van een gedistribueerde applicatie binnen een vastgestelde tijd en met een vastgestelde afwijking.

- Waarom Gedistribueerd?

  Gedistribueerde applicaties hebben een verbeterd prestatievermogen ten opzichte van vergelijkbare niet gedistribueerde applicaties.

Applicatie Programmeer Omgeving

Een groep applicatie programmeertalen voor realtime gedistribueerde simulatie applicaties op een cluster van civiele machines is ontwikkeld.

- Wat is een Applicatie Programmeertaal?

  Een gespecialiseerde programmeertaal die bedoeld is voor ontwikkeling van een bepaald type gespecialiseerde applicaties of applicatieonderdelen.

- Wat is een Realtime Gedistribueerde Applicatie?

  Een applicatie die verdeeld is over een aantal machines en waarvan de processen binnen een gegarandeerde tijd worden afgehandeld.

- Wat is een Cluster van Civiele Machines?

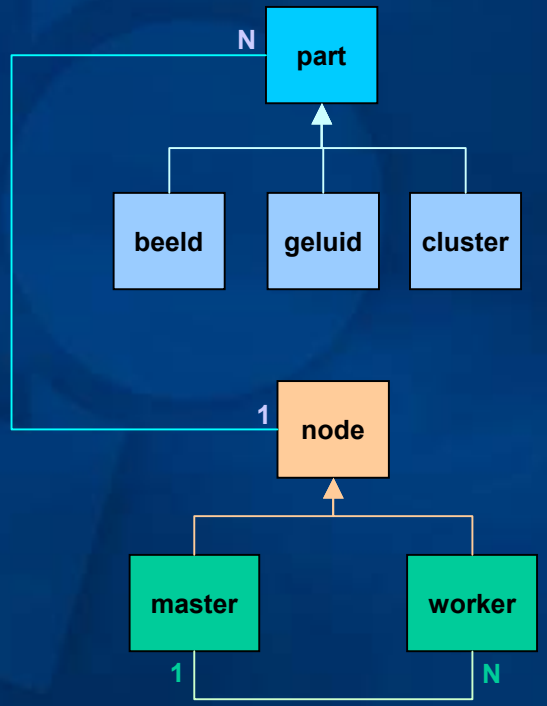
  Een groep samenwerkende machines uit een niet militaire of commercial off the shelf omgeving die tezamen een bepaald doel nastreven.

Cluster Configuratie Manager

Concentratie op ontwikkeling van een installatieomgeving voor het snel en gemakkelijk installeren en configureren van een cluster en de bijbehorende applicatie programmeertalen voor realtime gedistribueerde applicaties.

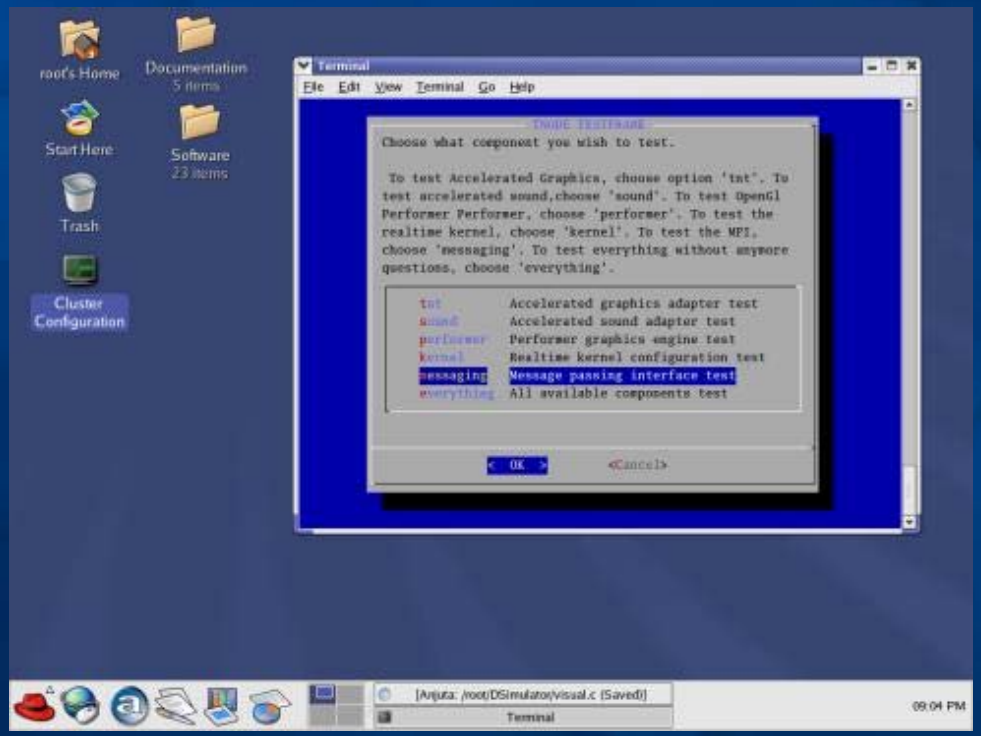
- RedHat Besturingssysteem
- Message Passing Interface
- Simple Direct Medialayer
- OpenGL Performer

- Directe Configuratie en Installatie



Cluster Configuratie Manager

- Schermvoorbeelden Machine Configuraties

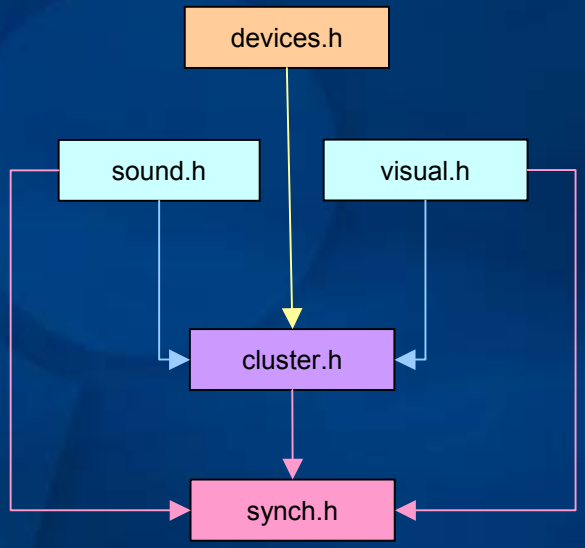


Applicatie Programmeertalen

Concentratie op ontwikkeling van een groep gemakkelijk herbruikbare applicatie programmeertalen voor realtime gedistribueerde simulatie applicaties met in het bijzonder een grafische applicatie programmeertaal.

- Beeld: visual.h
- Geluid: sound.h
- Randapparaten: devices.h
- Communicatie: cluster.h
- Realtime: synch.h

- Eenvoudige Programmeertalen



Applicatie Programmeertalen

- Schermvoorbeelden Applicatie Demonstraties

Probleemgebieden

- Gegevens synchronisatie (Datasynch)

  Synchronisatie van gegevensstromen zodat corresponderende gegevensstromen hetzelfde zijn op alle machines in een cluster.

- Beeld synchronisatie (Swapbuffersynch)

  Synchronisatie van beeld tekenprocessen in een cluster zodat alle bij elkaar horende beelden op hetzelfde moment worden getekend.

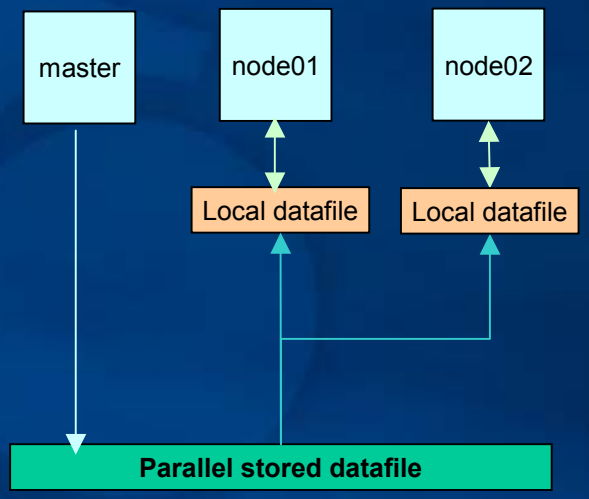
- Beeldpunt synchronisatie (Genlocking)

  Synchronisatie van beeldpunt tekenprocessen in een cluster zodat bij elkaar horende groepen beeldpunten op hetzelfde moment worden getekend.

Parallelle Gegevens (Datasynch)

- Gedeelde parallelle gegevens
- Dynamisch aanpasbare bestanden
- Message passing interface

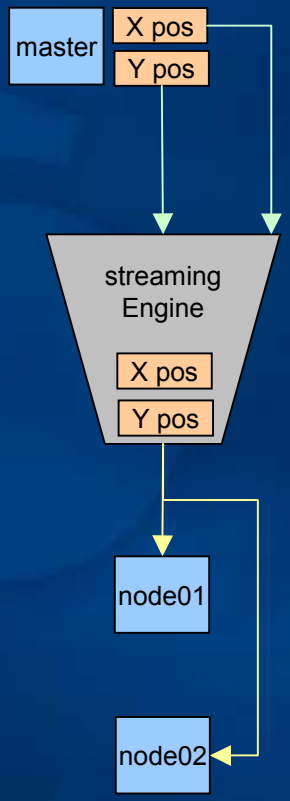
- Eenvoudige programmeertaal



Datastroom Machine (Datasynch)

- Parametrische gegevensstromen
- Scaleerbare datastroom machine
- Message passing interface

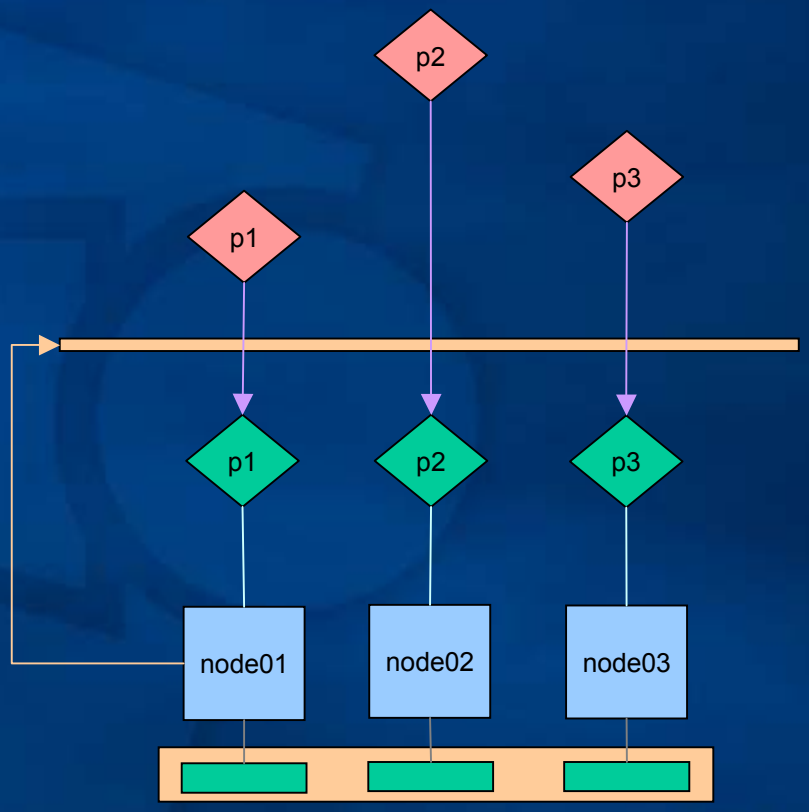
- Eenvoudige programmeertaal



Realtime Barrier (Swapbuffersynch)

- Parallelle realtime barrier
- Simple direct medialayer
- Message passing interface

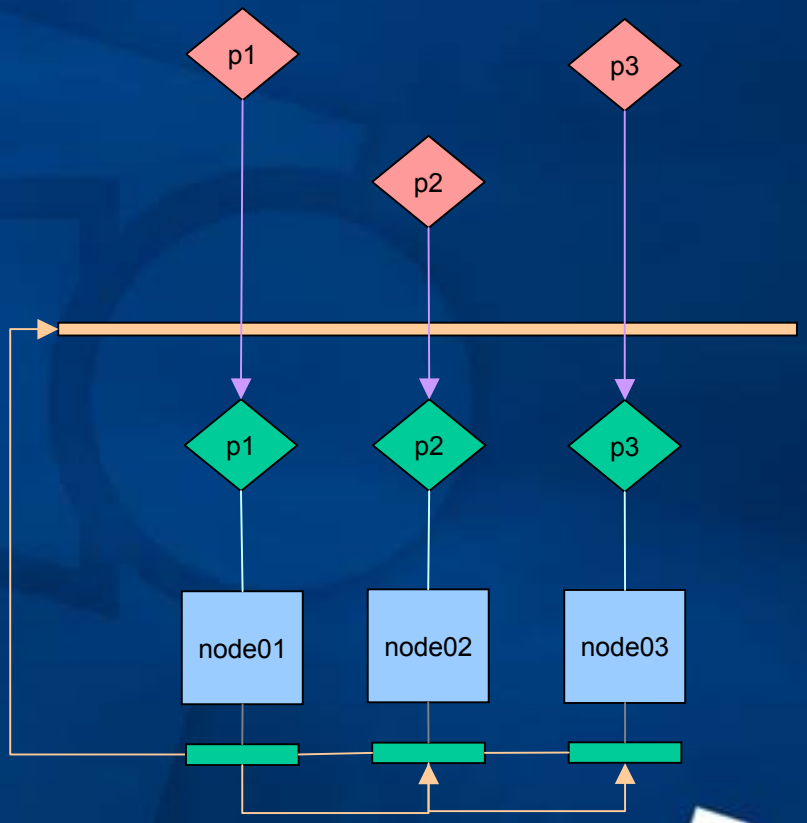
- Eenvoudige programmeertaal



Synchronisatie Barrier (Genlocking)

- Parallelle realtime barrier
- Gespecialiseerde hardware
- Directe genlock verbindingen

- Eenvoudige programmeertaal



Realtime Gedistribueerde Visualisatie

Ontwikkelde applicatie programmeertalen zijn gebruikt voor het programmeren van een gedistribueerde grafische applicatie programmeertaal.

- Gedistribueerde camera
- Gedistribueerde objecten
- Gesynchroniseerde beelden
- Scaleerbare applicatie
- Commercial off the shelf
- Direct installeerbaar

Prototype System Specification

RedHawk Specification

Processor:	Dual Processors
Memory:	Redhawk Specific
Graphics:	TNT Graphics
Network:	Redhawk Specific
Inputs:	Universal Serial Bus

TNode Specification

Processor:	Pentium 350 MHz
Memory:	128Mb DDR Ram
Graphics:	TNT Graphics
Network:	100Mbps Switch Ethernet
Inputs:	Universal Serial Bus

Commerciële applicaties

Gedistribueerde grafische applicaties zijn in een commercieel kader ruim toepasbaar voor een breed scala aan applicaties.

- Urban mission planning and intelligence
- Situation awareness and digital observation
- Business datavisualisation and datamining
- Military simulation and training prototypes
- Visualisation systems for very large datasets
- Instantaneously mobile military simulations