Bauhaus Universität Weimar
Fakultät Medien
Systeme der Virtuellen Realität

Projektdokumentation

4d-CAD

Kristin Bachmann
Benjamin Brombach
Erich Bruns
Franz Coriand
Alexander Kleppe
Kai Riege

4. Januar 2005

betreut durch
Professor Dr. Bernd Fröhlich
Dipl.-Inf. Hans-Friedrich Pabst

Inhaltsverzeichnis

1. Abstract (english)

Today complex architectural projects request qualified experts of all work areas to plan and realize such tasks. Therefore it is essential to create a field overlapping schedule such that work of all different experts is organized. Thereby the challenge consists of combining different workflows with different ways of visualization. In this case the problem consists in different experts using different applications to visualize and plan their work.

With 4d-CAD we provide a multi user visualization and planning tool, which enables architects to plan and visualize their building over time and space and give them the opportunity to share their material with other participants in real time.

Our project environment is divided in two main parts. The first part is an application to create and edit a Gantt chart. In this application it is possible to create tasks, change their properties and interact with the second part of the environment: the VR framework. The tasks of the planning process – shown as simple bars in the Gantt chart – are visualized in the VR framework in 3d mono or stereo as parts of the building. The advantage is that both parts are independent of their location. So, with the possibility of multiple users we provide, one could imagine a scenario two architects working at different locations on the same project by manipulating the Gantt chart, while a third person (e.g. the client) is watching the canvas to see the results as a virtual building.

To reach that state, we've done some basic implementations, including the development of a data model in the VR framework, representing the Gantt data structures (e.g. 'task', 'project'), and some methods to handle the various geometries of the building. We also implemented some communication modules, which took the task of transferring the data from the VR framework to the Gantt chart and vice versa.

To improve the VR visualization's as well as the Gantt chart's functionality we also implemented some techniques adding necessary features for such a distributed setting described above. So, for instance, each task now has three modes to visualize the status of its building process (non-visible, wireframe (under construction) and full visible) and there are controls in both the VR visualization and the Gantt chart to play the whole planning process as a virtual movie. Both features improve the planner's ability to navigate through the planning process and find the best way to build the building. Other improvements are the visualization of building resources (e.g. cranes and excavators) in both the VR framework and the Gantt chart and a cost calculation in the Gantt chart.

The whole 4d-CAD framework was developed to serve as a prototype and provides a basis for further research in distributed 4d-CAD systems.

2. Einleitung

Die zeitliche Planung für den Bau eines Gebäudes oder die Montage eines Autos erfordert von Planern, dass sie sich in Gedanken das Entstehen des fertigen Gebäudes oder Autos vorstellen. Im Allgemeinen arbeiten bei einem solchen Planungsprozess Experten aus unterschiedlichen Gebieten eng zusammen. Bisher gibt es aber wenige planerische Werkzeuge, die diesen Prozess geeignet unterstützen. Im Projekt 4D-CAD wurde eine interaktive und dynamische Produktplanungsumgebung entwickelt, die kontinuierlich den Zustand des 3D-Modells in den verschiedenen Bauphasen visualisiert. Diese 4D-Projektplanung erlaubt Architekten, Bauingenieuren und Planern gemeinsam am 4D-Modell über unterschiedliche Varianten des Bauablaufs zu diskutieren. Zusätzlich bietet unser 4D-CAD-System eine Unterstützung für virtuelle Teams, deren Experten sich an verschiedenen Orten aufhalten.

Die Aufgabe der Visualisierung der Baustelle bzw. des Produktionsstands haben wir hierbei Avango (siehe Anhang) übertragen. Als Planungstool für die Bauablaufplanung setzen wir eine freie Variante einer GanttChart-Software ein. Diese beiden kommunizieren über ein Kommunikationsmodul miteinander und bilden so unser verteiltes 4D-CAD System.

Projektleitung:	Professor Dr. Bernd Fröhlich Dipl.-Inf. Hans-Friedrich Pabst
Projekteilnehmer:	Kristin Bachmann Benjamin Brombach Erich Bruns Franz Coriand Alexander Kleppe Kai Riege

3. Avango

3.1. Architektur und Modelle

3.1.1. Objekt-Geometrien in Planungstools

Die Visualisierung des aktuellen Planungsstands eines Projekts erfordert eine etwas andere Herangehensweise beim Laden der Geometrien, als sie normalerweise im Avango-Framework angewendet wird.

Im Allgemeinen ist das Laden von Geometrien in Avango nicht schwer. Es genügt in der Regel, eine Instanz des Objekts als *fpLoadFile* zu erzeugen und ihm eine Geometrie zuzuweisen:

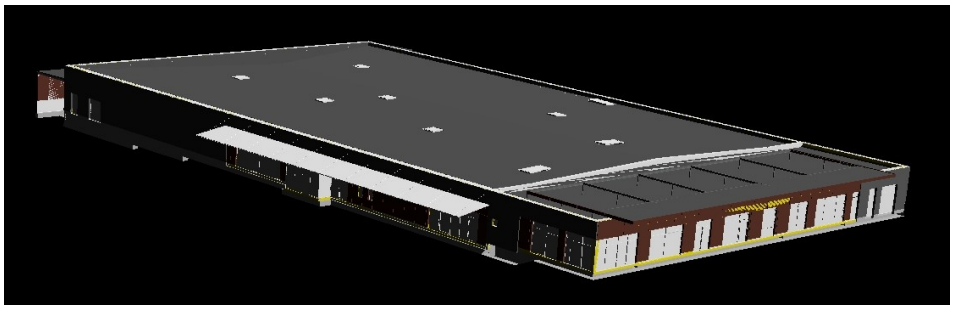

Abbildung 1: detaillreiches Modell eines Supermarkt-Komplexes

```
(define sphere
  (make-instance-by-name "fpLoadFile"))
  (fp-set-value sphere 'File "sphere.iv")
```

Zwar haben wir an dieser Methode, Objekte zu laden, nichts verändert, jedoch bringt es uns für unsere Zwecke offensichtlich nicht weiter, das Objekt (Gebäude, Fahrzeug, etc.) als Ganzes zu laden.

1. Es existieren n Geometrien, wobei jede einzelne den Planungsstand nach einem Arbeitsschritt visualisiert (siehe Abbildung 2). Dies geht jedoch auf Kosten des Speicherplatzes, da jede Geometrie ein Teil der neuen Geometrie ist und so mehrfach geladen wird.

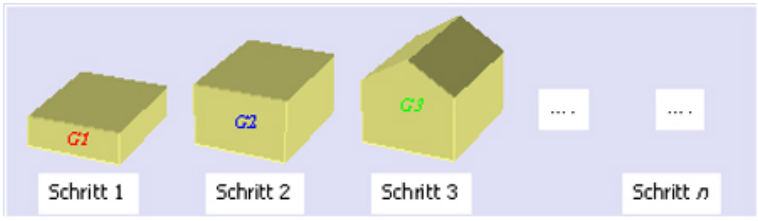

Abbildung 2: Objektaufbau Variante 1 – neue Geometrie entspricht neuer Zwischenansicht

2. Es existieren n Geometrien, wobei jede einzelne einen Arbeitsschritt visualisiert (siehe Abbildung 3 auf der nächsten Seite). Der neue Planungstand ergibt sich also aus dem aktuellen Stand plus der neuen Geometrie.

Ein Projektplanungstool erfordert es, das entsprechende Gebäude, Fahrzeug o. ä. nach und nach aufzubauen,was für eine Zahl n an Arbeitsschritten auf 2 mögliche Weisen realisiert werden könnte:

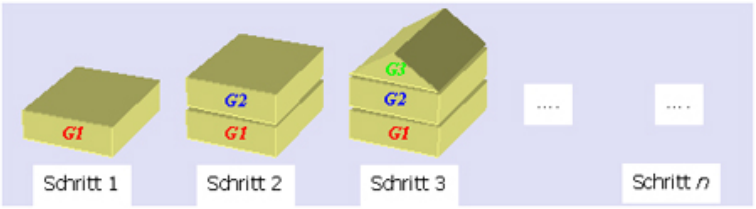

Abbildung 3: Objektaufbau Variante 2 – neue Geometrie entspricht aktuellem Arbeitsschritt

Zusätzlich zum Aspekt des Speicherbedarfs sprach auch der Umsortierungsaspekt für die zweite Variante, der mit der ersten Variante nicht zu realisieren wäre. Ein Planungstool sollte es nämlich zweifelsfrei auch ermöglichen, Abschnitte (und damit auch die Geometrien) in eine andere zeitliche Reihenfolge zu bringen, um sich anhand der Visualisierung für den besten Weg zum Ziel des fertigen Hauses oder Fahrzeugs zu entscheiden. Hierbei spielen auch Aspekte des Ressourcen-Managements eine Rolle, da es manchmal nicht möglich ist, eine Ressource heranzuschaffen und so zunächst ein anderer Teil gebaut werden muss. Aspekte des Ressourcen-Managements werden allerdings später noch näher betrachtet (siehe 3.4 auf Seite 15).

Um viele Geometrien während des Projektablaufs zu einem Ganzen zusammzusetzten, war zunächst der rückwärtige Schritt vonnöten, nämlich mangels vorhandener Gebäude- und Fahrzeugteile das Auseinandernehmen bereits vorhandener Objekte in mehrere Teilgeometrien, welche dann letztendlich in Avango hineingeladen werden konnten.

3.1.2. Laden von Projekten

Um die Anwendung für spätere Erweiterungen möglichst flexibel zu halten, entschlossen wir uns, das Laden einzelner Projekte nicht im normalen Quellcode zu integrieren. Vielmehr sollte es im Sinne der Transparenz in der fertigen Anwendung Möglichkeiten geben, Projekte zu laden bzw. möglichst einfach auszutauschen – idealerweise durch Auswählen in einem Dialog oder mittels Auswahl eines kartenähnlichen Icons – bei uns jedoch durch entsprechende Einzeiler im Code oder auf der Scripting-Konsole, welche

1. den zuvor als Binärfile abgespeicherten Szenengraphen inklusive der Eigenschaften sämtlicher Knoten des Graphen laden:
   - Speichern des Szenengraphen: `(writefile node project1)`
   - Laden des Szenengraphen: `(readfile project1)`

- Funktions-Definitionen:

```
(define (writefile node filename)
  (let ((temp (av-write-graph node filename)))
    (if temp (display "Output file written ...\n"))))
```

Der Parameter node gibt dabei an, ab welcher Position abwärts der Szenengraph abgespeichert werden soll.

```
(define (readfile filename)
(let ((temp (av-read-graph filename)))
  (if temp (begin
    (fp-remove-1value main_dcs 'Children gantt-node)
    (av-shared-define 'gantt-node (av-read-graph filename))
    (fp-add-1value main_dcs 'Children gantt-node)
    (display "Input file read ...\n")
    (make-all-triggers))))
```

oder

2. abgespeicherte scm-Dateien mittels *av-provide* anbieten und mit *av-require* abrufen. Diese Dateien beinhalten sämtliche Eigenschaften aller Tasks (Arbeitsschritte), u. a. natürlich auch deren Geometrien

   - Anbieten des Projekts in einer separaten .scm-Datei:

```
(av-provide 'project_disney)

;; Definition der Tasks in Project 'project_disney.scm'

(define task8 (make-instance-by-name "fpGanttTask"))
(fp-set-value task8 'TaskName "Task #8")
(make-task task8 gantt-node "040 SOG_nw.3ds"
  transform_mat  185 1 40 100 2200)
```

   - Abrufen des Projekts im Hauptquelltext:

```
(av-require 'project_disney)
```

In beiden Fällen wird also eine zuvor definierte Szenerie geladen, wobei der Vorteil von Variante 2 in der Lesbarkeit des scm-Files und damit möglichen Modifikationen der Szenen-Files liegt (im Gegensatz zum Binärfile in Variante 1).

3.2. Avango Datenmodell

Um verteiltes Arbeiten an einem Projekt zu ermöglichen, ist es notwendig, dass Avango alle relevanten Daten aus dem Planungstool (Gantt) für die anderen Teilnehmer bereithält. Aus diesem Grund haben wir drei Avango-Nodes implementiert, die diese Aufgabe übernehmen sollen. Diese Nodes halten alle projektbezogenen Daten aus Gantt und verteilen sie über den Verteilungsmechanismus von

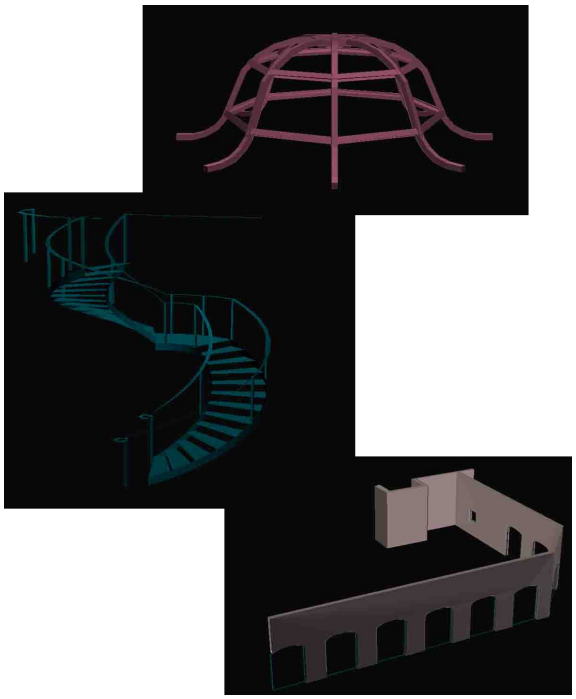

Abbildung 4: verschiedene Arbeitsschritte

Avango. Andere Teilnehmer melden sich mit ihrem Gantt an einem solchen verteilten Avango-Projekt an, um am 4D-Planungsprozess teilzunehmen. Die zweite Aufgabe dieses Datenmodells besteht darin, anhand dieser Daten eine Visualisierung der Baustelle zu ermöglichen.

Die Hierarchie im Szenengraphen ist auf Abbildung 6 auf Seite 11 zusehen.

3.2.1. fpGanttProject

In diesem Knoten werden alle projektweiten Parameter gespeichert. Hierzu sind folgende Felder implementiert:

- *fpSFString ProjectName;*
- *fpSFInt ID;*
- *fpSFInt StartProject;*
- *fpSFInt ProjectDuration;*

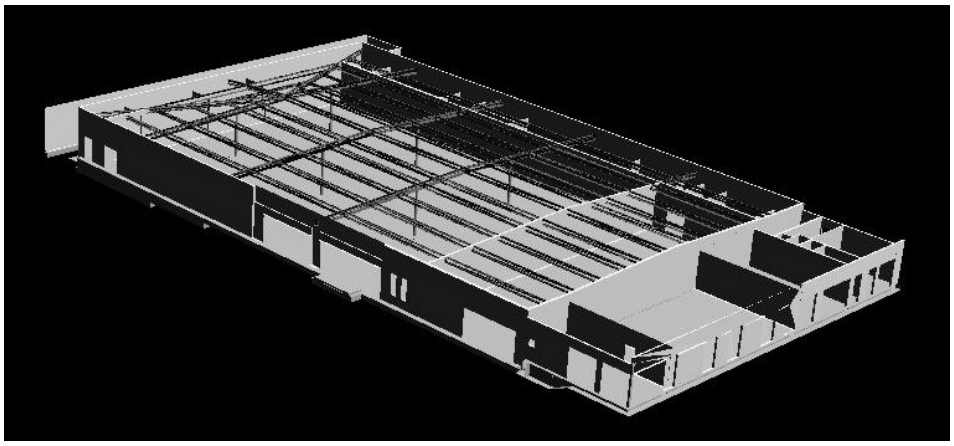

Abbildung 5: Zwischenansicht nach einigen Arbeitsschritten

- *fpSFInt ActualTime;*
- *fpSFInt TimeChanged;*
- *fpSFInt MaskTask;*
- *fpSFString Play;*

Jedes Projekt wird durch einen Projektnamen repräsentiert. Der Projektknoten wird mit der ID = 0 belegt und dadurch identifizierbar. Startdatum und Dauer des Projektes sind hier hinterlegt. In das Feld 'TimeChanged' wird von der Gantt-Seite aus ein Flag gesetzt, wenn dort die aktuelle Zeit geändert wird. In 'MaskTask' wird die ID desjenigen Tasks aus Gantt vermerkt, der dort gerade markiert ist. Das 'Play'-Feld dient der Steuerung der Play-Funktionalität der Zeitleiste.

3.2.2. fpGanttTask

In diesem Knoten werden alle taskbezogenen Parameter gespeichert. Hierzu sind folgende Felder implementiert:

- *fpSFString TaskName;*
- *fpSFInt ID;*
- *fpSFInt TaskStart;*
- *fpSFInt TaskDuration;*

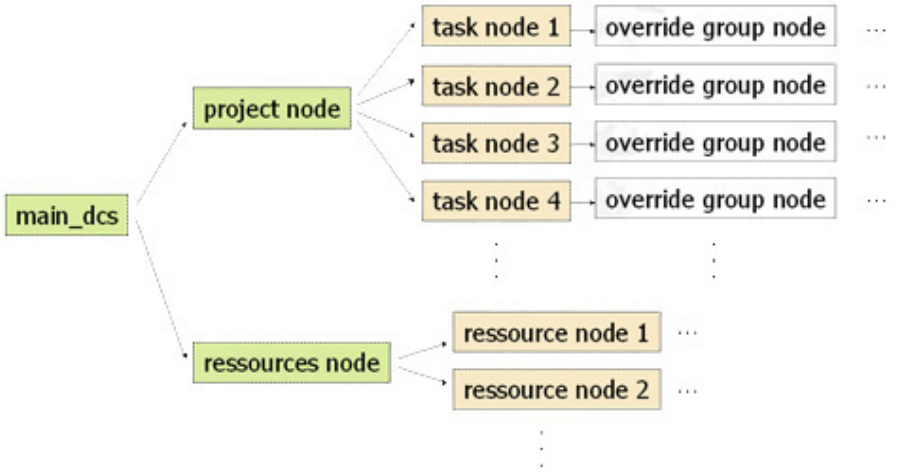

Abbildung 6: Hierarchie im Szenengraphen

- *fpSFInt ParentTaskID;*
- *fpMFInt PredecessorsIDs;*
- *fpMFFloat Ressources;*
- *fpSFInt Costs;*
- *fpSFInt Mark;*

Jeder Task trägt einen Namen, der in der Gantt-Komponente angezeigt wird. Das Feld '*ID*' dient auch hier zum Identifizieren. Beginn, Dauer und Kosten des Tasks sind hier gespeichert. Der Beginn eines Tasks ist relativ zum Beginn des Projekts, während die Startzeit einen absoluten Wert erhält. Aus Gründen der Variabilität wird hier nicht, wie in der Ganttkomponente, mit dem Datumsformat gearbeitet, sondern mit einfachen Integern. Das Feld '*Ressources*' enthält eine Liste, in der u.a. Beginn und Dauer der Benutzung einer Ressource stehen, diese beziehen sich relativ auf den Beginn des benutzenden Tasks (siehe Ressourcenmanagement). Im Feld '*ParentTaskID*' steht die ID des Vorgängertasks vermerkt. Das Feld '*PredecessorsIDs*' zeigt die Tasks an, von denen dieser Task abhängig ist. In '*Ressources*' sind die für diesen Abschnitt notwendigen Ressourcen und deren Auslastung vermerkt. '*Mark*' zeigt an, ob dieser Task in Gantt markiert ist oder nicht. Die Tasks sind im Szenengraphen unter den Projektknoten gruppiert.

3.2.3. fpGanttOpRessource

Dieser Knoten steht für eine zur Verfügung stehende Ressource. 'OpRessource' steht für Operational Ressource und meint wiederverwendbare Ressourcen wie beispielsweise einen Kran, Bagger oder Bau-Gerüste. Hierzu sind folgende Felder implementiert:

- *fpSFString Name;*
- *fpSFInt ID;*
- *fpSFInt Start;*
- *fpSFInt Duration;*
- *fpSFInt UtilizationRatio;*
- *fpMFInt CurrentUserIDs;*
- *fpMFFloat UtilizationRatioList;*

Auch die Ressourcen tragen Namen und werden durch ihre IDs identifiziert. Der Beginn ihrer Benutzung und die Dauer sind gespeichert. Dieser Zeitraum bezieht sich allerdings nur auf die allgemeine Verfügbarkeit der Ressource (z. B. von der ersten Verwendung bis zur letzten). Die Informationen über die spezifische Verwendung der operativen Ressource durch einen oder mehrere Tasks wird im Task-Knoten selbst gespeichert. In '*CurrentUserIDs*' sind die IDs der Tasks vermerkt, die diese Ressource gerade verwenden. '*UtilizationRatio*' zeigt die Auslastung dieser Ressource über die Zeit an. Zur Gruppierung mehrerer Ressourcen haben wir einen DCS-Knoten verwendet. Dieser steht in der Szenengraphen-Hierarchie mit dem Projektknoten auf gleicher Ebene.

3.3. Interaktion in Avango

3.3.1. Input

Ein gravierender Nachteil bei fertigen CAD-Zeichnungen ist u. a. die fehlende Interaktion. So hat man zum Beispiel nur eine feste und nicht unbedingt anschauliche Perspektive auf das Modell. Des Weiteren kann man nur das fertige Produkt begutachten, einen zeitlichen Ablauf hat man in üblichen CAD-Zeichnungen nicht. Ebenfalls ist es nicht möglich, den Fokus auf bestimmte Dinge zu legen (zum Beispiel das alleinige Anzeigen der Rohre in einem Haus). Man sieht immer alles oder wählt aus einer ganzen Sammlung an Zeichnungen. Bei komplexen Bauvorhaben kann so schnell mal der Überblick verloren gehen. Neben den Möglichkeiten in der Gantt-Applikation, den Bauprozess zu begutachten und zu verändern (siehe Gantt-Teil), ist es auch in Avango möglich, unterschiedliche Interaktionen auszuüben. Da

es sich um eine virtuelle dreidimensionale Umgebung handelt, erfordert dies spezielle Eingabegeräte, um intuitiv interagieren zu können. Wir verwenden dazu die sogenannte „Spacemouse“ (SM). Sie hat sechs Freiheitsgrade (DOF = Degrees Of Freedom) und wird bereits im CAD-Bereich verwendet. Die SM ermöglicht die Translation entlang dreier Achsen und die Rotation um die drei Achsen. Mit der SM ist es möglich, sich sowohl im Raum als auch in der Zeit zu bewegen (natürlich nur virtuell).

Abbildung 7: Spacemouse (6 DOF)

Die Bewegung im Raum ermöglicht es, das Modell während der Laufzeit aus allen Perspektiven zu begutachten. Man kann sich aus der Vogelperspektive einen Überblick über das Modell verschaffen oder sich direkt durch das Modell bewegen und dadurch einen viel besseren Eindruck erlangen, als es mit einer CAD-Zeichnung möglich wäre. Man bewegt sich in einer Art Flugmodus und kann sich dabei frei bewegen. Entsprechende Ausgabegeräte verstärken den Eindruck noch, dazu aber später. Es gibt keine Kollisionserkennung. Dies ermöglicht zum Beispiel die Etage eines Hauses zu wechseln, ohne das Treppenhaus nutzen zu müssen. (Eine Kollisionserkennung wäre aber grundsätzlich möglich.) Auch die Steuerung des Timesliders und somit die Bewegung durch die Zeit ist über die Spacemouse möglich, da die von uns verwendete SM acht diskrete Tasten anbietet. Jeder Taste wurde eine typische Player-Funktionalität zugeordnet, wie „Play“, „Stop“, „Pause“, „FastForward“ etc., um einen schnellen Einsteig ohne lange Einarbeitungszeit zu ermöglichen. Man kann sich so den kompletten Bauprozess vom Anfang bis zum Ende, wie eine Art Film, anschauen.

Neben der Steuerung der Zeit durch die SM ist es außerdem möglich, den Timeslider mittels einer Maus per Drag’n’Drop zu ziehen, was mitunter eine schnellere Interaktion erlaubt, als mit der Player-Funktionalität. Der Vorteil der Zeit als vierte Dimension liegt auf der Hand. Man kann sich die einzelnen Stadien des Bauprozesses anschauen und im Zusammenspiel mit der dreidimensionalen Visualisierung räumliche und zeitliche Konflikte auflösen, die in abstrakten und mitunter unübersichtlichen Plänen übersehen werden könnten. So kann der Auftraggeber

sich schon vorher einen Eindruck verschaffen und eventuelle Änderungswünsche vor der Fertigstellung an die Baufirma weiterleiten.

Im Zusammenhang mit operativen Ressourcen (siehe Ressourcenmanagement) können zeitliche und räumliche Konflikte schon erkannt werden, bevor sie passieren. So könnte zum Beispiel erkannt werden, dass zwei Prozesse eine Ressource zur gleichen Zeit an unterschiedlichen Orten benötigen oder sie zu über 100% auslasten. Auf visuellem Weg ist das besser zu erkennen als mit langen Listen. Die Interaktion ist natürlich auch über die Gantt-Komponente aus möglich. So kann in dieser ebenfalls der Timeslider gezogen und der Player bedient werden. Darüber hinaus ist es möglich, Tasks zu markieren. Diese werden dann in der Avango-Komponente mit einer Farbe maskiert (momentan rosa, grundsätzlich wären alle Farben möglich) und zusätzlich werden Informationen über diesen Task (Name etc.) in der Bildschirmebene ausgeben. So kann ein Teilnehmer einen Task markieren und die Gruppe weiß, welcher Task gerade Aufmerksamkeit verlangt.

3.3.2. Output

Die Applikation sollte auf unterschiedlichen Ausgabegeräten laufen können. So ist es möglich, sich das Modell swohl auf einem normalen Monitor, als auch von Stereoprojektionen visualisieren zu lassen. So kann man sich das Bauprojekt entweder auf dem Büro-PC bzw. bequem von zu Hause aus anschauen oder mit Hilfe einer Polarisationsbrille und zweier Projektoren das Baumodell in 3D. Letzteres ermöglicht einen sehr guten Eindruck über das fertige Produkt, da z. B. der Bauherr sein Haus virtuell betreten kann und so einen ersten Überblick bekommt.

Um das jeweilige Ausgabegerät nutzen zu können, gibt es unterschiedliche Startskripte, die spezielle Informationen über Gerät und Konfiguration enthalten. So gibt es momentan Startskripte für einen normalen Monitor und für Stereoprojektionen auf den Projektoren JVC SX21 und F1 von ProjectionDesign. Jeder weitere Projektortyp wäre möglich und darüber hinaus noch andere 3D-Visualisierungsarten. (Das VR-Framework Avango bietet eine Vielzahl solcher an, z. B. CaveTM) In den Konfigurationsdateien liegen auch die Informationen über das User-Interface (UI). Außer der grafischen Repräsentation des Modells, sieht der Benutzer zusätzliche Informationen:

- Timeslider (Anfangstag, aktueller Tag, Ende des Projektes)
- Teilnehmeranzahl im verteilten Modus
- Kostenangaben (Kosten des aktuellen Prozesses, Gesamt bis dato, Kosten insgesamt)
- Informationen über einen eventuell markierten Task

Beliebig viele weitere Informationen wären möglich und könnten in der Bildschirmebene angezeigt werden. Wie die Informationen allerdings angeordnet werden, ist variabel und wird in der Konfigurationsdatei gespeichert. Es ist also möglich ein individualisiertes UI zu „bauen“, um so den persönlichen Ansprüchen des Anwenders gerecht zu werden. Ein Laie kann auf bestimmte Informationen verzichten und lieber einen ungetrübten Blick auf das Modell haben wollen, während der Bauleiter eine Vielzahl an Informationen wünscht. Der Timeslider könnte horizontal unten oder oben bzw. vertikal rechts oder links angeordnet werden.

Abbildung 8: Timeslider in Avango

3.4. Ressourcenmanagement

Zu einem umfassenden Bauplanungsprozess gehört auch das komplexe Ressourcenmanagement. In dem Projekt sollten Ressourcen nicht nur als Teil eines Bauprozesses gespeichert werden, sondern auch visualisiert, d. h. in die Szene integriert werden. Eine vorherige Recherche in der entsprechenden Literatur zeigte den Umfang des für uns neuen Themenkomplexes auf. Es gibt eine Vielzahl an Ressourcen mit unterschiedlichsten Eigenschaften und dazugehörigen Theorien und Modellen. Neben den klassischen Ressourcen, die „verbraucht“ werden (z. B. Steine, Holz, etc.), werden auch Arbeitskräfte (Arbeitszeit, Pausen, Einsatzgebiet, etc.) und Arbeitsmittel (Auslastung, Abnutzung, etc.) als Ressourcen gehandhabt. Eine Ressource kann zum Beispiel Teil eines Prozesses sein oder ein Prozess kann auf eine Ressource zugreifen.

Wir haben uns auf so genannte operative Ressourcen beschränkt. Operative Ressourcen sind wiederverwendbare Arbeitsmittel, die eine bestimmte Arbeit verrichten. Eigenschaften wie Abnutzung und Regenerationsphasen haben wir zunächst außer Acht gelassen und uns auf die Auslastung und Positionierung durch Tasks beschränkt. Es gibt nun einen „Pool“ von operativen Ressourcen, auf den die Tasks zugreifen können (eine operative Ressource ist also nicht Teil eines Tasks, sondern es besteht eine Beziehung zu einem Task). Der Task sagt (die Informationen werden also beim Task gespeichert), wann er welchen Task wo braucht und wie groß die Auslastung ist.

Es können mehrere Tasks auf eine operative Ressource zugreifen (die entsprechenden Auslastungen werden dann addiert). Der Vorteil dabei ist, dass bei einer

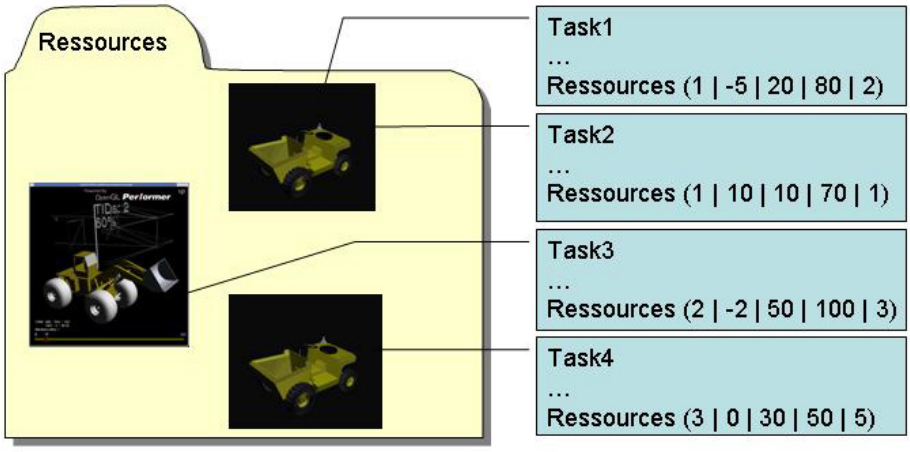

Abbildung 9: Ressourcen-Pool

Veränderung des Startdatums eines Tasks die von ihm verwendete Ressource davon nicht in Kenntnis gesetzt werden muss, da die Informationen ja nur beim Task gehalten werden (siehe genauere Beschreibung). Das Besondere ist nun, dass eine operative Ressource ebenfalls visualisiert wird, wenn sie an dem aktuellen Tag in Verwendung ist (sprich eine Auslastung größer als 0% hat).

Neben der grafischen Repräsentation eines Tasks (z. B. eine Wand) würde dann die verwendete operative Ressource (z. B. ein Kran) erscheinen. Über der Ressource befindet sich ein sich drehendes „Informationsfähnchen“ (damit man die Informationen aus allen Perspektive sehen kann), welches die IDs der aktuellen Tasks anzeigt, die auf sie zugreifen sowie die daraus resultierende Auslastung. Einer Ressource kann durch einen Task eine spezifische Position zugewiesen werden. Beispielsweise kann ein Kran an eine bestimmte Stelle positioniert werden und dann geschaut werden, ob räumliche Konflikte mit anderen Kränen oder Gebäuden auftreten würden. In Anlehnung an die Sichtbarkeitsstufen der Task gibt es also folgende Möglichkeiten:

- Unsichtbar — Die Ressource ist nicht in Verwendung.
- Sichtbar — Die Ressource ist in Verwendung.
- Rot maskiert — Die Ressource ist in Verwendung:

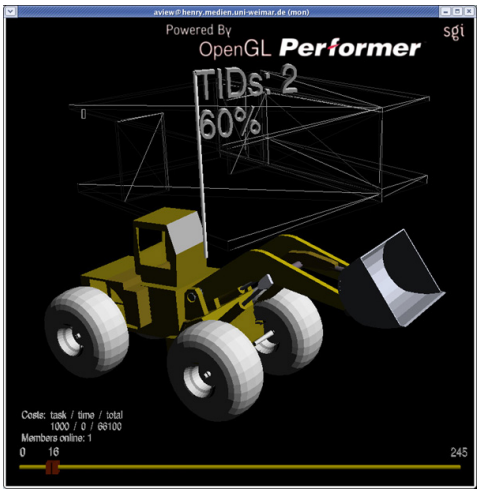

Abbildung 10: Ressource 1

- Es besteht ein räumlicher Konflikt zweier Tasks.
- Die Auslastung ist über 100%.

Es ist also möglich, sich den Bauprozess anzuschauen und anhand der Darstellungsform zu erkennen, ob Konflikte auftreten. Tritt beispielsweise ein Auslastungskonflikt ein, so sollten die entsprechenden Tasks nicht parallel laufen bzw. sich überschneiden. Wird eine operative Ressource zur selben Zeit an unterschiedlichen Orten benötigt, so sollte man entweder versuchen, einen Ort zu wählen, der für beide Task akzeptabel ist oder die Task nicht parallel laufen bzw. überschneiden lassen.

3.4.1. Genaue Funktionsweise

Ein Task vom Typ *fpGanttTask* besitzt u. a. das Feld *Ressources*. In diesem Feld befindet sich eine Liste mit 5 Werten:

1. ID der operativen Ressource
2. Startzeit
3. Dauer
4. Auslastung
5. Ort

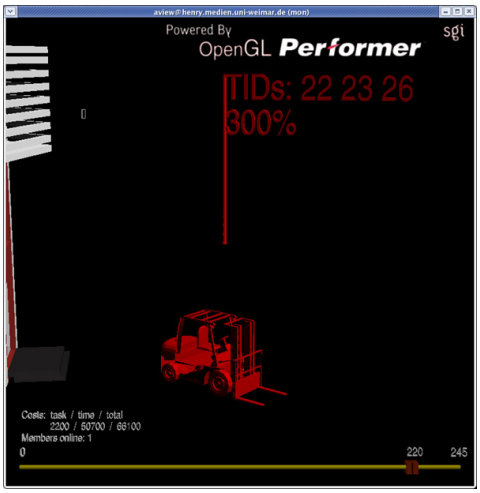

Abbildung 11: Ressource 2

Jeder operativen Ressource wird eine eindeutige ID zugewiesen, um sie identifizieren und zuordnen zu können. Die Startzeit ist relativ zur Startzeit des Tasks (die wiederum relativ zur Startzeit des Projekts ist). Sie kann positiv (Ressource wird jeweiligen Wert nach Beginn des Tasks benötigt) oder negativ (Ressource wird jeweiligen Wert vor Beginn des Tasks benötigt, z.B. Aufbauzeit) sein. Bei einer Änderung der Startzeit des Tasks (bzw. bei einer Änderung der Startzeit des Projekts) müsste also keine weitere Änderung der Liste erfolgen. Änderungen an dem Ressourcenknoten wären ebenfalls nicht notwendig, da die Informationen ja beim Task liegen.

Befindet sich die aktuelle Zeit in dem Intervall aus Startzeit des Tasks, addiert mit Startzeit der Ressource und der Dauer, so wird die Auslastung aus der Ressourcenliste in das Feld *UtilizationRatio* addiert und die ID des Tasks in die Liste CurrentUserIDs hinzugefügt (um die Informationen über der Ressource ausgeben zu können). Hat das Feld *UtilizationRatio* einen Wert über 100, so wird das Modell der Ressource rot maskiert. Dann wird geschaut, ob die Ressource an der richtigen Stelle steht, wenn nicht, dann wird es (sofern es denn noch nicht ist) rot maskiert. Die Position ist dabei nur eine Zahl beginnend bei 1. Da die Position (und Rotation und Skalierung) durch eine Matrix beschrieben wird, dies allerdings nicht gerade benutzerfreundlich ist, nutzen wir diese Abstraktion nach aussen. Intern wird jeder Zahl eine vorher definierte Matrix zugeordnet.

Diese Variante ist durch ihre relativen Verbindungen sehr flexibel. Die Auswertungen (Auslastung etc.) einer Ressource erfolgen erst zur Laufzeit. Es sind keine weiteren Schreiboperationen notwendig, wenn sich irgendwo ein Wert geändert

hat.

Möchte man sich jedoch ein Histogramm der Auslastung einer operativen Ressource anzeigen lassen, so muss anders vorgegangen werden. Da die Informationen ja bei den Tasks liegen und die Ressource erst zu Laufzeit weiß, wer sie verwendet, muss eben einmal durch das ganze Projekt gelaufen werden und die entsprechenden Auslastungen in einer Liste gespeichert werden (*UtilizationRatioList*). Regelmäßige Aktualisierungen wären nur für diesen Fall notwendig.

4. Kommunikation

4.1. Allgemeines

Um die 4D-Planung sinnvoll nutzen zu können, müssen der Ablaufplan und die Visualisierung aus denselben Daten hervorgehen, die synchron dargestellt werden müssen. Für die Synchronisation der Daten und somit die Kommunikation zwischen beiden Programmen wurde uns ein Remote User Interface für Avango zur Verfügung gestellt. Mit diesem Tool sollten wir einfacher auf Ereignisse und Werte von Avango zugreifen können. Leider stand uns das Tool am Anfang des Projektes nicht brauchbar zur Verfügung. Es konnte nicht kompiliert und eingebunden werden. Da wir nicht wussten, ob uns in angemessener Zeit eine kompilierfähige Version des Programms zur Verfügung steht, haben wir nach Alternativen gesucht. Unsere Idee war es, die Kommunikation über TCP oder UDP Protokolle laufen zu lassen. Da es für Avango schon ein UDP-Modul gibt, welches wir nutzen wollten, entschieden wir uns für UDP.

Für die Gantt-Applikation implementierten wir einen UDP-Listener und Receiver auf Basis von Sockets. Damit konnten Nachrichten gesendet und empfangen werden. Eine Auswertung der Nachrichten war auch vorgesehen und wurde ansatzweise implementiert.

Für den Avango-Teil der Applikation machten wir uns das avango-eigene UDP-Modul zunutze. Wir implementierten in den Projektknoten '*fpGanttProject*' entsprechende Felder zum An- und Ausschalten eines UDP-Listeners sowie zum Einrichten der Sender- und EmpfängerPorts und statteten ihn mit der Sende- und Empfangsfunktionalität aus.

Leider kam es bei der Übertragung zu Fehlern. Auch mussten wir feststellen, das es für komplizierte Ereignisse für uns schwierig werden würde, diese über UDP zu senden. Zu dieser Zeit gab es dann auch eine eingebundene und kompilierfähige Version des RUI, so dass wir wieder an unsere Anfangsidee zurückgingen.

Mit Hilfe des RUI war es möglich, Ereignisse von Avango abzufangen, entsprechende Werte abzufragen oder zu setzen. Diese Werte bzw. Ereignisse mussten nun aber noch an das GanttProject weitergegeben werden. Hierfür konnten wir uns eine Java-Funktionalität zu Nutzen machen: das Java Native Interface, welches im

Anschluss erläutert wird.

Der Teil der Kommunikation besteht nun also aus dem Java Native Interface auf der Java-Seite und dem RUI auf der Avango-Seite.

4.2. Java Native Interface

Java bietet mit dem Java Native Interface die Möglichkeit, auf „nativen“ Code zuzugreifen.

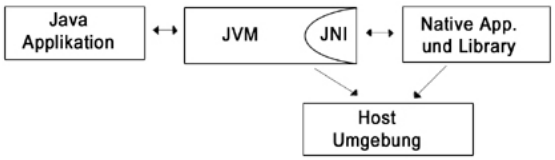

Abbildung 12: Java Native Interface

Das heißt, man kann von einem Java-Programm aus auf z. B. C oder C++ Funktionen zugreifen. Diese Funktionen müssen in einer shared Library (*.so bzw. *.dll) vorhanden sein, die in das Java Programm eingebunden wird:

```
static { System.loadLibrary("pfad") }
```

Die Funktionen, die von Java aus aufgerufen werden sollen, müssen im Java-Code deklariert werden und einer bestimmten Konvention folgen. Bsp.:

```
public native void setTaskName(String id, String name);
```

Auch bei der Implementierung der Methoden in der shared Library muss eine bestimmte Konvention beachtet werden.

```
JNIEXPORT void JNICALL Java_package_
  Klasse_Methode(JNIEnv *env, jobject obj)
```

JNIEXPORT und *JNICALL* sind Macros und dienen nur der Kennzeichnung der Funktion. Der Funktionsname setzt sich aus „Java“, dem Package-Namen, dem Klassennamen und dem Funktionsnamen zusammen. *JNIEnv* und *jobject* werden in jeder Funktion zusätzlich zu den Übergabeparametern übergeben. Bei *JNIEnv* handelt es sich um einen Pointer auf das JNI-Environment Interface, *jobject* ist eine Art „*this*“-Pointer auf das Klassenobjekt selbst. Die Konvertierung von Datentypen muss bei der Implementierung bei Übergabeparametern und Rückgabewerten beachtet werden. So wird ein Integer Wert von *Java(int)* über das JNI als *jint* übergeben. Ähnlich verhält es sich für *double – jdouble* oder *float – jfloat*. Die Parameter können dann in den C-Funktionen einfach als *int* benutzt werden. Auch ein *string* wird als *jstring* übergeben, kann aber nicht einfach als *char** benutzt

werden. Ein *jstring* wird als UTF-8 Format dargestellt und muss für die Benutzung in C konvertiert werden:

```
jstring name = "Test";
const char* taskName = env->GetStringUTFChars(name, 0);
```

Die Rückwärtskonvertierung erfolgt genauso:

```
const char* name = "Test";
jstring taskName = env->NewStringUTF(name);
```

4.2.1. Verbindung Gantt – Avango

Die Verbindung von Gantt und Avango wird von der Gantt-Seite aus gestartet. Zum Zeitpunkt des Verbindungsaufbau muss in Avango ein Projekt vorhanden sein. Wird dort auf über einen Button die Verbindung aufgebaut, wird die Funktion *initGantt* in der Library aufgerufen. Über diese Funktion werden der *JNIEnv*-Pointer und das *jobject* übergeben. Die Parameter werden gebraucht, um Funktionen aus der Library in Gantt aufzurufen.

```
static jclass ganttExtension;
static jmethodID task;
static jobject ganttObject;
static JNIEnv *ganttEnv = 0;

JNIEXPORT void JNICALL
Java_net_sourceforge_ganttproject_
extension_Communication_initGantt(JNIEnv *env, jobject obj) {
  env->NewGlobalRef(obj);

  ganttExtension = env->GetObjectClass(obj);
  ganttObject = obj;
  ganttEnv = env;

  task = ganttEnv->GetMethodID(ganttExtension, "newTask",
    "(Ljava/lang/String;Ljava/lang/String;"
    "Ljava/lang/String;Ljava/lang/String;"
    "Ljava/lang/String;Ljava/lang/String;"
    "Ljava/lang/String;)V");
  ...
}
```

Für die Methode *newTask*, die in Gantt implementiert ist und später in der Library aufgerufen werden soll, wird sich hier die MethodenID geholt. Dazu braucht man den *JNIEnv*-Parameter, über den man die Klasse bekommt, in der die Funktion steht. Das *jobject* wird später für den Aufruf der Java-Funktion gebraucht. Der Ausdruck *Ljava/lang/String* steht für den Übergabewert *String*, der später an die Java Funktion übergeben wird. In diesem Fall werden sieben Strings übergeben. *V* steht für *void*, dies meint den Rückgabewert.

Nach der Initialisierung ruft Gantt die Methode *getTask* in der Library auf. Diese Funktion holt sich über das RUI alle Tasks, die in Avango vorhanden sind. Für jeden Task wird die Methode *makeTask* aufgerufen, an die die Eigenschaften des Tasks übergeben werden, wie Id, Name, Startzeit usw.

```
void makeTask(const char* id,    const char* name,
              const char* start, const char* length,
              const char* costs, const char* father,
              const char* predecessors)
{
  jstring taskID = ganttEnv->NewStringUTF(id);
  jstring taskName = ganttEnv->NewStringUTF(name);
  jstring taskStart = ganttEnv->NewStringUTF(start);
  jstring taskLength = ganttEnv->NewStringUTF(length);
  jstring taskCosts = ganttEnv->NewStringUTF(costs);
  jstring taskFather = ganttEnv->NewStringUTF(father);
  jstring taskPredecessors = ganttEnv->NewStringUTF(predecessors);

  ganttEnv->CallVoidMethod(ganttObject, task, taskID,
                           taskName,    taskStart,
                           taskLength,  taskCosts,
                           taskFather,  taskPredecessors);
}
```

In dieser Funktion wird die JNIEnv und das jobject gebraucht, um die Methode newTask in Gantt aufzurufen. Diese erstellt einen Task mit angegebenen Parametern in Gantt.

4.2.2. Veränderungen in Gantt – Mitteilung an Avango

Veränderungen und Ereignisse, die in Gantt passieren und an Avango weitergeben werden müssen, können sein:

- Veränderung der Taskeigenschaften (Name, Startzeit, Dauer, Abhängigkeiten, VaterTask, Kosten, Ressourcen)
  - über Dialog
  - über Grafik
- Veränderungen am Projekt (Länge, Aktuelle Zeit, ?)
  - Über Timeslider
  - Über Player
  - Über Grafik (Länge)

Für jede Veränderung existiert eine Methode in der Klasse *Communication.java*, die als *native* deklariert ist, diese wird bei entsprechender Veränderung aufgerufen, was wiederum zum Aufruf der entsprechenden Funktion in der Library führt. Beispiel: Für das Ereignishandling des Timesliders gibt es in Java einen Slider-Listener (siehe Gantt). Bei Veränderung des Wertes wird hier die Methode *changeTimeSlider* mit der Position aufgerufen. Die entsprechende Funktion in der Library:

```
JNIEXPORT void JNICALL
Java_net_sourceforge_ganttproject_extension_Communication_
  changeTimeSlider(JNIEnv *env, jobject obj, jint pos) {
```

```
  int position = pos;
  rui.setIntField(*ns,"ActualTime",pos);
}
```

ruft die RUI-Methode auf und dort wird der Wert für Avango gesetzt. Der Timeslider ist also ein Listener in Gantt, welcher auf Ereignisse wartet und diese Veränderungen an die Library übergibt. Werden die Eigenschaften des Projektes oder eines Tasks geändert, wird neben der Speicherung im Programm die entsprechende Methode in der Library aufgerufen.

4.2.3. Veränderungen in Avango – Mitteilung an Gantt:

Tritt ein Ereignis in der Avango-Applikation auf, werden diese Werte direkt an das RUI übertragen. Diese müssen nun an das Gantt weitergegeben werden.

Das JNI bietet die Möglichkeit, Java-Funktionen aus dem nativen Code heraus aufzurufen. Dabei ist es allerdings nötig, über ein Java-Environment (JNIE) zu verfügen. Dieser wird beim Aufruf eine Funktion von Java nach C vom Java-Programm übergeben und nur für die entsprechende Funktion gültig. Wir haben uns daher ein globales JNIE-Objekt erzeugt. Dieses wurde beim ersten Aufruf von Java initialisiert und stand uns damit zur Verfügung. Leider gab es bei dieser Lösung viele Probleme bei der Behandlung von häufigen Ereignissen, mit deren Behebung wir sehr viel Zeit verbrachten. Wir entschieden uns, diese fehleranfällige Lösung nur für die Anfangsinitialisierung zu verwenden und für die Ereignisse eine andere Möglichkeit zu implementieren. Da wir nun nicht mehr die Methoden in Java direkt bei einer Veränderung aufrufen können, müssen die Veränderungen vom Gantt-Programm aus abgefragt werden. Dies muss in kurzen Zeitintervallen passieren. In das Gantt-Programm wurde also ein Timer implementiert, der alle 100ms die Funktion *run* in der Library aufruft.

```
public void timerRun() {
  timer = new Timer(100, new ActionListener() {
    public void actionPerformed(ActionEvent e){
      String handle = com.run();
      handleInput(handle);
    }
  });
  timer.start();
}
```

Diese Funktion ruft in RUI *handleEvent()* auf und gibt die jeweiligen Veränderungen als *String* zurück. Den String haben wir nach einer bestimmten Konvention zusammengesetzt: (*taskID*, *field*, *value*). Trat keine Veränderung im Intervall auf, ist der String leer. Treten mehr Veränderungen als eine auf, werden diese in einer Queue gehalten und nacheinander ausgegeben. Der Rückgabewert der Funktion wird an das Gantt-Programm zurückgegeben und dort ausgewertet. Dafür haben wir die Methode *handleMessage()* implementiert. Die Methode teilt den String in *taskID*, *field* und *value* und setzt die Parameter beim entsprechenden Task.

Beispiel-String: „5 taskLength 20“ – Die Tasklänge von Task5 wurde auf 20 gesetzt.

```
public void handleInput(String input) {
  StringTokenizer st = new StringTokenizer(input, " ");
  if (st.hasMoreTokens()) {
    int taskID = Integer.parseInt(st.nextToken());
    String field = st.nextToken();
    if (field.equalsIgnoreCase("TaskName")) {
      String value = st.nextToken();
      this.setName(taskID, value);
    }
    ...
    else if (field.equalsIgnoreCase("TaskDuration")) {
      int value = Integer.parseInt(st.nextToken());
      this.setTaskLength(taskID, value);
    }
    ...
  }
}
```

4.3. Remote User Interface - RUI

4.3.1. Überblick

Für die Kommunikation zwischen der Avango-Applikation und dem „Java Native Interface“ (kurz: JNI) kommt das in Avango schon vorhandene „Remote User Interface“ (kurz: RUI) zur Anwendung. Die ursprüngliche RUI-Anwendung besitzt zwei Teile. Einen Serverteil, welcher auf der Seite von Avango läuft, und eine Clientapplikation - eine unabhängige Qt-Anwendung, mit welcher man die in Avango vorhandenen Szenengraphen traversieren und verändern kann. Wir nutzen dieses in Avango vorhandene Interface und schrieben uns einen neuen, passenden Client, welchen wir einfach „C Remote Interface“ (kurz: CRI) nannten, da er keine graphische Oberfläche benötigt, sondern uns (vorrangig) nur als Shared Library im Javaprogramm dienen soll.

Das RUI-CRI soll uns die Möglichkeit bieten, die vorhandenen Szenengraphen in Avango komplett auszulesen (inklusive aller vorhandenen Knoten, Felder und deren Wertbelegungen), um somit den zeitlichen Planungsprozess im javabasierten Gantt-Chart modellieren zu können. Neben dem Auslesen der Daten ist es natürlich wichtig, dass auch Änderungen über dieses Interface möglich sind und diese an Avango weitergeleitet werden. Avango soll die benötigten Daten speichern und visualisieren, während der Planungsprozess komplett mit Hilfe des Gantt-Chart verwaltet werden soll.

Das von uns entwickelte CRI kann in zwei verschiedenen Art und Weisen angewandt werden. Als erste und wesentliche Aufgabe ist das SharedObject zu sehen, welches beim Starten der Java-GanttChart-Applikation geladen wird. Die Javaobjekte können mit Hilfe des JNI und dem CRI mit Avango kommunizieren. Das SharedObject läuft hierbei in demselben Systemprozess wie die Javaanwendung.

Als zweites besteht die Möglichkeit, das CRI als Konsolentool zu verwenden und den Szenengraphen auf Ebene der Eingabeaufforderung auszulesen und zu manipulieren.

Da ein implementierter Client schon vorhanden war, ist es nicht unser Ziel gewesen, die Schnittstellen zum Serversystem komplett neu zu schreiben, sondern vielmehr war es uns wichtig, die vorhandenen Methoden zu nutzen und die Clientseite nur von dem für uns unnötigen Overhead (z. B. Qt) zu befreien.

Um die vorhanden Methoden ausnutzen zu können, wird sobald ein Objekt unserer Klasse *RuiToCRI* erzeugt wird, auch ein öffentliches Objekt der (vorhandenen Client-) Klasse *RUI::RUIClientNet* erzeugt, mit welchem man auf die wesentlichen Methoden zugreifen kann.

4.3.2. Verbindungsaufbau

Ein sehr gutes Beispiel für das Wiederverwenden von vorhandenen Methoden ist das Aufbauen der Verbindung zwischen Client und Server. Die Methode hierzu heißt *RuiToCRI::init(...)* und bekommt als Signatur die Netzwerkadresse und den Port des Serversystems übergeben - Rückgabewert ist ein Integer, welcher signalisiert, ob der Verbindungsaufbau gelungen ist.

Wenn man sich die Implementation genauer ansieht, wird man feststellen, dass intern die Methode *RUIClientNet::connect* auf dem Objekt *mRuiClient_*, mit der entsprechenden Signatur aufgerufen wird. Diese *connect*-Methode wird von *RUI::-RUIClientNet* zur Verfügung gestellt und vollzieht den Verbindungsaufbau zwischen dem Clientsystem und dem Avango-Server. Unsere *RuiToCRI::init*-Methode wurde um ein Exception-Handling erweitert, damit falls der Verbindungsaufbau scheitert, dass Programm nicht unkontrolliert unterbricht, sondern den Benutzer über das Problem informiert.

4.3.3. Zugriff auf Knoten in der Avango-Datenstruktur

Um den Avango-Szenengraphen auslesen zu können, benötigt man Zugriff auf die einzelnen Elemente und deren Felder, in denen die benötigten Informationen gespeichert sind. Diesen Zugriff erhält man, indem man nachdem eine Verbindung zu dem Server aufgebaut worden ist, einen Startpunkt/-knoten mit der Methode *NodeStatePtr RuiToCRI::Scenegraph(...)* initialisiert. Als Signatur wird dabei der Name des gewünschten Rootknotens als *std::string* übergeben und Rückgabewert ist ein *std::auto_ptr* auf den gewünschten Knoten vom Typ *RUI::NodeState*. Wenn der Rootknoten initialisiert worden ist und zur Verfügung steht, dient dieser als Grundlage für diverse Methoden, um gewünschten Informationen zu erhalten, z. B. durch Traversierung.

Diesen Rootknoten oder andere Knoten vom Typ *RUI::NodeState* kann man nun als Teil einer Signatur für verschiedenste von uns implementierte *RuiToCRI-*

Methoden verwenden, die im folgenden kurz erläutert werden:

1. *unsigned int getChildrenCount(...)*
   Diese Methode liefert die Anzahl der Kinderknoten von einem in der Signatur übergebenen Knoten *node_state*, zurück. Die Anzahl der Kinderknoten wird für die Traversierung benötigt.

2. *NodeStatePtr getChildNode(...)*
   Diese Methode liefert einen *std::auto_ptr* vom Typ *RUI::NodeState* auf einen gefundenen Kinderknoten zurück. Die Methode existiert zweimal und unterscheidet sich nur in der Signatur dahingehend, dass eine Methode anstelle der Knotenindexnummer *uiChildIndex* den Namen *id* des Knotens übergeben bekommt. Beide Methoden suchen nur in der ersten Ebene unterhalb des übergebenen Knotens *node_state*. Des Weiteren ist zu beachten, dass die zweite Methode (Übergabe mit *std::string*) einen Pointer auf das erste gefundene Objekt zurückgibt – die Suche erfolgt hierbei mit Hilfe der Indexnummer und der ersten Methode (Übergabe mit *unsigned*).

3. *RUI::FieldState* getFieldNode(...)*
   Diese Methode liefert einen Pointer vom Typ *RUI::FieldState* auf ein Feld eines in der Signatur übergebenen Knotens *node_state* zurück. Die Identifikation des Felder erfolgt über den Feldnamen strFieldName vom Typ *std::string*, welcher zusätzlich in der Signatur angegeben wird.

4.3.4. Zugriff auf Felder von Knoten in der Avango-Datenstruktur

Wenn man einen Pointer vom Typ *RUI::FieldState* auf ein Feld von einem Avango-Knoten besitzt, dann gibt es nun die Möglichkeiten, die Werte der Felder auszulesen und gegebenenfalls neu zu belegen. Das Auslesen wird z. B. benötigt, um das in Avango vorliegende Modell im GanttProject aufzubauen. Nachdem der Benutzer dieses Modell im Gantt vorliegen hat, möchte er natürlich auch Werte ändern und diese Änderung sofort in Avango sehen – es ist daher nötig, dass man Werte auch setzen kann.

1. *std::string getValueOfField(...)*
   Diese Methode existiert auch wieder mit zwei unterschiedlichen Signaturen. Beide lesen die Werte eines Feldes aus und geben diese als *std::string* zurück. Die Signatur enthält in beiden Fällen den Knoten (*node_state*) und dann die Spezifikation des Feldes, entweder Feldname (*strFieldName*) oder Feld-ID (*uiFieldIndex*).

2. *std::string getValueOfIntVector(...)*
   Nach Angabe des Knotens (*node_state*), des Feldnamens (*strFieldName*) und

der Position in dem Vector (*pos*) liefert diese Methode den entsprechenden Integer-Wert als *std::string* zurück.

3. *std::string getIntVector(...)*
   Diese Methode liefert den ganzen Integer-Vector von einem Integer-Feld (*strFieldName*) eines Knotens (*node_state*) als *std::string* zurück – die einzelnen Wert sind durch Leerzeichen voneinander getrennt.

Das Setzen der Werte von der Seite des RUI-Clients erfolgt mit folgenden Methoden:

1. *void setIntField(...)*
   void setStrField(...)
   Diese beiden Methoden setzen vorhandene Felder in dem Avango-Szenegraphen neu. Wichtig hierbei ist, dass man zwischen dem Setzen von Integer-Werten (*void setIntField(...)*) und dem Setzen von String-Werten (*void setStrField(...)*) unterscheiden muss. In der Signatur wird der gewünschte Knoten (*node_state*), das zu ändernde Feld (*strFieldName*) und dann der zu setzende Wert (*iValue* oder *strValue*) angegeben.

2. *void addValueToIntVectorField(...)*
   Mit Hilfe dieser Methode wird ein Integer-Wert (*vctValue*) in ein vorhandenes Integer-Vectorfeld (*strFieldName*) von einem gegebenen Knoten (*node_state*) hinzugefügt. Der Integer-Wert repräsentiert in unserem Fall z. B. Taskabhängigkeiten.

3. *void removeValueFromIntVectorField(...)*
   Diese Methode löscht einen vorhandenen Integer-Wert (*vctValue*) aus einem Integer-Vectorfeld (*strFieldName*) von einem spezifizierten Knoten (*node_state*).

Für alle get- und set-Methoden ist es notwendig, einen entsprechenden Knoten und das jeweilige Feld anzugeben. Die Methoden suchen dann intern in dem Avango-Szenengraphen, ausgehend vom initialisierten Rootknoten, die Knoten und Felder heraus, indem sie den Szenengraphen traversieren. Diese Technik des ständigen Traversierens ist sehr langsam und zeitaufwendig. Aus diesem Grund waren wir gezwungen, eine Beschleunigung zu implementieren, welche in dem Abschnitt 4.3.6 auf Seite 29 genauer erläutert wird.

4.3.5. Änderungen auf Seite des Avango-Serversystemes

Alle bis jetzt beschriebenen Methoden beruhen darauf, dass das Clientsystem Aktionen auslöst. Was ist jedoch, wenn Änderungen auf Seite von Avango geschehen? Diese Änderungen müssen dem Client mitgeteilt werden, damit die Änderungen

auch im Gantt Chart ausgelöst werden können. Das einfachste Szenario hier ist, dass man ein verteiltes „4d-CAD“-System hat, wobei man für die Verteilung den Verteilungsmechanismus von Avango nutzt.

Damit das Clientsystem von den Änderungen des Avango-Szenengraphen erfährt, muss der Client die gewünschten Felder „abonnieren“, über dessen Änderungen er erfahren möchte.

1. *void subscribeField(...)*
   Mit Hilfer dieser Methode kann man ein Feld (*strFieldName*) eines Knotens (*node_state*) des Szenengraphen abonnieren. Jede Änderung, die auf Seite von Avango durchgeführt wird, wird dem Client mitgeteilt.

2. *void unsubscribeField(...)*
   Diese Methode dient zum Abbestellen von überwachten Feldern. Die Signatur ist gleich der Methode *subscribeField(...)* aus 1.

Damit man über Änderungen der abonnierten Felder erfährt, wurde noch eine Art „Listener“-Methode implementiert, der in regelmäßigen Zeitabständen den Server abfragt.

1. *void handleConnection()*
   Diese Methode wird in regelmäßigen Abständen vom JNI aufgerufen. Sie ruft intern *handleEvents()* von *RUI::RUIClientNet* auf, welches die Queue mit den Änderungen abarbeitet. Diese Methode speichert die Änderungen in einem String, welche vom JNI nach dem Aufruf von *handleConnection()* ausgelesen werden.

Interessant ist es, sich die Methode *handleEvents()* von *RUI::RUIClientNet* näher anzusehen. Diese Methode macht einen Broadcast mit der erhaltenen Message. Dieser Broadcast ist in *RUI::RUIClient* definiert – er ruft die Methode *processMessage(msg)* von jedem Listener auf, der in der Listener-Liste (*mListenerList*) enthalten ist.

```
void RUIClientNet::handleEvents() throw (TransportException)
{
  ...
  broadcast(msg);
  ...
}

void RUIClient::broadcast( Message* msg )
{
  for(int i=0; i<mListenerList.size(); i++)
  {
    mListenerList[i]->processMessage(msg);
  }
  delete msg;
}
```

Wie man anhand des Quelltextes erkennen kann, wird zum Abarbeiten der ankommenden Events eine eigene Listener-Klasse benötigt, welche von *RUI::MessageListener* abgeleitet worden ist. Diese Klasse haben wir implementiert – sie heißt *CRIListener* und hat *RUI::MessageListener* als abgeleitete Basisklasse. *CRIListener* besitzt lediglich einen Konstruktor *CRIListener(RuiToCRI &rc)*, einen virtuellen Destruktor *~CRIListener()* und eine virtuelle Methode *processMessage(...)*.

Sobald der Konstruktor angerufen wird, wird ein Objekt der Klasse erzeugt – wichtig ist hierbei, dass in der Implementierung des Konstruktors die Methode *addListener()* von *RUI::RUIClientNet* aufgerufen wird, die den aktuellen *CRIListener* zu der Liste aller Listener hinzufügt, die dann durch die oben gezeigte Implementieren von *handleEvents()* von *RUI::RUIClientNet* durchlaufen wird.

4.3.6. Optimierung des CRI

Das Auslesen und Setzen von Werten in Avango-Feldern erforderte immer eine Traversierung des Szenengraphen auf Seite des Avango-Servers, damit man den entsprechenden Pointer auf das gewünschte Feld bekam. Da dieses ständige Traversieren des Avango-Szenengraphen (inklusive alle Felder und deren Werte) sehr zeitaufwendig war und doch eine spürbare Verzögerung im Gantt Chart mit sich führte, waren wir gezwungen, den Vorgang der Traversierung zum Finden der erforderlichen Pointer zu optimieren. Die Optimierung erfolgt durch die Speicherung aller benutzten und noch nicht gespeicherten Pointer auf Avango-Felder in einer *std::map*.

Sobald ein Avango-Feld das erste Mal ausgelesen oder gesetzt wird, wird der Pointer zu einer *std::map* hinzugefügt. Da man zu Beginn den ganzen Szenengraphen auslesen muss, damit man die komplette Ansicht im GanttProject generieren kann, werden an dieser Stelle auch alle Pointer in der Map abgelegt. Bei jedem kommenden Zugriff auf ein Feld wird zuerst in der Map nach dem gewünschten Pointer gesucht (mit Hilfe der auf *std::map* definierten Methode *find()*) und nur wenn der Pointer noch nicht vorhanden ist, wird er in dem Szenengraphen gesucht und dann in der Map abgelegt. Der Zugriff und die Suche in einer *std::map* ist wesentlich effizienter als die Traversierung des Szenengraphen. Man kann den Unterschied sehen, wenn man das Konsolentool benutzt, da zu Beginn der Szenengraph zweimal ausgegeben wird - das erste Mal müssen alle benötigten Pointer von dem Avango-Server „geholt“ werden und bei den zweiten Durchlauf werden die Pointer aus der *std::map* benutzt - der Unterschied ist klar sichtbar.

4.3.7. Änderungen an den Originaldateien des Clientsystems

Damit unser Code sinnvoll lauffähig war, mussten wir in zwei Originaldateien von Avango Änderungen vornehmen:

1. *Value.h*
   In dieser Headerdatei ist für *class Value* eine Methode *virtual std::ostream& dumpValue(std::ostream& s) const = 0* definiert, welche im Original auf *protected* gesetzt ist, das heißt wir können sie mit unserer *RuiToCRI*-Klasse nicht verwenden. Da wir jedoch diese Funktion zum Ausgeben der Werte von Feldern eines Avango-Knoten benötigen, haben wir *protected* auskommentiert, was zur Folge hat, dass diese Methode nun *public* ist.
   Dasselbe gilt für *class VectorValue:public Value*. Dort haben wir auch die Methode *virtual std::ostream& dumpValue(std::ostream& s) const = 0* von *protected* auf *public* gesetzt, damit wir Vectorfelder direkt auslesen können.

2. *RUIClientNet.h*
   Im Originalfile befindet sich die Zeile *#inlcude <vector.h>* – diese Headerdatei ist jedoch nicht mehr aktuell und es werden entsprechend vom Compiler Warnungen ausgegeben. Wir haben diese Zeile in *#include <vector> geändert.*

4.3.8. RuiToCRI als Konsolentool

Man kann *RuiToCRI* auch als kleines Konsolentool verwenden, um sich von einer Linux-Shell zu einem Avango-Server zu verbinden. Die Funktionen des Tools erlauben es, sich von einem Rootknoten ausgehend alle Kinderknoten inklusive der Werte der einzelnen Felder ausgeben lassen. Des Weiteren kann man auch Avango-Felder abonnieren, über deren Änderung man dann entsprechend auf der Shellebene informiert wird. Der Syntaxaufruf für dieses Tool lautet: *./RuiToCRI hostname portnumber rootnode [subscribe fields].*

1. *./RuiToCRI*
   Konsolenprogramm

2. *hostname*
   Hostname oder IP-Nummer des Avango-Servers

3. *portnumber*
   Portnummer des Avango-Servers

4. *rootnode*
   Name des Knotens, von dem die Szenengraph-Traversierung beginnen soll

5. *subscribe fields*
   Liste von abonnierten Feldern - muss nicht angegeben werden

Wenn beim Aufruf des Konsolenprogramm die Syntax nicht richtig ist, bekommt der User einen (kleinen) Hilfetext auf den Konsole ausgegeben.

5. Gantt

5.1. Einleitung

5.1.1. Was war die Aufgabe

Da der Projektschwerpunkt nicht in der Entwicklung einer Gantt-Applikation lag, fiel die Entscheidung früh, ein vorhandenes Gantt-Programm zu wählen und dieses unseren Anforderungen anzupassen. Nach einer Internet-Recherchephase konnten wir die meisten angebotenen Programme aufgrund von unreifen bzw. trivialen Entwicklungen ausschließen. In Punkt 5.1.3 auf Seite 31 wird das Ergebnis dargestellt. In der darauffolgenden Zeit haben wir das Ganttprogramm (Punkt 5.2 auf Seite 32) so modifiziert, dass es als entferntes Planungstool Avango steuern kann (Punkt 5.3.1 auf Seite 35), sowie für uns relevante, zusätzliche Informationen visualisiert (siehe Abschnitt 5.3.2 auf Seite 39).

5.1.2. Was ist Gantt?

Das von Henry L. Gantt 1917 entwickelte Balkendiagramm diente in seiner Anfangszeit den Architekten und Bauingenieuren als Hilfsmittel zur visuellen Darstellung einer zeitlichen Bauprojektplanung. Im Laufe der Zeit entwickelte sich das Diagramm (hauptsächlich aufgrund des Aufkommens von Computern) zu einem Allroundtalent der allgemeinen Projektplanung und dient heutzutage in unterschiedlichen Branchen als elektronisches Planungstool.

Das allgemeine Ganttdiagramm ist sehr einfach aufgebaut und besteht hauptsächlich aus horizontal über die Zeit (Kalender) aufgetragenen Balken, die benannt werden und einzelne Projektabschnitte (z. B. im Baubereich das Errichten des Kellers eines Hauses) kennzeichnen (siehe Abbildung 13 auf der nächsten Seite).

5.1.3. Was sollte Gantt leisten?

Das Ganttprogramm (im Folgenden wird von Gantt gesprochen) als „Steuereinheit“ der Projektarbeitsumgebung sollte alle gängigen Funktionen, die ein Ganttdiagramm unterstützt, aufweisen sowie die Möglichkeit bieten, eine Programmierschnittstellen bereitzustellen.

Im Folgenden werden die Funktionen kurz skizziert:

Task einen Namen zuweisen/markieren	Unterscheidung
Bewegung eines Tasks (vor/zurück) durch Maus	Start-/Endzeitänderung
Länge eines Tasks ändern	Taskdaueränderung
Task einen Vatertask zuweisen	Hierarchiebildung
Task einen Vorgängertask zuweisen	Abhängigkeiten

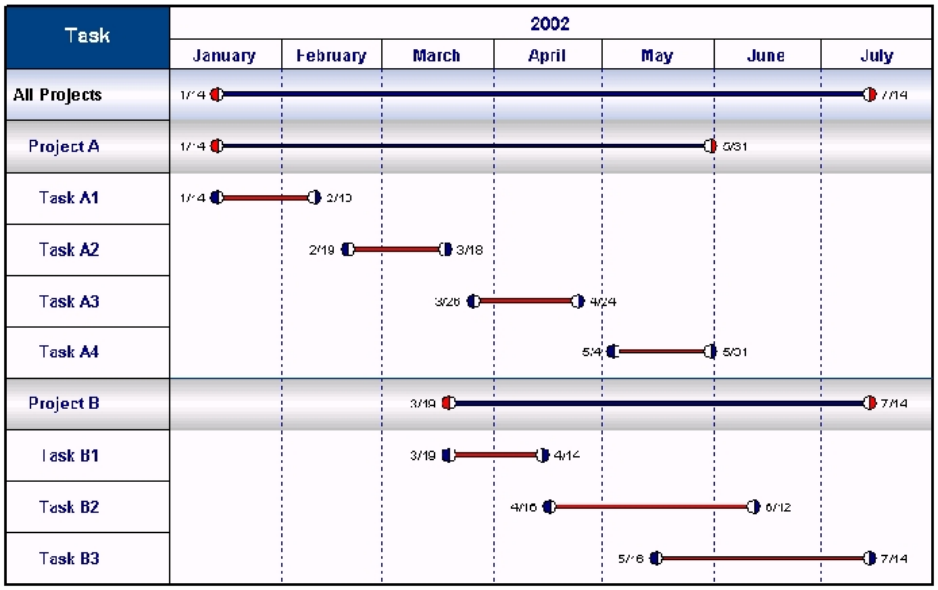

Abbildung 13: Gantt Chart

Zudem sollte das Programm bedienbar, erweiterbar und relativ ausgereift sein.

Am Ende der Recherche standen zwei Ganttapplikationen zur Auswahl: das proprietäre „Microsoft Project“ und das Open-Source-Programm „GanttProject“. Wir entschieden uns dafür, GanttProject zu nehmen, da es für die spätere Arbeit leichter handzuhaben erschien: „MS Project“ wäre durch die COM-Schnittstelle (Component Object Model) zwar auch programmiertechnisch interaktiv und erweiterbar gewesen, jedoch im Vergleich mit der Open-Source-Java-Applikation „GanttProject“ bietet sie zum einen keine Plattform-Unabhängigkeit und zum anderen wäre man von Lizenzen (Kosten!) abhängig gewesen.

5.2. GanttProject Anfangszustand

5.2.1. Aufbau GanttProject

GanttProject wurde an der französischen Universität Marne-la-Vallee als Open-Source-Applikation in Java entwickelt und wird stets erweitert.

Die GanttProject-Version 1.9.12 diente als Grundlage unserer Entwicklungen. Zu diesem Zeitpunkt bestand die Applikation aus 19 Packages (siehe Abbildung 14 auf der nächsten Seite), die insgesamt ca. 200 Dateien, also ungefähr 230 Klassen bein-

halten und zusammen ˜50.000 Zeilen Quellcode zur Interaktion bereitstellen. Da natürlich nicht alle Klassen von uns editiert bzw. geändert werden mussten und in manchen auch nur kleine Veränderungen vorgenommen wurden, werden im Folgenden nur die wichtigsten Klassen und Methoden vorgestellt, um einen Überblick über die Struktur einzelner Elemente zu geben.

Abbildung 14: Packages

Die Klassen in den grün gefärbten Packages wurden zum Teil geändert.

5.2.2. Wichtige Klassen und Methoden

GanttProject.java: *GanttProject.java* ist die Hauptklasse von GanttProject, in der die *main*-Funktion aufgerufen wird. Sie beinhaltet alle wichtigen Instanzen der Klassen für den Aufbau der Applikation. Neben der Erstellung des Menüs werden hier u. a. die zwei wichtigen Funktionen *newTask()* und *deleteTask()* definiert. *newTask()* (siehe Abbildung 15) erstellt dabei ein neues GanttTask-Objekt (einen Balken im Diagramm), welches danach mit Standardwerten (Länge, Start, Name, etc) gesetzt wird. Zusätzlich wird der Task relativ zu einem ggf. ausgewählten Task positioniert (Hierarchie), sowie im Taskmanager registiert und in den Baum („GanttTree“) aufgenommen.

GanttGraphicArea.java: In dieser Klasse wird das Kalenderfenster definiert, sowie die Zeichnung und die Interaktion der Task-Balken implementiert. Hierbei sind besonders die MouseListener wichtig, da diese die User Interaktion bereitstellen, sowie für unseren späteren Gebrauch die Schnittstelle zur Remote-Steuerung bieten.

```
public void newTask() {
  tabpane.setSelectedIndex(0);
  GanttTask current = tree.getSelectedTask();
  GanttCalendar cal = new GanttCalendar();
  if (tree.hasTasks())
    cal = new GanttCalendar(area.beg);
  DefaultMutableTreeNode node = null;
  GanttLanguage lang = GanttLanguage.getInstance();
  String nameOfTask = language.getText("newTask");
  ...
  GanttTask task = getTaskManager().createTask();
  task.setStart(cal);
  task.setLength(5);
  getTaskManager().registerTask(task);
  //create a new task in the tab
  // paneneed to register it
  task.setName(nameOfTask + "_" + task.getTaskID());
  task.setColor(area.getTaskColor());
  ...
  DefaultMutableTreeNode taskNode = tree.addObject(task, node);
  AdjustTaskBoundsAlgorithm alg = getTaskManager()
    .getAlgorithmCollection().getAdjustTaskBoundsAlgorithm();
  alg.run(task);
  ...
}
```

Abbildung 15: newTask()

GanttTask.java: Die nächste wichtige Klasse heißt *GanttTask* und repräsentiert einen Task in GanttProject. Die Struktur dieser Klasse ist etwas umfangreicher und wird kurz anhand Abbildung 16 auf der nächsten Seite näher erläutert:

Die oberste „Vaterklasse“ heißt „*MutableTask*“ und ist ein Interface, welches alle benötigten set-Funktionen für den späteren Gebrauch eines Tasks bereitstellt. Davon abgeleitet ist das Interface Task, welches die set-Methoden erbt und zusätzlich die get-Methoden und die Membervariablen (Name des Task, Länge, Start, etc.) des Tasks erstellt. Implementiert werden diese Funktionen (get-, wie auch set-Methoden) durch die Klasse „*TaskImpl*“. Von dieser abgeleitet ist die Klasse „*GanttTask*“, welche im Folgenden nun immer benutzt wird, um einen neuen Task zu erstellen.

Möchte man nun also, so wie wir, neue Felder einem Task zuweisen, so muss man diese in dem Interface „*Task*“ eintragen und ggf. Funktionen in „*TaskImpl*“ implementieren.

GanttDialogProperties.java, GanttTaskPropertiesBean.java: Diese zwei Klassen beschreiben den Eigenschaften-Dialog, den man in GanttProject aufrufen kann (über Menu oder Doppelklick auf Task), um Parameter eines Tasks zu ändern. Dabei implementiert *GanttTaskPropertiesBean.java* die grafische Oberfläche des Dialogs. Hier ist es also möglich, neue Eigenschaften eines Tasks, die zum Beispiel

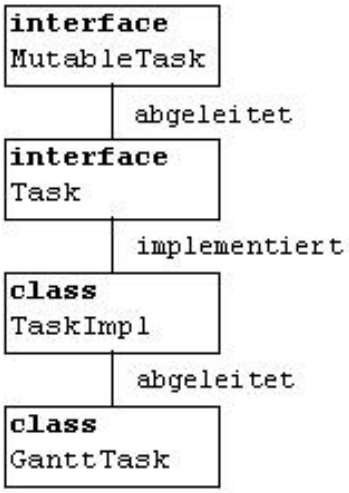

Abbildung 16: taskImpl

wie oben beschrieben, in dem Interface „*Task*“ angelegt wurden, dem Benutzer bereitzustellen und editierbar zu machen. *GanttDialogProperties.java* bietet die Funktionen an, um die eingegebenen Daten zu verarbeiten. Diese Klasse bietet wie *GanttGraphicArea.java* eine Schnittstelle zur späteren Interaktion mit Avango.

GanttResourcePanel.java: Diese Klasse ist zuständig für die Visualisierung der Ressourcen als Gantt-Diagramm, sowie für die Organisierung des „HumanResourceManager[s]“, welcher die gespeicherten und angelegten Ressourcen verwaltet. Eine solche Ressource ist ein Objekt der Klasse *HumanResource*.

In diesem Bereich ist es also möglich, für unsere Zwecke neue Ressourcen zu definieren, da das Ressourcenmanagement in dem ursprünglichen Gantt sich nur auf Ressourcen in Form von Personenangaben beschränken.

5.3. Projektarbeit

5.3.1. Remotesteuerung

In diesem Abschnitt werden die Kommunikation und die dazu nötigen Elemente in Gantt betrachtet. Zur Erläuterung des allgemeinen Aufbaus und der Struktur der Kommunikation zwischen Gantt und Avango verweise ich auf das Kapitel „Kommunikation“ (siehe 4 auf Seite 19).

Interne Erweiterung

Interne Erweiterungen umfassen die Implementierung von set- und get-Funktionen für die Tasks und Ressourcen, die Implementierung von Funktionen zur Handhabung von Objekten, Variablen, etc., sowie die modifizierten *newTask()*- und *deleteTask()*-Funktionen.

Ein Beispiel solcher Hilfsfunktionen ist beispielsweise die *getGanttTaskfromTree()*-Funktion (siehe Abbildung 17): alle erzeugten Tasks werden sinnigerweise in GanttProject in einem Baum gespeichert. Leider gab es keine Möglichkeit, ein GanttTask-Objekt anhand seines Namens aus dem Baum zu lesen, was zum Beispiel bei der *deleteTask()*-Funktion gebraucht wird. Aufgrund dessen musste die in Abbildung 17 dargestellte Funktion erstellt werden.

```
public GanttTask getGanttTaskFromTree(String name)
{
  GanttTask returnTask = null;
  ArrayList temp = new ArrayList(500);
  temp = tree.getAllGanttTasks();
  GanttTask tmp;
  for (int i = 0; i < temp.size(); i++)
  {
    tmp = (GanttTask) temp.get(i);
    if (tmp.getName() == name)
    {
      returnTask = tmp;
      break;
    }
  }
  return returnTask;
}
```

Abbildung 17: getGanttTaskFromTree()

Weitere Beispiele solcher Funktionen sind *getTaskByID()*, oder *getLastID()*.

Die Implementierung von (besonderen) set-Funktionen für die Tasks war aus zwei Gründen wichtig: zum einen gab es nicht für alle Möglichkeiten (siehe 5.1.3 auf Seite 31) eine set-Funktion, da entweder die Struktur dies nicht zuließ (Beispiel: eine Funktion „setFather“ existiert nicht. Ein entsprechender Befehl wird nur implizit durch die Baumstruktur hervorgerufen, d. h. der Knoten unter dem man den neuen Task einfügt, ist der Vater – man kann jedoch nicht explizit sagen – der Task hat den Task als Vater), oder aber manche Funktionen noch nicht vorhanden sein konnten, da wir die nötigen Felder (Kosten, Resourcen, usw.) erst später eingefügt haben. Zudem waren die Funktionen in unterschiedlichen Klassen verteilt, so dass wir zusätzlich einen gemeinsamen Pool all dieser Funktionen bereitstellen wollten.

Der zweite Grund war, dass die Funktionen natürlich Memberfunktionen waren, d. h. dass man sie direkt auf dem Objekt ausführen musste. Da wir jedoch bei einer Remote-Steuerung das Objekt erstmal nicht besitzen, erschien es uns leichter, die

Task-Identifikation als Parameter der betreffenden Funktion zu übergeben, um dann diese Funktion überall aufrufen zu können. Ein typisches Beispiel sieht man in Abbildung 18:

```
public void setTaskPredecessor(int id, int predecessorID)
{
  currentTask = getTaskByID(id);
  if (currentTask != null)
  {
    GanttTask predecessorTask = getTaskByID(predecessorID);
    try
    {
      ganttProject.getTaskManager().getDependencyCollection()
        .createDependency(currentTask, predecessorTask);
      //successor, predecessor
    }
    catch (TaskDependencyException e1)
    {
      e1.printStackTrace();
    }
  }
  else System.err.println("Error setting predecessor
    Task doesn't exist");
}
```

Abbildung 18: setTaskPredecessor()

Die Struktur dieser Funktionen ist prinzipiell immer die gleiche: zuerst wird der Task anhand der übergebenen ID identifiziert, dann der zugehörige Wert, der sich ändert. In diesem Fall wird die Vorgänger-ID übergeben, so dass nun festgelegt wird, dass der Task mit der ID „*id*“, den Vorgänger-Task mit der ID „*predecessorID*“ bekommt. Meist wird an dieser Stelle auch noch ein *repaint()* aufgerufen, um die aktuelle Grafik neu zu generieren, damit die Veränderungen sichtbar werden.

Einer der wichtigsten Funktionen ist die *newTask()*-Funktion. Diese ist, wie eingangs erwähnt, schon in GanttProject vorhanden gewesen und generiert einen neuen Task mit Standardwerten. Da jedoch die Tasks zu einem Projekt mit ihren spezifischen Parametern in Avango gespeichert sind, musste es nun möglich sein, einen Task zu generieren, der mit individuellen Parametern initialisiert wird.

Abbildung 19 zeigt nun die veränderte Funktion (vgl. Abbildung 15):

Man erkennt dort, dass die eben erwähnten set-Funktionen hier verwendet werden, um den Task zu initialisieren. Alle diese Funktionen findet man kommentiert in der Extension.java Datei.

Externe Erweiterung

Wie im Kapitel Kommunikation beschrieben und nachzulesen, benötigt man für die Kommunikation von Gantt zu Avango (also Gantt schreibt in ein Feld von

```
public void newTask(int ID, String name, int start,
  int length, int costs,  int father, int predecessors[])
{
  ...
  GanttTask task = (ganttProject.getTaskManager()).createTask();
  //new task

  task.setTaskID(ID);
  task.setStart(cal);
  task.setLength(length);
  task.setCosts(costs);
  ganttProject.getTaskManager().registerTask(task);
  //create a new task in the tab

  task.setName(name);
  task.setColor(area.getTaskColor());

  //set father
  GanttTask parent;
  node = tree.getNode(father);
  DefaultMutableTreeNode taskNode = tree
    .addObject(task, node); //child, parent

  try {
    for (int i = 0; i < predecessors.length; i++) {
      System.out.print("Abhaengigkeit gebildet"
        + predecessors.length);
      ganttProject.getTaskManager().getDependencyCollection()
        .createDependency(task, getTaskByID(predecessors[i]));
      //successor, predecessor
    }
  }
  catch (TaskDependencyException e1) {
    e1.printStackTrace();
  }
  ...
}
```

Abbildung 19: newTask()

Avango) native Methoden (deswegen extern), die in Gantt in der Klasse *communication.java* definiert sind (Bsp.: Abbildung 20). Diese werden dann in der *communication.cpp* implementiert.

Die Kommunikation von Avango nach Gantt wird von der *HandleInput*-Funktion geregelt, welche nach dem Initialisierung von Gantt gestartet wird.

Die wichtigste Klasse für die Kommunikation ist die *communication.java*, welche die Bibliothek lädt, die nötig ist, um außerhalb von Gantt Funktionen auszuführen, die nativen Funktionen definiert, sowie die Funktionen beinhaltet, die Avango auf Gantt Seite ausführen kann (Bsp. *newTask()*).

Im Folgenden wird zur Vereinfachung als Avango nicht nur das VR Framework bezeichnet, sondern auch die Kommunikationsschnittstelle (Remote User Interface, RUI) derselbigen.

Zusätzlich werden die Funktionen, auf die ich mich beziehe, die explizit die Kommunikation darstellen, nicht weiter erklärt. Erklärungen kann man im Abschnitt 4 „Kommunikation“ nachlesen.

5.3.2. Erweiterung

Unter den folgenden zwei Punkten werden die einzelnen von uns erstellten Erweiterungen von GanttProject, logisch, also strukturell, sowie visuell dargestellt.

Logische Erweiterungen
Die logischen Erweiterungen umfassen alle Erweiterungen, die nötig waren, um die Gantt-Applikation als Steuerungstool für Bauplanung mit Avango zu nutzen.

Projektstart
Task-Start, wie auch -Ende werden in GanttProject als Datum angegeben (Klasse „*GanttCalender*“ abgeleitet von „*GregorianCalendar*“). Da dieses Format für die Kommunikation (nur Strings und Integer sind übertragbar), sowie für Speicherung in Avango unhandlich ist, haben wir uns entschlossen, einen eindeutigen Projektstart zu definieren. Alle Tasks werden dann relativ zu diesem Projektstart definiert. In dem Startfeld des Tasks steht also zum Beispiel eine „4“, wenn der Task vier Tage nach dem Projektstart anfängt. Wir haben Tage als Einheit gewählt, da uns dies als guter Mittelweg zwischen großen und kleinen Zeiteinheiten erschien. Zur Darstellung der Tasks wird intern das wahre Datum (in der richtigen Form) berechnet, da die Tasks bezüglich des Kalenders dargestellt werden und dies für zeitliche Berechnungen das ideale Format darstellt (Operationen für Monatslänge, Schaltjahre, etc. vorhanden).

Den Projektstart, den man nun über das Menu editieren kann, wird als schwarzer senkrechter Strich in der Kalenderdarstellung angezeigt. An diesem Datum orientieren sich alle Tasks.

Implementiert ist das Menu in „*GanttProject.java*“, die Visualisierung in „*GanttGraphicArea.java*“.

TimeSlider
Der Timeslider ist eins der wichtigsten und ersten Erweiterungen, die wir realisiert haben.

Da Avango als Visualisierung dienen sollte, musste ein Weg gefunden werden, um die aktuelle Zeit für den jeweiligen Bauabschnitt, in dem man sich gerade befindet, anzugeben, sowie dem Benutzer die Möglichkeit zu geben können, schnell und bequem über die Zeit von Anfang bis Ende des Projekts im Ganttchart zu navigieren, um die aktuelle Bauphase zu sehen. Als Angabe, wo wir uns gerade in welchem Bauabschnitt befinden, haben wir einen Slider eingebaut, welcher über dem Ganttchart positioniert ist und von 0 bis Ende des Projekts in Tagen geht.

Wird nun der Slider bewegt, bewegt sich (skaliert) ein grüner senkrechter Strich über den Ganttchart – in den Grenzen des Projekts und die aktuelle Position (in Tagen als Integer) wird Avango übergeben (die native Funktion „*changeTimeSlider*“ wird aufgerufen). Die Objekte (Tasks) links von dem grünen Marker sind logischerweise in Avango sichtbar, die auf der rechten Seite noch nicht. Das Objekt, auf dem der Marker gerade steht, wird als Wireframe angezeigt (siehe dazu auch Kapitel 3 auf Seite 5).

Implementiert und dokumentiert ist der Timeslider (*JSlider*) in „*GanttProject.java*“ mit seinem *changeListener* (siehe Abbildung 20). Die grafische Darstellung des Slider ist in „*GanttGraphicArea.java*“ definiert.

```
private class SliderListener implements ChangeListener {
  public void stateChanged(ChangeEvent e) {
    JSlider source = (JSlider)e.getSource();
    int position = source.getValue();
    //avango.projectCosts(position);

    area.timeSliderPosition = position;
    area.repaint();
    //draw.pos();
    //cost.repaint();
    if (avango.sliderPosition!=position) {
      com.changeTimeSlider(position);
    }
  }
}
```

Abbildung 20: stateChanged()

Play-Buttons

In der 3D-Ansicht des Projektes ist es möglich, das Projekt abspielen zu lassen. Dazu wurden in Avango verschiedene Funktionen implementiert, mit denen man sich, wie bei einem Player zeitlich im Projekt bewegen kann. Da Gantt für alle zeitlichen Komponenten des Projektes verantwortlich ist und um dem Nutzer ein komfortableres Interface als die Konsolse für diese Steuerung zu geben, haben wir Play-Buttons in das GanttProject integriert. Diese Buttons steuern die Funktion des Abspielens und erschließen sich eigentlich durch das jeweilige Icon (siehe Abbildung 24 auf Seite 44). Folgende Funktionen sind möglich: vorwärts abspielen, rückwärts abspielen, Pause, Stop, an den Anfang springen, an das Ende springen, vorspulen, zurückspulen, nächster Abschnitt, vorheriger Abschnitt. Der Nutzer kann durch Klicken auf die jeweiligen Icons eine dieser Funktionen wählen.

Bei Auswahl eines Buttons wird ein String mit dem jweiligen Ereignis an die Funktion *playProject* in der Library übergeben und dort an Avango weitergegeben, wo das Projekt dann abgespielt wird.

Connection-Panel

Das ConnectionPanel (*JDialog*) wird über einen Button in der Menuleiste aufgerufen und bietet dem Nutzer die Möglichkeit, Projektname, sowie Host und Port anzugeben.

Der Projektname bestimmt das aktuelle Projekt, welches von Avango heruntergeladen wird. Da mehrere Projekte in Avango gespeichert werden können, ist diese Auswahl wichtig. Standardmäßig wird ein Defaultwert angezeigt.

Die Felder für Host und Port bestimmen den Rechner, auf dem das betreffende Avango läuft (entfernt oder Localhost), von dem die Daten geladen werden sollen. Standardmäßig stehen die Werte auf „Localhost“ und „8081“. Bei Betätigung des connect-Buttons wird die Verbindung zu dem entfernten Rechner aufgebaut und es wird automatisch der Download der Taskdaten gestartet.

Sind alle Daten übermittelt worden, verschwindet der Verbindungsdialog und die erstellten Tasks werden angezeigt. Programmiertechnisch sieht dies folgendermaßen aus, wenn man den connect-Button drückt:

```
public void actionPerformed(ActionEvent e) {
  ...
  com.setConnection(extension.host, extension.port);
  com.searchForProject(extension.projectName);
  appli.initCommunication();
  extension.timerRun();
  this.dispose();
  ...
}
```

Abbildung 21: actionPerformed()

Die beiden Methoden von dem com-Objekt legen die Parameter für die Verbindung fest – wie eben beschrieben. Mit *appli.initCommunication()* wird u. a. in *communication.java* die native Funktion *getTask()* aufgerufen. Diese erstellt dann aus der Library heraus (mit der Funktion *makeTask*, welche in *extension* die Funktion *newTask* aufruft (siehe Abbildung 19). Der Zwischenschritt wird gemacht, da alle nötigen Parameter in einem String aus der Library ankommen, der dann zuerst geparst und in einzelne Token geteilt werden muss (siehe auch Abbildung 23 auf Seite 43) – die einzelnen Tasks mit den gespeicherten Informationen aus Avango (neben den Tasks werden auch die Ressourcen generiert, siehe Ressourcen).

extension.timerRun() ruft die Handle-Funktion auf, die in Punkt „Externe Erweiterung“ auf Seite 37 behandelt wurde.

Kosten

Um dem Nutzer eine Übersicht über die Baukosten zu geben, haben wir ein zusätzliches Panel (*GraphicPanel.java*) eingebaut, das die Kosten im Projektverlauf darstellt (siehe Abbildung 24 auf Seite 44).

Die Kosten für einen Task werden normalerweise bei der Erstellung eines Tasks in Avango festgelegt. Sie können über Gantt verändert werden.

Das Kostendiagramm zeigt die Projektkosten im Verlauf der Zeit, bei Bewegung des Timesliders wird die jeweilige Position im Zeit-Kosten Diagramm angezeigt.

Verändern sich die Kosten eines Tasks, wird ein *repaint* für das GraphicPanel aufgerufen. Beim Aufruf der *repaint*-Funktion für ein Panels in Java wird die *paint*-Funktion dieses Panels aufgerufen und das Panel wird neu gezeichnet.

In der *paint*-Funktion des GraphicPanels wird zuerst die Dauer des Projektes ermittelt. Dann erfolgt ein Aufruf der Funktion *projectCosts* (in *Extension.java*) mit der Dauer des Projektes als Übergabeparameter, diese addiert die Kosten aller Tasks bis zum Ende des Projektes und gibt diese als Rückgabewert zurück.

Der Übergabeparameter wurde eingebaut, um nicht nur die Gesamtkosten ermitteln zu können, sondern auch die Kosten nach 10 Tagen, nach 50 Tagen, usw. Die Funktion *projectCosts* ermittelt nur die Kosten der Tasks bis zu dem Zeitpunkt, den man übergibt. Um die Berechnung einfach zu halten, werden die Kosten der Tasks, die bis zu diesem Zeitpunkt beendet sind, zu 100% mit in die Summe eingerechnet. Die Kosten der Tasks, die noch in Bearbeitung sind, werden zu 50% mit eingerechnet und die Kosten der Tasks, die noch nicht anfangen wurden, werden nicht mit eingerechnet.

Sind die Dauer des Projektes und die Gesamtkosten ermittelt, wird die Dauer des Projektes in 20 Schritte unterteilt. Zu jedem Zeitschritt werden die Kosten errechnet, so dass eine Zeitkurve für die Kosten entsteht.

Auch die Achsenbeschriftungen ändern sich dynamisch mit der Höhe der Kosten und der Dauer des Projektes. Auf der Kostenachse gibt es die Möglichkeiten für „Costs“, „Costs in T“ und „Costs in Mio“, je nach Höhe der Gesamtkosten. Die Zeitachse entspricht natürlich der Dauer des Projektes.

Ressourcen

Ressourcen können von ihrer Struktur mit Tasks verglichen werden: sie haben wie die Tasks eine eigene Darstellung in Gantt und in Avango und auch ihre interne Handhabung ist ähnlich der von Tasks (Speicherung in Avango, Initialisierung in Gantt). Zur Theorie und Ansätzen der Ressourcenverwaltung verweise ich auf den Abschnitt Ressourcen in Avango.

Die prinzipielle Idee war es, einen Pool von Ressourcen einzurichten, aus dem sich dann die Tasks einzelne Ressourcen abonnieren können. Um dies zu ermöglichen, haben wir in Gantt die vorhandene Ressourcenverwaltung etwas abgeändert und verwendet. Dabei werden zuerst alle Ressourcen erzeugt und dann in einem zweiten Schritt den Tasks zugewiesen. Dies läuft wie folgt ab:

Wie oben bei den Tasks beschrieben, wird die *getTask()*-Funktion aufgerufen, welche neben den Tasks auch die Ressourcen generiert. Dabei wird zuerst die *makeResource*-Funktion aus *communication.java* durch die Bibliothek aufgerufen. Diese Funktion ruft nun mit den nötigen Parametern (ID, Name, Auslastung) aus

dem verarbeiteten String (s.o.) die Methode *newResource* aus *extension.java* auf. Diese erstellt ein „*HumanResource*“-Objekt, welches in Punkt 5.2.1 auf Seite 32 erklärt wurde und unsere Ressource repräsentiert.

Neben der *makeResource*-Funktion wird gleichzeitig auch „*assignResource*“ ausgeführt, die nach dem gleichen Ablauf mit den IDs für Task und Ressource als Parameter einem Task eine Resource zuweist. Dies passiert mit „*setAssignment*“ (siehe Abbildung 22):

```
public void newResource(int id, String name, String u) {
  HumanResource r = new HumanResource(name, id, u);
  ganttProject.getHumanResourceManager().add(r);
}

public void assignResource(int taskID, int rID) {
  currentTask = getTaskByID(taskID);
  currentTask.setAssignment(ganttProject
    .getHumanResourceManager().getById(rID));
}
```

Abbildung 22: newResource()

Abbildung 23 zeigt noch einmal bildlich den Ablauf von der Entstehung eines Tasks oder einer Ressource beginnend (mit dem Aufruf der nativen *getTask*-Funktion in der Library) bis zu den neuen Funktionen *newTask* und *newResource* in unserer Erweiterung.

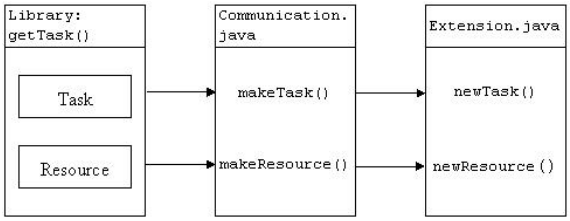

Abbildung 23: Ressource

Neben der Erstellung und Zuweisung einer Ressource, ist es außerdem möglich, die Auslastung einer Ressource anzeigen zu lassen.

Jede Ressource besitzt dabei in Avango ein Feld *Utilization* (Auslastung), welches, wie oben beschrieben, der Funktion „*newResource*“ als Parameter übergeben und in „*HumanResource*“ als Variable gespeichert wird.

Wird nun eine Ressource ausgewählt, so wird die Funktion *changeResourceView()* aus *GanttResourcePanel* aufgerufen, welche die Funktion *initHistogramms* aufruft. Diese erstellt dann ein Objekt der von uns geschriebenen Klasse *ResourcePanel*. Diese Klasse ist von *JPanel* abgeleitet und konstruiert aus dem überge-

benen String ein Auslastungshistogramm. Der String ist eine Aneinanderreihung von Auslastungen einer Ressource über die Zeit: für jeden Tag wird die Auslastung als Integer in Prozent angegeben und zwischen zwei Tagen ein Leerzeichen gesetzt. Dadurch ist es dann möglich, den String in einzelne Integerwerte zu zerlegen. Diese werden dann verwendet, um ein Balkendiagramm zu erstellen (*paintComponent*). Da ein Projekt in der Dauer stark variieren kann, wurde die Funktion *scaling()* implementiert, welche anhand von Maximalwerten der Projektdauer die Breite der Balken und der Zwischenräume anpasst, so dass ein Histogramm immer komplett angezeigt werden kann.

Kleinigkeiten

Neben den genannten wichtigen Erweiterungen haben wir natürlich auch viele Kleinigkeiten geändert, die jedoch keiner größeren Erläuterung bedürfen. Zu nennen sind da z. B. das Menu, die Möglichkeit Tasks zu markieren (Die Markierung eines Tasks führt dazu, dass das entsprechende Element in Avango auch markiert wird, indem die native Funktion *setMaskTask* aufgerufen wird, die ein entsprechendes Feld eines Objekts in Avango belegt), oder das Lösen kleinerer Bugs (Bsp.: falsche set- oder get-Methoden implementiert).

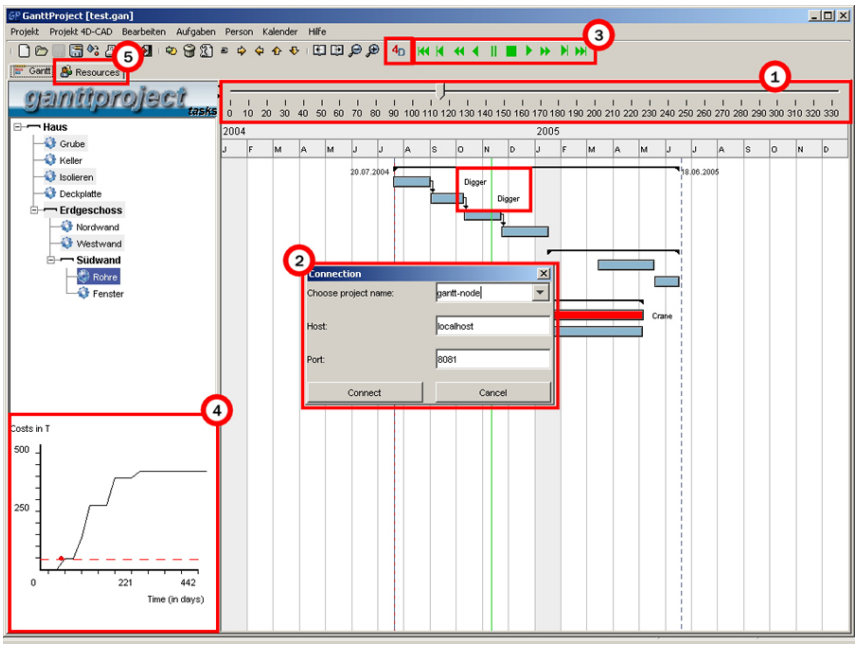

Abbildung 24: Projekt

Visuelle Erweiterung

1. Timeslider: Scroll-Mechanismus durch das Projekt. Einheit in Tagen (10-Tages-Schritte)
2. ConnectionPanel: Verbindungsaufbau. Aufruf über 4D-Button (neben Abspiel-Buttons)
3. Play-Buttons: Möglichkeit des automatischen Abspielens durch das Projekt
4. Kostendiagramm: Kosten über die Zeit. Aktueller Standpunkt gekennzeichnet
5. Tab für Ressourcendarstellung: Ressourcen-Histogramm (siehe Abbildung 25)

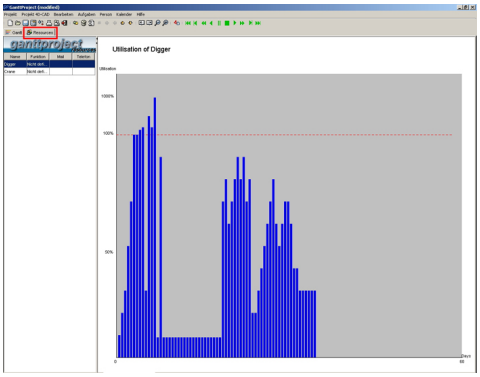

Abbildung 25: Ressourcen

6. Conclusion

Bei dem Forschungsprojekt „4d-CAD“ wurde ein hybrider Softwareverbund entwickelt, welcher die Vorteile eines VR-Frameworks wie Avango (Visualisierung in mono/stereo, multiple Ein- und Ausgabemöglichkeiten) und die eines Gantt-Charts (Zeitplanungsmanagement, Erzeugen und Darstellen von Abhängigkeiten, Ressourcenplanung) kombiniert. Mit diesem Softwaretool ist es nicht nur möglich, sich eine statische und zeitlich gebundene Vorstellung von Planungsprozessen zu machen, es ist zudem auch möglich, mit den enthaltenen Daten interaktiv zu arbeiten. Der Benutzer kann sich mit dem „4d-CAD“-System eine Vorstellung von Planungsprozessen der Zukunft machen.

Bau oder Entwicklungsprozesse werden mit Hilfe des Gantt-Charts benannt, geplant und gegebenenfalls in verschiedene Abschnitte gruppiert, die sich hierarchisch ein- und ausblenden lassen. Es besteht zudem die Möglichkeit zur Erzeugung von Abhängigkeiten. Dies bedeutet, wenn sich Teilabschnitte zum Beispiel zeitlich verschieben, werden alle davon abhängigen Komponenten entsprechend angepasst und verschoben. Neben der Planung von zeitlichen Einheiten lassen sich auch benötigte Ressourcen für einzelne Abschnitte definieren, welche sich graphisch in Diagrammen darstellen und auswerten lassen.

Parallel zu dem Gantt-Planungstool gibt es noch das Visualisierungstool Avango, welches uns den aktuellen Stand der Planung in verschiedenster Weise (mono oder stereo) mit verschiedensten Medien (Monitor, PowerWall, Cave, ...) visualisieren kann. Es lassen sich im Wesentlichen zwei Modi darstellen – ein zeitlich statischer Modus, welcher die Planung zu einem festen Zeitpunkt visualisiert und ein „Filmmodus“, bei dem man sich den modellierten Bauprozess als eine Art Film abspielen lassen kann (inklusive aller allgemein bekannten Funktionen wie Stopp, Vorwärts, Rückwärts, Suchlauf). In beiden Modi besteht die Möglichkeit, mit Hilfe von verschiedenen Eingabegeräten im Visualisierungstool zu interagieren – man kann sich so zum Beispiel durch die Szene bewegen.

Die Synchronisation dieser beiden Komponenten erfolgt selbstständig durch das Programm und in Echtzeit. Dies gibt dem Nutzer eine neue Möglichkeit, Probleme und Schwachstellen schnell und einfach zu erkennen und sie gegebenenfalls zu beheben (zum Beispiel ein Konflikt von zeitgleich benötigten Ressourcen).

Als wesentliche Neuerung neben der Kombination der zwei von Natur verschiedenen Programme Avango und Gantt-Chart muss erwähnt werden, dass man dieses „Paket“ auch verteilt nutzen kann. Es können sich mehrere Benutzer an einem Avango-Server anmelden, welcher die Objektansicht entsprechend verteilt – es war uns hierbei wichtig, dass alle Benutzer dieselbe Ansicht von dem Objekt haben, um somit besser über diese kooperativ reden zu können. Des Weiteren können sich auch mehrere Gantt-Charts anmelden und sich den Planungsprozess darstellen lassen. Unsere Implementierung lässt auch ein gleichzeitiges Arbeiten mehrerer Personen an einem Bauobjekt mit Gantt-Chart zu – dieses Feature wird aber leider nicht vom Remote User Interface (RUI) der aktuellen Avango-Version unterstützt. Wird dieser Bug in Avango beseitigt, ist jedoch ein verteiltes gleichzeitiges Arbeit sofort möglich.

Bei einem Evaluationseinsatz in einer reellen Arbeitsumgebung durch einen Bauingenieur der Bauhaus-Universität Weimar hat sich gezeigt, dass in dem GanttProject die Möglichkeiten einer komfortablen Kostenkalkulation (zum Beispiel anhand einer Tabellenkalkulation, wie in Microsoft Excel angeboten) fehlen, aber durch die Zielgruppe benötigt werden. Des Weiteren wurde der Wunsch geäußert, dass man zwischen verschieden Ansichten wählen kann. Der Gedanke hierbei geht dahin, dass verschiedene Gewerke ihre spezifische Sicht auf Bauprozesse haben. Es wäre wünschenswert, wenn man verschiedene Sachen einzeln ein- oder ausblenden

könnte, zum Beispiel Einblenden des Lüftungs-, Heizungssystem und Abwassersystems.

Bei einem in der Praxis genutzten verteilten Arbeiten an einem Bauprojekt müsste man sich auch noch Gedanken über die Frage des zeitgleichen Bearbeitens desselben Bauabschnittes machen (Stichwort: CSCW). Es wurden aktuell keine Schutzmechanismen, wie Semaphoren oder ein hierarchisches Rechtessystem implementiert. Eine Art Nutzerdatenbank wäre für eine verteilte Nutzung zusätzlich sinnvoll, welche dann eventuell die Rechte beinhaltet.

Das entwickelte „4d-CAD“-System soll als Prototyp und Researchplattform zur Darstellung der Möglichkeiten eines verteilten 4d-CAD-System dienen. Die wesentlichen Grundfunktionalitäten wurden durch das Entwicklerteam implementiert.

7. Credits

Ansprechpartner bzw. Verantwortliche für die einzelnen Teilbereiche:

Bereich	Ansprechpartner	E-Mail
Avango	B. Brombach K. Riege A. Kleppe	benjamin.brombach@uni-weimar.de kai.riege@uni-weimar.de alexander.kleppe@uni-weimar.de
Kommunikation/JNI	K. Bachmann E. Bruns	kristin.bachmann@uni-weimar.de erich.bruns@uni-weimar.de
Kommunikation/RUI	F. Coriand K. Riege	franz.coriand@uni-weimar.de kai.riege@uni-weimar.de
Gantt	E. Bruns K. Bachmann	erich.bruns@uni-weimar.de kristin.bachmann@uni-weimar.de

Des Weiteren danken wir Knut Giebel, aus der Fakultät Bauingenieurwesen der Bauhaus Universität, für das detaillreiche Modell des Supermarkt-Komplexes sowie seiner fachliche Evaluation der Anwendung im Rahmen seiner Bachelor-Arbeit.

A. Avango

Avango ist ein Framework für interaktive verteilte virtuelle Umgebunden in Echtzeit und wurde am Frauenhofer Institut für Medienkommunikation entwickelt. Avango bietet eine Vielzahl an möglichen Eingabegeräten mit unterschiedlichen Freiheitsgraden (DOF) an und ebenfalls eine große Vielzahl ein Ausgabegeräten, vom einfachen Monitor über Stereoprojektionen bis zu „Cave“-Anwendungen. Es setzt auf OpenPerformer auf und ist in C++ programmiert (also objektorientiert), bietet aber zusätzlich ein Scripting Interface in der interpretierten Sprache Scheme an, wodurch noch während der Laufzeit Modifikationen an der Szene durchgeführt werden können („Rapid Prototyping“ möglich). Die räumliche objektorientierte Beschreibung der virtuellen Welt erfolgt in Avango durch einen so genannten Szenengraph, d. h. die einzelnen Objekte mit ihren speziellen Eigenschaften (repräsentiert durch „Fields“) werden als Knoten hierarchisch in einen azyklischen Graph einsortiert. Der Szenengraph spielt auch eine tragende Rolle in Bezug auf die Verteilung. Der Graph ist allen verteilten Teilnehmern zugänglich und wird über das Netzwerk versendet und konsistent gehalten, d. h. jeder Client hat eine lokale Kopie des Szenengraphen. Veränderungen in der lokalen Kopie eines Clienten werden dann per „Updatemessage“ über das Netzwerk an alle teilnehmenden Clienten versendet, so dass sie ihre lokale Version des Szenengraphs aktualisieren können. Die Verteilung erfolgt über ein „Peer2Peer“-Netzwerk.