Politechnika Warszawska
Wydział Elektroniki i Technik Informacyjnych
Instytut Automatyki i Informatyki Stosowanej
2008

Praca dyplomowa inżynierska

Paweł Szufladowicz

Wizualizacja pracy robotów w systemie MRROC++

Promotor pracy:
mgr inż. Tomasz Winiarski

Ocena ....

....
Podpis Przewodniczącego
Komisji Egzaminu Dyplomowego

Specjalne podziękowania kieruję do Piotrka Trojanka za rady jakie udzielił mi podczas realizacji pracy dyplomowej. Dziękuję także mojemu promotorowi Tomkowi Winiarskiemu za wyrozumiałość. Mojej narzeczonej Agnieszce oraz rodzicom za mobilizowanie do wykonania niniejszej pracy dyplomowej.

Specjalność: Informatyka –
Systemy Informacyjno-Decyzyjne

Data urodzenia: 24 grudnia 1983 r.

Data rozpoczęcia studiów: 16 luty 2003 r.

. . . .
podpis studenta

Egzamin dyplomowy

Złożył egzamin dyplomowy w dn. . . . .

Z wynikiem . . . .

Ogólny wynik studiów . . . .

Dodatkowe wnioski i uwagi Komisji . . . .

. . . .

Streszczenie

W ramach pracy powstała aplikacja kliencka wizualizująca pracę systemu manipulatorów. Serwerem jest wątek procesu EDP w systemie MRROC++ udostępniający informację o stanie manipulatorów (ich obecnej konfiguracji w sensie układu członów). Transmisja między klientem a serwerem oparta jest na protokole UDP. System pozwala na symulację pracy systemu MRROC++ z wizualizacją manipulatorów bez konieczności angażowania rzeczywistych robotów. Jako platformę implementacyjno-uruchomieniową wybrano Sun JAVA co pozwala na uruchamianie aplikacja na różnych platformach systemowych (w szczególności Linux i Windows).

Słowa kluczowe: *robotyka, manipulator, wizualizacja 3D, Java, Java3D*

Abstract

Title: *Visualization of robots in the MRROC++ framework.*

The aim of this thesis was to create client application for visualization of the industrial robots. Effector driver process of the MRROC++ system acts as a server, which produces information about manipulators state (current joints positions). Communication between a client and a server is based on UDP. It is possible to simulate work of the MRROC++ system with visualization of manipulators, without need to using real robots. Implementation is based on Sun Microsystems Java environment, which allows to run application on diverse operating systems (Linux, Windows, etc.).

Key words: *robotics, manipulator, visualisation 3D, Java, Java3D*

Spis treści

1. Wstęp 1

1.1. Podstawowe zagadnienia 1

1.2. Cel pracy 2

2. Wprowadzanie do systemu MRROC++. 4

2.1. Architektura systemu MRROC++. 4

2.1.1. Warstwa pierwsza: Interfejs użytkownika. 5

2.1.2. Warstwa druga: Część zależna od zadania. 5

2.1.3. Warstwa trzecia: Część zależna od sprzętu. 6

3. Model wizualizacji obiektów 3D - opis technologii 8

3.1. 3D Studio MAX - projektowanie wirtualnego świata 9

3.2. Format pliku X3D - tekstowa reprezentacja sceny 3D 10

3.3. Java - język programowania 10

3.4. Sposób opisu wirtualnego świata w Java3D 11

3.4.1. Sposób opisu oraz implementacja grafu sceny w Java3D 12

3.4.2. Gałąź View Branch 13

3.4.3. Gałąź Content Branch 15

3.4.4. Tworzenie własnego kodu wplecionego w graf sceny 19

3.5. Biblioteka Xj3D 20

4. Realizacja zadania 21

4.1. Modele robotów w 3D Studio MAX 21

4.2. Struktura aplikacji *Virtual Robo 3D* 21

4.2.1. Interfejs Użytkownika 21

4.2.2. Moduł menadzera sceny 25

4.2.3. Moduł menadżera klientów 26

4.3. Graf sceny - model 3D wirtualnego świata robotów 29

4.3.1. Korzeń grafu sceny 29

4.3.2. Kostrukcja gałęzi treści grafu sceny. 30

4.3.3. *User-defined actions* - definiowanie zachowań w wirtualnym świecie. . . 33

4.4. Konfiguracja aplikacji. 40

4.4.1. Format pliku konfiguracyjnego. 40

4.4.2. Zawartość pliku konfiguracyjnego. 41

5. Instalacja i uruchamianie aplikacji 43
5.1. Konfiguracja środowiska 43
5.1.1. Instalacja biblioteki Java 3D 43
5.1.2. Biblioteka Xj3D 43
6. Podsumowanie 45
6.1. Perspektywy rozwoju 45
Bibliografia 47

1. Wstęp

1.1. Podstawowe zagadnienia

Robotyka (ang. robotics) jest dziedziną nauki zajmującą się wszystkimi problemami dotyczącymi mechaniki, sterowania ruchem, sensoryki, inteligencji maszynowej, projektowania, zastosowań i eksploatacji manipulatorów, robotów i maszyn kroczących. Robotyka jest stosunkowo nową dziedziną nauki, która łączy różne tradycyjne gałęzie nauk technicznych. Zrozumienie zawiłości budowy robotów i ich zastosowań wymaga znajomości zagadnień elektrycznych, mechanicznych, inżynierii przemysłowej, nauk komputerowych, ekonomii i matematyki. Nowe działy inżynierii, takie jak inżynieria wytwarzania, inżynieria zastosowań i inżynieria wiedzy, w znacznym stopniu dotyczą problemów z obszaru robotyki i szeroko pojętej automatyki przemysłowej.

Robot jest mechanicznym urządzeniem wykonującym zadania w sposób automatyczny. Działanie robota może być kontrolowane przez człowieka, przez wprowadzony wcześniej program, bądź przez zbiór ogólnych reguł, które zostają przełożone na działanie robota przy pomocy technik sztucznej inteligencji. Roboty często zastępują człowieka przy monotonnych, złożonych z powtarzających się kroków czynnościach, które mogą wykonywać znacznie szybciej od ludzi. Domeną ich zastosowań są też te zadania, które są niebezpieczne dla człowieka, na przykład związane z manipulacją szkodliwymi dla zdrowia substancjami lub przebywaniem w nieprzyjaznym środowisku. Stały rozwój technologii stosowanych w robotyce i coraz bardziej zaawansowane systemy wpływają na częstsze i coraz powszechniejsze stosowanie urządzeń robotyki w życiu codziennym człowieka - od przemysłu i fabryk, poprzez szpitale, wojsko i policję, po gospodarstwa domowe.

Pojęcia robot używamy też do nazywania autonomicznie działających urządzeń odbierających informacje z otoczenia przy pomocy sensorów i wpływających na nie przy pomocy efektorów. Roboty takie budowane są przez badaczy zajmujących się sztuczną inteligencją lub kognitywistką w celu modelowania zdolności poznawczych, sposobu myślenia lub zachowania zwierząt bądź ludzi.

Manipulator to uradzenie mechaniczne będące odpowiednikiem reki człowieka. Pozwala on na realizacje efektywnego przenoszenia, przekładania, obracania i przełączania urzadzen lub przedmiotów. Manipulator zbudowany jest najczęściej z ramion i obrotowego chwytaka. Najprostsza budowa manipulatora opiera się na wykorzystaniu serwomechanizmów (układ sterowania przemieszczeniem).

Roboty przemysłowe **IRp-6** są uniwersalnymi środkami automatyzacji procesów przemysłowych, przede wszystkim procesów uciążliwych lub trudnych do wykonania przez człowieka. Roboty mogą być stosowane do automatyzacji prac wykonywanych przez maszyny lub mogą same wykonywać pewne prace przy użyciu narzędzi, jak np.: spawanie łukowe, szlifowanie, stępianie krawędzi.

System sterowania robota jest oparty na technice komputerowej, co umożliwia robotowi wykonanie skomplikowanych funkcji, jak: układanie lub pobieranie

przedmiotów według wzoru, realizację bardzo długich programów, ruch prostoliniowy - pionowy lub poziomy, poszukiwanie przedmiotów o nieznanym z góry położeniu. Możliwe jest również dokonanie poprawek w programach, wykonywanie skoków warunkowych itp. Funkcje te umożliwiają użytkownikowi stosowanie robotów do różnych skomplikowanych zadań bez konieczności stosowania specjalnych urządzeń współpracujących z robotami.

Roboty składają się z części manipulacyjnej i oddzielonej konstrukcyjnie szafy sterowniczej. W szafie sterowniczej są umieszczone moduły układu sterowania łącznie ze sterownikami mocy silników, dzięki czemu część manipulacyjna jest nieduża i lekka, szafa sterownicza z elektronicznymi elementami układu sterowania może być umieszczona oddzielnie z dala od części manipulacyjnej, co stosuje się w przypadku pracy robota w szczególnie ciężkich warunkach otoczenia.

Robot IRp-6 jest manipulatorem o pięciu stopniach swobody. Dokładny opis manipulatora IRp-6 znajduje się na witrynie [1].

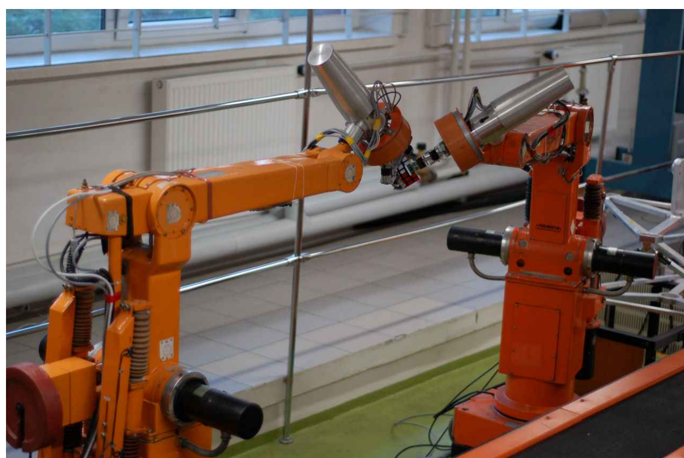
W laboratorium Zespoły Robotyki na wydziale Elektroniki i Technik Komputerowych znajdują się dwa manipulatory IRp-6. Ich budowa różni się do fabrycznych manipulatorów. Posiadają sześć stopni swobody (została dodana ruchoma kiść). Dodatkowo jeden z robotów posadowiony jest na torze jezdnym, co znacznie poszerza jego zakres ruchu.

Wizualizacja 3D jest sposobem prezentacji rzeczywistych przedmiotów za pomocą technik komputerowych. Dzięki wizualizacji 3D projektant grafiki komputerowej potrafi wiernie odwzorować rzeczywiste obiekty na ich wirtualne odpowiedniki. Projektowanie wirtualnych światów może odbywać się na wiele sposobów. Wśród podstawowych narzędzi do tworzenia skomplikowanych scen 3D można zaliczyć: bibliotekę OpenGL (Open Graphics Library) oraz interfejs Microsoft DirectX. Narzędzia służą do tworzenia grafiki dwu- i trójwymiarowej wysokiego poziomu (OpenGL) lub średniego poziomu (DirectX). Są interfejsami pomiędzy programistą (najczęściej języka C/C++) oraz sprzętem. W oparciu o wyżej wymienionych narzędzia zostały stworzone bardziej złożone aplikacje do tworzenia grafiki 3D. Wymienić tutaj można aplikację 3D Studio MAX do tworzenia skomplikowanych scen 3D lub AutoCAD służący do prezentacji wyników analityczno-cyfrowych. Modelowanie sceny i wizualizacja 3D są bardzo popularne w współczesnym programowaniu gier komputerowych.

1.2. Cel pracy

Podczas prowadzenia badań związanych z projektowaniem i eksploatacją robotów pojawia się potrzeba stworzenia aplikacji wizualizującej działanie systemu robotów. Aplikacja wiernie odwzorowująca środowisko otaczające roboty, jak i również same roboty, staje się bardzo przydatna.

Aplikacja ta pomoże studentom we wstępnym zapoznaniu się ze sterowaniem robotami. Pozwoli także na analizę niektórych problemów związanych z programowaniem robotów, bez potrzeby uruchamiania ich w rzeczywistości. Zaoszczędzi to wiele czasu i kłopotów które mogą pojawić się podczas eksploatacji rzeczywistych manipulatorów. Podczas symulacji, ewentualna kolizja robotów z otoczeniem, czy też inne nieprzewidziane zachowania robota, nie będą niosły ryzyka zranienia operatora lub uszkodzenia rzeczywistych manipulatorów. Ograniczy to liczbę



Rysunek 1.1. *Zdjęcie manipulatorów z uczelnianego laboratorium.*

awarii, i pozwoli efektywniej wykorzystać rzeczywiste stanowisko eksperymentalne.

Celem mojej pracy jest stworzenie aplikacji, która będzie wiernie wizualizować budowę, zakres ruchów i zachowanie robotów - manipulatorów IRp-6, wraz z taśmociągiem, składających się na stanowisko badawcze w laboratorium robotyki IAiIS PW. Jeden z manipulatorów posadowiony jest na torze jezdnym, drugi przytwierdzony jest do nieruchomego postumentu.

Rysunek 1.1 przedstawia wizerunek rzeczywistych robotów z uczelnianego laboratorium.

2. Wprowadzanie do systemu MRROC++.

System **Multi-Robot Research-Oriented Controler** (MRROC++) jest systemem wspomagającym tworzenie i rozwój aplikacji sterującej robotami. Został napisany na platformie systemu czasu rzeczywistego QNX. Jest systemem uniwersalnym. Przy projektowaniu systemu jego twórcy kierowali się następującymi wymaganiami:

1. Programowanie robotów powinno być off-line, tzn. kod wynikowy jest napisany przez programistę i w całości wczytany przez robota.
2. Sterownik robota powinien umożliwiać przeprowadzanie różnorodnych badań.
3. Powinien mieć strukturą otwartą, umożliwiającą modyfikacje i zmiany poszczególnych jego części.
4. System powinien umożliwić dołączanie czujników różnych typów oraz sporządzenie dokładniej dokumentacji technicznej programu.
5. Zmiany w niektórych częściach sterownika nie powinny powodować zmian w innych częściach systemu.

W efekcie powstał system o strukturze ramowej (*framework*) napisany w języku zorientowanym obiektowo C++, którego głównymi cechami są: enkapsulacja danych, modularność oraz dziedziczenie. Umożliwia on twórcom oraz użytkownikom osiągnięcie pożądanych celów w programowaniu sterowników dla wszystkich dostępnych robotów w laboratorium.

W dalsze części rozdziału zostaną omówione najważniejsze moduły sterownika MRROC++. Cały rozdział został napisany w oparciu o raport [2].

2.1. Architektura systemu MRROC++.

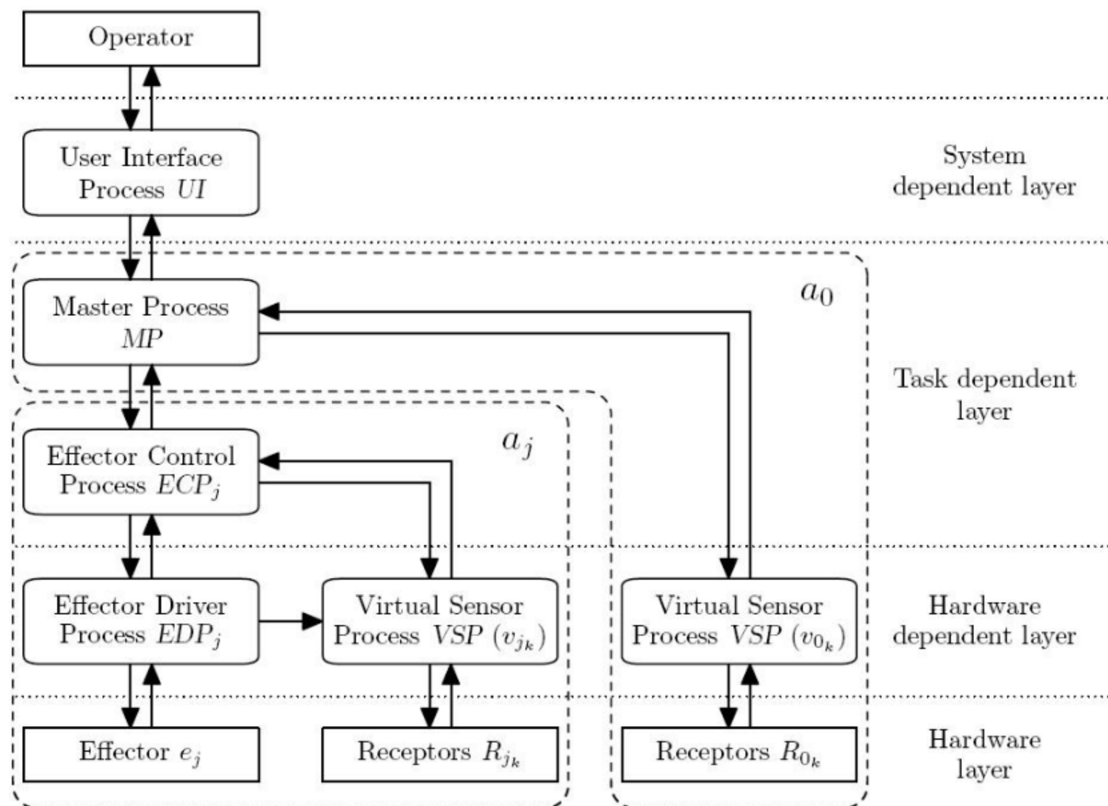
Sterownik MRROC++ ma hierarchiczną strukturę funkcjonalną. Poszczególne funkcje sterownika realizowane są w postaci odrębnych procesów działających w węzłach sieci lokalnej w uczelnianym laboratorium pod systemem operacyjnym QNX.

W strukturze systemu MRROC++ można wyróżnić trzy warstwy. Każda z nich zawiera zbiór kilku procesów o zróżnicowanej funkcjonalności. Rysunek 2.1 przedstawia szczegółową budowę sterownika z podziałem na warstwy.

Warstwy sterownika można podzielić na:

— Warstwa zależna od systemu operacyjnego, której kod nie jest zmieniany przez programistę - Interfejs użytkownika.
— Warstwa zależna od zadania - kod pisany przez programistę robota
— Warstwa zależna od sprzętu, zawiera głównie sterowniki robotów

W dalsze części rozdziału, zostaną omówione z podziałem na warstwy poszczególne procesy wchodzące w skład systemu MRROC++.



Rysunek 2.1. *Ogólny schemat działania systemu MRROC++.*

2.1.1. Warstwa pierwsza: Interfejs użytkownika.

Interfejs użytkownika jest warstwą komunikacji użytkownika ze sterownikiem robota. Składa się z dwóch wątków. Pierwszym jest wątek obsługujący polecenia użytkownika - *User Interface* (UI). Drugim jest wątek odpowiadający za wyświetlanie na ekranie stanu systemu sterującego - *System Response Process* (SRP).

Wątki UI i SRP mają zamkniętą strukturę i nie podlegają żadnym modyfikacją, ani rekompilacji przez programistę. Jedynym ich zadaniem jest realizacja komunikacji między użytkownikiem, a systemem za pośrednictwem okienkowego interfejsu.

Proces UI inicjuje i kończy działanie sterownika oraz obsługuje polecenia operatora. Moduł SRP służy do pobierania i formatowania informacji znajdujących się wewnątrz działającego systemu, które prezentuje w przyjazny sposób użytkownikowi.

2.1.2. Warstwa druga: Część zależna od zadania.

W skład warstwy drugiej wchodzą dwa rodzaje procesów:

— Master Proces (MP) - koordynuje pracę całego systemu.
— Effector Control Process (ECP) - koordynuje pracę robota, do którego przynależy.

Każdy z nich zbudowany jest z części stałej (powłoki) oraz części zależnej od zadania (jądra). Na ogół części stałe służą do komunikacji z innymi procesami w systemie, natomiast części zmienne wykonują konkretne zadanie.

Rola procesu MP.

Rola części stałej (powłoki)- nie zależnej od zadania:

- kontakt z operatorem (nasłuch jego poleceń),
- powoływanie, a po zakończeniu pracy likwidacja procesów ECP i VSP,
- utworzenie łączy komunikacyjnych z innymi procesami,
- obsługa zgłoszonych wyjątków, czyli reakcja na zaistniałe błędy,
- sygnalizacja operatorowi sytuacji awaryjnych (przekazanie SRP informacji o
- zaistniałej sytuacji),
- realizacja programu użytkowego zawartego w jądrze dostarczanym przez programistę.

Rolą jądra procesu MP jest:

- deklaracje obiektów reprezentujących robota (efektor), sensor (receptor), generator oraz condition, które będą używane w programie użytkowym,
- realizacja programu użytkowego,

Rola procesu ECP.

Rola części stałej (powłoki)- nie zależnej od zadania:

- kontakt z procesem MP (nasłuch jego poleceń),
- powoływanie, a po zakończeniu pracy likwidacja procesów VSP współdziałających z tym procesem ECP,
- utworzenie łączy komunikacyjnych z innymi procesami,
- obsługa zgłoszonych wyjątków, czyli reakcja na zaistniałe błędy,
- sygnalizacja procesowi MP sytuacji awaryjnych, a ponadto przekazanie procesowi SRP informacji o zaistniałej sytuacji,
- realizacja programu użytkowego zawartego w jądrze dostarczanym przez programistę.

Jądro procesu ECP odgrywa taką samą rolę jak jądro procesu MP, czyli realizuje kod programu napisanego przez programistę.

2.1.3. Warstwa trzecia: Część zależna od sprzętu.

Procesy *Effector Driver Process* (EDP) oraz *Virtual Sensor Process* (VSP) stanowią warstwę trzecią w strukturze każdego sterownika MRROC++. Komunikują się ze sterownikami robota (proces EDP) oraz z rzeczywistymi receptorami (proces VSP) zamocowanymi na efektorach robota.

Główną rolą procesu EDP jest interpretowanie poleceń przesłanych z procesów ECP i UI. Do jego zadań należą:

- zdekodowanie rozkazu i sprawdzenie jego poprawności,
- wykonanie adekwatnych operacji na parametrach zlecenia (np. przeliczenie współrzędnych),

— wykonanie rozkazu (np.: zmiana definicji narzędzia, odczytanie wejść binarnych, zapisanie wyjść binarnych, odczyt aktualnego położenia ramienia)
— zlecenie wykonania ruchu, jeżeli przysłano rozkaz ruchu (SET ARM), odczytu położenia (GET ARM) lub synchronizacji (SYNCHRO),
— uformowanie przesyłki zwrotnej dla zleceniodawcy (ECP lub UI),
— wysłanie do ECP przesyłki zwrotnej w odpowiedzi na zlecenie QUERY.

Proces VSP agreguje dane odczytane z rzeczywistych czujników. Zagregowane dane, czyli odczyty z wirtualnych czujników są przesyłane do procesu ECP. Odczyt może być wymuszony przez ECP lub MP, lub poprzez realizację okresowego odczytu z czujników rzeczywistych. Za każdym razem dane docierają do procesów ECP lub MP za pośrednictwem instrukcji wykonywanych przez owe procesy. Czujniki wirtualne wskutek zastosowania odpowiednich metod żądają nadesłania nowych danych oraz odbierają przesyłki, a następnie przechowują otrzymany odczyt i udostępniają go instrukcjom ruchowym.

3. Model wizualizacji obiektów 3D - opis technologii

Istnieje co najmniej kilka narzędzi i technologii wspomagających tworzenie aplikacji 3D. Nie wszystkie są darmowe. Są też takie, które nie pasują do realizacji zadania postawionego w temacie pracy dyplomowej. Z pośród dostępnych narzędzi i technologii rozważyłem kilka pozycji i wybrałem zarówno najbardziej przyjazne programiście jak i również najbardziej popularne. Proces tworzenia aplikacji wymagał realizacji kilku punktów:

1. Wizualizacja robotów - aplikacja powinna być napisana dla platformy QNX, lecz nie jest to kryterium krytyczne i można wykorzystać inny system operacyjny.
2. Wybrór narzędza do tworzenia skomplikowanych scen 3D oraz projekt wirtualnego modelu rzeczywistych robotów z laboratorium.
3. Decyzja w jakim formacie zostanie zapisana scena. Format musi udostępniać zapis kluczowych elementów sceny.
4. Decyzja w jakim języku zostanie napisana aplikacja, którą będzie można prosto połączyć z systemem MRROC++ napisanym w języku C++.

Wstępne plany zakładały, że aplikacja wizualizująca pracę robotów zostanie napisana w języku C++, ponieważ system MRROC++ jest napisany właśnie w tym języku. Jednak w trakcje instalowania niezbędnych bibliotek 3D na systemie operacyjnym QNX, okazało się, że wsparcie przez sterowniki graficzne po stronie systemu QNX jest niewystarczające. Udało mi się zainstalować bibliotekę OpenGL w wersji dla systemu operacyjnego Linux, jednak sterowniki do kart graficznych w QNX nie są tak wydajne jak w Linux-ie. Istniało duże prawdopodobieństwo, że gotowa aplikacja będzie bardzo przeciążać procesor i wizualizacja robotów nie będzie wystarczająco wydajna.

Inną trudnością był format zapisu sceny do pliku. Istnieje wiele formatów, ale nie każda technologia udostępnia wczytywanie każdego z nich. Program napisany w języku C++ przy użyciu biblioteki OpenGL wymagał biblioteki, która wczytałaby do pamięci sformatowany plik i na jego podstawie tworzyła graf sceny. Niestety nie znalazłem takiej biblioteki, która mogłaby pogodzić wszystkie moje wymagania. Z pośród rozważanych dwóch bibliotek: *lib3D3* oraz *libASE*, które wczytują odpowiednio: pliki .3ds oraz .ase, żadna nie okazała się dobrym rozwiązaniem. Głównym problemem były: brak szczegółowej dokumentacji owych bibliotek oraz brak ich rozwoju. Ostatnie wprowadzone poprawki były w latach 2004 i 2005.

W związku z powyższym stwierdzeniem, język C++ nie sprosta moim wymaganiom i uwagę skupiłem na technologii Java. Jak wiadomo owy jeżyk jest uniwersalny. Aplikacje napisane w tym języku moga być uruchamiane na wielu platformach. Stanowi to istotny atut tej technologi.

W poszukiwaniu rozwiązania natknąłem się na ciekawy format zapisu sceny 3D - *Virtual Reality Modeling Language* (VRML). Dowiedziałem się również, że nie

jest już rozwijany, ale zastąpił go format *eXtensible 3D Graphics* (X3D), stworzony w celu prezentacji scen 3D przez przeglądarki internetowe. Jak wiadomo Java udostępnia interfejsy do tworzenia apletów, czyli aplikacji, które są uruchamianie i obsługiwane przez przeglądarki www takie jak: Internet Exploler, Opera, Mozilla Fire Fox itp.

Java zawiera standard Java3D, oferujący obsługę grafiki trójwymiarowej oraz budowanie wirtualnych światów 3D. Rozszerzenie dostarcza bibliotekę opartą o sterowniki OpenGL, a także interfejs programistyczny umożliwiający dostęp do funkcji i obiektów graficznych na wysokim poziomie abstrakcji.

Problem połączenia Java3D z ładowaniem formatu X3D przez aplikację rozwiązuje darmowa biblioteka Xj3D stworzona w celu wczytywania do aplikacji 3D formatów VRML lub X3D. Jest ciągle rozwijana i posiada dość bogatą dokumentację, co znacznie ułatwia zapoznanie się z projektem i wykorzystanie go przy tworzeniu własnych wizualizacji.

Format X3D jest standardem ISO dla grafiki trójwymiarowej w czasie rzeczywistym, następcą Virtual Reality Modeling Language (VRML). Wprowadza pewne rozszerzenia do VRML i jest ciągle rozwijany. Zdolność do kodowania sceny w składni XML jest jego dodatkowym atutem.

Aplikacja napisana w języku Java z wykorzystaniem biblioteki Java3D oraz Xj3D jest najlepszym rozwiązaniem. Wystarczy dołączyć odpowiednie biblioteki do Wirtualnej Maszyny Javy (JVM). Daje to dodatkowe korzyści użytkownikom mojej aplikacji. Każdy mógłby bez trudu korzystać z programu w przeglądarce internetowej bez jakich kolwiek utrudnień związanych z instalacją dodatkowego oprogramowania. Jedyne co jest wymagane, to przeglądarka internetowa, odpowiednie biblioteki i maszyna wirtualna Java; nie trzeba kompilować kodu źródłowego.

3.1. 3D Studio MAX - projektowanie wirtualnego świata

Zadanie jakie wybrałem na temat pracy dyplomowej, wymagało ode mnie zapoznania się z dostępnymi technologiami do tworzenia aplikacji 3D. Podstawą było stworzenie wirtualnego świata - modeli robotów i ich otoczenia. Jednym z dostępnych narzędzi do tworzenia trójwymiarowych modeli jest bez wątpienia 3D Studio MAX, zwane potocznie 3DS.

3DS [3] jest jednym z najbardziej zaawansowanych i najpopularniejszych programów do obróbki i tworzenia grafiki trójwymiarowej dostępnych na rynku. Znajduje on zastosowanie w najróżniejszych przedsięwzięciach artystycznych i komercyjnych. Miedzy innymi w architekturze, grach komputerowych, produkcjach filmowych, publikacjach internetowych, wizualizacjach medycznych i naukowych, projektowaniu rzeczywistości wirtualnej oraz sztukach pięknych.

Innym z dostępnych aplikacji do tworzenia wirtualnych światów jest między innymi Lightwave3D . Jednak 3DS ma swoje zalety, które ułatwiły mi napisanie pracy. Najważniejszą cechą 3DS, która zadecydowała o moim wyborze jest przyjazność interfejsu użytkownika, a także łatwość tworzenia zaawansowanych obiektów 3D. Inną nie mniej ważną zaletą, która przesądziła o moim wyborze jest możliwość eksportu przez 3DS wirtualnego świata do formatu (VRML/X3D).

3.2. Format pliku X3D - tekstowa reprezentacja sceny 3D

Standard X3D jest następcą modelu *Virtual Reality Modeling Language*(VRML). Opisuje grafikę trójwymiarową lub interaktywną grafikę wektorową, projektowane głównie z myślą o stronach WWW. W plikach .x3d zapisane są szczegółowe informacje na temat budowy całej sceny.

X3D jest językiem programowania, dającym wiele możliwość:

— za jego pomocą programista może budować nowe obiekty i indywidualne profile.
— został stworzony z myślą wykorzystania go w aplikacjach od superkomputerów do telefonów komórkowych.
— umożliwia prezentację w czasie rzeczywistym wysokiej jakości rozwiązań graficznych.
— interaktywny, pozwala łączyć w całość pliki audio, wideo oraz dane 3D.
— jest łatwy do opracowania - możliwość eksportu do tego formatu w wielu popularnych aplikacjach do tworzenia scen 3D (w tym 3DS).
— darmowy dostęp do specyfikacji.

Informacje dotyczące sceny 3D zawierają nie tylko dane dotyczące współrzędnych obiektów znajdujących się w wirtualnym świecie. Wierzchołki i krawędzie brył mogą być opisane dodatkowymi cechami takimi jak kolor powierzchni, jej przezroczystość, odblaskowość lub mapowanymi na nie teksturami. Zawarte są tu także informacje dotyczące oświetlenia, tła, kamery czy obiektów służących do nawigowania widokiem na scenie przez użytkowników.

Podstawową cechą charakteryzującą język X3D jest to, że jednym z celów postawionych przez jego twórców jest minimalizacja ilości informacji przesyłanych przez łącza sieciowe. Odpowiednie komendy pozwalają przeglądarkom internetowym na wygenerowanie modeli 3D, które mogą być oglądane pod dowolnym kątem i dowolnej odległości.

X3D udostępnia mechanizm SAI (Scene Accescc Interfajs). Daje on programiście następujące możliwości dostępu do sceny:

— dostęp do opcji funkcjonalnych przeglądarek witryn internetowych
— rejestruje i powiadamia o akcjach podejmowanych przez przeglądarkę www (błędne adresy, start, zamknięcie sesji itp.)
— czyta ostatnie wartości pól węzłów wewnątrz sceny
— rejestruje zmiany wartości pól węzłów wewnątrz sceny

Więcej informacji o standardzie X3D znajduje się na stronie www [4].

3.3. Java - język programowania

Język Java powstał w latach dziewięćdziesiątych ubiegłego wieku. Po kilku latach zdobył ogromną popularność. Programy napisane w Javie są niezależne od architektury. Dzieje się tak, ponieważ programy są kompilowane do kodu pośredniego, który jest niezależny od systemu operacyjnego. Powstały kod jest wykonywany przez maszynę wirtualną, która tłumaczy go na format dostosowany do konkretnej platformy.

Java jest silnie ukierunkowana obiektowo. Jej podstawowe koncepcje zostały przyjęte z języka C++. Obiektowość pozwala na sprawniejsze manipulowanie składnikami wirtualnego świata, które także są obiektami. Obiekty Javy i obiekty 3D łączy kilka cech. Przyjmują one określone stany i posiadają określone zachowania, które mogą zmieniać te stany (np. zmiana położenia lub kształtu obiektu 3D odzwierciedlana jest zmianą stanu obiektu w Javie).

Biblioteki Javy udostępniają wyspecjalizowane funkcje umożliwiające programowanie rozproszone, także między aplikacjami napisanymi w innych językach. Ta koncepcja również została wykorzystana przeze mnie do zbudowania interfejsu sieciowego, który umożliwi komunikację między systemem MRROC++ a moją aplikacją.

3.4. Sposób opisu wirtualnego świata w Java3D

Niniejsza sekcja została napisana w oparciu o specyfikację biblioteki Java3D [5].

API Java3D zostało zaprojektowane w celu tworzenia trójwymiarowych aplikacji graficznych oraz apletów. Umożliwia wysokopoziomowe tworzenie i przetwarzanie trójwymiarowej sceny oraz zawartych w niej obiektów. Dzięki efektywnemu wsparciu renderowania i modułom dostarczonym przez jego twórców, programiści mogą opisywać wielkie i złożone wirtualne światy.

Swoją szybkość Java3D czerpie z faktu, iż została zbudowana ponad interfejsami OpenGL lub Direct3D i korzysta z wszystkich usprawnień jakie mają do dyspozycji te interfejsy, czyli jest to przede wszystkim dostęp do akceleratorów graficznych. Tak więc programując w Java3D, Java zajmuje się jedynie reprezentacją tego co znajduje się na scenie, natomiast cały rendering odbywa się już na poziomie kodu napisane dla danej platformy. Takie rozwiązanie zapewnia nie tylko przenośność aplikacji na wiele systemów, ale również wysoką wydajność.

Innym mechanizmem zwiększającym wydajność tego interfejsu jest wielowątkowość. Obecnie implementacja Java3D jest w pełni wielowątkowa, co oznacza, że w dowolnym punkcie czasu zadania mogą równolegle wykonywać różne czynności takie jak: wykrywanie widoczności obiektów (jeśli obiektu nie widać nie należy go rysować), rendering, obsługa zachowania obiektów (czyli ich ruchu), obsługa dźwięku, urządzeń sterujących (klawiatura, mysz, joystick), wykrywanie kolizji modeli itd.

Java3D jest bardzo funkcjonalnym interfejsem i ma duże możliwości, spośród których można wymienić jeszcze:

- Sterowanie poziomem szczegółowości modeli (modele znajdujące się w głębi sceny nie muszą być tak dokładnie renderowane jak te na pierwszym planie)
- Mgła (używana nie tylko po to aby zwiększyć realizm sceny, ale również aby zasłonić dalsze elementy sceny i tym sposobem przyspieszyć rendering)
- Java3D łączy grafikę 3D z przestrzennym dźwiękiem, można określić typ głośników, ich odległość od słuchacza, symulować efekt Dopplera itd.
- Stereowizja umożliwia współpracę z hełmami wirtualnej rzeczywistości
- Obsługiwane są najczęściej spotykane typy oświetlenia i cieni
- Kompresja modeli (redukcja ilości pamięci używanej przez Java3D i szybsze przesyłanie geometrii między rozproszonymi w sieci aplikacjami Javy)

— Morfing geometrii (tzn. zmiana kształtu modeli)
— Interpolacja położenia, koloru, przeźroczystości, obrotu, skali itd.
— Materiały modeli, filtrowanie tekstur...

Podobnie jak inne środowiska graficzne, Java3D pozwala na tworzenie grafiki trójwymiarowej w oparciu o hierarchiczną strukturę grafu sceny, w którego węzłach znajdować się będą wszystkie elementy wirtualnego świata (tj. geometria obiektów, przekształcenia, kamery, oświetlenia, animatory itd.) konieczne do właściwej reprezentacji projektowanej sceny.

3.4.1. Sposób opisu oraz implementacja grafu sceny w Java3D

Jak już zostało wspomniane, obiekty tworzące wirtualną rzeczywistość zorganizowane są hierarchicznie, co oznacza, że są ze sobą powiązane w strukturę drzewiastą co ściśle określa przeznaczenie każdego obiektu.

Graf sceny składa się z węzłów macierzystych (rodziców), węzłów pochodnych (dzieci) oraz obiektów przechowujących informacje dotyczące wyglądu brył w wirtualnej przestrzeni, jak również obiektów odpowiedzialnych za komunikację z urządzeniami wejścia/wyjścia.

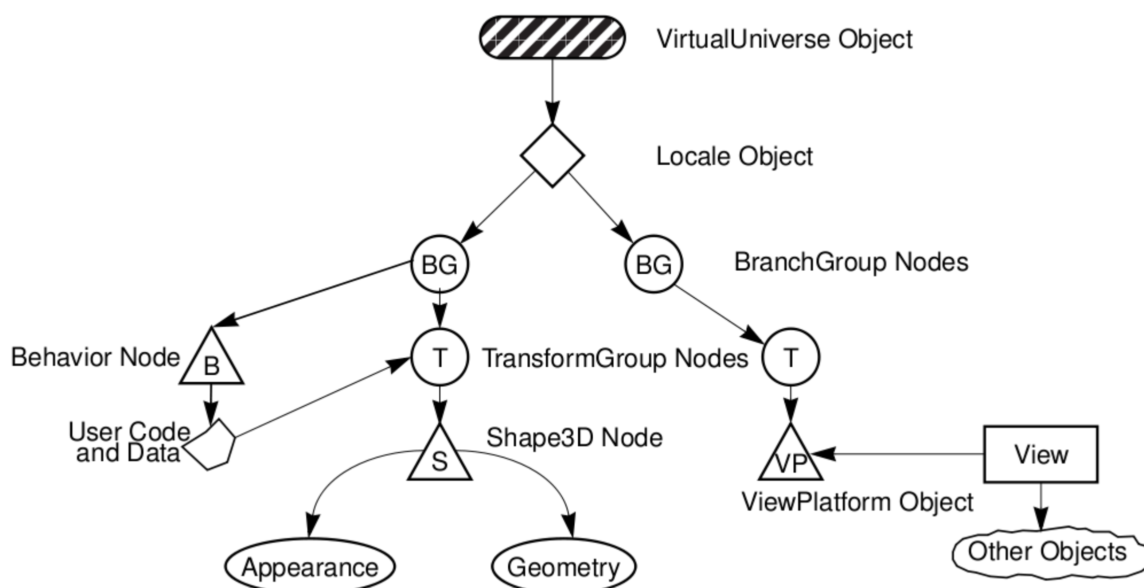
Węzły macierzyste reprezentują grupy węzłów. Organizują i czasami kontrolują to, w jaki sposób Java3D ma interpretować węzły potomne. W niektórych przypadkach węzeł rodzic może pełnić rolę łącznika kliku podgrafów w jeden, większy graf. Natomiast węzły dzieci mogą stanowić kolejną podgrupę w grafie, lub liść.

Liście z punktu widzenia grafu sceny nie są węzłami, ponieważ nie posiadają potomków. Zawierają informacje na temat wyglądu obiektów 3D i/lub referencję do innych obiektów. Do wyglądu zaliczamy m.in. kształt, zbiór punktów reprezentujących bryłę, widzialność, kolor lub teksturę mapowaną na bryłę lub płaszczyznę.

Budowanie i manipulacja grafem sceny są możliwe po skonstruowaniu potrzebnych obiektów Javy 3D i odpowiedniemu powiązaniu ich ze sobą. Program tylko raz buduje graf sceny. Raz zbudowana scena może być wielokrotnie przekształcana. Programista może manipulować albo pojedynczym obiektem, albo wszystkimi obiektami (wraz z nim) znajdującymi się pod danym obiektem w grafie sceny. Typowy schemat połączeń został przedstawiony na rysunku 3.1.

W korzeniu drzewa znajduje się obiekt *VirtualUniverse*, jest to obiekt, który stanowi kontener dla wszystkich grafów sceny, aplikacja lub aplet może posiadać wiele takich obiektów jak również jeden obiekt *VirtualUniverse* może być współdzielony pomiędzy wieloma aplikacjami lub apletami, niemniej jednak obiekty znajdujące się w drzewie należącym do jednego obiektu *VirtualUniverse* nie mogą należeć jednocześnie do innego. *VirtualUniverse* składa się ze zbioru obiektów *Locale*.

Obiekt *Locale* stanowi kolekcję obiektów *BranchGroup*, które stanowią korzeń podgrafu, wszystkie obiekty należące do *Locale* używają współrzędnych względem położenia globalnego. Klasa *BranchGroup* może być rozpatrywana jako jednostka podlegająca kompilacji. Obiekty *TransformGroup* mają dokonywać transformacji na modelach sceny przy czym grupa modeli może mieć wspólną transformację, a każdy model grupy może mieć jeszcze dodatkowe transformacje. Transformacje mogą być sterowane przez obiekty dziedziczące po obiekcie *Behavior* co umożliwia interaktywne sterowanie modelem przy pomocy klawiatury, myszy itp.



Rysunek 3.1. *Podstawowa struktura grafu sceny.*

Graf sceny dzieli się na dwie, podległe obiektowi *Locale* gałęzie: *View Brach* oraz *Content Branch*. Rysunek 3.2 przedstawia podstawowy podział na podgrafy w grafie sceny.

W dalszej części rozdziału zostaną omówione składowe obu najważniejszych gałęzi w grafie sceny.

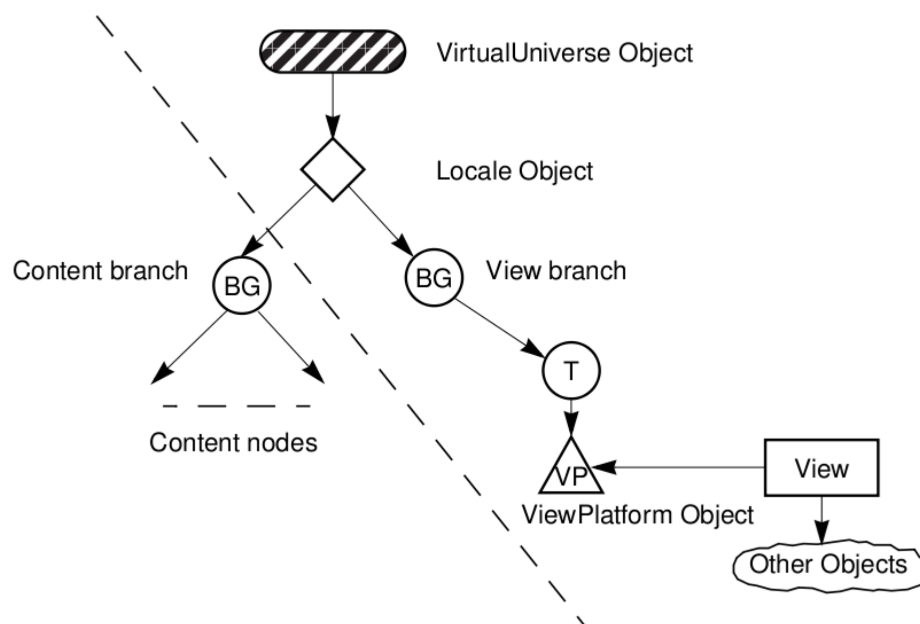
3.4.2. Gałąź View Branch

Gałąź Viem Branch (gałąź widoku) zajmuje ważne miejsce w hierarchicznym grafie sceny. Jej zadaniem jest przechowywanie informacji jak użytkownik końcowy widzi scenę 3D.

Aplikacje lub aplety napisane przy użyciu modelu widoku Java3D mogą być renderowane na wielu zróżnicowanych urządzeniach. Widok osiąga tę wszechstronność dzięki rozdzieleniu świata wirtualnego z fizycznym. Model odróżnia w jaki sposób aplikacja tworzy i kontroluje punkt widzenia w wirtualnym świecie od obliczeń potrzebnych do renderowania sceny dla końcowego użytkownika na określonej platformie systemowej.

Programista umieszcza i łączy w relacje wirtualne obiekty. Zadaniem Javy 3D jest renderowanie sceny, która jest widziana na ekranie monitora i modyfikowana (manipulowana) przez użytkownika końcowego. Technologia dostarcza bogaty interfejs łączący fizyczny świat z wirtualnym. Interfejs pomiędzy dwoma światami definiuje pojedynczą, wspólną przestrzeń, w której efekty oddziaływania użytkownika końcowego na przedmioty z wirtualnego świata widoczne są na ekranie monitora.

Obiekt *ViewPlatform* (punkt widzenia), inny dla każdej platformy systemowej, za pośrednictwem obiektów *BranchGroup* oraz *TransformGrour* (zostaną one omówione w kolejnej sekcji) jest przyłączony do obiektu *Locale*. Definiuje sposób w jaki



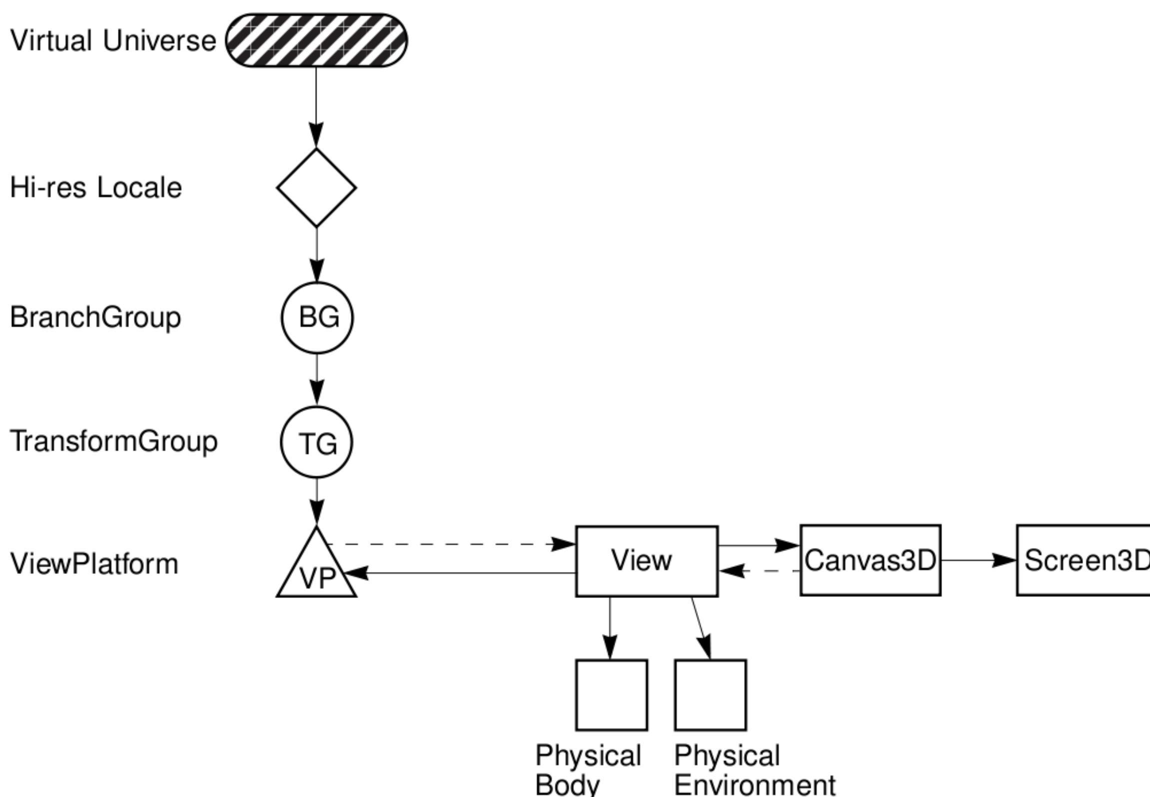
Rysunek 3.2. *Superstrutkura grafu sceny wraz z podstawowym podziałem na gałęzie treści i widoku.*

scena jest widziana oczami użytkownika końcowego. Dzięki niemu poprzez lokalizację, orientację i skalowanie punktu widzenia możliwe jest poruszanie się wewnątrz sceny. Za każdym razem kiedy zmienimy punkt widzenia w scenie, zmieniane jest położenie obiektu *ViewPlatform* i/lub kąt, pod jakim oglądamy jej zawartość. W efekcie zmienia się wynik renderowania, który widać na na monitorze.

ViewPlatform jest liściem w hierarchii grafu. Ten obiekt kontroluje położenie, orientację i skalę obserwatora. Chcąc zmienić wartość któregoś z tych parametrów należy odpowiednio ustawić transformację (obiekt TransformGroup znajdujący się nad obiektem ViewPlatform w strukturze grafu sceny). Do obiektu ViewPlatform dołączony jest obiekt *View* koordynujący wszystkie aspekty związane z procesem renderowana. Zawiera wszystkie parametry albo referuje do obiektów, które zawierają parametry, niezbędne do renderowania sceny wynikowej. Konfigurację graficzną i inne informacje potrzebne do renderowania okna wynikowego przechowuje obiekt typu *Canvas3D*. Obiekt *View* zawiera listę obiektów *Canvas3D*, do których widok jest renderowany, ten obiekt zawiera również referencje do obiektów *PhysicalBody* i *PhysicalEnvironment*. Obiekt *PhysicalBody* zawiera specyfikację głowy użytkownika, tzn. pozycje oczy i uszu, natomiast obiekt *PhysicalEnvironment* zawiera specyfikację środowiska, czyli urządzeń sterujących oraz urządzenia generującego dźwięki (karta dźwiękowa).

Canvas3D jest używany albo do natychmiastowego renderowania sceny na ekranie monitora (on-screen rendering), albo wykonuje niezbędne obliczenia do renderowania w pamięci (off-screen rendering). Wszystkie obiekty *Canvas3D* przechowują referencję do obiektów *Screen3D*, które odpowiadają m.in. za rozmiar pixeli w fizycznych wielkościach. Dzięki relacji obiektów *Canvas3D* oraz *Screen3D* Java3D potrafi przeliczać wielkości pixeli do fizycznych rozmiarów, np. metrów. *Canvas3D*

potrafi określić pozycję i orientację przedmiotów w rzeczywistym świecie. Dokładny schemat gałęzi widoku przedstawia Rysunek 3.3.



Rysunek 3.3. *Gałąź sceny grafu zawierająca model widoku.*

Java 3D automatycznie i ciągle przekazuje informacje o obiektach, które pojawiły się w polu widzenia do *Canvas3D*. Proces przepływu danych między obiektami opiszę w skrócie na przykładzie:

Przykład 3.1: Nawigując np. myszą, zmieniamy punkt widzenia w scenie 3D. Obiekt *ViewPlatform* przekazuje odpowiednie informacje do obiektu *View*. Obiekt *View* interpretuje je w następujący sposób: Zmienił się punkt widzenia i nowe, widzialne części sceny powinny zostać ponownie poddane procesowi renderowania i przedstawione użytkownikowi w odpowiedniej postaci. Ważne informacje docierają do obiektu *Canvas3D*. Po wykonaniu odpowiednich procesów przez *Canvas3* obiekt *View* odbiera dane wynikowe, które za pośrednictwem obiektów *PhysicalBody* i *PhysicalEnvironment* prezentowane są końcowemu użytkownikowi.

3.4.3. Gałąź Content Branch

Graf sceny składa się z kolekcji węzłów połączonych w strukturę drzewiastą. Opisywana wcześniej cześć drzewa (gałąź widoku) odpowiadała za wyświetlanie obiektów znajdujących się w scenie na ekranie wynikowym. W niniejszej sekcji zostanie przedstawiona gałąź odpowiadająca za przechowywanie informacji o budowie obiektów sceny i relacjach zachodzących między nimi.

Wszystkie węzły należące do modelu grafu sceny dziedziczą po obiekcie *Node*. Elementy drzewa są węzłami, które są reprezentantami dwóch, ukierunkowanych rodzai węzłów: węzły grupujące i liście. Klasyfikujemy je również w relacji rodzic dziecko, gdzie rodzic może mieć wiele dzieci, a dziecko może przynależeć tylko do jednego rodzica - graf sceny jest acykliczny.

Z pośród węzłów grupujących najważniejsze są obiekty: *BranchGroup*, *TransformGroup*, *Link* oraz *SharedGroup*.

Obiekt BranchGroup

Obiekt *BranchGroup* jest korzeniem każdego podgrafu. Każdy podgraf może osiągać niezależnie dwa stany: może zostać skompilowany lub „żywy". Obiekt *BrachGroup* charakteryzuje się następującymi cechami:

— Każdy obiekt *BranchGroup* może zostać skompilowany metodą *compile*. Powoduje to, że każdy podgraf zakorzeniony w *BranchGroup* także zostanie skompilowany.
— Każdy obiekt *BranchGroup* może zostać przyłączony do obiektu wirtualnego świata co można uzyskać dzięki przymocowaniu go do obiektu *Locale*. Cały podgraf staje się „żywy".
— Każdy obiekt *BranchGroup*, który jest zawarty w innym podgrafie może zostać odłączony od grafu sceny w czasie rzeczywistym - podczas działania aplikacji. W czasie rzeczywistym graf sceny może rozbudowany lub zredukowany.

Różnica między określeniem żywy od skompilowanego polega na tym, że żywe podgrafy zostają poddawane procesowi renderowania i tylko ich zawartość zostaje prezentowana użytkownikowi końcowemu.

Jeżeli obiekt *BranchGroup* jest włączony do grafu, jako dziecko innego węzła grupującego, to nie może zostać powiązany z obiektem *Locale*.

Obiekt TransformGroup

Obiekt *TransformGroup* wyszczególnia pojedynczą, przestrzenną transformację za pośrednictwem obiektów *Transform3D*. Możliwa jest zmiana pozycji, orientacji oraz skalowania wszystkich obiektów znajdujących się pod obiektem *TransformGroup* w hierarchicznym grafie sceny.

Przekształcenie musi być afiniczne, czyli wzajemnie jednoznaczne przekształcenie bryły lub płaszczyzny na siebie z zachowaniem współliniowości punktów. Natomiast jeśli obiekt *TransformGroup* użyty jest jako przodek obiektu *ViewPlatform*, to dozwolone są tylko przekształcenia: rotacja, zmiana orientacji i jednolite skalowanie.

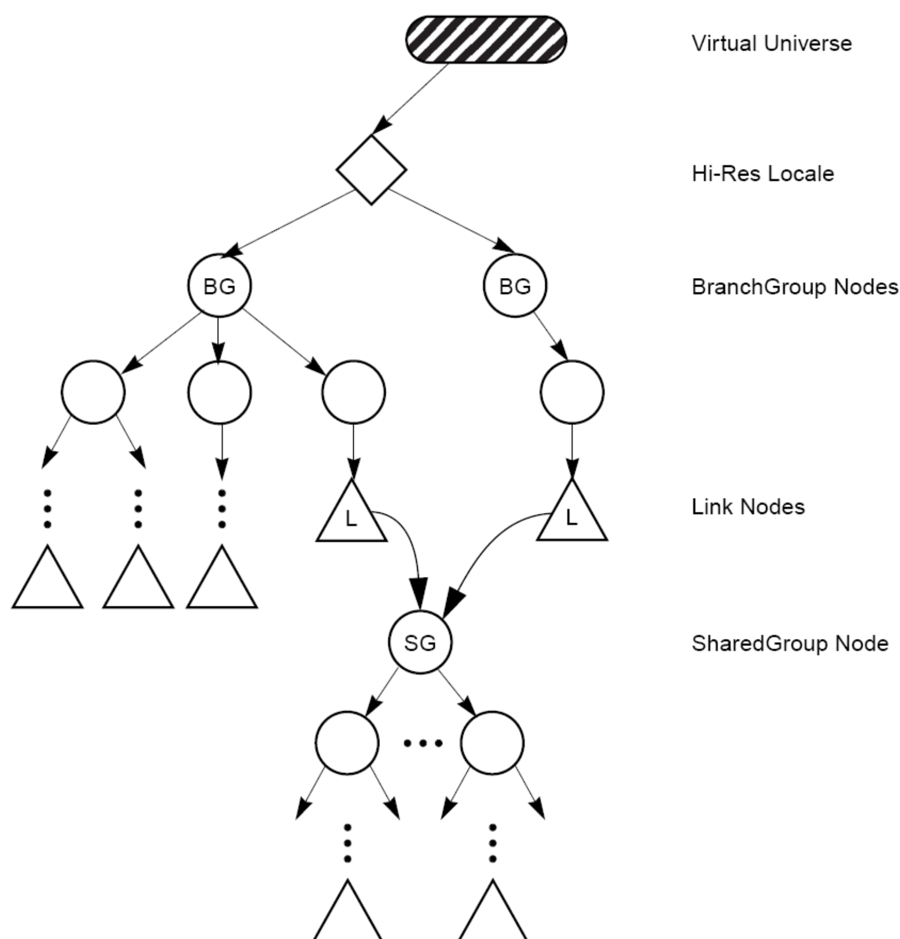
Efekty transformacji są łączne. Każda transformacja pociąga za sobą transformację innych, potomnych części podgrafu. Połączenie transformacji każdego obiektu *TransformGroup* w bezpośredniej ścieżce od *Locale* do liścia *Leaf* tworzy złożony model transformacji (*Composie Model Transformation*). CMT przekształca wszystkie obiekty (punkty, normalne i odległości) począwszy od liścia do danego obiektu *TransformGroup* i dopasowuje je do współrzędnych wirtualnego świata. Punkty są przekształcane przez CMT, normalne przez inwersję CMT, a odległości przez skalę CMT.

Java 3D dostarcza programistom dwa różne sposoby do wtórnego użycia grafu sceny. Zbiorowe grafy sceny mogą dzielić tą samą część grafu lub graf może zostać sklonowany.

Opiszę pierwszy sposób, ponieważ drugi nie został przeze mnie wykorzystany w tworzeniu pracy dyplomowej.

Obiekty Link i SharedGroup

Za pomocą obiektów *Link* i *SharedGroup* w grafie sceny programista może podłączyć kilka węzłów-rodziców do jednego węzła-dziecka. Wymienione grupy węzłów w grafie sceny zawsze występują w parze. Obiekt *SharedGroup* może być dowolnie złożony, ale musi występować jako węzeł liścia w grafie sceny oraz musi zostać opakowany w instancji obiektu *Link* zanim zostanie dodany do grafu sceny. *Link* może mieć tylko jednego, unikalnego rodzica. Java3D pozwala na przywiązanie jednego obiektu *SharedGroup* do wielu obiektów typu *Link*. Tylko obiekty *Link* mogą być rodzicami obiektu *SharedGroup*. Typowe złączenie grafu sceny za pomocą wyżej wymienionych obiektów obrazuje Rysunek 3.4.



Rysunek 3.4. *Gałąź sceny grafu zawierająca model widoku.*

Ten typ struktury jest możliwy tylko z wykorzystaniem obiektów *Link*. Inne węzły grupowania mogą mieć tylko jednego rodzica.

Następujące węzły nie mogą być połączone z węzłem *SharedGroup*:

— *ViewPlatform*
— *Background* - obiekt reprezentujący tło sceny
— Obiekty dziedziczące po abstrakcyjnej klasie *Behavior*
— *Clip* - spinacz, zawiera informacje niezbędne dla obiektu *ViewPlatform*
— *Fog* - efekt mgły na scenie, który przydaje się również prze renderowaniu sceny.

Obiekt Shape3D

Podstawową klasą do definiowania widocznej geometrii w Java3D jest liść z grafu sceny - obiekt *Shape3D*. Zawiera informacje o geometrii wizualnego przedmiotu (referencja do obiektu *Geometry*) oraz o wyglądzie przedmiotu (*Appearance*) potrzebne do renderowania sceny. W dodatku obiekt *Shape3D* otrzymuje informacje o granicach wizualnej bryły (obiekt *Bounds*). Granice są wykorzystywane podczas detekcji kolizji między przedmiotami.

Obiekt Appearance

Obiekt *Appearance* nie definiuje żadnych własności, które kontrolują obiekt *Shape3D*. Komponentami obiektu *Appearance* są atrybuty, między innymi: *PointAttribute* (określa w jaki sposób renderowane są punkty w scenie) czy *RenderingAttributes* (kontroluje operacje renderowania). Obiekt *Appearance* kontroluje dostęp do własnych atrybutów poprzez ustawienie poszczególnych flag reprezentujących dany atrybut. Wszystkie atrybuty muszą zastać zdefiniowane przez programistę zanim obiekt stanie się „żywy” w grafie sceny.

Każda z wyżej wymienionych flag może zostać poprzedzone odpowiednimi prefiksami. Każda flaga zaczyna się przedrostkiem ALLOW_ - czyli pozwól na: _READ lub _WRITE dalej. Prefiksy odpowiadają za kontrolę dostępu do poszczególnych atrybutów.

Obiekt Geometry

Obiekt *Geometry* jest abstrakcyjną klasą, która precyzuje szczegóły komponentów geometrii. Obiekt *Geomety* opisuje geometrię oraz topologię obiektu *Shape3D*. Zawiera cztery ogólne typy geometrii: *GeometryArray*, *CompressedGeometry* (przechowuje skompresowane informacje o bardzo złożonych kształtach), *Raster* (używany prze imporcie rastrowych obrazków) oraz *Text3D* (trój wymiarowe teksty). Każdy z wymienionych typów definiuje pojedynczy widoczny obiekt lub zbiór widocznych obiektów.

Obiekty dziedziczące po klasie *GeometryArray* wyszczególniają komplet prymitywów, w których skład wchodzą współrzędne przedmiotu oraz współrzędne budowy (poszczególnych punktów należących do przedmiotu), normalne, kolory oraz maskę bitów wskazującą, które z tych komponentów mają być widoczne. Informacje o wierzchołkach mogą zostać przekazane do obiektu *GeometryArray* na dwa sposoby: poprzez kopiowanie danych przez zewnętrzne metody lub poprzez podanie referencji do nich.

3.4.4. Tworzenie własnego kodu wplecionego w graf sceny

Java3D umożliwia interakcję użytkownika ze wirtualnym światem. Możliwe jest zmienianie dowolnych parametrów obiektów. Trzeba jednak pamiętać o poprawnych wartości. Drugim warunkiem, który musi być spełniony, jest utworzenie obiektu, który będzie odpowiedzialny za obsługę zdarzeń generowanych przez użytkownika. Obiekt taki musi byś obiektem klasy, która rozszerza klasę *Behavior*.

Abstrakcyjna klasa Behavior dostarcza strukturę, która może być wykorzystana do dodawania do grafu sceny obsługi zdarzeń generowanych przez użytkownika. Posiada metody, które muszą zostać przesłonięte:

— *initialize()*, która jest używana w momencie, gdy obiekt zostaje dołączony do grafu sceny
— *processStimulus()*, która używana jest, gdy konieczne jest obsłużenie zdarzenia

Dla obiektu rozszerzającego klasę *Behavior* określa się obszar działania (*SchedulingBounds*) oraz warunek wzbudzenia (*WakeUpCondition*).

Kod metody *initialize()* odpowiedzialny jest za definicję początkowego warunku wzbudzenia.

Kod metody *processStimulus()* odpowiedzialny jest za obsługę przychodzących zdarzeń oraz ustalenie następnego warunku wzbudzenia.

Obiekt obsługujący zdarzenia generowane przez użytkownika należy powiązać z kilkoma innymi obiektami. Te obiekty to obszar działania.

Dla każdego obiektu obsługującego zdarzenia określa się warunek wzbudzenia. Warunkami mogą być zdarzenia takie jak kolizja obiektów, naciśnięcie przycisku myszy, fakt wyrenderowania określonej liczy klatek, upłynięcie określonego czasu itp. Dwa obiekty obsługujące zdarzenia nie mogą posiadać identycznych warunków wzbudzenia.

Wykonanie zewnętrznego kodu odbywa się w czasie rzeczywistym, podczas działania aplikacji 3D. Poniżej zostały przedstawione poszczególne kroki podczas procesu ładowania i wykonywania zewnętrznego kodu:

1. Stworzenie obiektu dziedziczącego po *Behavior* oraz rejestracja obiektów *WakeUpCondition* zakończone jest wywołaniem metody *initialize()*, która czyni obiekt „żywym” w grafie sceny.
2. Java3D dodaje stworzony obiekt do pętli wykonania programu.
3. Co jakiś czas sprawdza, sprawdzane jest krytrium, czy obiekt *WakeUpCondition* dla odpowiedniego obiektu *Behavior* osiągnął pożądany stan.
4. Zostaje wywołana metoda *processStimus()*, zdefiniowana przez programistę.
5. Następuje wykonanie metody *processStimulus()*. Metoda odpowiedzialna jest za zmiany w grfie sceny, które najczęście widoczne są na ekranie.
6. Po jej zakończeniu ustawiane są nowe kryteria - następny obiekt *WakeUpCondition* dla innego obiektu *Behavior*.
7. Kolejny obiekt *Behavior* wchodzi w pętlę wykonania.

Wykonanie zewnętrznego kodu, zdefiniowanego przez programistę jest analogią do wykonaniu programu sterującego rzeczywistym robotem w systemie MRROC++. Każde polecenie (w Java3D i MRROC++) ma taki sam charakter: aby program mógł je wykonać, muszą zostać spełnione początkowe warunki - kryteria, dzięki którym możliwa jest zmiana stanu systemu.

Kod obiektu dziedziczącego po *Behavior* zależy od potrzeb aplikacji. Proces tworzenia własnego kodu wplecionego w graf sceny zostanie dokładnie omówiony w rozdziale poświęconym realizacji zadania w sekcji **4.3.3**

Oprócz samodzielnie tworzonych obiektów odpowiedzialnych za obsługę zdarzeń pochodzących od użytkowników mozliwe jest wykorzystanie klas dostarczonych przez pakiety *com.sun.j3d.behaviors.mouse* oraz *com.sun.j3d.behaviors.keybord.* Klasy te służą do obsługi zdarzeń generowanych przez mysz i klawiaturę, służących do poruszania obiektów w scenie.

3.5. Biblioteka Xj3D

Xj3D [8] jest projektem Web3D Consortium skupiony na tworzeniu zestawu narzędzi dla plików formatu VRML i X3D napisanym w języku Java. Ten zestaw narzędzi może zostać użyty do tworzenia aplikacji, albo jako przeglądarka plików VRML oraz X3D. Początkowo całkowicie opierał się on na technologii Java3D, jednak z biegiem czasu stał się samodzielnym API do tworzenia aplikacji 3D.

Wymagania mojej aplikacji wobec biblioteki Xj3D nie są duże. Skupiłem się jedynie na wczytaniu pliku .x3d do grafu sceny dzięki klasą dostarczonym przez Xj3D. Resztę operacji na grafie dokonuję za pomocą API Java3D. Zatem przedstawię podejście teoretyczne w tej właśnie kwestii.

Jest wiele różnych metod wczytania pliku tekstowego, w którym zapisana jest struktura grafu sceny. Najprostszą metodą jest wykorzystanie standardowej klasy Javy *com.sun.j3d.loaders.Loader*. Obiekt *Loader* jak sama nazwa wskazuje może wczytać pliki w formacie VRML lub X3D, jeśli zostanie przypisany mu właściwy obiekt pochodny, którego metody są zorientowane na konkretny format.

Xj3D udostępnia ukierunkowane wczytywacze dla plików formatu VRML lub X3D. Są nimi odpowiednio: *VRMLLoader* i *X3DLoader*.

Wczytanie sceny 3D ze sformatowanego pliku odbywa się za pomocą dwóch operacji:

1. Stworzenie określonego obiektu, dziedziczącego do klasie *Loader*.
2. Wywołanie na nim metody *load()* z argumentem ścieżki do pliku. Wynikiem parsowania pliku przez metodę *load()* jest obiekt typu *Scene*, reprezentujący graf sceny.

4. Realizacja zadania

4.1. Modele robotów w 3D Studio MAX

Model pojedynczego robota został wcześniej stworzony w podobnej pracy inżynierskiej [6]. Na podstawie tego modelu zostałem aklualny model robotów z laboratorium.

Różnica między modelami polega na stopniu złożoności wirtualnego świata. Wcześniej świat wirtualny składał się z wolno stojącego robota (postumentu) i taśmociągu. W mojej aplikacji są trzy roboty - jeden wolno stojący (*postument*), drugi poruszający się po torze jezdnym (*track*) oraz taśmociąg (*conveyor*). Poprzednia aplikacja wizualizująca została napisana w języku C++, w środowisku Windows. Jej wadą był brak przenośności na inne platformy, co mogło utrudniać jej eksploatację. W mojej aplikacji ten problem został rozwiązany przez wybór innych technologii.

Stworzony wcześniej model robota wymagał aktualizacji. Od czasu poprzedniej pracy dyplomowej zmienił się wygląd rzeczywistych robotów. Została dołączona kiść, którą należało zaprojektować oraz dodać do modelu 3D w aplikacji 3DS. Dzięki pomiarom wykonanym w laboratorium kiść została dokładnie odwzorowana w programie.

Następnie został zmierzony i odwzorowany tor jezdny. Jego położenie i odległości od postumentu w programie 3DS odpowiada rzeczywistem położeniom. Taśmociąg również uległ modyfikacji - został wydłużony do rzeczywistej długości.

Pozostało tylko sklonowanie gotowego robota i modyfikacja jego podstawy. Należało skrócić ją do odpowiedniej wysokości. Po wykonaniu tych czynności drugi robot został umieszczony na jednym z krańców toru jezdnego.

Posiadając gotowy wirtualny model robotów w 3DS z uczelnianego laboratorium (rysunek 4.1), można było wyeksportować informacje o wirtualnym świecie do pliku w formacie X3D.

4.2. Struktura aplikacji *Virtual Robo 3D*

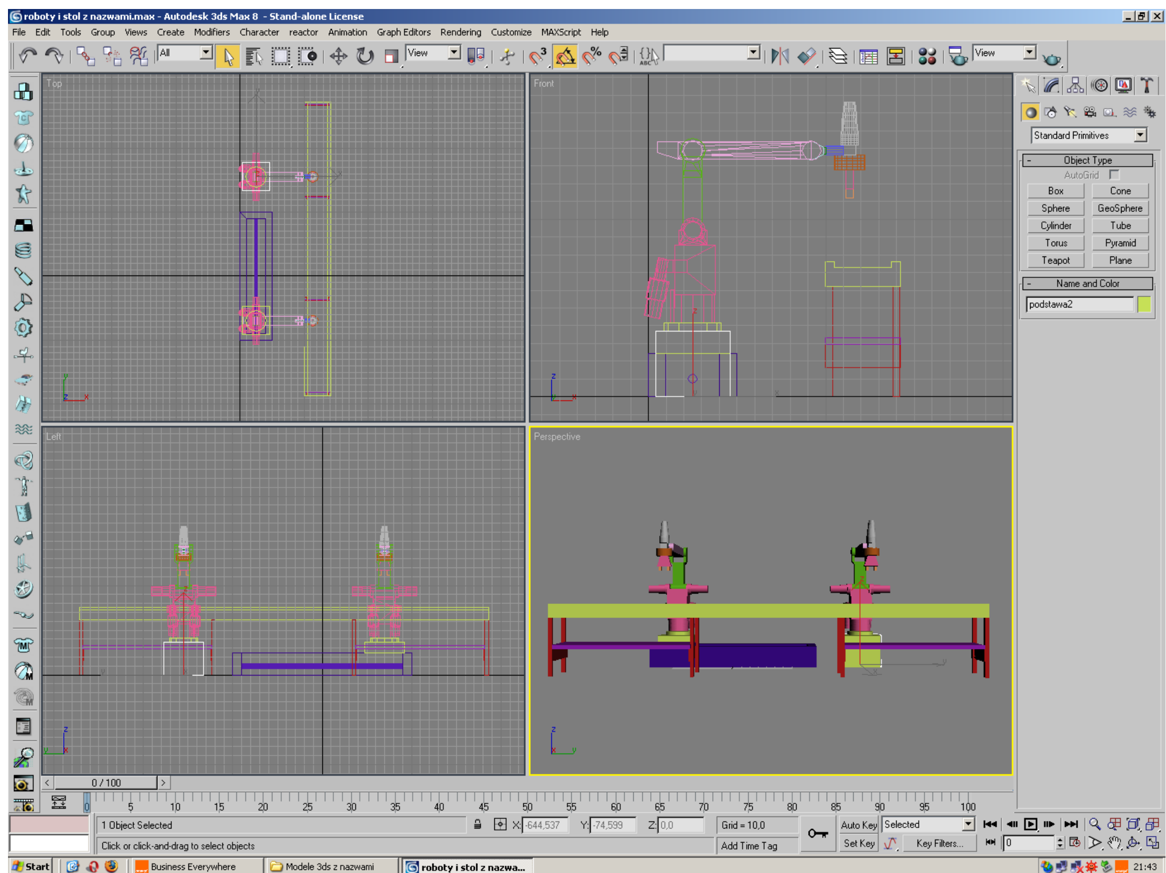
Rysunek 4.2 przedstawia strukturę aplikacji *Virtual Robo 3D*.

Aplikacja *Virtual Robo 3D* składa się w trzech głównych modułów:

1. Interfejs Użytkownika
2. SceneManager - w skład modułu wchodzą klasy i metody do zarządania innymi modulami oraz całą aplikacją
3. ClientsManager - moduł zarządzający klientami systemu MRROC++

4.2.1. Interfejs Użytkownika

W skład modułu wchodzą trzy klasy:



Rysunek 4.1. *Model robotów zaprojektowany w programie 3D Studio MAX.*

1. *VirtualRobo3D* - klasa rozszerzająca klasę *javax.swing.JFrame*
2. *WarpVirtualRobo3D* - rozszerzająca klasę *javax.swing.JPanel*
3. ConnectionStatusListner - wątek nasłuchujący status połączenia każdego klienta z odpowiednim serwerem EDP

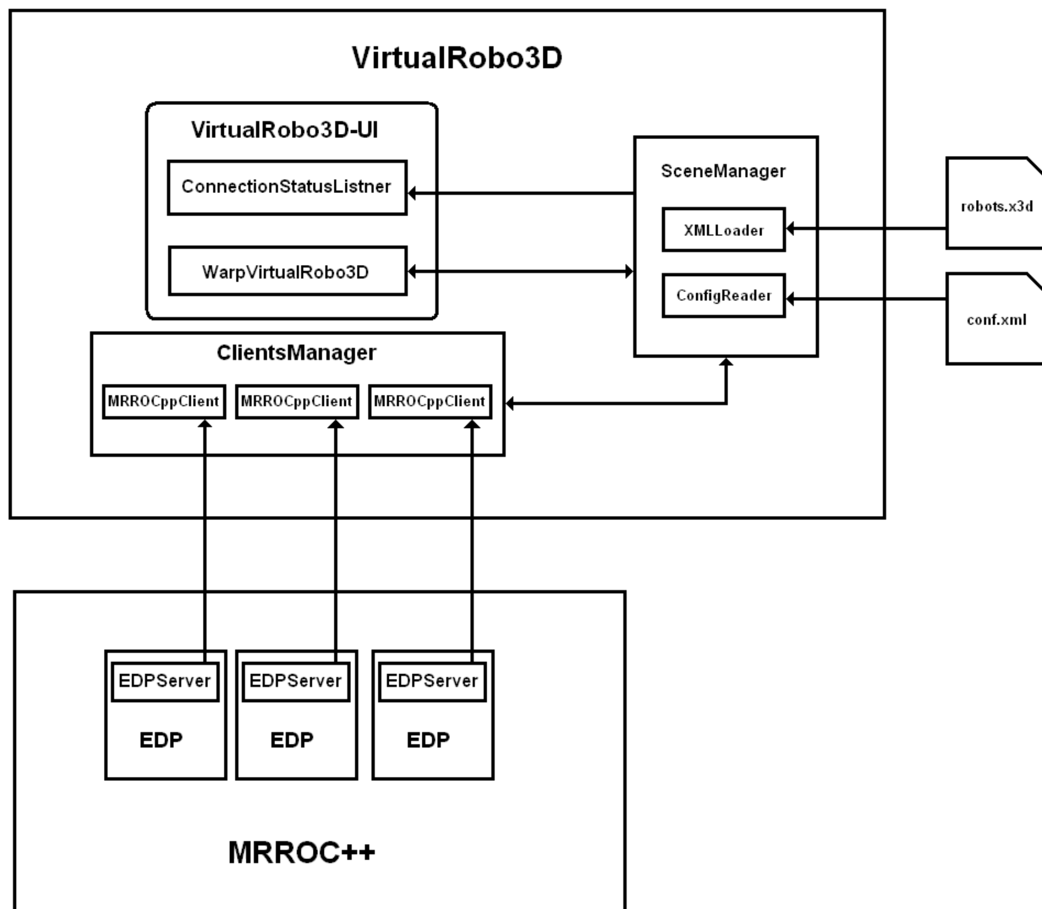
Główna klasa aplikacji - *VirtualRobo3D*

Klasa *VitrualRobo3D* rozszerza klasę JFrame. Jej ciało zawiera definicję wszystich przycisków, etykiet oraz panelu *WarpVirtualRobo3D* renderującego wirtualny świat robotów na ekranie monitora. Skupia i wiąże w całość interfejs użytkownika (UI). Po uruchmieniu metody *main()* na ekranie monitora pojawia się okno aplikacji *Virtual Robo 3D*.

W dolnej części okna znajdują się etykiety oraz przyciski dla każdego robota. Etykieta prezentuje nazwę robota i domyślnie ma kolor czerwony. Inne kolory i ich znaczene opisuje paragraf poświęcony klasie *ConnectionStatusListner* (**4.2.1**).

Po prawej stronie każdej etykiety znajdują się dwa przyciski:

— pierwszy jest dwustanowyn przyciskiem. Służy do włączania i wyłączanie pracy robota. Jeśli robot jest włączony na przycisku widnieje napis „stop”, w przeciwnym przypadku - „start”. Po pierwszym naciśnięciu przycisku start, dla którego kolwiek robota, włączany jest nałuch obiektu *RobotsBehavior*. Od tej pory, kiedy



Rysunek 4.2. *Struktura aplikacji „Virtual Robo 3D”.*

zostanie nawiązane połączenie z serwerem i klient będzie odbierał zsynchronizowane dane obiekt *RobotsBehavior* będzie aktualizował graf sceny. Zmiany będą widoczne na ekranie monitora.

— drugi przycisk służy do ponownego wczytania konfiguracji robota z pliku konfiguracyjnego. Ponowne wczytanie powoduje ustawienie trzech parametrów. Są nimi adres IP i port serwera, z którym łączy się klient oraz częstliwość z jaką klient odpytuje serwer o nowe pozycje.

Klasa *WarpVirtualRobo3D*

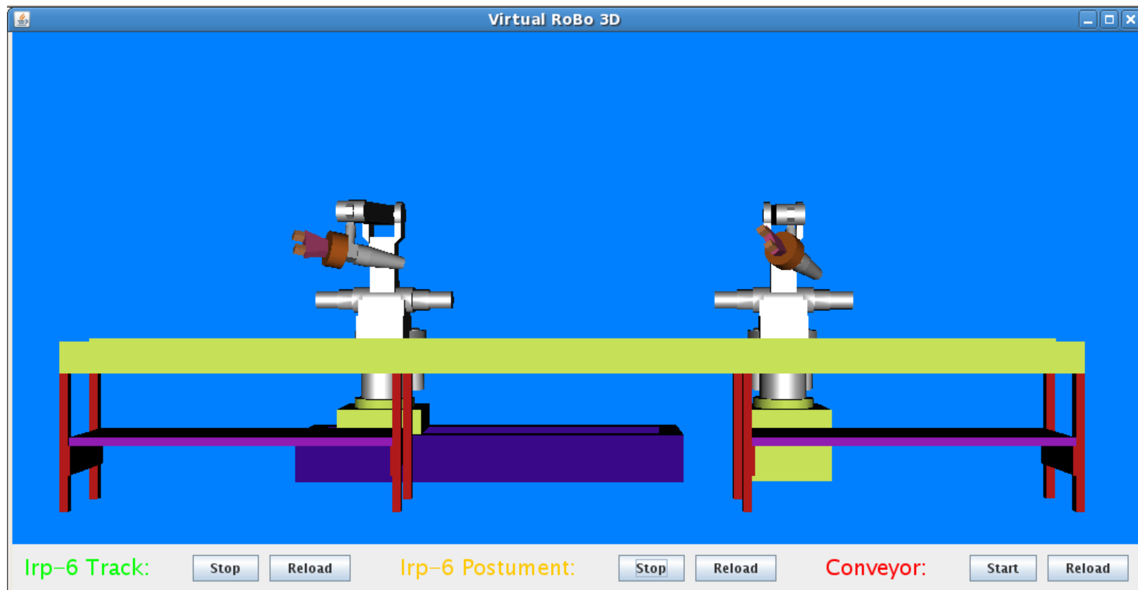
Klasa *WarpVirtualRobo3D* rozszerza klasę JPanel. Podczas inicjacji aplikacji obietk *WarpVirtualRobo3D* zostaje dodany do kontenera tworzącego okno aplikacji *Virtual Robo 3D*.
Rolą *WarpVirtualRobo3D* jest:

— pobranie konfiguracji sprzętowej i przekazanie jej do obiektów Java3D
— budowa grafu sceny
— przechowywanie aktualnych informacji dotyczącej grafu sceny
— renderowanie aplikacji Java3D

— stanowi interfejs między obiektem SceneManager a ramą VirtualRobo3D

Klasa *ConnectionStatusListner*

Klasa *ConnectionStatusListner* rozszerza klasę *java.lang.Thread*. Jest wątkiem, który za pośrednictwem menadzera aplikacji (obiekt SceneManager) prezentuje użytkownikowi stan połączenie klientów z serwerem. Efekty działania wątku *ConnectionStatusListner* widać na rysunku 4.3.



Rysunek 4.3. *Aplikacja Virtual Robo 3D. Kolory etykiet oznaczają stan połączenia.*

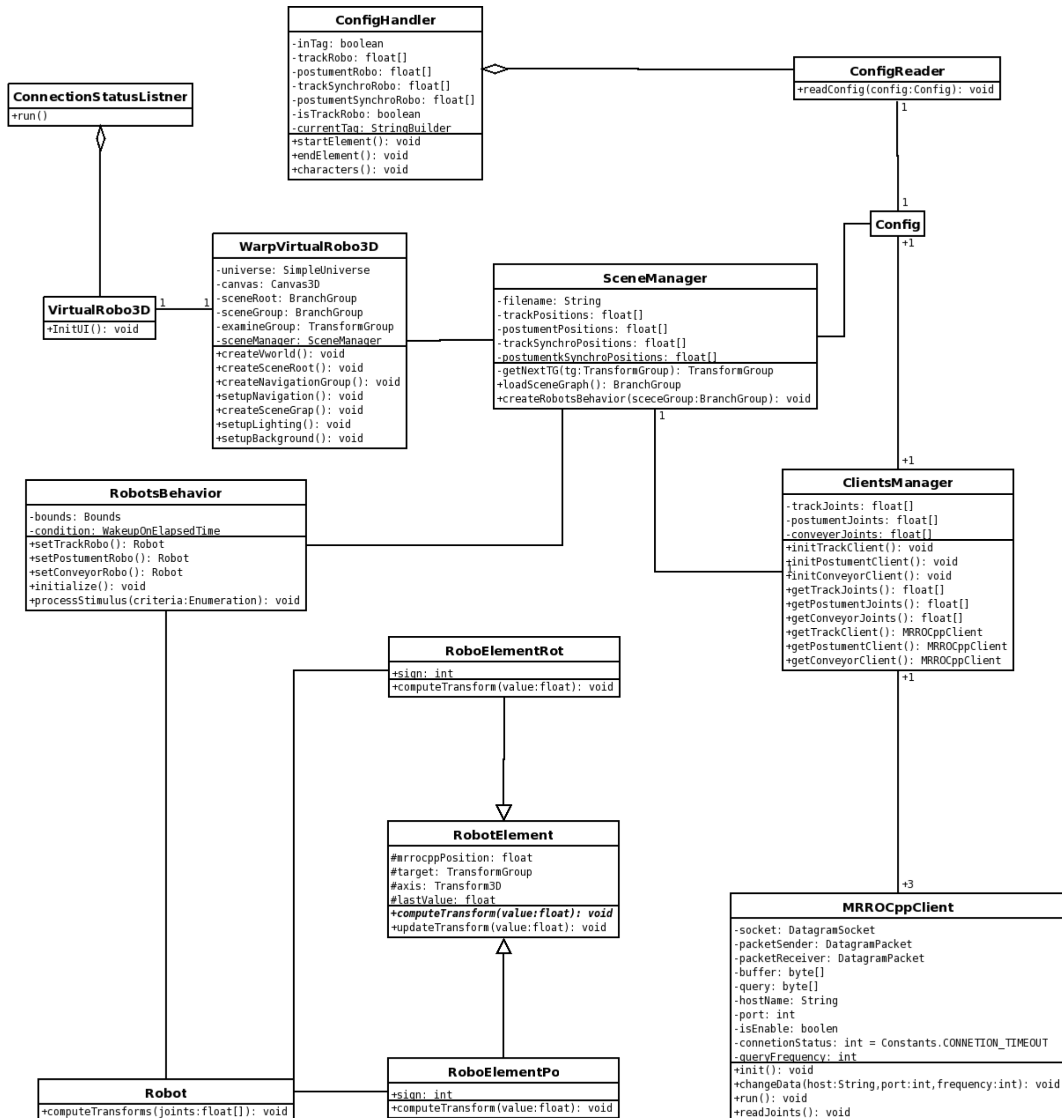
Dobór kolorów etykiet jest intuicyjny. Połączenie klienta z serwerem może osiągać trzy stany:

— brak połączenia, oznaczony jest kolorem czerwonym
— połączenie zostało nawiązane lecz roboty są niezsynchronizowane - kolor pomarańczowy
— połączenie zostało nawiązane, roboty zostały zsynchronizowane, czyli są gotowe do użytku - kolor zielony.

Dodatkowo obiekt *ConnectionStatusListner* sygnalizuje utratę połączenia w serwerem za pomocą zmiany etykiety na przycisku „start” - „stop”. Po naciśnięcu przycisku „start" klient próbuje połączyć się z odpowiadającym mu serwerem. Jeśli połącznie nie zostanie nawiązane w ciągu określonego czasu (domyślnie po dwóch sekundach), obiekt *ConnectionStatusListner* zmienia napis na przycisku na „stop” i zatrzymuje klienta. Nasłuch odbywa się także podczas połączenia. Jeśli połącznie zostanie przerwane po dwóch sekundach, następuje zmiana napisu na „stop”, a robot ustawia się w pozycji niezsynchronizowanej. Kolor etykiety zmieniany jest na czerwony oraz klient przestaje próbować łączyć się z serwerem.

4.2.2. Moduł menadzera sceny

Główną klasą modułu menadzera sceny jest *manager.SceneManager*. Zarządza przepływem informacji wewnątrz aplikacji oraz między aplikacją a systemem MRROC++. Zamin zostanie opisana jej struktura i role, należy zapoznać się z diagramem klas aplikacji *Virtual Robo 3D* znajdującym się na rysunku 4.4.



Rysunek 4.4. *Diagram klas aplikacji Virtual Robo 3D.*

Na podstawie rysunków 4.2 oraz 4.4 zostały sformułowane i wymienione role modułu *SceneManager*:

— wczytanie z zewnętrznego pliku *conf.xml* konfiguracji aplikacji *Virtual Robo 3D*
— wczytanie z zewnętrznego pliku *robots.x3d* grafu modeli wirtualnego świata i dołączenie go do grafu sceny
— konstrukcja obiektu *RobotsBehavior* oraz dołączenie go do grafu modeli wirtualnego świata

— inicjacja obiektu *ClientsManager* i powiązanie go z obiektem *RobotsBehavior*. *ClientsManager* zostanie opisany w dalszej części rozdziału
— za pośrednictwem obiektu *ClientManager* włączanie i wyłączanie klientów oraz ponowne wczytanie konfiguracji dla klientów
— włączenie i wyłączenie obiektu *RobotsBehavior*

Oprócz klasy *SceneManager* w skład manadzera sceny zalicza się obiekt wczytujący plik konfiguracyjny - *config.ConfigReader*. *ConfigReader* posiada klasę prywatną, która parsuje plik konfiguracyjny w formacje XML. Wynik parsowania zapisywany jest w kontenerze JavaBean - *config.Config*. Obiekty JavaBean charakteryzują się specyficzną budową. Oprócz składowych prywatych posiadają tylko metody zwane setterami i getterami, ustawiające lub zwracające odpowiednie pola klasy.

Wcztanie i parsowanie pliku konfiguracyjnego zostało osiągnięta za pomocą metody *readConfig()* w klasie *config.ConfigReader*:

```
public void readConfig() {
   XMLReader xr;
   try {
      xr = XMLReaderFactory.createXMLReader();
      ConfigHandler handler = new ConfigHandler();
      xr.setContentHandler(handler);
      xr.setErrorHandler(handler);

      FileReader r = new FileReader(Config.configFilePath);
      xr.parse(new InputSource(r));
   } catch (Exception e) {
      e.printStackTrace();
   }
}
```

Metoda oprócz wykorzystania klas dostępnych w pakiecie *org.xml.sax* korzysta z prywatnej klasy *ConfigHandler* rozszerzającej klasę *DefaultHandler*. Proces parsowania wykonuje nadklasa *DefaultHandler*. Programiście pozostaje tylko przysłonić trzy metody:

— *startElement()* - definiuje działanie, kiedy parser wejdzie w treść znacznika
— *endElement()* - definiuje działanie, kiedy parser zakończy pobieranie treści znacznika - tworzy odpowiednie pole w obiekcie *config.Config*
— *characters()* - zapisuje (znak po znaku) treść znacznika do obiektu tymczasowego *java.lang.StringBuilder*

4.2.3. Moduł menadżera klientów

Moduł menadzera klientów składa się z dwóch klas:

— *manager.ClientsManager* - zarządza klientami systemu MRROC++

— *client.MRROCppClient* - kllient systemu MRROC++

Klasa *ClientsManager*.

Rola obiektu *ClientsManager*:

— przechwuje referencje do trzech obiektów *MRROCppClient*: *trackClient*, *postumentClient*, *conveyorClient*
— inicjuje oraz uruchamia wątki klientów, przekazując im dane serwera, z którym mają się połączyć oraz odpowiednio tablice: trackJoints, postumentJoints, conveyerJoints, w których będzie przechwywać aktualne pozycje, pobrane z odpowiedniego serwera.
— przekazuje aktualne pozycje oraz poszczególnych klientów do obiektu *RobotsBehavior*

Klasa *MRROCppClient*.

Klasa *MRROCppClient* rozszerza klasę *java.lang.Thread*. Jest oddzielnym wątkiem, którego jedynym zadaniem jest połączenie się z serwerem EDP oraz pobranie aktualnych pozycji przypisanego do niego robota. Działanie wątku zostalo uproszczone do wywoływania w nieskończonej pętli metody *readJoints()*, której kod wygląda następująco:

```
public void readJoints() {
   try {
      socket.send(packetSender);
      try {
         socket.receive(packetReceiver);
      } catch(SocketTimeoutException e ) {
         connetionStatus = Constants.CONNETION_TIMEOUT;
         return;
      }
   } catch( IOException e ) {
      e.printStackTrace();
   }

   ByteBuffer buf = ByteBuffer.allocate(Constants.BYTE_BUFFER_SIZE);
   buf.order(ByteOrder.LITTLE_ENDIAN);
   buf.put(buffer);

   int ans = buf.getInt(0);

   if(ans == 0) {
```

```
            connetionStatus = Constants.NOT_SYNCHRONIZED;
            return;
        }
        else {
            for(int i = 4; i <
                Constants.FLOAT_BYTE_SIZE * this.joints.length + 4;
                    i += Constants.FLOAT_BYTE_SIZE) {
                joints[(i-4) / Constants.FLOAT_BYTE_SIZE] =
                    buf.getFloat(i);
            }
            connetionStatus = Constants.SYNCHRONIZED;
        }
    }
```

Metoda *readJoints()* jest wywoływana w częstotliwością zadaną w pliku konfiguracyjnym.

W każdym procesie EDP przypisanym do robota uruchamiany jest wątek, w którym przeglądane są poszczególne pozycje członów robota. Zebrane informacje są zapisywane do tablicy bajtów (bufora). Po nadejściu żadania od kilenta wątek w procesie EDP wysyła tablicę bajtów do klienta. Pierwsze cztery bajty są zarezerwowane na oznaczenie w jakim stanie znajduje się robot. Wyróżniono dwa stany:

— robot jest niezsynchronizowany - wysyłana jest liczba całkowita o wartości "0"
— robot osiągnął zsynchronizowane pozycje - wysyłana jest liczba całkowita o wartości "1"

Reszta bajtów w tablicy niesie informacje o pozycjach poszczególnych członów robota. Rozmiar pakietu jest stałego rozmiaru i wynosi trzydzieści sześć bajtów. Wiąże się to z rozmiarem liczby całkowitej (int) oraz ośmiu liczb zmiennopozycyjnych (float). Rozmiar tablicy (buffora) bajtów można obliczyć za pomocą prostego wzoru:

```
BUFFER_SIZE = sizeof(int) + 8*sizeof(float) = 4 + 8*4 = 32 [bajty]
```

Komunikacja między klientami a serwerami jest bardzo prosta. Klient za każdym razem kiedy chce otrzymać informację o aktualnej pozycji robota, wysyła do serwera jeden bajt. Umownie bajt ma wartość **0xFF**. Po wysyłce czeka na odpowiedź przez określony czas (domyślnie dziesięć milisekund). Jeśli odpowiedź nie nadeszła, ustawia status połączenia na kod *Constants.CONNETION_TIMEOUT* metoda kończy działanie. Jeśli odpowiedź nadzeszła i pierwsze cztery bajty mają wartość "1", aktualizuje tablicę liczb zmiennopozycyjnych (*float []*) przekazaną przez menadzera klientów i ustawia status połączenia na wartość *Constants.SYNCHRONIZED*. Jeśli pierwsze cztery bajty oznaczają brak synchronizacji (liczba "0"), status połączenie ustawiany jest na wartość *Constants.NOT_SYNCHRONIZED* i metoda kończy działanie.

Wyżej wymieniony mechanizm można było wykonać za pomocą jednego z dwóch popularmych protokołów komunikacyjnych. Pierwszym z nich jest bardziej popularny *Transmission Control Protocol / Internet Protocol* (**TCP/IP**). Drugim, mniej popularnym jest *User Datagram Protocol* (**UDP**).

UDP to protokół bezpołączeniowy, więc nie ma narzutu na nawiązywanie połączenia i śledzenie sesji (w przeciwieństwie do TCP). Nie ma też mechanizmów kontroli przepływu i retransmisji. Korzyścią płynącą z takiego uproszczenia budowy jest większa szybkość transmisji danych i brak dodatkowych zadań, którymi musi zajmować się host posługujący się tym protokołem.

Na potrzeby aplikacji *Virtual Robo 3D* w zupełności wystarcza protokół UDP. Nie trzeba kontrolować przepływów i retransmisji danych. Pakiet klienta jest bardzo mały (wynosi jeden bajt). Lepiej ponawiać zapytanie co określony czas, niż czekać i kontrolować, to co się dzieje w sieci komputerowej. Serwer EDP zawsze wyśle komplet rzeczywistych odczytów z czujników. Pakiet z serwera zawsze będzie miał taką samą długość. Pakiet będzie wypełniany danymi w zależności od robota. Dla maniputalora na torze jezdnym tablica bajtów będzie w całości przetwarzana przez klienta. Klient reprezentujący postument będzie przetwarzał tablicę bez ostatnich czterach bajtów, ponieważ liczba pozycji jest o jeden mniejsza od manipuratora na torze jezdnym. Z kolei dla klienta taśmociągu ważne dane zawierają się w ośmiu pierwszych bajtach. Pierwsze cztery dla określenia synchronizacji, kolejne cztery - przesuniecie taśmy.

4.3. Graf sceny - model 3D wirtualnego świata robotów

W rodziale poświęconemu opisowi technologi w sekcji dotyczącej *Java3D* (**3.4**) zotały opisane poszczególne obiekty wchodzące w skład każdej aplikacji 3D. W niniejszej sekcji zostanie opisany proces inicjacji aplikacji, budowa wczytanego grafu sceny z pliku *robots.x3d* oraz modyfikacja grafu na potrzeby wizualizacji.

4.3.1. Korzeń grafu sceny

Podczas konstruowania wirtualnego świata należy pobrać informację na temat graficznej konfiguracji maszyny, na której aplikacja ma działać. Służą do tego klasy znajdujące się w pakiecie *java.awt*. Kod znajdujący się poniżej przedstawia pobranie konfiguracji sprzętu graficznego oraz utworzenie obiektu *Canvas3D*, który bedzie przechowywał niezbędne informacje dotyczące sterowników graficznych:

```
GraphicsConfigTemplate3D template =
            new GraphicsConfigTemplate3D();
template.setDoubleBuffer(template.REQUIRED);
GraphicsEnvironment env =
            GraphicsEnvironment.getLocalGraphicsEnvironment();
GraphicsDevice dev = env.getDefaultScreenDevice();
GraphicsConfiguration gfxConfig =
            dev.getBestConfiguration(template);
```

```
canvas = new Canvas3D(gfxConfig);
```

Po utworzeniu obiektu *Canvas3D* i spakowaniu do niego konfiguracji sprzętowej, możliwa jest inicjacja obiektu *SimpleUniverse*, dziedziczącego po obiekcie *VirtualUniverse*. Klasa *SimpleUniverse* jest powszechnie używana do tworzenia korzenia wirtualnego świata. Tworząc obiekt tej klasy programista nie musi się martwić o obiekt *Locale*, który jest przytwierdzony do korzenia grafu sceny. Obiek *SimpleUniverse* automatycznie tworzy obiekty *Lokale*, *VirtualUniverse* oraz model widoku (View Branch), w którego skład wchodzą obiekty *ViewingPlatform* oraz *Viewer*.

Następnie kierując się zgodnie zasadami Java3D inicjowany jest graf sceny (gałąź Content Branch).

4.3.2. Kostrukcja gałęzi treści grafu sceny.

Tworząc poszczególne węzły w grafie sceny trzeba pamiętać o dwóch zasadach:

1. Węzłów dodanych do żywego grafu sceny nie można modyfikować
2. Przed dodaniem węzła do żywego grafu należy ustawić niezbędne flagi

W pierwszej kolejności należy stworzyć korzeń dla gałęzi treści - obietk *sceneRoot*typu *BranchGroup*. Odpowiada za to kod:

```
sceneRoot = new BranchGroup();
```

Dalej trzeba ustawić niezbędną flagę:

```
sceneRoot.setCapability(BranchGroup.ALLOW_CHILDREN_EXTEND);
```

Flaga *BranchGroup.ALLOW_CHILDREN_EXTEND* pozwala dodawać węzły-dzieci do korzenia gałęzi treści grafu sceny.

Następnie należy stworzyć obiek *examineGroup* typu *TransformGroup* i dodać do niego węzły-liście definiujące zachownie myszki oraz klawiatury. Niestety obsługa klawiatury na dzień dzisiejszy nie jest dostępna, ponieważ gdy aplikacja obsługuje klawiaturę w 80% obciąża procesor - co jest nie dopuszczalne z punktu widzenia użytkowania aplikacji. Tworzenie funkcji manipulacji sceną jest wykonane za pomocą kodu:

```
examineGroup = new TransformGroup();
examineGroup.setCapability(TransformGroup.ALLOW_CHILDREN_EXTEND);
examineGroup.setCapability(TransformGroup.ALLOW_CHILDREN_READ);
examineGroup.setCapability(TransformGroup.ALLOW_CHILDREN_WRITE);
examineGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
examineGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);

sceneRoot.addChild(examineGroup);
BoundingSphere behaviorBounds =
          new BoundingSphere(new Point3d(),  Double.MAX_VALUE);
```

```
MouseRotate mr = new MouseRotate();
mr.setTransformGroup(examineGroup);
mr.setFactor(0.01,0.01);
mr.setSchedulingBounds(behaviorBounds);
sceneRoot.addChild(mr);

MouseTranslate mt = new MouseTranslate();
mt.setTransformGroup(examineGroup);
mt.setFactor(0.5);
mt.setSchedulingBounds(behaviorBounds);
sceneRoot.addChild(mt);

MouseZoom mz = new MouseZoom();
mz.setFactor(5);
mz.setTransformGroup(examineGroup);
mz.setSchedulingBounds(behaviorBounds);
sceneRoot.addChild(mz);
```

Ważne jest, aby przed dodaniem obiektu *examineGroup* do korzenia grafu treści, ustawić flagi umozliwiające dalsze operacje na grafie. Szczególną uwagę trzeba wzrócić na flagi: TransformGroup.ALLOW_TRANSFORM_READ oraz TransformGroup.ALLOW_TRANSFORM_WRITE, które umożliwą modyfikację sceny i prezentację ruchów robotów.

Dla myszki należy stworzyć trzy obiekty definiujące jej zachowanie:

1. *MouseRotate* - odpowiada za obracanie sceną, gdy naciśnięty jest prawy przycisk myszy powoduje obrót sceny,
2. *MouseTranslate* - odpowiada za przesuwanie sceną, gdy naciśnięty jest lewy przycisk myszy powoduje przesuwanie sceny w płaszczyźnie XY,
3. *MouseZoom* - odpowiada za skalowanie sceny, gdy naciśnięty jest środkowy przycisk myszy powoduje, przesuwanie sceny wzdłuż osi Z

Należy zwrócić tutaj uwagę na pewien ważny szczegół. W aplikacji *VirtualRobo3D* efekt "pływania"po scenie został rozwiązany w następujący sposób:

Dla obiektu *examineGroup* macierz transformacji w postaci obiektu *Transform3D* reperzentuje środek globalnego układu współrzędnych. Podczepionie do niej zachowania myszki, zmieniają położenie całej sceny względem środka globalnego układu odniesienia pozostawiając widok (kamerę) w niezmiennej pozycji. Jest to lepsze rozwiązanie niż manipulacja widokiem (kamerą), ponieważ takie rozwiązanie jest mało intuicyjne i mniej funkcjonalne. Ciężko byłoby poruszać się w scenie użytkownikowi końcowemu.

W dalszej kolejności należy dodać do korzenia gałęzi treści obiekt reprezentujący tło (*Background*), ustawiając odpowiedni kolor:

```
Background backrgound = new Background();
backrgound.setColor(new Color3f(0.0f,0.5f,1.0f));
BoundingSphere bounds =
         new BoundingSphere(new Point3d(0,0,0),100);
backrgound.setApplicationBounds(bounds);
BranchGroup bg = new BranchGroup();
bg.addChild(backrgound);

sceneRoot.addChild(bg);
```

oraz obiekt źródła światła:

```
BranchGroup lightBG = new BranchGroup();
BoundingSphere lightBounds =
new BoundingSphere(new Point3d(), Double.MAX_VALUE);
DirectionalLight headLight =
   new DirectionalLight(new Color3f(1.0f,1.0f,1.0f),
      new Vector3f(0,0,-1));
headLight.setCapability(Light.ALLOW_STATE_WRITE);
headLight.setInfluencingBounds(lightBounds);
lightBG.addChild(headLight);

sceneRoot.addChild(lightBG);
```

Pozostało najważniejsze - wczytanie modelu robotów. Odczyt sceny z pliku wyeksportowanego przez 3DS jest jest możliwy za pośrednictwem biblioteki Xj3D. Twórcy Xj3D umozliwili programistom wczytanie modelu sceny do aplikacji Java na dwa sposoby:

1. *O*dczyt za pomocą SAI - *Scene Access Interface*. Dalsze manipulacja na grafie sceny odbywa się za pomocą biblioteki Xj3D.
2. *U*życie biblioteki Xj3D jako loader'a plików .x3d do Java3D. Dalsza manipulacja odbywa się z poziomu Java3D.

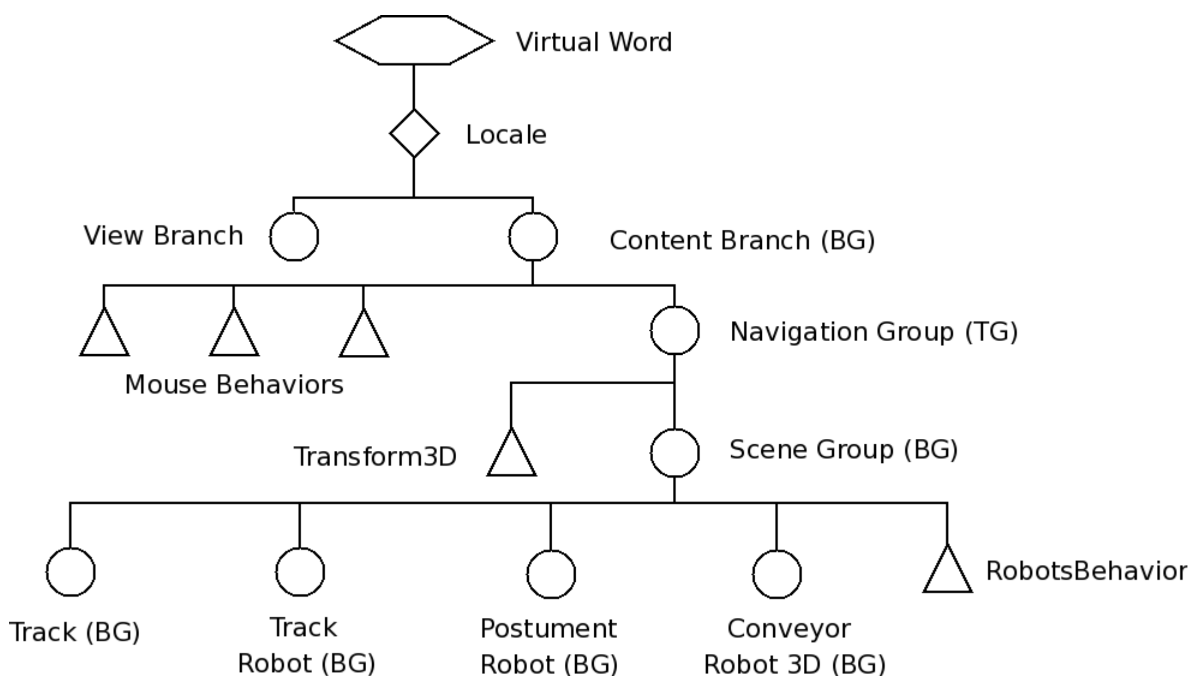
Wybrałem drugi wariant, który był dla mnie bardziej zrozumiały i prostrzy w użyciu:

Java3D w pakiecie *com.sun.j3d.loaders* udostępnia klasę *Scene*, do której za pośrednictwem „loader'ów" zapisywana jest informacja o struktórze sceny zapisanej w plikach tekstowych. Za pomocą poleceń:

```
Scene scene =  (new X3DLoader()).load(filename);
sceneGroup = scene.getSceneGroup();
```

W wyniku wyżej wynienionych operacji obiet *sceneGroup* zawiera informacje dotyczące korzenia grafu sceny wczytanego z pliku w formacie X3D.

Rysunek 4.5 zawiera uproszczony schemat grafu sceny wirtualnego świata, stworzonego na potrzeby aplikacji. Witać na nim jeszcze nie omawiany węzeł-liść *RobotsBehavior*. Obiekt *RobotsBehavior* został zdefiniowany i stworzony w celu modylikacji obiektów w wirtualnym świecie. Za jego pomocą wirtualne roboty będą symulowac pracę rzeczywistych robotów w laboratorium.



Rysunek 4.5. *Górna część grafu sceny.*

Ostatnim etapem w inicjacji wirtualnego świata jest przejście w pętli przez graf sceny, zebranie danych niezbędnych do wizualizacji pracy robotów. Na podstawie zagregowanych danych możliwa jest inicjacja oraz dołączenie do grafu sceny obiektu dziedziczące po klasie *Behavior* - *RobotsBehavior*.

4.3.3. *User-defined actions* - definiowanie zachowań w wirtualnym świecie.

API Java 3D zostało wyposażone w interpolatory. Zadaniem interpolatorów jest interpolowanie stanu obiektu między rónymi stanami. Java3D dostarcza bogatą kolekcję klas pozwalających na interpolowanie stanu obiektów. Tego typu operacje są wykorzystywane przy tworzeniu animacji. Stan obiektu zawsze zmienia się na podstawie upływającego czasu. Stan obiektu przechodzi ze stanu początkowego do końcowego w określonym czasie.

W aplikacji *Virtual Robo 3D* można było wykorzystać gotowe interpolatory. Jednak nie byłoby to optymalnym rozwiązaniem. Wymagałoby stworzenia kilkunastu interpolatorów, ponieważ przy korzystaniu z interpolatorów należy określić obiekt, na którym interpolator ma operować.

W przypadku interpolacji pozycji robotów należałoby przypisać interpolator dla każdego ważnego węzła TransformGroup. W konsekwencji liczba przyłączonych interpolatorów do grafu sceny byłaby równa osiemnaście (dziewięć dla robota na torze jezdnym, osiem dla postumentu i jeden dla taśmociągu). Animacja zużywałaby niepotrzebną pamięć i nadmiernie obciążałaby procesor.

Na potrzeby aplikacji wystarczyłby jeden obiekt rozszerzający klasę *Behavior* skupiający wszystkie potrzebne węzły transformacji (*TransformGroup*). Po otrzymaniu aktualnych pozycji robotów zaktualizowałby odpowiednie węzły transformacji i czekałby na następny ruch. Jest to rozwiązanie prostrze i bardziej wydajne.

Obiekt *RobotsBehavior*

Wynikiem analizy budowy obietku rozszerzającego klasę *Behavior* jest obiekt *utils.RobotsBehavior*. Pełni rolę koordynatora zachowań wszystkich robotów. W jego skład wchodzą dwa obiekty typu *utils.Robot* reprezentujące rzeczywiste manipulatory. Robot *Conveyor* został uwzględniony w projektowaniu i implementacji, ale obecnie nie jest wizualizowany ruch taśmociągu. Każdy z obiektów *utils.Robot* zbudowany jest z obietków rozszerzających abstrakcyjną klasę *utils.RobotElement*.

Elementy robota zostały sklasyfikowane na dwie klasy:

1. *utils.RobotElementPos* - obiekt reprezentujący translację sprecyzowanej części robota.
2. *utils.RobotElementRot* - obiekt reprezentujący rotację sprecyzowanej części robota.

Każdy *RobotElement* w kostruktorze przyjmuje obiekt *TransformGroup*. *TransformGroup* umożliwia grupowanie obiektów i wykonywanie na całej grupie pewnych operacji geometrycznych. Operacje obejmują przekształcenia takie jak translacja, obrót, czy skalowanie, są wykonywane z wykorzystaniem obiektów *Transform3D*. Dzięku grupowaniu *RobotElement* potrzebuje tylko jednego, właściwego dla danej cześci robota obiektu *TransformGroup*.

Metoda *updateTransform()* aktualizuje węzeł transformacji *TransforGroup* za pomocą zmiany zawartości obiektu transformacji *Transform3D*:

```
public void updateTransform(float value) {
   Transform3D t = new Transform3D();
   target.getTransform(t);
   if(value != lastValue) {
      computeTransform(value - lastValue);
      lastValue = value;
      inverseAxis.invert();
      axis.mul(inverseAxis);

      target.setTransform(axis);
      inverseAxis.invert(axis);
```

```
    }
}
```

Metoda *updateTransform()* jako argument otrzymuje kolejną (aktualną) pozycję części robota. Jest nią liczba zmiennoprzecikowa. Odejmując od aktualnej wartości poprzednią pozycję, otrzymuje skok o jaki trzeba zaktualizować macierz transformacji (obiet *axis* typu *Transform3D*). Zaktualizowaną macierz należy pomnożyć przez macierz do niej odwrotną. Wynik mnożenia jest aktualną macierzą transformacji, którą należy ustawić w węźle transformacji (*TransformGroup*).

Operacja aktualizacji pozycji jest wykonywana za pomocą metody *computeTransform()*. Każda klasa rozszerzająca abstrakcyjną klasę *RobotElement* musi przesłonić abstrakcyjną metodę *computeTransform()*. Przesłonięcie polega na zdefiniowaniu, co dokładnie rozszerzony *RobotElement* powinien obliczać.

Dla *utils.RobotElementPos* aktualizacja macierzy transformacji polega na aktualizacji wektora transformacji, a dokładnie współrzędnej z:

```
@Override
public void computeTransform(float value) {
    axis.setTranslation(new Vector3d(0,0,value));
}
```

Natomiast dla *utils.RobotElementRot* przesłonienie metody *computeTransform()* wygląda następująco:

```
@Override
public void computeTransform(float value) {
    axis.rotY(this.sign*(value - mrrocppPosition));
}
```

W tym przypadku należy zaktualizować macierz rotacji o podaną wartość, z uwzględnieniem różnicy między rzeczywistymi robotami i ich modelami wirtualnymi (stała mrrocppPosition odczytana z pliku konfiguracyjnego). Obrót odbywa się wokół osi *Y*.

Mając zdefiniowane elementy robotów, można było przystąpić do budowy klasy *utils.Robot*. Budowa manipulatorów różni się tylko w jednej kwestii. Robot *Track* ma dodatkową funkcjonalności. Mianowicie: możliwa jest translacja manipulatora po torze jezdnym. Poza tym elemeny robota z punktu widzenia aplikacji *Virtual Robo 3D* niczym się nie nie różnią.

W skład klasy *utils.Robot* wchodzą następujące prywatne składowe:

```
private RoboElementPos postument
private RoboElementRot [] angles
private RoboElementPos [] fingers
```

Składowa *postument* typu *RoboElementPos* odpowiada podstawie manipulotorów, która jest inicjowana tylko w przypadku maniputalowa usytuowanego na torze jezdnym. Zatem robot-postument nie ma zainicjowanej tej składowej.

Składowa *angels* jest tablicą typu *RoboElementRot []*. Zawiera informacje dotyczące poszczególnych przegubów, począwszy od kolumny, kończąc na kiści.

Tablica *fingers* zawiera obiekty typu *RoboElementPos* i służy do wizualizowania zaciskania kiści przez manipulatory. Jest inicjowana dla obu robotów.

Klasa *utils.Robot* na potrzeby wizualizacji została wyposażona w metodę *computeTransforms()*, której parametrem wywołania jest tablica aktualnych pozycji robota:

```
public void computeTransforms(Float [] params) {
   int startindex = 0;
   if(params != null) {
      if(params.length == Constants.TRACK_SERVOS) {
         postument.updateTransform((-100)*params[startindex]);
         startindex = 1;
      }

      int angle = 0;
      for(int i = startindex; i < params.length - 1; i++) {
         changeAngle(angle++, params[i]);
      }
         updateFingers(params[params.length - 1]);
   }
}
```

Metoda aktualizuje wszystkie obiekty *utils.RobotElement*. Rozróżnia, dla którego robota jest wywołana, poprzez sprawdzenie długości tablicy parametrów:

```
params.length == Constants.TRACK_SERVOS
```

Jeżeli długość tablicy parametrów jest równa ilości parametrów dla manipulatora z torem jezdnym, aktualizowana jest pozycja podstawy manipulatora. W przeciwnym razie aktualizowane są kąty przegubów i roztaw szczęki chwytaka.

Aktualizacja całego robota odbywa się z poziomu obiektu rozszeżającego klasę *Behavior - RobotsBehavior*. Rozszerzenie wymusza na programiście przysłonięcie dwóch metod: *initialize()* oraz *processStimulus()*.

Metoda *initialize()* jest wywoływana tylko raz - podczas dołączenia węzła-liścia *RobotsBehavior* do żywego grafu sceny. Wykonuje następujące zadania:

— inicjuje kryterium wyzwolenia. Jako kryterium wyzwolenia dla obiektu *RobotsBehavior* przyjmuje obiekt *WakeupOnElapsedTime*. Obiekt *WakeupOnElapsedTime* jest ustawiany na czas, po którym ma zostać aktywowany. Częstotliwość z jaką będzie aktywowany jest definiowany i przekazywany do aplikacji z pliku konfiguracyjnego.
— inicjacjuje wątki - klienty dla serwera EDP (klasa MRROCppClient zostanie omówiona w następnej sekcji) innego dla każdego robota.

Metoda *processStimulus()* za pośrednictwem klientów aktualizuje scenę:

```
@Override
public void processStimulus(Enumeration criteria) {
   Float [] tmp = manager.getTrackJoints();
   if(tmp != null && manager.getTrackClient().getConnetionStatus()
               == Constants.SYNCHRONIZED)
   trackRobo.computeTransforms(tmp);

   tmp = manager.getPostumentJoints();
   if(tmp != null &&
               manager.getPostumentClient().getConnetionStatus()
                       == Constants.SYNCHRONIZED)
      postumentRobo.computeTransforms(tmp);

   tmp = manager.getConveyerJoints();
      if(tmp != null &&
               manager.getConveyorClient().getConnetionStatus()
                      == Constants.SYNCHRONIZED)
         ;

   this.wakeupOn(condition);
}
```

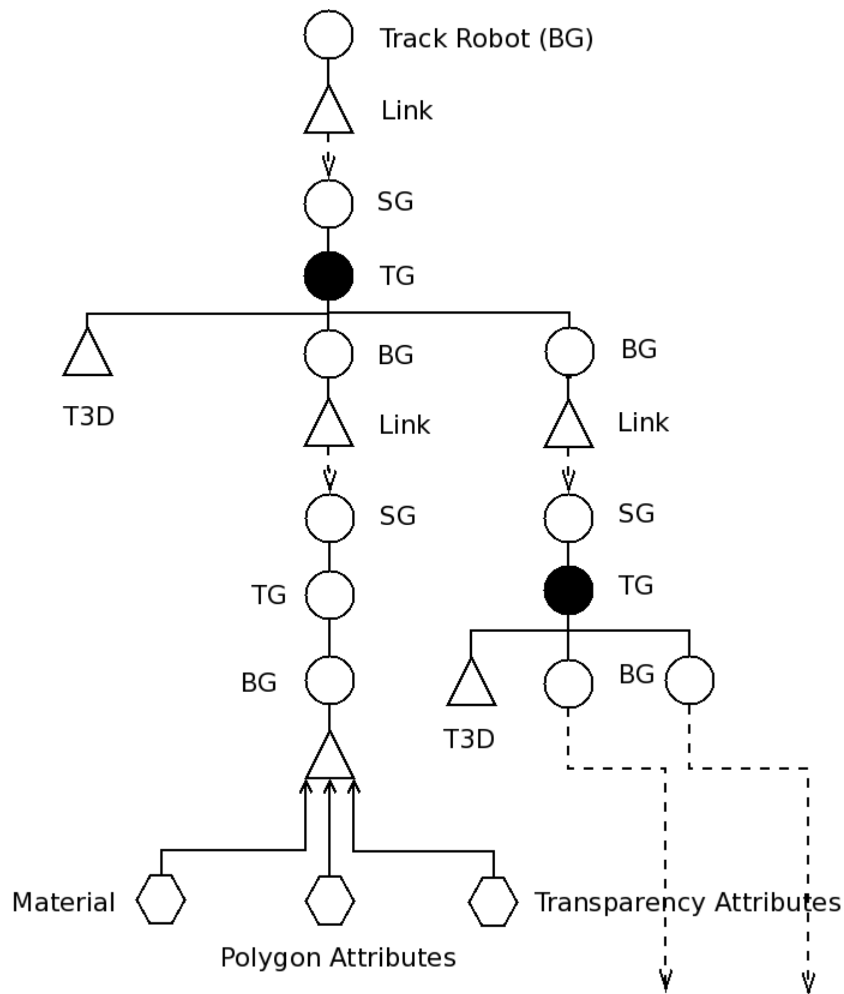
Metoda *processStimulus()* wykorzystuje obiekt *manager* typu *SceneManager*. Klasa *SceneMaganager* zarządza całą aplikacją. Zostanie omówiona w sekcji *4.5*.

Dołączenie obiektu *RobotsBehavior* do grafu sceny.

Na rysunku 4.6 został przedstawiony początek podgrafu sceny zawierający informacje na temat budowy modeli manipulatorów.

Zostały w nim wyróżnione (w kolorze czarnym) dwa węzły transformacji (TG). Dalsza część grafu maniputalora nie została przedstawiona z uwagi na powtarzalność w budowe oraz problemy z przestrzenią jaką zajmowałby rysunek. Przerywane strzałki w prawej, dolnej części grafu symbolizują dalszą, powtarzalną podwzględem budowy część grafu. Schemat przedstawia węzły, które należy rozpatrzyć oraz wyznaczyć te, które są potrzebne przy interpolacji robotów. Poniżej znajduje się kod, dzięki któremu pobierane są kolejne węzły:

```
private Group getNextTG(Group previousGroup, int place) {
   BranchGroup bg = (BranchGroup) previousGroup.getChild(place);
```



Rysunek 4.6. *Górna część grafu sceny.*

```
        Link link = (Link) bg.getChild(0);
        SharedGroup sg = link.getSharedGroup();
        TransformGroup tg = (TransformGroup) sg.getChild(0);

        return tg;
    }
```

Metoda wywoływana jest w pętli, która przechodzi prze cały graf. Zaczynając od węzła-korzenia manipulatora, kończąc na kiści zapisując chwytaki (dwa małe prostopadłościany). Nie wszystkie węzły są potrzebne i trzeba ręcznie wskazać, które są istotne.

Węzły, które należy przypisać elementom robota zostały wyznaczone metodą „prób i błędów”. Na przykładzie wyżej przedstawionego rysunku opiszę dobór węzłów transformacji. Pętla wykonuje się dwa razy i tworzy obiekty typu *utils.Robot*

dla każdego robota.
Przykład 4.1:

— Dla każdego z manipulatorów pętla rozpoczyna się wywołaniem wyżej wymienionej metody z parametrem korzenia grafu manipulatora. Zwraca węzeł transformacji, który reprezentuje podstawę maniputatora. Węzeł jest znaczący dla manipulatora poruszającego się na torze jezdnym i dla niego zostanie zainicjowany obiet *RobotElementPos*. Po inicjacji należy zsynchronizować część modelu 3D z rzeczywistą pozycją. Odpowiada za to kod:

```
el.updateTransform(-tempPositions[0]);
```

Dla postumentu węzeł zostaje pominęty.

— W kolejnym wywołaniu, metoda pobiera węzeł transformacji dla podstawy oraz zwraca węzeł dla pierścienia przymocowanego do górnej części podstawy. Węzeł jest pomijany dla obu robotów.

— Następnym węzlem zwróconym przez metodę *getNextTG()* jest węzeł typu *TransformGroup* reprezentujący kolumnę monipulatora. Kolumna obraca się dla każdego robota. Zatem jest zapisana na pierwszym miejscu w tablicy *angles* przechowyjącej informacje na temat węzłów reprezentujących przeguby (obiekty typu *RobotElementRot*). Analogicznie jak w przypadku podstawy po inicjacji należy zsynchronizować część modelu 3D z rzeczywistą pozycją. Następnie ustawić cześć robota w pozycji zsynchronizowanej. Odpowiada za to kod:

```
angles[0].updateTransform(-tempPositions[1]);
angles[0].setLastValue(0);
angles[0].updateTransform(tempSynchroPositions[1]);
```

— W kolejnych iteracjach zapisywane są odpowiednie węzły dla przegubów. Pętla kończy sie, kiedy zostanie dodana końcówka manipulatora - dodanie do tablicy *fingers* dwóch obiektów *RobotElementPos* reprezentujących zaciski.

Po dokonaniu agregacji porządnych węzłów w każdej iteracji, zostaje stworzony obiekt robota i przekazany do obiektu *RobotsBehavior*:

```
tempRobo.setAngles(angles);
tempRobo.setFingers(fingers);
tempRobo.updateFingers(tempPositions[7]);


if(i == 2)
    this.behavior.setTrackRobo(tempRobo);
else
    this.behavior.setPostumentRobo(tempRobo);
```

Gdy obiekt *RobotsBehavior* jest uzupełniony w składowe robotów, zostaje wyłączony (nie aktualizuje grafu sceny):

```
this.behavior.setEnable(false);
```

oraz dodany do grafu sceny:

```
sceneGroup.addChild(this.behavior);
```

Gotowy graf sceny wczytany w pliku *robots.x3d* oraz dołączony do niego węzeł *RobotsBehavior* należy jeszcze dołączyć do żywego grafu sceny. Cały graf z modelami robotów za pomocą jednej instrukcji zostaje dołączony do grafu sceny:

```
examineGroup.addChild(sceneGroup);
```

Od tej pory cały graf jest żywy (jego struktura nie może zostać zmieniona) oraz gotowy do manipulacji.

4.4. Konfiguracja aplikacji.

Jedną z funkcjonalności każdej aplikacji jest możliwość jej konfiguracji. Do najprostszych sposobów konfigurowania aplikacji należą:

- uruchomienie aplikacji z linii poleceń z odpowiednimi argumentami, które zostaną przypisane odpowiednim obiektom
- wczytanie konfiguracji z pliku konfiguracyjnego

Aplikacja *Virtual RoBo 3D* jest konfigurowana za pomocą pliku konfiguracyjnego. Dobór nie jest przypadkowy i ma swoje uzasadnienie. Podczas eksploatacji aplikacji mógłby pojawić się następujący problem:

Użytkownik uruchomił aplikację skonfigurowaną dla jednego robota (w konfiguracji podał adres IP i port servera EDP). Podczas działania aplikacji zaszła potrzeba połączenie się z drugim serwerem, reprezentującym innego robota. W konsekwencji trzeba podać adres IP i port innego serwera EDP. Jeśli konfiguracja została wczytana z linii poleceń wymagałoby to zakończenia jej działania oraz ponownego uruchomienia z nowymi ustawieniami. W tym momencie plik konfiguracyjny ma przewagę. W każdej chwili użytkownik może zmodyfikować plik i jeszcze raz wczytać konfigurację, która na nowo zostanie przypisana odpowiednim obiektom.

4.4.1. Format pliku konfiguracyjnego.

Format pliku konfiguracyjnego może być dowolny. Istnieją jednak standardowe formaty, które wprowadzają pewien schemat.

Jednym z nich jest format *Extensible Markup Language* (XML) przeznaczony do prezentowania różnych danych w ustruktualizowany sposób. XML jest niezależny od platformy, co umożliwia łatwą wymianę dokumentów w sieci, a w przypadku aplikacji plik konfiguracyjny nie będzie miał wpływu na dobór platformy.

Dokumenty XML tworzy się według pewnych reguł za pomocą tak zwanych *znaczników*. Dokument poprawny składniowo może być przetworzony przez parser XML.

Technologia Java udostępnia co najmniej kilka parserów, które wczytują dane z plików XML. Najbardziej powszechne są *Sample API for XML* (SAC) oraz *Dokument Objęść Model* (DOM). Dla potrzeb aplikacji został wykorzystany SAX z uwagi na łatwość pisania parserów oraz umożliwia on wczytywanie dużych plików, co w obecnej sytuacji nie ma znaczenia, ale w przyszłości może ułatwić rozszerzenie pliku konfiguracyjnego.

4.4.2. Zawartość pliku konfiguracyjnego.

Plik konfiguracyjny obejmuje definicję trzech serwerów udostępniających bieżące położenia manipulatorów zaimplementowanych w procesie EDP. W skład definicji wchodzi *adres IP* lub *nazwa hosta* serwera oraz *port*, na którym udostępnia on połączenie z klientami. Klientami w aplikacji *Virtual Robo 3D* są wątki odpowiedzialne za odbieranie informacji z serwerów i przesyłanie do obiektów animujacych scenę. Dodatkowo plik konfiguracyjny musi zawierać częstotliwość z jaką klienci wysyłają zapytania do serwerów oraz ścieżkę do pliku definiującego wirtualny świat. Opis sceny 3D znajduje się w katalogu *$Virtual_Robo3D_HOME/models* w pliku *robots.x3d*.

Roboty w 3DS zostały zaprojektowane w pewnych początkowych pozycjach. W systemie MRROC++ pozycje synchronizacji robotów różnią się od pozycji wirtualnych robotów, dlatego po załadowaniu modeli robotów, trzeba uwzględnić różnicę między rzeczywistymi członami, a ich odpowiednikami w aplikacji. Dane nie powinny się zmienić, ale dla pewności zostały umieszczone w pliku konfiguracji. Dzięki temu jeśli zajdzie potrzeba ich zmiany, będzie można zmienić je z poziomu konfiguracji, a nie korygować kodu aplikacji.

Po uwzględniu różnic okazało się, że roboty przyjmują nieintuicyjne wizualnie pozycje. Rozwiązaniem tego kosmetycznego problemu, było umieszczenie w pliku konfiguracyjnym wartości, które roboty przyjmują kiedy są zsynchronizowane. W efekcje po inicjacji aplikacji roboty wiernie odwzorują pozycje rzeczywistych robotów po przeprowadzonym procesie synchronizacji.

```
<Virtual-Robo-3D>
   <file-name>models/robots.x3d</file-name>
   <query-frequency>10</query-frequency>

   <track-ip-address>127.0.0.1</track-ip-address>
   <track-port>50000</track-port>

   <postument-ip-address>192.168.18.8</postument-ip-address>
   <postument-port>50001</postument-port>

   <conveyor-ip-address>192.168.18.7</conveyor-ip-address>
   <conveyor-port>50002</conveyor-port>

   <track-robo>
      <d1>12.5</d1>
      <synchro_d1>0.0</synchro_d1>
      <Q2>0</Q2>
      <synchro_Q2>-0.08746202</synchro_Q2>
```

```
      <Q3>1.57</Q3>
      <synchro_Q3>-1.5418735</synchro_Q3>
      <Q4>0</Q4>
      <synchro_Q4>-0.004859416</synchro_Q4>
      <Q5>0</Q5>
      <synchro_Q5>1.1946632</synchro_Q5>
      <Q6>-1.57</Q6>
      <synchro_Q6>2.5911849</synchro_Q6>
      <Q7>-1.57</Q7>
      <synchro_Q7>-2.6643867</synchro_Q7>
      <Q8>3.8</Q8>
      <synchro_Q8>0.074</synchro_Q8>
   </track-robo>

   <postument-robo>
      <Q2>0</Q2>
      <synchro_Q2>-0.10063291</synchro_Q2>
      <Q3>1.57</Q3>
      <synchro_Q3>-1.5419428</synchro_Q3>
      <Q4>0</Q4>
      <synchro_Q4>0.019737557</synchro_Q4>
      <Q5>0</Q5>
      <synchro_Q5>1.1491433</synchro_Q5>
      <Q6>-1.57</Q6>
      <synchro_Q6>2.1009765</synchro_Q6>
      <Q7>-1.57</Q7>
      <synchro_Q7>-2.7311954</synchro_Q7>
      <Q8>3.8</Q8>
      <synchro_Q8>0.074</synchro_Q8>
   </postument-robo>
</Virtual-Robo-3D>
```

5. Instalacja i uruchamianie aplikacji

5.1. Konfiguracja środowiska

Aplikacja *Virtual RoBo 3D* została napisana w języku programowania Java w systemie Linux Fedora Core 7. W celu uruchomienia aplikacji należy odpowiednio skonfigurować środowisko na wybranej platformie. W dalszej części sekcji zostanie opisany proces konfiguracji środowiska dla platformy Linux, jednak instalacja wymaganych bibliotek jest analogiczna dla innych systemów operacyjnych w tym systemu Windows.

Wymagane biblioteki i aplikacje:

1. Wirtualna Maszyna Java w wersji JRE 1.6.0_03-b05.
2. biblioteka Java 3D w wersji 1.5.1.
3. biblioteka Xj3D.

5.1.1. Instalacja biblioteki Java 3D

Pod adresem [7] znajdują się linki do pobrania biblioteki dla wybranego systemu operacyjnego. W przypadku platformy Linux'opodobnej w skład paczki wchodzą dwa katalogi: *ext/* oraz *i386/*. Należy skopiować je do katalogu **$JAVA_HOME/jre/lib**.

Katalog **ext/**:

— j3dcore.jar
— j3dutil.jar
— vecmath.jar

Katalog **i386/**:

— libj3dcore-ogl.so
— libj3dcore-ogl-cg.so

5.1.2. Biblioteka Xj3D

Biblioteka Xj3D została wykorzystana do wczytania grafu sceny z pliku *$Virtual_Robo3D_HOME/models/robots.x3d*. Wszystkie niezbędne archiwa dołączone zostały do aplikacji. Znajdują się w katalogu *$Virtual_Robo3D_HOME/lib*.

Aktualna wersja biblioteki Xj3D znajduje się pod adresem [8]. W skład paczki wchodzą archiwa:

— j3d-org.jar

— xj3d-j3d.jar
— xj3d-common.jar
— xj3d-render.jar
— xj3d-parser.jar
— xj3d-runtime.jar
— xj3d-script-base.jar
— xj3d-ecmascript.jar
— xj3d-jsai.jar
— xj3d-core.jar
— xj3d-sav.jar
— uri.jar
— vlc_uri.jar
— j3d-org.jar
— xj3d-j3d.jar
— xj3d-java-sai.jar
— xj3d-sai.jar
— xj3d-xml.jar

Aplikację uruchamiamy za pomocą skrypty *run.sh* znajdującym się w katalogu *$Virtual_Robo3D_HOME*.

6. Podsumowanie

W ramach pracy powstała aplikacja kliencka wizualizująca pracę systemu manipulatorów. Serwerem jest wątek procesu EDP w systemie MRROC++ udostępniający informację o stanie manipulatorów (ich obecnej konfiguracji w sensie układu członów). Transmisja między klientem a serwerem oparta jest na protokole UDP/IP. System pozwala na symulację pracy systemu MRROC++ z wizualizacją manipulatorów bez konieczności angażowania rzeczywistych robotów. Jako platformę implementacyjno- uruchomieniową wybrano Sun JAVA co pozwala na uruchamianie aplikacja na różnych platformach systemowych (w szczególności Linux i Windows).

Podczas analizy technologii, w jakich można by było wykonać niniejszą pracę dyplomową zapoznałem się z różnymi rozwiązaniami. Rozszerzyłem moją wiedzę na temat dostępnych narzędzi do tworzenia grafiki trójwymiarowej. Zdecydowałem się na język programowania Java, który jest prostszy w implementacji od języka C/C++. Implementacja aplikacji w języku Java nie ogranicza jej pod względem platformy, pod którą program będzie używany.

Zapoznałem się z również z narzędziami wspomagającymi tworzenie zaawansowanych scen 3D, jakim jest 3D Studio MAX. Dowiedziałem się o trędach jakie są rozwijane w grafice komputerowej w dobie internetu. Zainteresował mnie format zapisu sceny do pliku tekstowego X3D dający szerokie perspektywy na przyszłość.

Podstawowym założeniem technologii Java jest uniwersalność. Java dostarcza bardzo bogatą bibliotekę do tworzenia grafiki trójwymiarowej - Java3D, jest dobrze udokumentowana. W sieci internet można znaleźć wiele przykładów prezentujących zastosowanie tej technologii. Jest ciągle rozwijana i także daje szerokie perspektywy na przyszły rozwój aplikacji.

Dzięki połączeniu wszystkich wyżej wymienionych technik zaprojektowałem i napisałem aplikację, która wiernie odwzorowuje pracę rzeczywistych manipulatorów. Podejście obiektowe w języku programowania Java uelastycznia aplikację, którą łatwo można rozwijać w przyszłości.

Zastosowanie protokołu komunikacyjnego UDP/IP w aplikacji, daje możliwości połączenia się z każdą platformą.

6.1. Perspektywy rozwoju

Rozwój aplikacji może podążać w dowolnym kierunku:

1. Do sceny można dodać kolejne roboty, nie koniecznie manipulatory typu IRp-6.
2. Kod aplikacji można wykorzystać w aplikacji internetowej. Stworzyć aplet, który na potrzeby dydaktyczne wizualizowałby scenę robotów.
3. Funkcja nagrywania ruchu robotów.

4. Na podstawie poznanych technologii 3D możnaby zaprojektować strukturę ramową, umożliwiającą definicję składu wirtualnego świata. Umożliwiałby definiowanie i osadzanie w nim rożnego typu robotów, przeszkód.

Przedstawiłem tylko kilka pomysłów rozwoju aplikacji. Myślę, że każdy wtajemniczonych w badania w wydziałowym laboratorium podsunąłby ciekawy pomysł.

Bibliografia

[1] *http://www.zap-robotyka.com.pl/irp6_10.html*

[2] C. Zieliński, W. Szynkiewicz, T. Winiarski, and T. Kornuta - „MRROC++ based system description", Technical Report 06-9, IAiIS, Warsaw, 2006.

[3] Michele Matossian - „Po prostu 3ds max 5"

[4] *http://web3d.org/x3d*

[5] *http://java.sun.com/products/java-media/3D/forDevelopers/j3dguide/j3dTOC.doc.html*

[6] Grzegorz Górski - „Wizualizacja pracy robota", praca inżynierska, PW IAiIS , Warszawa 2004.

[7] *http://java.sun.com/products/java-media/3D/download.html*

[8] *http://www.xj3d.org/snapshots.html*