Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Leszek Bartkowski,
Mikołaj Dądela,
Paweł Marczewski,
Jan Szejko
Nr albumu: 262488, 262484, 262952, 248325

Netacka 3D - sieciowa gra czasu rzeczywistego na platformy PC i Android

Praca licencjacka
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dra Janusza Jabłonowskiego
Instytut Informatyki

Czerwiec 2010

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora (autorów) pracy

Streszczenie

Netacka3D to gra sieciowa czasu rzeczywistego, w której gracze sterują wstęgami umieszczonymi na trójwymiarowej planszy. Celem gry jest jak najdłuższe prowadzenie własnej wstęgi, tak, aby nie zderzyła się ze wstęgami innych graczy ani ze ścianami.

Gra występuje w dwóch wersjach - na platformy PC (Java SE) i Android.

Słowa kluczowe

gra komputerowa, 3D, Android, OpenGL, octree, wykrywanie kolizji

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

D. Software
D.m. Miscellaneous

Tytuł pracy w języku angielskim

Netacka 3D - a real-time network game for PC and Android platforms

Spis treści

Rozdział 1

Opis projektu

1.1. Definicje

klient
: aplikacja łącząca się z serwerem i umożliwiająca graczom udział w grze

serwer
: aplikacja zajmująca się prowadzeniem gry i wysyłaniem oraz odbieraniem informacji o jej przebiegu od klientów

plansza
: fragment (wirtualnej) przestrzeni trójwymiarowej wraz z jego granicami (ścianami) i poruszającymi się w nim wstęgami prowadzonymi przez graczy

gracz
: podmiot kierujący wstęgą i biorący udział w grze za pośrednictwem klienta

bot
: gracz sterowany przez komputer

Java SE
: Java Platform, Standard Edition — podstawowa wersja platformy Java [JSE]

Android
: platforma dla telefonów komórkowych oparta na systemie Linux

1.2. Wizja

Naszą inspiracją do napisania Netacki 3D była gra Zatacka [ADK]. Zatacka (oraz jej klon napisany przez Pawła Marczewskiego, Netacka [NET]) to prosta wieloosobowa gra zręcznościowa. Gra w Zatackę polegała na kierowaniu kolorową, wydłużającą się linią z użyciem dwóch klawiszy (jeden skręcał w prawo, drugi w lewo), tak, aby nie zderzyć się z liniami innych graczy. Netacka dodaje do rozgrywki możliwość gry przez sieć.

Netacka 3D przenosi tę mechanikę w przestrzeń trójwymiarową. Zmieniło to wygląd gry - linie stały się wstęgami, obszar gry przestał być płaski i stał się bryłą - jednak cel i zasady pozostały takie same.

1.3. Mechanika gry

Gra składa się z wielu krótkich rund.

Podczas każdej rundy każdy z graczy kieruje końcem swojej wstęgi. Wstęga "rośnie" w stałym tempie, i gracz może, poprzez zmiany kierunku tego końca, skręcać. Możliwy jest skręt w trzech kierunkach, odpowiadających obrotom wokół trzech osi: X, Y i Z. Steruje się sześcioma przyciskami, gdzie każdy przycisk zagina wstęgę w danej osi - kąt zagięcia jest stały; nie ma tu żadnych "poślizgów". Kiedy gracz nic nie naciska, wstęga przedłuża się prosto do przodu.

Po rozbiciu się o którąś wstęgę lub brzeg planszy gracz wypada z gry do końca rundy, a pozostali gracze (ci, którzy jeszcze się nie rozbili) dostają po punkcie. Po ustalonej liczbie rund lub osiągnięciu przez któregoś z graczy określonej liczby punktów, punkty się podlicza i zwycięzcą zostaje ten, kto zdobył ich najwięcej.

1.4. Możliwości dostosowywania gry

- **Ustawienia gry**:
  - szybkość gry
  - **plansza** (dowolna bryła, z dodatkową możliwością przeskalowania)
  - promień skrętu
  - szerokość wstęg
  - limit punktów
- Ustawienia gracza:
  - nick
  - kolor
- **Boty**: możliwość podłączenia do gry komputerowego przeciwnika. Gra ma API pozwalające na łatwe pisanie własnych botów.

1.5. Motywacje i cele

Tworząc nasz projekt, chcieliśmy pokazać, że niektóre gry 2-wymiarowe można przenieść w trójwymiarową przestrzeń. Dzięki temu otrzymujemy ciekawe zmiany w mechanice gry. Gra Zatacka była na przykład przez wielu graczy uznawana za grę strategiczną, bo można w niej było "odgrodzić się" od innych poprzez odpowiednie manewrowanie na początku rundy. W wersji trójwymiarowej przestaje to być możliwe, więc gra zyskuje charakter bardziej zręcznościowy.

1.6. Lista produktów projektu

- gra na PC
- gra na Androida

Podstawowa funkcjonalność jest dostępna na obu platformach, ale tylko w wersji PC możliwe jest założenie serwera.

1.7. Licencja

Gra jest przez nas udostępniana na licencji Apache 2.0 [APA]. Jest to licencja typu open source pozwalająca na dalsze wykorzystywanie kodu w projektach zamkniętych.

Rozdział 2

Opis zastosowanych rozwiązań

2.0.1. Technologia

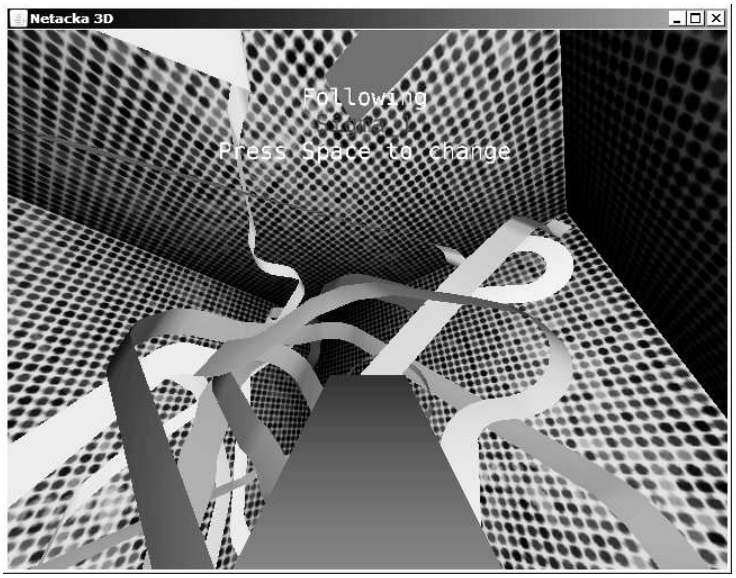
Gra występuje w dwóch wersjach - na platformy Java SE (zwana dalej PC) i Android. Na obu platformach dostępny jest język Java, a także część bibliotek standardowych Javy. Umożliwiło nam to napisanie głównych modułów gry w sposób niezależny od platformy.

2.0.2. Interfejs użytkownika

Interfejs użytkownika w wersji PC został wykonany z użyciem środowiska NetBeans i wykorzystuje bibliotekę Swing.

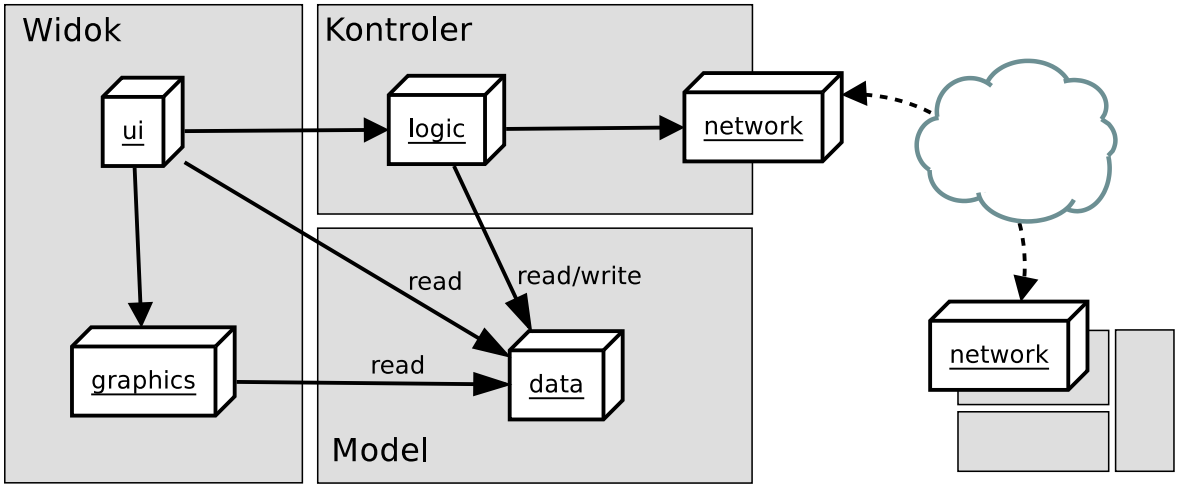
Wersja Android używa do wyświetlania interfejsu standardowych bibliotek dostępnych na tej platformie.

2.0.3. Grafika



Rysunek 2.1: Trójwymiarowy widok gry w Netacce 3D

W wersji PC moduły grafiki korzystają z biblioteki JOGL [JOGL] realizującej standard OpenGL. JOGL umożliwia aplikacjom w Javie wywoływanie odpowiedniego kodu natywnego,



Rysunek 2.2: Architektura wewnętrzna Netacki 3D

który dostępny jest dla większości systemów Javy SE (m. in. Windows, Linux, OS X).

Wersja Android korzysta z wbudowanej implementacji standardu OpenGL ES [OES].

2.0.4. Obliczenia

Do obliczeń wykorzystaliśmy bibliotekę javax.vecmath, pozwalającą wykonywać działania na wektorach, macierzach i kwaternionach.

2.0.5. Komunikacja sieciowa

Moduł komunikacji sieciowej wykorzystuje standardową bibliotekę `java.nio.channels` [NIO]. Proste dane liczbowe przesyłane są bezpośrednio, natomiast do serializacji obiektów (takich jak np. parametry gry, lista graczy) użyliśmy biblioteki Jackson [JAC].

2.1. Opis modułów

Architektura naszej aplikacji oparta jest na wzorcu **MVC** (model-widok-kontroler). Rolę widoku pełni tutaj interfejs graficzny (zależny od platformy; są to moduły `ui` i `graphics`), modelem są klasy reprezentujące dane gry (moduł `data`), a część kontrolerową stanowią moduły `logic` i `network`.

Podział na moduły nie wyróżnia u nas części klienckiej i serwerowej. Jest to spowodowane tym, że zdecydowana większość kodu jest wykorzystywana po obu stronach. Główne różnice między klientem i serwerem pojawiają się w modułach `logic` oraz `network`, w których odpowiednie klasy istnieją w dwóch odpowiadających sobie wersjach.

Zadbaliśmy o to, żeby zależności między modułami były jednostronne. Jeżeli komunikacja między pewnymi modułami występuje w obie strony, stosowany jest wzorzec projektowy Obserwator. Moduł nadrzędny rejestruje u podrzędnego *listener* - obiekt reagujący na określone komunikaty.

2.1.1. Dane (`data`)

Moduł zawierający struktury danych gry (wstęgi, segmenty, dane o planszy i graczach) i odpowiedzialny za algorytmiczną część logiki gry (wykrywanie kolizji). Struktury z modułu `data`

używane są we wszystkich pozostałych modułach. Tam, gdzie jest to możliwe, struktury danych nie ulegają zmianom, dzięki czemu unikamy potencjalnych problemów z synchronizacją pomiędzy wątkami.

Wstęgi zostawiane przez graczy przechowywane są jako ciągi *segmentów*. Segment wstęgi składa się z dwóch trójkątów (niekoniecznie znajdujących się w jednej płaszczyźnie). Oprócz czterech wierzchołków do danych związanych z segmentem należą początkowa i końcowa pozycja gracza, oraz obrót w przestrzeni (reprezentowany przez kwaternion).

Kod grafiki liczy położenie oraz obrót kamery, wykorzystując powyższe wartości do liniowej interpolacji, co daje złudzenie płynnego ruchu. Rysowanie kolejnych segmentów z tymi samymi wektorami normalnymi na wspólnych wierzchołkach pozwala na gładkie cieniowanie i daje złudzenie gładkości wstęgi.

Wciśnięcie przez gracza przycisku powoduje, że następny segment ma zmieniony kształt, co przy odpowiedniej ilości segmentów na sekundę pozwala na płynne sterowanie.

Podmoduły

`data.arena` — zajmuje się ładowaniem geometrii planszy z zewnętrznych plików. Na chwilę obecną wspierany jest tylko format OBJ [OBJ] z materiałami MTL [MTL]. Działa też ładowanie tekstur z zewnętrznych plików. Taki importer pozwala nam używać aren zaprojektowanych w zewnętrznych programach do obróbki grafiki 3D, jak np. Blender.

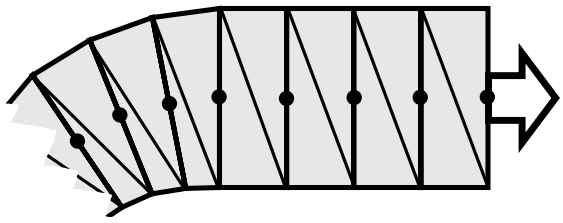
`data.serialization` — umożliwia serializację danych gry do formatu JSON, co pozwala na ich łatwe przesyłanie przez sieć. Skorzystaliśmy z biblioteki Jackson.

System wykrywania kolizji

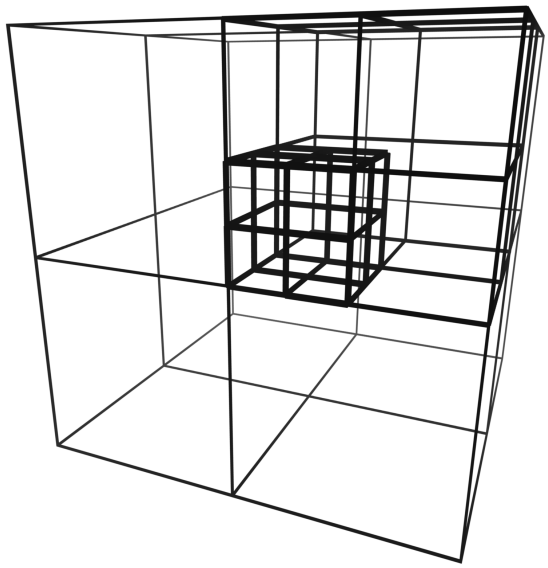
Mechanika gry wymusza ciągłe sprawdzanie pozycji gracza pod względem kolizji - zarówno z zawczasu ustaloną geometrią poziomu, jak i z pojawiającymi się nowymi segmentami wstęgi. Uwzględnianie za każdym razem wszystkich trójkątów na planszy powodowałoby zbyt duże obciążenie procesora. W celu poprawienia wydajności używane jest więc drzewo ósemkowe (*octree*).

Węzeł drzewa opisuje prostopadłościan przestrzeni gry. Jest w nim przechowywana tablica trójkątów. W momencie, gdy zawartość węzła przekroczy ustaloną wartość progową, jest on dzielony na osiem mniejszych, a jego zawartość odpowiednio do nich przekazywana.

Wobec tego sprawdzenie kolizji dla danego gracza wymaga przejrzenia tylko najbliższych mu węzłów.



Rysunek 2.3: Wstęga złożona z segmentów. Przy każdym segmencie zaznaczona została pozycja gracza; przy ostatnim dodatkowo ukazany został kierunek gracza.



Rysunek 2.4: Graficzna reprezentacja drzewa ósemkowego. Węzły, w których znajduje się więcej segmentów, podzielone są na mniejsze części.

2.1.2. Sterowanie przebiegiem gry (`logic`)

Moduł zawierający klasy `ServerController` oraz `ClientController` (zwane dalej kontrolerami), odpowiedzialne za przebieg rozgrywki i kierowanie pracą wszystkich pozostałych modułów. Kontrolery korzystają z modułów `data` oraz `network`, natomiast przyjmują polecenia i powiadamiają o zmianach moduły `ui` i `graphics`.

- `ServerController` — do jego odpowiedzialności należy kontrolowanie upływu czasu, zarządzanie graczami i koordynowanie ogólnego przebiegu rozgrywki. Z przyczyn wydajnościowych nie jest używany w wersji Android.
- `ClientController` — odbiera dane przychodzące od serwera i powiadamia o nich wszystkie pozostałe moduły. Jest używany w obu wersjach gry.

Wątek kontrolera

Każdy kontroler na początku działania uruchamia własny wątek, który nadzoruje przebieg rozgrywki. W wątku tym odbywa się również działanie komunikacji sieciowej oraz powiadamianie modułu grafiki o zmianie stanu gry - dzięki temu tylko jeden wątek ma dostęp do danych gry i znika większość problemów z synchronizacją.

Boty

Moduł `logic` zawiera też podmoduł `logic.bots`, który zawiera zmodyfikowaną wersję kontrolera klienckiego, umożliwiającego łatwe programowanie botów - komputerowych graczy. W pakiecie umieściliśmy bota Sigma, korzystającego z prostej sztucznej inteligencji, jak również 2 proste, przykładowe boty, których używaliśmy do testów.

2.1.3. Obsługa sieci (`network`)

Moduł zajmujący się komunikacją sieciową.

Gra ma architekturę klient-serwer - dedykowany serwer udostępnia planszę do gry, a następnie podłączają się do niego klienci i rozpoczynają rozgrywkę. Klienci na bieżąco przekazują do serwera informacje o swoich ruchach, a serwer przekazuje je dalej innym klientom. Aplikacja kliencka została też wyposażona w prostą kompensację lagów.

Wybieranie serwera ułatwia wbudowany moduł automatycznie wyszukujący serwery w sieci lokalnej.

Moduł korzysta z Javowej biblioteki `java.nio.channels` [NIO] w sposób całkowicie asynchroniczny, co sprawia, że nie są potrzebne dodatkowe wątki.

Protokół

Na potrzeby komunikacji klienta z serwerem opracowaliśmy protokół oparty o UDP. Jest to standardowy wybór w przypadku gier czasu rzeczywistego, w których najważniejsza jest szybkość dostarczania danych.

Przed możliwą w UDP utratą pakietów zabezpieczyliśmy się wprowadzając następujące mechanizmy:

- potwierdzanie przez klienta informacji o zmianie stanu gry (np. przejście do nowej rundy), i przesyłanie przez serwer tych informacji na nowo
- żądania retransmisji segmentów, wysyłane przez klienta w przypadku wykrycia niespójności danych.

Protokół umożliwia wyszukiwanie serwerów w sieci lokalnej, co realizowane jest przez wysłanie pakietu UDP broadcast na standardowym porcie.

2.1.4. Interfejs użytkownika (`ui`)

Moduł realizujący interfejs użytkownika występuje w dwóch wersjach:

`ui.pc` -- okienkowy interfejs wykorzystujący bibliotekę Swing. Zawiera proste menu główne, z którego można m. in. rozpocząć nową grę, dołączyć się do trwającej rozgrywki. Przy uruchamianiu nowej gry jest również dostępny edytor ustawień meczu.

`ui.android` -- implementacja wykorzystująca biblioteki Androida. Jest nieco prostsza od wersji PC, gdyż nie pozwala na zakładanie nowych gier.

2.1.5. Grafika 3D (`graphics`)

Moduł korzystający z interfejsu OpenGL (dokładniej części wspólnej standardowego OpenGL i OpenGL ES) do wyświetlania grafiki. Między wersją Android i PC kodu występują różnice. Wobec tego zdecydowaliśmy, że najlepszym rozwiązaniem będzie generowanie przez preprocesor wersji Android na podstawie wersji PC.

Moduł ten zajmuje się również wyświetlaniem na ekranie komunikatów oraz statystyk. Ponieważ OpenGL nie ma wbudowanych mechanizmów wyświetlania tekstu, odpowiednia klasa ładuje czcionkę w formie bitmapy, a następnie wyświetla napisy jako teksturowane wielokąty.

2.1.6. Pozostałe narzędzia (`misc`)

W module `misc` zawarliśmy narzędzia, z których korzystaliśmy w naszym kodzie. Najważniejszym z nich jest *logger*, czyli narzędzie umożliwiające łatwe śledzenie wiadomości wysyłanych przez program na konsolę. Dzięki niemu każda wiadomość wysyłana przez program jest opatrzona nazwą wątku i miejscem w kodzie, z którego pochodzi (te informacje pobiera automatycznie poprzez Javowy mechanizm refleksji). Dodatkowo nasz `logger` obsługuje kilka różnych priorytetów wiadomości, umożliwiając filtrowanie i pozwalając na ich łatwiejsze czytanie.

Drugim narzędziem z tego pakietu jest wspólny interfejs do obsługi plików dla PC i Androida. Potrzebowaliśmy takiego interfejsu, gdyż Android nie pozwala na bezpośrednie otwieranie plików. Narzuca on pewne ograniczenia (np. na nazwy plików), ale za to umożliwia uniknięcie duplikacji kodu. Dane pomiędzy projektem Androida i PC są synchronizowane automatycznie, przez skrypt, zintegrowany z Eclipse.

2.2. Napotkane problemy

2.2.1. Użycie bibliotek OpenGL na obu platformach

Platforma Android udostępnia interfejs OpenGL w wersji ES, natomiast użyta przez nas w wersji PC biblioteka JOGL udostępnia standardową edycję OpenGL. Mimo tego, że z pewnymi wyjątkami funkcjonalność OpenGL ES jest podzbiorem standardowej, Javowe interfejsy różniły się - wskutek niezgodności nazw pakietów, oraz nazw i sposobów użycia niektórych metod, nie było możliwe skompilowanie tego samego kodu źródłowego na obu platformach.

Możliwym rozwiązaniem byłoby tu potraktowanie kodu OpenGL w naszej aplikacji jako zależnego od platformy i napisanie go w dwóch wersjach. Uznaliśmy jednak, że nieznaczne różnice w API nie usprawiedliwiają duplikacji kodu i związanych z nią problemów.

Innym możliwym podejściem byłoby napisanie warstwy pośredniej, pozwalającej wywoływać funkcje OpenGL w sposób niezależny od platformy. Ten sposób również odrzuciliśmy jako mało efektywny.

Aby rozwiązać problem, stworzyliśmy prosty preprocesor w języku Python. Kod używający OpenGL powstał w wersji dla platformy PC, ale zawierał komentarze z dyrektywami preprocesora mówiącymi, jak analogiczny kod wygenerować dla systemu Android. Preprocesor został zintegrowany z automatycznym procesem kompilacji w środowisku Eclipse.

2.2.2. Problemy z komunikacją sieciową

Pierwsza wersja modułu komunikacji sieciowej realizowała hybrydowy protokół, wykorzystujący zarówno mechanizmy TCP, jak UDP. W założeniu miało to pozwolić skorzystać z zapewnianej przez TCP niezawodności dostarczania danych dla wymagających potwierdzenia informacji. Umożliwiło to również zrealizowanie części TCP jako protokołu tekstowego opartego o JSON.

Okazało się jednak, że komunikacja TCP powoduje znacząco większe opóźnienia. Problem sprawiało również buforowanie strumienia TCP, którego nie udało nam się wyłączyć we wszystkich konfiguracjach.

Opracowaliśmy więc nową wersję protokołu, korzystającą wyłącznie z UDP i realizującą własny, prosty mechanizm potwierdzeń. Oprócz rozwiązania powyższych problemów, zmiany w protokole uprościły także znacząco implementację.

2.2.3. Obsługa wielu klawiszy jednocześnie

Okazało się, że na większości klawiatur PC nie jest możliwy w grze obrót w trzech osiach naraz. Jest to spowodowane sprzętowym zjawiskiem zwanym *keyboard ghosting* [GHO], polegającym na nierozpoznawaniu przez klawiaturę pewnych kombinacji 3 klawiszy.

Możliwym rozwiązaniem byłaby zmiana konfiguracji klawiszy w grze. Uznaliśmy jednak, że inny sposób sterowania byłby mniej wygodny, a problem 3 klawiszy można zignorować jako mało widoczny i rzadko wpływający na wygodę gry.

2.2.4. Czytelność danych wypisywanych na konsolę

Wraz z rozwojem projektu wiadomości wypisywane przez program na konsolę stawały się coraz bardziej nieczytelne i nieuporządkowane. Sprawiło to, że potrzebowaliśmy narzędzia, które by pozwoliło nam łatwo śledzić:

- z którego miejsca w kodzie została wypisana dana wiadomość,
- z którego wątku pochodzi,
- ile czasu mija pomiędzy wiadomościami,
- jak ważna jest dana wiadomość (potrzebowaliśmy takiego wypisywania, żeby można było szybko "wyłowić" ważne wiadomości spośród mniej ważnych).

Aby rozwiązać powyższe problemy, napisaliśmy moduł `Logger`. Wykonuje on wymienione zadania automatycznie, używając m.in. mechanizmu refleksji — w kodzie piszemy tylko wiadomość do wypisania i jej ważność.

Dodatkowo `Logger` ma różne implementacje obsługi wyjścia dla obu platform.

2.2.5. Problemy z wydajnością

Platforma Android i ograniczona wydajność dostępnego sprzętu (a także ograniczona wydajność emulatora) stanowiły dla naszego projektu poważne ograniczenie. W procesie dostosowywania aplikacji okazało się, że pewne niezauważalne na PC zadania zajmują na Androidzie zaskakująco dużo czasu.

Szczególnie niska okazała się wydajność operacji związanych z przetwarzaniem tekstu, Przykładem może być Androidowa implementacja metody String.format(), która przy każdym użyciu wykonywała alokacje nowych obiektów i przetwarzała ustawienia regionalne. W efekcie dużą część czasu procesora zabierał moduł logujący, korzystający z tych metod.

Podobny problem wystąpił przy ładowaniu poziomów. Są one przechowywane w tekstowym formacie OBJ, ich ładowanie wiąże się więc z wczytywaniem liczb. Standardowe procedury takie jak `Float.parseFloat()` również okazały się wyjątkowo nieefektywne - do tego stopnia, że załadowanie prostego poziomu z torusem zajmowało 1 sekund.

Trudności sprawiała używana przez nas do serializacji danych o grze biblioteka Jackson [JAC]. O ile jej szybkość działania okazała się zadowalająca, pierwsze jej użycie dla danej klasy powodowało dość długą inicjalizację. Rozwiązaniem okazał się "rozruch" biblioteki Jackson w oddzielnym wątku uruchamianym przy starcie programu.

Androidowa implementacja OpenGL wymaga użycia do danych tzw. bezpośrednich buforów pamięci. Nie podlegają one kontroli Javowego *garbage collectora*, co przyspiesza wczytywanie ich przez OpenGL, natomiast bardzo zwalnia pisanie do nich przez program. Uzyskanie odpowiedniej wydajności wymagało bardzo ostrożnego korzystania z buforów pamięci.

Na szczęście dzięki narzędziu profilującemu na platformę Android mogliśmy zidentyfikować "wąskie gardła", sprawdzić, które metody zabierają najwięcej niepotrzebnego czasu, i wyeliminować problemy.

2.3. Organizacja pracy zespołowej nad projektem

2.3.1. Początki

Pierwsze iteracje projektu odbywały się bez wyraźnego podziału odpowiedzialności pomiędzy członków zespołu. Nie prowadziliśmy także dokładnej dokumentacji, ograniczając się do ustalenia wysokopoziomowego podziału na moduły. Wkrótce jednak okazało się, że złożoność projektu zmusza nas do dokładniejszego ustalenia sposobu pracy i bardziej szczegółowego planowania architektury.

2.3.2. Podział prac nad modułami na 3 etapy

Aby zorganizować pracę naszego zespołu, po podziale na moduły ustaliliśmy, że tworzenie każdego modułu ma składać się z trzech etapów: analiza, projekt i kod. Analiza oznacza przemyślenie odpowiedzialności modułu i jego przypadków użycia. Projekt to dokładne opisanie klas i metod, które powinny się znaleźć w danym module. Natomiast kod to praktyczna realizacja tego, co zostało napisane w projekcie. Trzymaliśmy się zasady, że osoba wykonująca analizę nie powinna pisać projektu, a osoba pisząca projekt nie powinna implementować danego modułu. (Ustalony podział został opisany w tabelce w rozdziale poniżej).

2.3.3. Narzędzia zarządzania projektem

Większość projektu powstała w środowisku **Eclipse**. Jego wybór podyktowany był dostępnością wtyczki do programowania pod Androidem. Niestety Eclipse nie pozwalał na łatwą edycję interfejsu korzystającego z biblioteki Swing. Ta część programu powstała więc w środowisku **NetBeans**.

Cały kod projektu przechowywaliśmy od początku we wspólnym repozytorium **SVN**. SVN bardzo dobrze sprawdził się dla naszych potrzeb. Przez kilka miesięcy pracy nad projektem wysłaliśmy do repozytorium około 700 zmian.

Używaliśmy systemu **Trac** do śledzenia zmian w projekcie i organizowania zadań. Podczas tworzenia Netacki 3D utworzyliśmy w tym systemie około 50 zadań.

Dokumenty Google bardzo pomogły nam przy tworzeniu dokumentacji projektu. Przydała się możliwość szybkiej edycji oraz szybkiego współdzielenia edytowanych treści. Zmiany można było łatwo śledzić dzięki wbudowaniu systemu wersji. Możliwość wstawiania komentarzy z oznaczeniem autora pozwalała toczyć dyskusję nad odpowiednimi aspektami architektury.

Rozdział 3

Wkład własny

3.1. Macierz odpowiedzialności

Zgodnie z podziałem opisanym wcześniej (analiza, projekt, kod), uzgodniliśmy następujący rozkład odpowiedzialności:

	Analiza	**Projekt**	**Kod**
Obsługa danych (`data`)	Paweł Marczewski	Jan Szejko	Paweł Marczewski
Mechanika gry (`logic`)	Leszek Bartkowski	Paweł Marczewski	Mikołaj Dądela
Sieć (net)	Leszek Bartkowski	Paweł Marczewski	Leszek Bartkowski
UI, wersja Android	Mikołaj Dądela	Leszek Bartkowski	Jan Szejko
UI, wersja PC	Mikołaj Dądela	Paweł Marczewski	Jan Szejko
Grafika 3D (`graphics`)	Jan Szejko	Mikołaj Dądela	Paweł Marczewski

Narzędzia - Mikołaj Dądela

Serwer - Mikołaj Dądela

3.2. Leszek Bartkowski

3.2.1. Analiza modułu `logic`

Analiza części kodu zajmującego się logiką gry została przedstawiona w dokumencie Logic, w dziale Analiza. Opisano w niej zakres odpowiedzialności, przypadki użycia oraz wymagane funkcjonalności. Wyróżniono dwa byty, które miały stanowić główną zawartość tego modułu: `Game` oraz `Controller`. Jako zadanie tego pierwszego określono pośredniczenie w dostępie do modelu, czyli danych gry. Miał on w tym celu udostępniać interfejs umożliwiający modyfikowanie stanu gry i graczy. Rolą `Controller`a miało być sterowanie przebiegiem gry przy pomocy tego interfejsu oraz komunikacja z pozostałymi modułami. Ze względu na centralne umiejscowienie `Controller`a w architekturze gry, w ramach analizy sporządzono też 3 diagramy, umieszczone w dokumencie "Interakcje między modułami", obrazujące działanie modułu, z uwzględnieniem różnic między kontrolerem klienta, a serwera.

3.2.2. Analiza modułu net

Analizę modułu służącego do komunikacji sieciowej umieszczono w dokumencie Network, w dziale Analiza. Zawiera ona opis przewidywanego działania sieci, odpowiedzialności oraz możliwe przypadki użycia. Wyszczególnia różnicę w działaniu tego modułu u serwera oraz u klienta. Proponuje możliwe rozwiązania dla protokołu sieciowego, którego szczegółowa wersja miała zostać później wykonana w projekcie. Prezentuje również przykładowy schemat komunikacji między klientem a serwerem. Zwraca też uwagę na kwestie identyfikacji i uwierzytelniania graczy.

3.2.3. Projekt modułu `ui`, wersja dla Androida

Projekt modułu `ui` znajduje się w dokumencie UI, w dziale Projekt - Android. We wstępie przedstawia on pokrótce najważniejsze w Androidowym frameworku klasy oraz zasady tworzenia interfejsu użytkownika dla tej platformy. W dalszej części opisuje jak powinno być zbudowane menu, wskazując klasy i metody, które należałoby użyć. W ostatniej części umieszczone zostały odnośniki do dokumentacji, samouczków oraz przykładów kodu androidowego, aby osoba implementująca moduł wiedziała, gdzie szukać wszelkich technicznych informacji.

3.2.4. Implementacja początkowej wersji menu dla UI Android

Zamiast pisać bardzo szczegółowy projekt UI dla Androida, postanowiłem lepiej wykorzystać czas implementując pierwszą wersję interfejsu. Utworzono klasę Netacka3D będącą punktem wyjścia dla androidowej wersji aplikacji oraz przy pomocy XMLa i udostępnianego przez androidową wtyczkę UIBuildera wyprodukowano proste menu. Uprzątnięto tez kod pozostały po wersji prototypowej i zintegrowano go z pozostałymi modułami (kontrolerem i grafiką).

3.2.5. Implementacja modułu net

Większa część kodu sieciowego znajduje się w klasach `ServerNetworkNode` i `ClientNetworkNode`, które odpowiadają za działanie sieci po stronach odpowiednio serwera i klienta. Obie korzystają ze statycznej klasy `UDPProtocol`, która zawiera pola reprezentujące stałe z protokołu oraz metody wspomagające odczyt/zapis obiektów do bufora z bajtami. Istotną część modułu stanowiła też klasa `TCPMessage`, która wyszła z użycia po gruntownej modyfikacji protokołu i związanego z nim refactoringu kodu sieciowego.

Do implementacji sieci skorzystano z pakietu `java.nio.channels` [NIO]. Opiera się on na tzw. `Channel`ach, czyli kanałach, które są wydajniejsze niż standardowe rozwiązania, oferowane przez `java.net`. Ponadto kanały te mogą być wykorzystywane w sposób asynchroniczny, co pozwoliło zrezygnować z tworzenia nowych wątków w kodzie sieciowym - kod jest wykonywany tylko przez wątek kontrolera.

W module sieciowym znalazła się również wyszukiwarka serwerów. Wysyła ona na adres broadcastowy zapytanie o grę. Na podstawie otrzymanych na ten komunikat odpowiedzi klient jest w stanie poznać adresy serwerów z grą w sieci lokalnej i przyłączyć się do nich.

W trakcie tworzenia aplikacji pojawiły się kilka razy problemy związane z siecią. Zaobserwowano nieoczekiwane działanie pewnych metod z biblioteki `java.nio`, np. metoda `Selector.select(int timeout)` zamiast blokować się do czasu otrzymania zdarzenia, lub upłynięcia timeout milisekund, natychmiast kończyła się i zwracała 0, co według dokumentacji nigdy nie powinno się zdarzyć. Problem udało się rozwiązać wprowadzając pewne drobne

poprawki w kodzie (podsunięte przez intuicje, dokumentacja bowiem nie była szczególnie przydatna pod tym względem). Oprócz tego na Androidzie z nieznanych przyczyn klient nie otrzymywał pakietów UDP od serwera, poprawę zaobserwowano dopiero, gdy zamiast używać metod `connect(adres)` i `write(dane)`, zastosowano wszędzie tylko metody `send(dane,adres)`. To również było nieoczekiwane, ponieważ według dokumentacji oba rozwiązania powinny być równoważne. Tymczasem praktyka pokazała co innego.

Kod sieciowy po tym jak został ukończony przechodził jeszcze poważne zmiany. Największą gdy okazało się, że przy większym obciążeniu sieci pewnym komunikatom TCP zdarzało się zatrzymywać na buforach. Przez to przychodziły przesunięciem i gra się rozsynchronizowywała. Pomimo licznych prób, stosowania różnych metod z bibliotek sieciowych i szukania informacji w Internecie przyczyn problemu, ani rozwiązania nie udało się znaleźć. W konsekwencji protokół został zmodyfikowany, zrezygnowano z TCP i całość komunikacji oparto o UDP. Wiązało się to z pewnymi istotnymi zmianami w kodzie oraz koniecznością opracowania i zaimplementowania mechanizmu potwierdzeń, który w przypadku TCP ma się "za darmo". Wymagało to odrobinę wysiłku, ale pozwoliło rozwiązać problem i przywrócić należyte działanie sieci. Pozytywnym skutkiem ubocznym był też bardziej spójny kod oraz bardziej przejrzysty protokół sieciowy.

3.3. Mikołaj Dądela

3.3.1. Administrowanie serwerem

Do moich obowiązków należało administrowanie serwerem, na którym działały narzędzia wspomagające naszą pracę zespołową - przede wszystkim chodziło tu o uruchomienie i konfigurację serwera HTTP i z jego pomocą SVN i Trac-a. Dodatkowo dostosowałem Trac-a do pracy zgodnie z naszym wcześniej opisanym w rozdziale 2.3.2 podziałem prac.

3.3.2. Kod modułu `logic`, boty

Zgodnie z tym, co zostało napisane wcześniej w tabelce - napisałem cały kod modułu `logic`, to jest: kontroler kliencki i serwerowy, wraz z kilkoma klasami pomocniczymi i interfejsem do powiadomień. Napisałem też boty: początkowo dwa proste, uzywane przez nas do testów, a później nieco bardziej wyrafinowanego (`Sigma`).

Działanie `Sigmy` zasadza się na tym, że bot w kazdym kroku sprawdza, jak daleko można pojechać w każdym z zadanych 5 kierunków (prosto, w górę, w dół, w lewo i w prawo). Aby to sprawdzanie nie powodowało spadku wydajności działania gry, odpowiednie ciągi segmentów zostały przybliżone jednym dłuższym, o nieco zmienionym kącie skrętu. Po sprawdzeniu wybierany jest zawsze ten kierunek, w którym można jechać najdłużej. Taka implementacja radzi sobie dosyć dobrze w większości przypadków.

3.3.3. Analiza modułu `ui`

Zajmowałem się analizą modułu `ui` (obu wersji) - w praktyce chodziło o sporządzenie rysunków, jakie komponenty powinny się znaleźć w GUI i jaki powinien być ich układ.

3.3.4. Projekt modułu `graphics`

Rozdzieliłem odpowiedzialność modułu `graphics` na 3 klasy. Zaprojektowałem obsługę komunikacji pomiędzy kontrolerem a grafiką.

Przy okazji pisania tego projektu powstał problem, jak poradzić sobie z tym, że kontroler ma dyskretny, a grafika ma ciągły upływ czasu. Innymi słowy, chodziło o to, że podczas jednego kroku grafika mogła narysować wiele klatek i należało to uwzględnić przy rysowaniu ostatniego segmentu i obliczeniu pozycji kamery.

3.3.5. Testy

Przygotowałem projekt do działania z biblioteką JUnit i stworzyłem kilka prostych testów, testujących niektóre klasy z modułu `data` i ogólny test kontrolera, uwzględniający wiele różnych przypadków.

3.3.6. Preprocesor

W związku z problemem, opisanym w rozdziale 2.2.1, napisałem w Pythonie preprocesor, który na podstawie specjalnych komentarzy przetwarza wersję PC kodu grafiki na jego wersję dla Androida. Następnie zintegrowałem go z procesem automatycznej kompilacji w środowisku Eclipse.

Najważniejszą funkcjonalnością preprocesora jest dodawanie lub ukrywanie odpowiednich linijek kodu. Zostało to zaimplementowane w ten sposób, że w odpowiednich linijkach dodawane są komentarze postaci `//#.`*tag*. Jeśli taki tag znajduje się na początku linii, zostaje przesunięty na koniec, co powoduje ''odsłonięcie'' danej linii kodu; jeśli został umieszczony na końcu linii, preprocesor przesuwa go na początek, ''zasłaniając'' dany fragment kodu.

3.3.7. `Logger`

Napisałem `Logger`a na potrzeby projektu. Jego działanie zostało szczegółowo opisane powyżej, w rozdziale 2.2.4.

3.3.8. Importer OBJ

Napisałem importer plików OBJ [OBJ], aby umożliwić załadowanie do gry poziomu, przygotowanego w zewnętrznym programie. Importer ten prawie wcale nie korzysta z kodu naszego projektu, więc można go bez żadnego problemu wykorzystać później. Dodatkowo stworzyłem prosty unit-test, sprawdzający poprawność tego modułu.

3.3.9. FileIO

Napisałem klasę udostępniającą wspólne wejście-wyjście dla plików, którego mogliśmy używać na PC i na Androidzie. Pozwoliło to nam umieścić importowanie plików OBJ w części wspólnej wersji PC i Android.

3.3.10. Plansze

Stworzyłem na potrzeby gry kilka plansz w Blenderze [BLE]. W pakiecie gry zostały umieszczone plansze:

- kostka z teksturą
- torus - efekt pasków osiągnąłem poprzez przypisanie ścianom różnych materiałów
- octa - bryła o symetrii ośmiościanu
- icosi - bryła o symetrii dwudziestościanu

- cage - prostopadłościan przedzielony na pół kratami. Wymaga to od gracza użycia skrętu wokół osi Z (tj. jak korkociąg).

Przy eksporcie pliku do formatu OBJ pojawiły się problemy z nazwami plików - można je było rozwiązać poprzez zmianę nazw i ręczną edycję pliku.

3.4. Paweł Marczewski

3.4.1. Pomysł i prototyp

Netacka 3D pochodzi od napisanej przeze mnie gry Netacka [NET] (będącej oczywiście klonem wcześniejszych gier o tej samej mechanice). Przed rozpoczęciem pracy zespołowej napisałem w C++ prosty prototyp eksplorujący możliwości, jakie stwarza dla mechaniki Netacki trzeci wymiar. Stał się on podstawą wyboru tematu, a później pomógł ustalić jasną wizję projektu.

3.4.2. Koordynacja

Wyniesiona z prototypu dobra znajomość specyfiki problemu pozwoliła mi zająć się najwyższym poziomem architektury oraz koordynacją pracy członków zespołu. Byłem więc odpowiedzialny za działanie programu jako całości.

3.4.3. Projekt modułów `logic` i `network`

Należało do mnie zaprojektowanie modułów `logic` i `network`, oraz związanego z nimi schematu działania głównej pętli gry. Zadanie utrudniał fakt korzystania z danych gry przez praktycznie wszystkie moduły. Przy narzucającym się rozwiązaniu - jeden wątek do obsługi grafiki, jeden dla logiki, jeden (lub więcej) do komunikacji sieciowej - mogłoby to spowodować problemy z synchronizacją.

Rozdzielenie działania grafiki i reszty programu na oddzielne wątki było nieuniknione, udało się jednak uniknąć dalszego podziału. Wprowadzony ostatecznie przeze mnie projekt przesuwa całą komunikację sieciową do wątku logiki (co jest możliwe dzięki użyciu poleceń I/O w trybie asynchronicznym). Aby uniknąć kołowych zależności między modułami, komunikacja odbywa się z użyciem wzorca projektowego Obserwator - tak więc np. moduł logiki powiadamia moduł grafiki o zmianach za pomocą zarejestrowanego przez moduł grafiki *listenera*.

Schemat ten upraszcza także zarządzanie danymi gry - ich autorytatywną wersję posiada na własność moduł logiki. Dostęp do danych inne moduły mają tylko podczas wywołań odpowiednich metod przez ten moduł, a więc wyłącznie w wątku logiki. Pozwala to uniknąć problemów, jakie stworzyłoby współdzielenie danych przez różne wątki i spowalnianie dostępu do nich niepotrzebnymi muteksami. Natomiast w sytuacji, w której dany moduł potrzebuje danych w swoim wątku (np. do wyświetlenia na ekranie), są one kopiowane; bądź też - jeśli są niemodyfikowalne - kopiowana jest po prostu referencja do nich.

3.4.4. Projekt modułu UI PC

Zajmowałem się również projektem wersji PC interfejsu użytkownika, opartej o bibliotekę Swing. Główny widok gry oraz wszystkie menu korzystają z jednego elementu jako głównego kontenera, co w przyszłości pozwoli na łatwe umieszczenie Netacki 3D na stronie internetowej jako apletu Javy.

3.4.5. Projekt protokołu sieciowego

W ramach projektu modułu `network` opracowywałem protokół klient-serwer. Jego pierwsza wersja była dość skomplikowana i korzystała ze standardów TCP i UDP. Eksperyment z zastosowaniem TCP w grze czasu rzeczywistego okazał się niestety nieudany - problemy z uzyskaniem szybkiego czasu reakcji sprawiły, że zrezygnowałem z oferowanych przez TCP ułatwień i zaprojektowałem drugą wersję protokołu, opartą wyłącznie o UDP. W efekcie uzyskaliśmy przejrzyście napisany, szybki moduł komunikacji sieciowej sprawdzający się zarówno w sieciach lokalnych, jak i w Internecie.

3.4.6. Analiza i kod modułu `data`

Przeprowadziłem wysokopoziomową analizę, a później na podstawie sporządzonego z jej użyciem projektu napisałem moduł `data` odpowiedzialny za reprezentację danych o grze oraz ich obliczanie. Wykorzystałem w tym celu bibliotekę `javax.vecmath`, ułatwiającą obliczenia na wektorach, macierzach i kwaternionach. Orientacja gracza w przestrzeni przechowywana jest w reprezentacji kwaternionowej, aby ułatwić składanie obrotów oraz ich interpolację przy obliczaniu pozycji kamery.

3.4.7. Kod modułu `graphics`

Zakodowałem też moduł `graphics`, wykorzystujący OpenGL do rysowania planszy i segmentów, a także tekstu pojawiającego się w grze za pomocą czcionki bitmapowej.

Użycie biblioteki OpenGL w Javie jest dobrym przykładem styku kodu interpretowanego z natywnym. Było to źródłem problemów z wydajnością - OpenGL wymaga użycia buforów pamięci nie podlegających Javowemu *garbage collection* i związanej z nim relokacji pamięci. Jest to wygodne dla biblioteki, natomiast niewygodne dla programisty - dostęp od strony Javy do takich buforów jest znacznie spowolniony.

Przekazanie kontroli bibliotece natywnej utrudniło także śledzenie błędów związanych z przepełnieniem bufora. OpenGL nie sprawdza, czy dane mu przesłane mają odpowiedni rozmiar, co czasami powodowało rzadko spotykany w Javie komunikat "Segmentation fault".

3.5. Jan Szejko

3.5.1. Projekt modułu `data`

Zajmowałem się projektowaniem modułu `data`. Głównym problemem było tu opracowanie algorytmu wydajnego sprawdzania kolizji, tak by nie było trzeba za każdym razem sprawdzać, czy nowy segment nie przecina się ze wszystkimi innymi. Wykorzystałem tu schemat drzewa ósmekowego (ang. *octree*), który polega na dzieleniu planszy na fragmenty, jednocześnie pilnując, żeby w każdym fragmencie liczba wielokątów nie była zbyt duża (jeśli przekroczy pewien próg, to wykonywany jest kolejny podział). Na podstawie współrzędnych segmentu łatwo jest określić, z którymi obszarami się przecina i odzyskać z drzewa ósemkowego kolekcję wielokątów leżących w tych obszarach.

3.5.2. Analiza modułu `graphics`

Należała do mnie analiza modułu `graphics`. Tutaj jednym z głównych problemów było rozgraniczenie odpowiedzialności: czy moduł grafiki ma decydować o przejściu w tryb spectator

mode, czy powinien się zajmować obliczaniem pozycji kamery, czy ma obsługiwać wyświetlanie statystyk podczas gry itp. Wszystkie powyższe zadania ostatecznie przydzieliliśmy modułowi `graphics`.

3.5.3. Kod modułów UI PC i UI Android

Na PC UI składa się z głównego okna, w którym zmieniają się panele odpowiadające np. głównemu menu, ustawieniu nowej gry lub właściwej grze.

Oprócz paneli wypełniających na zmianę główne okno, został napisany też panel edytora ustawień gry, który wydobywa informacje o ustawialnych parametrach gry z modułu `data` i wyświetla pola do ich wpisania, jednocześnie zaznaczając, czy wpisana wartość jest prawidłowa (zakresy wartości są podawane przy definiowaniu pól klasy za pomocą adnotacji). Panel edytora może być umieszczany wewnątrz innych paneli tam, gdzie może być przydatne ustawianie paramterów gry (obecnie to jest jedynie przy ustawianiu nowej gry na PC).

Na Androidzie do programowania menu doszło też obmyślenie metody sterowania w trzech osiach. Zdecydowałem się na metodę łączącą akcelerometr z ekranem dotykowym. Akcelerometrem pozwala zaprogramować skręcanie przez przechylanie telefonu - np. aby skręcić w lewo, wystarczy przechylić telefon w tym kierunku. To daje dwie osie. Skręcanie wokół osi z, czyli wirowanie wstążki, odbywa się przez dotykanie dwóch boków ekranu (z lewej strony obrót przeciwnie do ruchu wskazówek zegara, z prawej strony zgodnie z tym ruchem).

Rozdział 4

Spis zawartości dołączonej płyty CD

- `praca/` — praca licencjacka
  - `netacka.pdf` — praca licencjacka (PDF)
  - `netacka.tex` — kod źródłowy pracy
  - `pracamgr.cls` — wydziałowy arkusz stylów prac licencjackich
  - `screen.png`, `octree.png`, `logiczna_mvc.dia`, `segmenty.svg` — obrazki
  - `Makefile` — skrypt kompilacji
- `dok/` — dokumentacja projektu:
  - `moduly/` — dokumentacja architektury: opis poszczególnych modułów
    - `data.pdf`
    - `logic.pdf`
    - `network.pdf`
    - `graphics.pdf`
    - `ui.pdf`
  - `interakcje-miedzy-modulami.pdf` — architektura
  - `wizja.pdf` — wizja projektu
  - `specyfikacja-funkcjonalna.pdf` — specyfikacja funkcjonalna
- `kod/` — kod źródłowy projektu, w trzech katalogach — projektach Eclipse:
  - `netacka3d-common.zip` — część wspólna dla obu platform
  - `netacka3d-pc.zip` — źródła wersji PC
  - `netacka3d-android.zip` — źródła wersji Android
  - `README.txt` — instrukcja kompilacji
- `gra/` — kod wykonywalny gry
  - `netacka3d-pc.zip` — wersja PC
  - `netacka3d-android.apk` — aplikacja dla Androida

Rozdział 5

Podsumowanie

Napisanie Netacki 3D wymagało od nas wiele pracy i pomogło zdobyć doświadczenie w różnorodnych aspektach programowania gier, a także doświadczenie w prowadzeniu wieloosobowego projektu.

Stworzoną przez nas grę planujemy rozszerzyć, dodając m. in. centralną bazę serwerów, wersję dostępną jako aplet Java oraz ciekawsze sztuczne inteligencje przeciwników. Silnik gry stanowi także dobry punkt wyjścia do eksperymentów z nowymi trybami rozgrywki.

Bibliografia

[ADK] Zatacka (Achtung, die Kurve!)
http://en.wikipedia.org/wiki/Achtung,_die_Kurve!

[APA] Licencja Apache 2.0
http://www.apache.org/licenses/LICENSE-2.0

[BLE] Blender 3D
http://www.blender.org/

[GHO] Key jamming and ghosting
http://en.wikipedia.org/wiki/Rollover_%28key%29#Key_jamming_and_ghosting

[JAC] Biblioteka Jackson
http://jackson.codehaus.org/

[JOGL] Java OpenGL (JOGL)
http://jogamp.org/jogl/www/

[JSE] Java, Standard Edition
http://java.sun.com/javase/

[MTL] Material Template Library
http://en.wikipedia.org/wiki/Material_Template_Library

[NET] Netacka
http://students.mimuw.edu.pl/~pm262952/netacka/

[NIO] java.nio.channels
http://java.sun.com/j2se/1.4.2/docs/api/java/nio/channels/package-summary.html

[OBJ] OBJ Format
http://en.wikipedia.org/wiki/Obj

[OES] OpenGL ES
http://www.khronos.org/opengles/