Uniwersytet Warszawski
Wydział Matematyki, Informatyki, Mechaniki

Andrzej Gąsienica-Samek
Nr albumu: 181258

Prototyp platformy dla gier sieciowych opartej na mikrowątkach

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dr Janiny Mincer-Daszkiewicz
Instytut Informatyki

Wrzesień 2004

Pracę przedkładam do oceny

Data Podpis autora pracy:

Praca jest gotowa do oceny przez recenzenta

Data Podpis kierującego pracą:

Streszczenie

W pracy opisuję prototyp platformy do tworzenia gier sieciowych, która może znaleźć zastosowanie w nauczaniu programowania. W prototypie zaimplementowałem nowatorski model programowania logiki gier oparty na mikrowątkach oraz zegarze logicznym, jak również model automatycznej synchronizacji rozgrywek sieciowych oparty na wirtualizacji kontrolerów. Częścią pracy jest obszerne studium przypadku kompletnej gry sieciowej napisanej przy pomocy prototypu.

Słowa kluczowe

gra komputerowa, gra sieciowa, mikrowątek, logika gry, kontroler, wirtualizacja, duszek, serwer synchronizacyjny, system graficzny.

Klasyfikacja tematyczna

D. Software
D.1. Programming Techniques
D.1.m. Miscellaneous

Spis treści

1. Wprowadzenie........5
   Nauka programowania........5
   Platforma dla gier sieciowych........6
   Obecnie stosowane technologie........6
   Wyzwania technologiczne........7
   Plan pracy........7
2. Gra Dynia........9
   Elementy gry Dynia........9
3. Model prototypu........11
   System graficzny........12
   Kontrolery........12
   Zegar logiczny........12
   Procesor logiki gry........13
4. Mikrowątki........15
   Mikrowątki w prototypie........15
   Wnioski........16
   Dodatkowe informacje........16
5. Automatyczna synchronizacja sieciowa........19
   Wnioski z implementacji........20
   Utrwalanie stanu logiki gry........21
6. Studium przypadku – gra sieciowa Aster........23
   Krok 0. Sprawdzenie środowiska........23
   Krok 1. Pierwszy duszek........25
   Krok 2. Poruszanie duszkiem........26
   Krok 3. Obsługa bezwładności........26
   Krok 4. Strzały........28
   Krok 5. Jednoczesny ruch........29
   Właściwy krok 5. Mikrowątki........30
   Krok 6. Dwóch graczy........31
   Krok 7. Trafienia........32
   Krok 8. Poprawienie grywalności........34
   Krok 9. Sieciowość........35
   Wnioski........36
7. Podsumowanie........37
   Mikrowątki i gry sieciowe........37
   Platforma jako produkt........37
8. Dodatek A. Listing gry Aster........39
9. Dodatek B. Zawartość płyty CD........43
10. Bibliografia........45

Rozdział 1

Wprowadzenie

W pracy opisuję prototyp platformy dla gier sieciowych. Celem tej platformy jest wspomaganie powszechnej nauki programowania, dzięki umożliwieniu tworzenia ciekawych gier w bardzo prosty sposób. Implementacja prototypu umożliwiła zbadanie przydatności do tego celu modelu programowania opartego na mikrowątkach i zegarze logicznym oraz zbadanie możliwości automatycznej synchronizacji rozgrywek sieciowych.

Nauka programowania

Nauka programowania, nawet jeśli nie jest potrzebna w życiu zawodowym, wyrabia takie sposoby myślenia, które pomagają zrozumieć procesy zachodzące w otaczającym nas świecie. Głęboko wierzę, że w przyszłości powszechna nauka programowania zrealizuje nadzieje, jakie niegdyś pokładaliśmy w powszechnej nauce matematyki. Zanim tak się stanie, musimy rozwiązać dwa bardzo poważne problemy związane z nauką programowania: programowanie jest skomplikowane i nieciekawe. Powoduje to, że w szkołach średnich programowanie omija się dużym łukiem, a uczy się jedynie obsługi komputera.

Programowanie jest nieciekawe. Jeśli chcę zrobić rysunek, używam Painta. Jeśli chcę zrobić prezentację odpowiada mi PowerPoint. Jeśli chcę przeprowadzić interakcję z użytkownikiem, stworzę zbiór połączonych stron WWW. Jeśli chcę nagrać dźwięk, użyję Rejestratora Dźwięku. Po co mi więc programowanie? Oczywiście, aby to wszystko razem połączyć, dodać logikę i uzyskać w ten sposób całkowicie nowy efekt. Niestety, za pomocą programowania mogę jedynie zapytać użytkownika o liczbę i wypisać coś na konsoli. Mimo dojrzewania informatyki i powstawania wielu narzędzi programistycznych nadal przygoda z programowaniem rozpoczyna się od tak bolesnych doświadczeń.

Programowanie jest skomplikowane. Gdy uczeń przebije się przez podstawowe zasady programowania, stworzy i zrozumie pierwszy program, powinien mieć możliwość twórczego wykorzystania swoich umiejętności. Niestety, poznawanie każdej kolejnej niewielkiej technologii jest tak samo bolesne, jak stawianie pierwszych kroków w programowaniu. Efekt jest taki, że po rocznym kursie programowania na uczelni wyższej, studenci rzeczywiście mają silne podstawy teoretyczne, dobrze rozumieją podstawy metod programowania i struktur danych, natomiast program zaliczeniowy wymagający stworzenia interfejsu graficznego tworzą metodą prób i błędów, do końca zmagając się z technologią.

Platforma dla gier sieciowych

Aby nauka programowania była prosta i ciekawa proponuję uczyć programowania przez tworzenie gier sieciowych. Za pomocą specjalnej platformy dla gier sieciowych uczniowie tworzyliby gry, które mogliby przetestować na zajęciach, bądź wysłać do znajomych. Tworząc gry uczyliby się programować oraz lepiej obsługiwać komputer.

Programowanie może być ciekawe. Tworzenie gier sieciowych zostało wybrane nieprzypadkowo:

- Większość osób, które zaczynają przygodę z programowaniem, chce stworzyć grę,
- Grą można się pochwalić każdemu, nawet osobie nie rozumiejącej programowania,
- W grze najważniejszy jest pomysł, technologia jest drugorzędna,
- Gry sieciowe są bardzo atrakcyjne, gdyż są formą interakcji międzyludzkiej. Nawiązanie interakcji w grze pozwala na rozszerzenie jej na poziom tworzenia gry.

Programowanie może być proste. Platforma dla gier sieciowych jest zamkniętą całością. Oznacza to, że istnieje ściśle określony zestaw wiedzy, którą należy posiąść, aby móc w pełni wykorzystywać platformę. Osiąganie efektów następuje w wyniku twórczego wykorzystywania tej wiedzy, poznawania nowych sposobów jej wykorzystywania, a nie poprzez poznawanie coraz to nowych konstrukcji języka i bibliotek. Wszelkie problemy technologiczne, z którymi borykają się programiści w rzeczywistym świecie są usunięte, gdyż celem platformy jest nauka programowania, a nie nauka konkretnej technologii.

Obecnie stosowane technologie

Do tworzenia prostych gier są obecnie wykorzystywane głównie dwie technologie:

Java J2ME [9]. Technologia J2ME powstała przez okrojenie technologii ogólnego przeznaczenia Java na potrzeby małych urządzeń. W przypadku gier jest ona stosowana głównie do pisania prostych gier na telefony komórkowe. Jako że J2ME nie została stworzona do pisania gier, aby ich pisanie było prostsze wymagane jest używanie bibliotek pomocniczych, np. J2MEGL [7]. Językiem programowania jest język Java.

Macromedia Flash [6]. Technologia Macromedia Flash powstała do tworzenia animowanych efektów na stronach WWW. Dzięki temu posiada bardzo dobry silnik graficzny. W trakcie jej rozwoju został dodany prosty język skryptowy ActionScript [2] oraz biblioteki programistyczne. W przypadku gier jest ona używana do tworzenia prostych gier umieszczanych na stronach WWW. Ze względu na nieprzemyślany język programowania i biblioteki, tworzenie gier jest trudne.

Istnieje również szereg bibliotek do tworzenia gier dla różnych języków programowania, które wspierają zazwyczaj proces wyświetlania grafiki.

Należy zwrócić uwagę, że za pomocą tych narzędzi tworzone są proste gry, bez żadnego wsparcia dla gier sieciowych. Model programowania zawsze dziedziczy cechy języka na jakim został oparty. Jako że języki te nie były projektowane z myślą o programowaniu logiki gier, ich zastosowanie wprowadza dodatkową komplikację.

Wyzwania technologiczne

W pracy opisuję prototyp platformy dla gier sieciowych. Główne wyzwania, jakie stoją przez twórcą platformy to dobranie odpowiedniego modelu programowania, umożliwienie synchronizacji gier sieciowych oraz stworzenie kompletnego zestawu bibliotek, pozwalających na tworzenie gier.

Model programowania powinien umożliwiać opisywanie logiki gier w naturalny sposób, a jednocześnie powinien być na tyle silny, aby zapoznać początkującego programistę z różnymi aspektami programowania. W prototypie zastosowałem bardzo nowatorskie podejście oparte na mikrowątkach i zegarze logicznym.

Synchronizacja gier sieciowych nie powinna wprowadzać dodatkowej komplikacji przy tworzeniu gier. Należy zwrócić uwagę, że w obecnie wykorzystywanych technologiach do tworzenia prostych gier, rozgrywki sieciowe nie są w ogóle wspierane. W prototypie zastosowałem automatyczną synchronizację rozgrywek sieciowych, poprzez wirtualizację kontrolerów. Dodatkowo opisałem możliwości rozszerzenia schematu synchronizacji. Ta cecha platformy udostępnia początkującym programistom możliwości, które do tej pory były zarezerwowane tylko dla ekspertów.

Plan pracy

Praca składa się z następujących rozdziałów:

Rozdział 2. Gra Dynia. W tym rozdziale przedstawię przykładową grę, do której będę się odwoływał przy omawianiu prototypu.

Rozdział 3. Model prototypu. W tym rozdziale wprowadzę model prototypu, z podziałem na komponenty. Wyjaśnię podstawowe pojęcia takie jak zegar logiczny czy procesor logiki gry.

Rozdział 4. Mikrowątki. W tym rozdziale wprowadzę pojęcie mikrowątku i przedstawię jego zastosowania w programowaniu logiki gry.

Rozdział 5. Automatyczna synchronizacja sieciowa. W tym rozdziale przedstawię schemat automatycznej synchronizacji rozgrywek sieciowych oraz przeprowadzę dyskusję o jego rozbudowie.

Rozdział 6. Studium przypadku – gra sieciowa Aster. W tym rozdziale przedstawię kompletny przykład prostej gry sieciowej, zastosowanie poszczególnych komponentów prototypu, mikrowątków oraz synchronizacji sieciowej. Dodatkowo dokładnie opiszę obsługę prototypu.

Rozdział 7. Podsumowanie. Ten rozdział zawiera podsumowanie pracy oraz informacje na temat możliwych rozszerzeń i udoskonaleń platformy.

Rozdział 2

Gra Dynia

Gra Dynia została opracowana na podstawie gry DynaBlaster firmy Hudson Soft. Oryginalna gra pochodzi z 1992 roku, a jej strona utrzymywana przez fanów mieści się na:

http://www.geocities.com/Eureka/4979/dynablst.html

Logo gry wygląda następująco:

O popularności gry może świadczyć fakt, że Google znajduje 13600 odniesień do DynaBlaster oraz 6740 do „Dyna Blaster", mimo iż gra powstała przed rozwinięciem się Internetu. W sieci można znaleźć bardzo dużo klonów, zazwyczaj nie mających tego czegoś, co oryginalna Dyna posiadała. Mój prototyp doskonale (niestety) wbija się w tą grupę.

W Dynię grają dwie osoby. Plansza składa się z 225 pól, z których 108 jest dostępnych dla graczy (zrzut ekranu 1). Gracze poruszają się po planszy i stawiają bomby. Bomba po pewnym stałym czasie wybucha, a jej siła rażenia rozchodzi się na odległość 3 pól w pionie i w poziomie. Wybuch bomby nie przechodzi przez ściany. Gracz trafiony przez bombę ginie, a bomba trafiona przez bombę wybucha natychmiast.

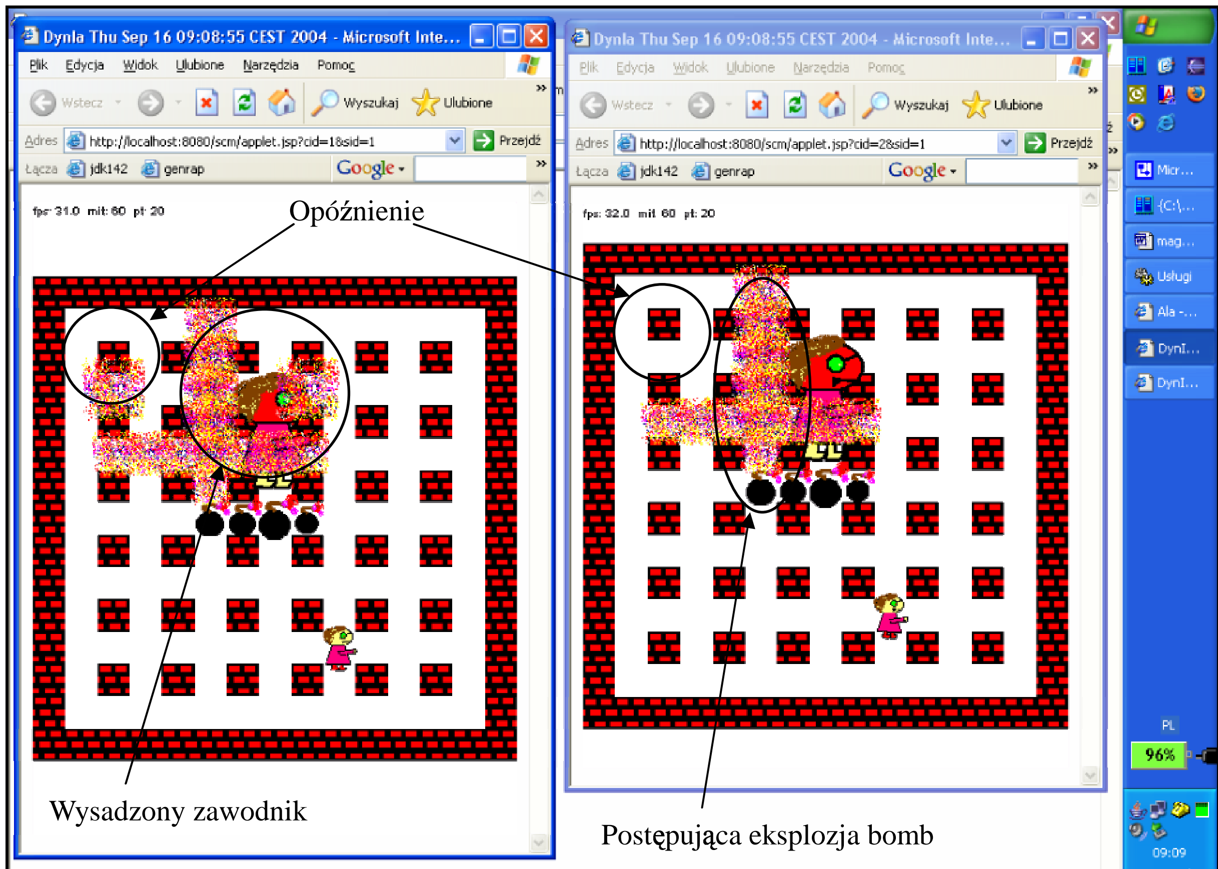
Zrzut ekranu 1 przedstawia grę sieciową, rozgrywaną w dwóch przeglądarkach Internetowych. Widać na nim gracza bezpiecznego oraz gracza trafionego przez bombę, który wylatuje w powietrze (postać powiększona). Widać również postępującą eksplozję bomb. Przeglądarka z prawej strony o kilkadziesiąt milisekund wyprzedza przeglądarkę z lewej strony, gdyż eksplozja w lewym górnym rogu w jednej przeglądarce już wygasła, a w drugiej dopiero wygasa.

Elementy gry Dynia

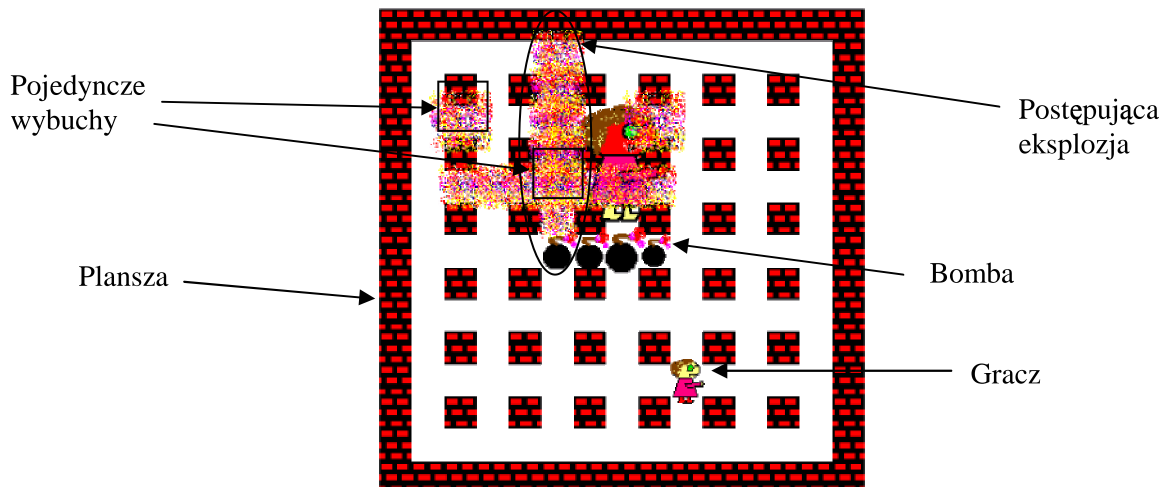
Dynia składa się z następujących elementów:

- Plansza,
- Gracz,
- Bomba,

- Eksplozja,
- Wybuch.



Zrzut ekranu 1. Gra Dynia – rozgrywka sieciowa



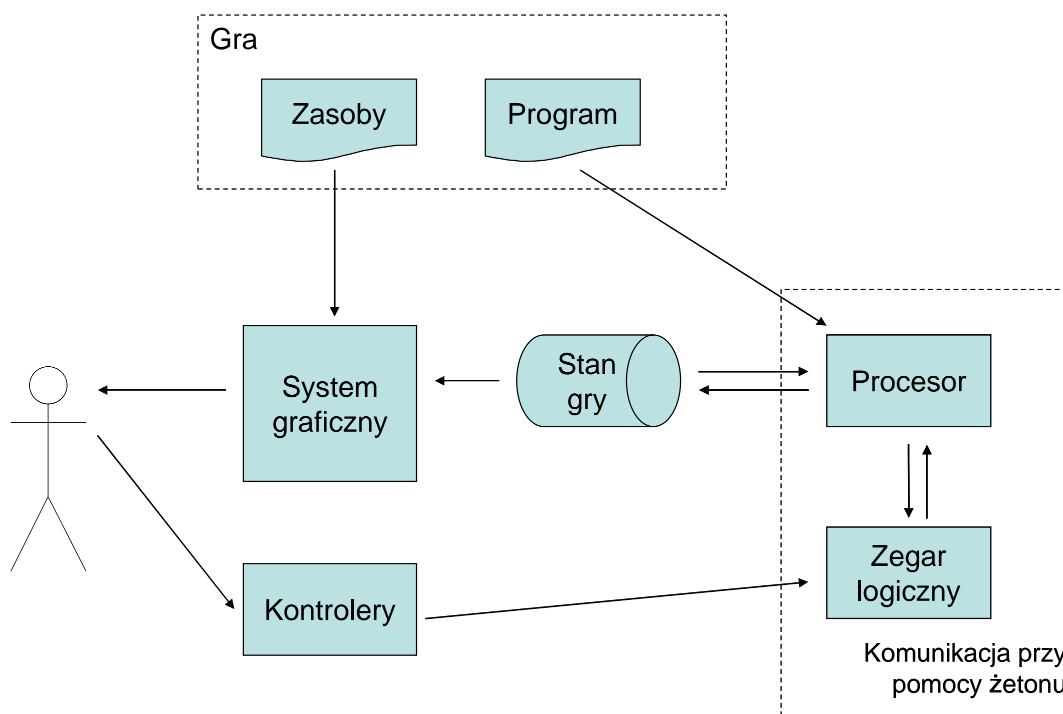
Zrzut ekranu 2. Elementy gry Dynia

Plansza stanowi przestrzeń, w której poruszają się gracze. Gracz może się poruszać po planszy w pionie i w poziomie oraz może stawiać bomby. Gracz nie może przechodzić przez bomby. Bomba wybuchając rozpoczyna proces eksplozji. Eksplozja, w odstępach czasowych i w ograniczonej przestrzeni, tworzy wybuchy tam, gdzie nie ma bomb oraz powoduje eksplozje bomb. Wybuch trwa krótką chwilę, jeśli nastąpi kolizja wybuchu z graczem, to gracz ginie.

Rozdział 3

Model prototypu

W modelu prototypu wszystkie nazwy odnoszą się do abstrakcyjnych pojęć, a nie do urządzeń komputera. Jeśli mówimy o procesorze logiki gry, to nie mamy na myśli rzeczywistego procesora, tylko wirtualny komponent realizujący pewną funkcjonalność.



Rysunek 1. Przepływ danych w prototypie

Program logiki gry jest to program w **języku programowania logiki gry**. **Procesor logiki gry** jest to jednostka wykonująca program. Procesor logiki gry podczas wykonywania programu modyfikuje **stan logiki gry**. Procesor logiki gry komunikuje się z **zegarem logicznym**, który daje programowi logiki gry poczucie czasu. Do opisu komunikacji między procesorem a zegarem logicznym używane jest pojęcie **żetonu**. Tylko urządzenie posiadające żeton wykonuje swoje zadania, a drugie jest wstrzymane. **System graficzny** wizualizuje stan logiki gry dla gracza. **Kontrolery** zapewniają sterowanie programem logiki gry przez gracza. **Zasoby** to katalog obrazków, do których dostęp ma system graficzny. **Gra** jest to program i zasoby.

System graficzny

System graficzny jest to komponent, który odpowiada za wizualizację stanu logiki gry dla gracza. W prototypie jest to komponent, który może być uruchomiony w przeglądarce internetowej lub w oknie systemu operacyjnego.

Część stanu logiki gry, która jest wizualizowana to **drzewo duszków**. Węzłem drzewa duszków jest **duszek** albo **rysunek**, z tym że węzłami wewnętrznymi są duszki, a liśćmi – rysunki. Rysunki mogą być prostymi figurami, albo mogą odwoływać się do obrazków umieszczonych w zasobach. Każdy duszek składa się z ciągu węzłów drzewa oraz przechowuje dodatkową informację w postaci ciągu przekształceń afinicznych. Słowo duszek jest przyjętym tłumaczeniem angielskiego słowa *sprite*, którego jednym ze znaczeń jest mały obrazek, często używany przy animacji grafiki.

W przypadku gry Dynia plansza jest osobnym duszkiem, dzieckiem korzenia. Plansza zawiera po jednym duszku dla każdego kwadratu ściany. Każdy kwadrat ściany zawiera jedną translację. Poza tym każdy gracz, bomba oraz wybuch są skojarzone z pojedynczymi duszkami, dziećmi korzenia. Każdy z tych duszków zawiera translację oraz inne przekształcenia realizujące efekty (np. powiększanie, przechylanie). Wszystkie rysunki odnoszą się do obrazków umieszczonych w zasobach.

System graficzny pobiera drzewo duszków ze stanu logiki gry, ale w żaden sposób nie zmienia stanu logiki gry. **Procedura rysująca** systemu graficznego odpowiada za narysowanie drzewa duszków na ekranie monitora. Procedura ta przechodzi rekurencyjnie po drzewie duszków składając przekształcenia afiniczne, a po dojściu do rysunku rysuje go, potencjalnie pobierając obrazki z zasobów. Tak więc rysunki są rysowane w naturalnej kolejności, a każdy rysunek jest poddawany złożeniu wszystkich przekształceń afinicznych znajdujących się na ścieżce od korzenia do rysunku. System graficzny wywołuje procedurę rysującą w chwili bezczynności procesora logiki gry. Dzięki temu gra dostosowuje płynność animacji do wydajności komputera i złożoności programu logiki gry. Częstotliwość odświeżania nie jest obserwowalna przez procesor logiki gry i nie wpływa na sposób wykonywania programu logiki gry. Schemat ten działa dobrze w przypadku, gdy wykonywanie programu logiki gry zużywa niewielką część mocy procesora, co jest cechą gier dwuwymiarowych.

Kontrolery

Kontrolery stanowią interfejs wejściowy dla gracza. W prototypie zaimplementowano obsługę klawiatury, z tym że można tworzyć wiele kontrolerów, z których każdy ma przypisany zestaw klawiszy. Gra Dynia korzysta z dwóch kontrolerów, po jednym dla każdego gracza. Każdy kontroler składa się z pięciu klawiszy. W przypadku gry dwóch graczy siedzących przy jednym komputerze, pierwszy dysponuje klawiszami kursora i klawiszem Ctrl, a drugi – klawiszami A, S, D, W oraz klawiszem Shift.

Z punktu widzenia programu logiki gry, prototyp udostępnia dwa sposoby odwoływania się do klawiatury. Pierwszy polega na testowaniu stanu konkretnych klawiszy. Aby nie przepuścić żadnego klawisza, należy robić to odpowiednio często. Jest to sposób, który odpowiada potrzebom gier zręcznościowych. Drugi sposób dostępu to instalacja metody wywoływanej w reakcji na wciśnięcie klawisza.

Zegar logiczny

Zegar logiczny, zwany dalej zegarem, jest to komponent, który daje poczucie czasu procesorowi logiki gry. Komunikacja między procesorem a zegarem odbywa się przy pomocy żetonu. Tylko komponent posiadający żeton może wykonywać swoje zadania. Jeśli żeton jest w posiadaniu procesora logiki gry,

to czas logiczny stoi, a procesor wykonuje program logiki gry, zmieniając w ten sposób stan logiki gry. W przypadku gdy żeton jest w posiadaniu zegara logicznego, to czas logiczny płynie, a procesor logiki gry nie wykonuje żadnych czynności, co powoduje, że stan logiki gry jest zamrożony. Przekazanie żetonu następuje tylko w odpowiedzi na żądanie komponentu, który go posiada.

Czas logiczny to liczba mówiąca o ilości czasu jaki upłynął od uruchomienia gry. Jednostką miary czasu logicznego są **sekundy logiczne**. Zegar logiczny zarządza czasem logicznym. Zegar ma za zadanie utrzymywanie upływu czasu logicznego w synchronizacji z upływem czasu rzeczywistego. Dla uniknięcia niejednoznaczności, sekundy czasu rzeczywistego będziemy czasem określali mianem **sekund rzeczywistych**. Zadanie utrzymania upływu czasu logicznego w synchronizacji z upływem czasu rzeczywistego jest o tyle trudne, że w przeciwieństwie do czasu rzeczywistego, który płynie nieustannie, czas logiczny może płynąć tylko wtedy, gdy zegar logiczny posiada żeton. Procesor przekazuje żeton do zegara logicznego na określoną liczbę sekund logicznych. Po upływie określonej liczby sekund logicznych zegar zwraca żeton procesorowi.

Po uruchomieniu gry żeton jest w posiadaniu procesora logiki gry, a więc czas logiczny nie płynie. Wykonywanie programu logiki gry zużywa czas rzeczywisty. Schematy zarządzania czasem logicznym są więc następujące:

- Jeśli procesor przekaże żeton bezpośrednio po uruchomieniu gry na 5 sekund logicznych, to zadanie zegara logicznego jest proste – musi odczekać 5 sekund rzeczywistych, przestawić czas logiczny o 5 sekund logicznych do przodu i zwrócić żeton procesorowi.
- Jeśli procesor przetrzyma żeton przez 2 sekundy rzeczywiste i przekaże go na 5 sekund logicznych, to zegar musi zamortyzować różnicę, czyli przestawić czas logiczny o 5 sekund logicznych do przodu, wyczekać 3 sekundy czasu rzeczywistego i wtedy zwrócić żeton.
- Jeśli procesor przetrzyma żeton przez 6 sekund rzeczywistych i przekaże go na 5 sekund logicznych, to zegar musi przestawić czas logiczny o 5 sekund logicznych do przodu, zapamiętać że 1 sekunda rzeczywista jest w niedomiarze oraz zwrócić żeton. Oczywiście problemy pojawiają się, gdy taka sytuacja staje się częsta, a liczba sekund w niedomiarze przestaje wracać do zera.

W grze Dynia zegar logiczny jest używany do utrzymywania tempa gry oraz płynności animacji. Dzięki niemu postacie poruszają się ze stałą prędkością, niezależnie od mocy obliczeniowej komputera. Czas rzeczywisty, który zużywa procesor logiki gry na przetwarzanie programu logiki gry jest niezauważalnie mały, w porównaniu do czasu jaki zużywa system graficzny na potrzeby rysowania.

Procesor logiki gry

Procesor logiki gry jest jedynym komponentem, który może zmieniać stan logiki gry. Procesor zmienia go wykonując program logiki gry. Jedynym wejściem dla procesora jest komunikacja z kontrolerami. Komunikacja ta pozwala na interakcję z graczem. Od procesora wymagamy, aby posiadał **właściwość deterministycznego działania**:

> Wykonanie programu logiki gry, rozpoczęte od stanu logiki gry `X`,
> w przypadku gdy kontrolery zgłaszają ciąg zdarzeń w czasie $Z=(time_i,event_i)$,
> musi zawsze prowadzić do tego samego stanu logiki gry `Y`.

Zauważmy, że procesor działa tak samo, niezależnie od sposobu zarządzania czasem logicznym przez zegar logiczny. Sposób zarządzania czasem logicznym jest obserwowalny przez gracza w postaci płynności animacji, nie jest natomiast obserwowalny z wnętrza programu logiki gry.

Rozdział 4

Mikrowątki

Mikrowątki to wątki o specyficznych właściwościach. Są podobne do wątków w następujących aspektach:

- każdy mikrowątek posiada stos rekordów aktywacji,
- w stanie programu jednocześnie może istnieć wiele mikrowątków,
- mikrowątki mają dostęp do wspólnej pamięci,
- mikrowątki wykonują kod programu niezależnie od siebie.

Mikrowątki różnią się od wątków następującymi właściwościami:

- jednocześnie może działać tylko jeden mikrowątek, zwany **aktywnym mikrowątkiem**, a w przypadku wątków możliwe jest jednoczesne działanie kilku wątków na maszynie wieloprocesorowej,
- przełączanie mikrowątków może następować tylko na żądanie aktywnego mikrowątku, a w przypadku wątków przełączaniem zarządza system operacyjny i może się ono odbywać w dowolnym momencie,
- mikrowątki są tanie, w sensie pamięci i czasu są porównywalne z wywoływaniem metod, a wątki są wprawdzie tańsze niż procesy, jednak znacznie droższe niż metody.

Mikrowątki w prototypie

Aby używać mikrowątków musimy określić sposób ich przełączania. Przełączanie mikrowątków w prototypie może następować tylko wtedy, gdy aktywny mikrowątek wykona operację czekania przez określoną liczbę sekund logicznych. Wykonanie tej operacji polega na sprawdzeniu, czy nie należy obudzić jednego z oczekujących mikrowątków. Jeśli nie, to procesor wylicza czas logiczny, na jaki należy oddać żeton do zegara logicznego, aby wzbudzić kolejny mikrowątek w odpowiedniej chwili. Po ponownym otrzymaniu żetonu procesor wzbudza właściwy mikrowątek.

Aby zrealizować powyższy schemat, stan logiki gry musi posiadać kolejkę priorytetową stosów mikrowątków. Kluczem w kolejce jest czas logiczny obudzenia mikrowątku, a wartością – stos rekordów aktywacji.

W grze Dynia zastosowane są następujące mikrowątki:

- po jednym dla obsługi sterowania każdą z postaci,
- po jednym dla efektu chybotania się każdej postaci przy chodzeniu,
- po jednym dla obsługi każdej bomby – animacji i rozpoczynania procesu eksplozji,

- po jednym dla każdego kierunku procesu eksplozji,
- po jednym dla obsługi każdego wybuchu – animacji i badania kolizji.

W trakcie normalnej rozgrywki używanych jest około stu mikrowątków jednocześnie. Mikrowątki okazały się być bardzo wygodnym mechanizmem.

Wnioski

Mikrowątki doskonale sprawdzają się przy opisywaniu logiki duszków. Dzięki nim można:

- opisywać zachowanie duszka, z perspektywy tego duszka,
- udostępniać na zewnątrz tylko tą część stanu duszka, która jest wymagana do komunikacji między duszkami, np. pozycję duszka można umieścić w globalnych zmiennych, a prędkość i zwrot przechowywać w zmiennych lokalnych procedury obsługi duszka,
- dodawać efekty animacyjne, np. pulsowanie, chybotanie itp. bez żadnej ingerencji w stan zmiennych globalnych.

Istnieją jednak pewne problemy, które mikrowątki posiadają, a które nie zostały jeszcze zbadane w przypadku wątków. Główny problem to odpluskwianie. Nie ma jeszcze programów odpluskwiających, które w sposób zadowalający wspierałyby programowanie wielowątkowe. Nie znaczy to, że nie mogą one istnieć w ogóle. W tej dziedzinie obserwujemy duży postęp, nadal jednak programy wielowątkowe odpluskwia się trudniej niż jednowątkowe.

Dodatkowe informacje

W prototypie język programowania logiki gry jest mapowany na język programowania Java [3]. Procesor logiki gry jest mapowany na maszynę wirtualną Javy [5]. Mikrowątki są mapowane na wątki Javy. Zarządzanie wątkami jest realizowane przez prototyp i zabronione jest korzystanie z mechanizmów wielowątkowości języka Java.

W przypadku Javy 1.4 wątki są implementowane przez system operacyjny. Sprawia to, że mikrowątki nie są tak tanie, jak mogłyby być. Do zarządzania mikrowątkami jest użyta pula wątków, dzięki czemu na procesorze Pentium M 1300 z WinXP i Javą 1.4, można wykonać około 100 000 tysięcy przełączeń wątków w ciągu sekundy. Prawdziwa implementacja powinna zarządzać wątkami tak tanio, w sensie czasu i pamięci, jak zarządza wywoływaniem procedur.

W swoich poszukiwaniach natknąłem się na jedną próbę efektywnej implementacji mikrowątków dla języka Python [10]. Niestety implementacja ta jest od 1999 roku na etapie prototypu. Na komputerze AMD Athlon XP 1500+ autor uzyskał ponad 1,6 miliona przełączeń mikrowątków w ciągu sekundy, a jeden mikrowątek zajmował poniżej 800 bajtów pamięci. Należy zwrócić uwagę, że implementacja była optymalizowana pod kątem efektywnego wykonywania programu, co jest przyczyną aż 800 bajtów potrzebnych na pojedynczy mikrowątek.

W 1998 roku został opisany wzorzec projektowy mikrowątku [4] w programowaniu obiektowym, nazwijmy go wzorcem mikrowątku. Wzorzec mikrowątku definiuje standardowy sposób na radzenie sobie w przypadkach, w których przydatne byłby mikrowątki, a jednocześnie brak jest dostępu do systemu wątków na poziomie języka programowania. Wzorzec mikrowątku formalizuje konieczność stworzenia obiektu stanu dla każdego potencjalnego miejsca przełączania mikrowątków oraz określa standardowy sposób zarządzania przełączaniem wykonywania operacji na takich obiektach stanu. Powoduje to ogromną komplikację w implementacji oraz wprowadza potencjalne miejsca popełniania

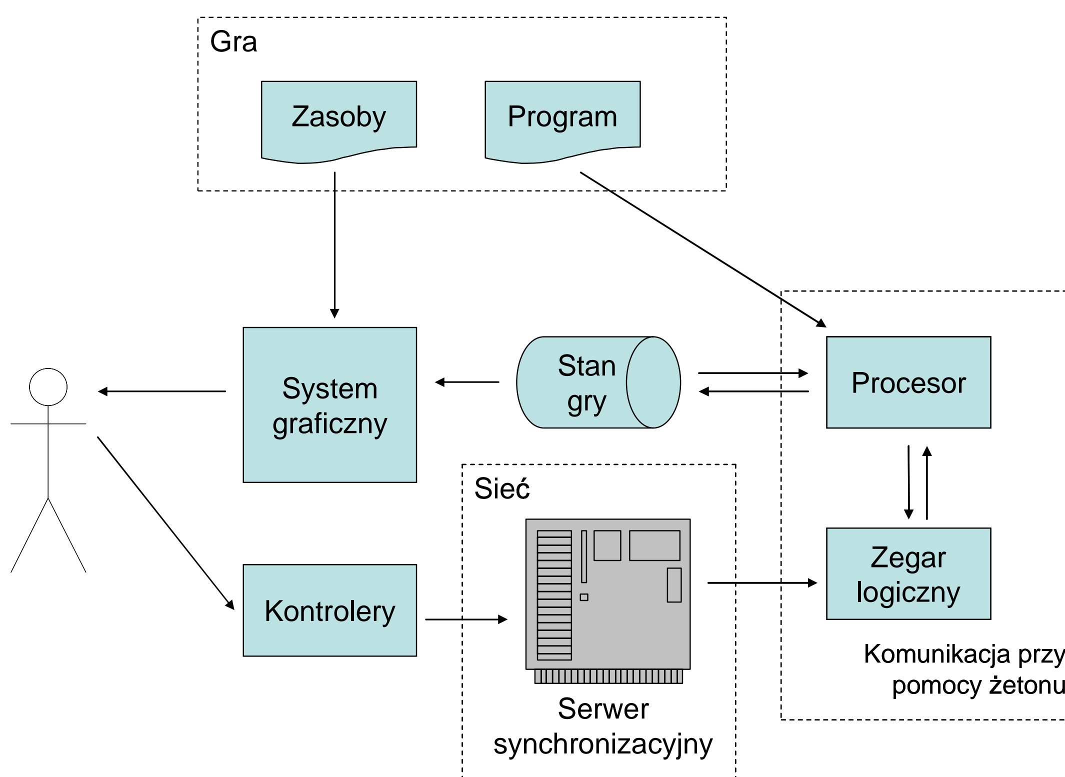
błędów. Zastosowanie wzorca mikrowątku jest analogiczne do kroku 5 w studium przypadku (zob. rozdział 6).

Pozycja [1] zawiera informacje o zastosowaniu mikrowątków do implementacji sztucznej inteligencji w grach komputerowych. Mikrowątki były stosowane bez zegara logicznego. Szczególna przydatność mikrowątków w tym przypadku wynika z faktu, że procedury implementujące sztuczną inteligencję nierzadko rozwijane są reaktywnie. Pisany jest podstawowy szkielet, a następnie w przypadku znalezienia niepoprawnego zachowania sztucznej inteligencji dodawane są na bieżąco nowe reguły. To prowadzi do procedur o bardzo skomplikowanym przepływie sterowania. W takich przypadkach mikrowątki nadają się znakomicie, gdyż dzięki nim nie trzeba zarządzać bardzo skomplikowanym obiektem stanu.

Rozdział 5

Automatyczna synchronizacja sieciowa

W tym rozdziale analizuję możliwości automatycznej synchronizacji gier sieciowych w prototypie. Automatyzm tej czynności rozumiem jako brak konieczności implementowania i testowania kodu, który dotyczy komunikacji sieciowej. W zaproponowanym modelu platformy możemy ten efekt uzyskać poprzez wirtualizację kontrolerów. Powoduje to, że każdy komputer uczestniczący w grze steruje jednym z kontrolerów, a cała rozgrywka toczy się tak, jakby wszystkie te kontrolery podłączone były lokalnie.



Rysunek 2. Synchronizacja sieciowa

Kluczowym elementem obsługi gier sieciowych jest **serwer synchronizacyjny**. Serwer synchronizacyjny w przypadku gry sieciowej zarządza upływem czasu logicznego na klientach oraz szeregowaniem zdarzeń z kontrolerów. Serwer synchronizacyjny co **kwant czasu** wysyła powiadomienie o upływie czasu i zmianach stanów kontrolerów do wszystkich klientów.

W przypadku gry sieciowej procesory logiki gier wszystkich klientów wykonują program logiki gry niezależnie. Kluczowe jest utrzymanie takiego samego stanu logiki gry na wszystkich klientach. Jest to osiągane dzięki szeregowaniu zdarzeń z kontrolerów przez centralny serwer synchronizacyjny oraz dzięki właściwości determinizmu działania procesora logiki gry, opisanej w rozdziale 2.

Sterowanie upływem czasu na klientach przez serwer jest konieczne, gdyż to serwer decyduje, jaki czas logiczny zostanie przypisany zdarzeniom z kontrolerów, a więc tylko serwer może zdecydować, że w pewnym przedziale czasu nie zdarzy się już żadna akcja i bezpieczne jest dalsze wykonywanie programu logiki gry. Zauważmy, że procesor nie ma możliwości cofania się do poprzednich stanów logiki gry.

Aby zapewnić płynność animacji w grze serwer synchronizacyjny musiałby wysyłać co najmniej 50 powiadomień o upływie czasu w ciągu sekundy w równych odstępach czasowych. Jest to mało realne. Z tego powodu czas logiczny na klientach jest utrzymywany o kwant czasu wstecz w stosunku do potwierdzonego czasu z serwera. Czas logiczny na klientach biegnie przy tym niezależnie, co zapewnia płynność działania aplikacji nawet przy dużym kwancie czasu. Powoduje to niestety, że czas reakcji na klawisz jest równy sumie czasu obrotu pakietu w sieci i kwantu serwera. Kwant serwera został przyjęty na poziomie 50 ms, co umożliwia płynną grę po sieci lokalnej.

Niekiedy pakiety w sieci mogą się opóźnić. Może się to zdarzyć nawet wtedy, gdy sieć działa poprawnie, a jedynie komputer jednego z graczy przez chwilę został przyblokowany przez działający w tle proces. W takim przypadku czas na serwerze biegnie swobodnie, a więc inni gracze nie odczuwają żadnych problemów. Na komputerze, który obsłużył pakiet z opóźnieniem, nastąpi jednak przerwa. Aby uniknąć nagłego przeskok stanu gry o czas opóźnienia wprzód, utrzymujemy czas logiczny opóźniony w stosunku do otrzymywanych informacji, nawet gdy te zaczną przychodzić poprawnie. Aby płynnie wyrównać powstałą różnicę, czas na kliencie biegnie o 1% szybciej niż na serwerze. Dzięki temu, gdy z jakiś chwilowych przyczyn ulegnie on opóźnieniu, to i tak po pewnym czasie wróci do stanu optymalnego. Nie powoduje to natomiast zauważalnych problemów z płynnością animacji, jeśli czas logiczny na kliencie zrówna się z czasem potwierdzonym przez serwer. Dzieje się to nieustannie i powoduje wstrzymywanie gry, ale tylko na milisekundy. W przypadku animacji nie jest to zauważalne dla oka ludzkiego.

Gra Dynia pozwala na grę sieciową dla dwóch graczy na dwóch komputerach. Każdy z graczy widzi ten sam stan, jednak każdy ma przypisany kontroler sterujący inną postacią. W przypadkach obu graczy kontroler składa się z klawiszy kursora oraz klawisza Ctrl do stawiania bomb na ich lokalnych klawiaturach.

Wnioski z implementacji

Istnieją pewne ograniczenia przyjętego schematu. Implementacja obsługi gier sieciowych działa bardzo dobrze. Rzeczywiście można napisać aplikację sieciową nie przejmując się ani protokołem, ani zagadnieniami współbieżności. Bieżący projekt i implementacja ma jednak swoje ograniczenia. Główne to:

1. konieczność wyświetlania tego samego widoku na monitorach wszystkich graczy, co znacznie ogranicza możliwości zastosowań,
2. wymaganie szybkiego dostępu do serwera,
3. konieczność częstej komunikacji z serwerem synchronizacyjnym,
4. brak możliwości dołączania zawodników w trakcie rozgrywki.

Poniżej przedstawiam pomysły na rozwiązanie problemów 2-4. Problem 1 można rozwiązać poprzez umożliwienie instalacji funkcji, która na podstawie stanu logiki gry tworzy drzewo duszków. Funkcja ta mogłaby mieć dostęp do informacji o numerze lokalnego zawodnika i na tej podstawie zmieniać parametry wizualizacji. Jeśli funkcja ta nie miałaby efektów ubocznych, to przedstawiony schemat synchronizacji sieciowej działałby poprawnie.

Należy stworzyć własny język programowania logiki gry. Aby umożliwić używanie platformy przez programistów konieczne jest stworzenie języka, który zapewnia własność determinizmu działania procesora logiki gry. W przypadku prototypu zaimplementowanego w Javie, aby zapewnić właściwość determinizmu działania procesora logiki gry, w programie logiki gry można odwoływać się tylko do podzbioru biblioteki standardowej Javy o interfejsie czysto funkcyjnym, nie mającego efektów ubocznych. Można więc korzystać z funkcji sinus zdefiniowanej w bibliotece standardowej, ale nie można odwoływać się do bibliotek dających dostęp do systemu operacyjnego, gdyż stają się one urządzeniami wejściowymi dla procesora logiki gry, które nie podlegają synchronizacji. Główny problem języka Java to brak rozgraniczenia, które funkcje mają efekty uboczne, a które ich nie posiadają.

Utrwalanie stanu logiki gry

Stan logiki gry jest inicjowany przed uruchomieniem gry, a następnie jest zmieniany podczas wykonywania programu logiki gry, w oparciu o zdarzenia z kontrolerów. Możliwość utrwalenia stanu logiki gry pozwoliłaby na zapis i odczyt stanu rozgrywki oraz na dołączanie zawodników do rozgrywki sieciowej w trakcie jej trwania. Jeśli operacja zachowania stanu logiki gry działałby bardzo szybko, to możliwe byłoby prowadzenie rozgrywki w sieci nawet poprzez wolne łącze modemowe oraz możliwe byłoby znaczne ograniczenie ruchu w sieci.

Zapis i odczyt stanu rozgrywki

Zapis i odczyt stanu jest to opcja niezbędna w przypadku gier, których rozgrywki trwają dłużej niż godzinę. Przykładem są gry strategiczne. Aby zaimplementować taką funkcjonalność należy w kodzie gry wyodrębnić jej stan oraz zaimplementować zapis tego stanu do pliku. To wymaga rezygnacji z przechowywanie części stanu na stosie, co znacznie ogranicza zakres stosowania mikrowątków. Możliwość utrwalania stanu logiki gry jest rozwiązaniem, które zaspokaja większość potrzeb związanych z zapisem gry, a jego niewątpliwą zaletą jest prostota użycia. Oczywiście nie jest to rozwiązanie całościowe. Aby dobrze zaimplementować obsługę zapisu gry, wyodrębnienie jej stanu jest niezbędne. Tylko takie podejście pozwala na pełną kontrolę formatu pliku, a co za tym idzie – możliwość zachowywania zgodności tego formatu między wersjami gry, co w przypadku utrwalenia stanu procesora nie jest możliwe.

Dołączanie zawodników do rozgrywki sieciowej w trakcie jej trwania

Podstawą spójności stanu logiki gry jest spójność początkowych stanów logiki gry. Łatwo jest ją zachować w przypadku gry wszyscy gracze rozpoczynają grę jednocześnie. Jeśli jednak chcemy mieć możliwość dołączania zawodnika w trakcie rozgrywki, to stan logiki gry na jego komputerze musi zostać zainicjowany stanem logiki gry trwającej już rozgrywki. Stany wszystkich graczy powinny oczywiście być spójne, więc możemy wybrać stan logiki gry z dowolnego komputera jako źródło. Do takie operacji niezbędna jest możliwość utrwalania stanu logiki gry.

Prowadzenie rozgrywki w sieci poprzez wolne łącze modemowe

Prototyp nie umożliwia gry poprzez wolne łącze modemowe. W takim przypadku oczywiście uruchomienie gry jest możliwe, ale czas reakcji na zdarzenia z kontrolerów jest zbyt wolny. Jest to spowodowane koniecznością wstrzymywania czasu logicznego na lokalnym komputerze do chwili potwierdzenia go przez serwer. To powoduje, że w przypadku wciśnięcia klawisza czas reakcji jest równy sumie czasu obrotu pakietu w sieci i kwantu serwera. W przypadku próby gry przez dwa modemy podłączone do Internetu, z których jeden jest serwerem, dla jednego z graczy oznacza to opóźnienie rzędu 500 ms. Dla informacji: opóźnienie 100 ms jest lekko wyczuwalne, 200 ms jest już zauważalne i przeszkadza w grze, a 500 ms praktycznie uniemożliwia normalną rozgrywkę.

Czy w takim razie istnieje możliwość pokonania odległości jaka dzieli komputery? Rozwiązanie polega na spekulatywnym wykonywaniu programu logiki gry. Jeśli komputer jest oddalony od serwera, to przyspieszamy na nim rozgrywkę o czas obrotu pakietu w stosunku do czasu potwierdzonego przez serwer. Zakładamy przy tym, że stan kontrolerów od chwili potwierdzonej przez serwer się nie zmienił. Jeśli w międzyczasie dojdzie do nas pakiet z informacją, że jednak ktoś nacisnął klawisz, to musimy cofnąć się w czasie i jeszcze raz przeprowadzić spekulację od tego momentu. Strategia ta sprawdza się szczególnie dobrze w przypadku gier 3D, w których zawodnik widzi bardzo ograniczoną przestrzeń, postacie poruszają się bardzo szybko i względnie rzadko zmieniają kierunek poruszania się. Efekt jest taki, że generalnie całość działa dobrze, ale czasem można zobaczyć rakietę dopiero w momencie gdy do nas dolatuje, gdyż informacja o jej wystrzeleniu dotarła z opóźnieniem. W przypadku gier 2D takich jak Dynia, w których widzimy całą planszę, może to powodować, że przeciwnicy będą lekko przeskakiwali na planszy, ale jest to znacznie mniej uciążliwy efekt niż powolna reakcja na klawisze.

Oczywiście cofanie czasu wymaga zapisu i odtwarzania stanu rozgrywki. Operacja ta jest wykonywana w przypadku każdej zmiany stanu kontrolera u przeciwnika, gdyż zawsze dowiadujemy się o tym po fakcie. Powoduje to, że operacje zapisania i odtworzenia stanu muszą działać bardzo szybko. Pomocne mogą się tutaj okazać struktury niezmienialne, znane z języków funkcyjnych. Można za ich pomocą zaimplementować tablice o czasie dostępu *log(n)* w taki sposób, że po każdej operacji modyfikacji mamy dostęp do dwóch tablic, które współdzielą większość pamięci. Język programowania logiki gry może całkowicie przykrywać takie detale, czyniąc je przezroczystymi dla programisty.

Ograniczenie ruchu w sieci

Spekulatywne wykonywanie programu logiki gry pozwala również na ograniczenie ruchu w sieci. W chwili obecnej serwer synchronizacyjny 20 razy na sekundę wysyła informacje o upływie czasu do wszystkich uczestniczących w rozgrywce, gdyż tylko to daje pewność, że zdarzenia na klientach zostaną przetworzone we właściwej kolejności, a czas logiczny będzie płynnie posuwał się do przodu. Spekulatywne wykonywanie pozwoliłoby na wysyłanie komunikatów tylko w przypadku zmian stanów kontrolerów, razem z informacją o upływie czasu. Upływ czasu na klientach następowałby z optymistycznym założeniem, że nic się nie dzieje, a czas płynie właściwie. W przypadku gdyby okazało się to nieprawdą, zawsze można cofnąć się w czasie.

Rozdział 6

Studium przypadku – gra sieciowa Aster

W tym rozdziale przedstawię studium przypadku gry sieciowej. Pokażę jak stworzyć kompletną, prostą grę sieciową przy pomocy prototypu. Dodatkowo dokładnie opiszę wszystkie konieczne do wykonania czynności, tak aby człowiek potrafiący programować mógł zapoznać się z prototypem. Aby bezpośrednio stosować polecenia pojawiające się w tym rozdziale, musimy użyć następującego środowiska:

- system operacyjny MS Windows,
- zainstalowany Sun J2SE SDK w wersji 1.4 z ustawioną ścieżką dostępu do katalogu bin,
- biblioteka scm.jar z prototypu pod ręką.

Prototyp działa równie dobrze w systemie Linux. Należy tylko pamiętać, że przy wywołaniu programu `java` elementy listy parametru `-cp` rozdziela dwukropek, a nie średnik:

```
java -cp .:scm.jar Aster
```

Krok 0. Sprawdzenie środowiska

W tym rozdziale pokażę jak przebić się przez narzut technologiczny związany z językiem Java i sprawdzić działanie szkieletu. Rozpocznijmy od założenia katalogu `C:\case` i przekopiowania do niego pliku `scm.jar`, który jest załączony do niniejszej pracy (zob. dodatek B, zawartość załączonej płyty CD). Następnie stwórzmy w tym katalogu plik `Aster.java` o następującej zawartości:

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import scm.game.*;
import scm.graph.*;

public class Aster extends Game {
    public void start(ScmManager mgr, int playerCount) {
    }

    public void initLocalGame(ScmManager mgr) {
    }

    public static void main(String[] args) {
```

```
            Game.startLocalGame(Aster.class, "AsterWar", 1);
        }
    }
```

Teraz po uruchomieniu:

```
C:\aster> javac -classpath scm.jar Aster.java
```

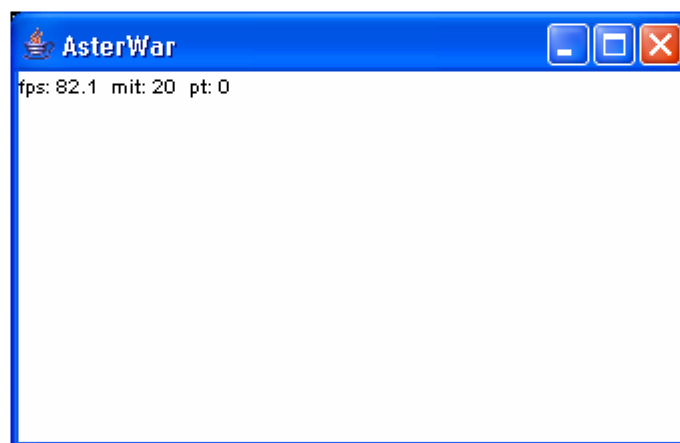
Powinien powstać plik `Aster.class`, a polecenie nie powinno wypisać żadnych komunikatów. Jeśli tak się stało, to znaczy, że środowisko jest skonfigurowane właściwe. Teraz wpiszmy:

```
C:\aster> java -cp .;scm.jar Aster
```

Spowoduje to pojawienie się białego okna o tytule `AsterWar`, co oznacza, że szkielet działa. Okno jest przedstawione na zrzucie ekranu 3. Można to okno zamknąć. W kolejnych częściach tego studium będziemy kolejno:

- modyfikować plik `Aster.java` zgodnie z przykładem,
- kompilować przykład za pomocą polecenia `javac`,
- uruchamiać przykład za pomocą polecenia `java`.

Do większości kroków dołączony jest przykładowy zrzut ekranu.



Zrzut ekranu 3. Okno prototypu

Plik `Aster.java` składa się z następujących części:

- ciąg deklaracji `import` pozwala na odwoływanie się do bibliotek zewnętrzny za pomocą krótkich identyfikatorów,
- deklaracja klasy jest wymagana przez każdy program w języku Java, w naszym przypadku jest to klasa `Aster`,
- metoda `main` pozwala na proste wywołanie przykładu za pomocą polecenia `java`. Określa przy tym nazwę dla okienka oraz liczbę graczy,
- metoda `initLocalGame` służy do inicjowania kontrolerów i będzie rozwijana w kolejnych przykładach,
- metoda `start` to główna metoda logiki gry.

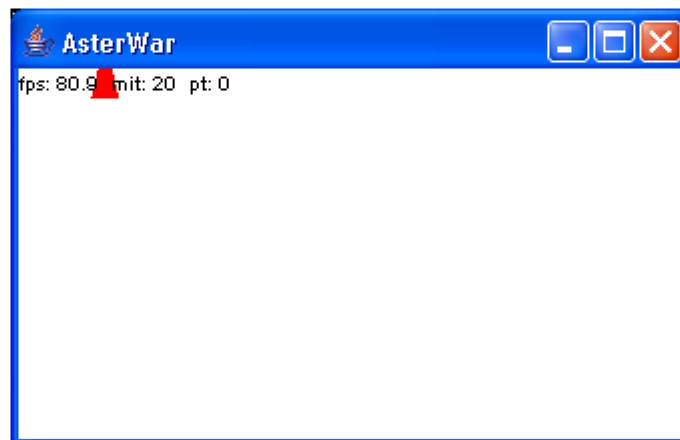
Należy zwrócić uwagę, że metody `main` oraz `initLocalGame` według projektu prototypu nie są częścią programu logiki gry. Jest to narzut technologii.

Krok 1. Pierwszy duszek

Do pliku `Aster.java` należy dodać deklarację klasy `TrianglePic`, pól `mgr`, `surf` i `clock` oraz należy podmienić definicję metody `start`:

```
class TrianglePic implements Paintable {
    public void paint(Graphics2D g) {
        g.setColor(Color.RED);
        g.fillPolygon(new int[] { 0, 1, -1 }, new int[] { -2, 2, 2 }, 3);
    }
}

ScmManager mgr;
Surface surf;
Clock clock;

public void start(ScmManager mgr, int playerCount) {
    this.mgr = mgr;
    this.surf = mgr.getSurface();
    this.clock = mgr.getClock();
    surf.setVirtualRect(0, 0, 100, 75);
    Sprite sprite = new Sprite(new TrianglePic());
    surf.addPic(sprite);
}
```

Po kompilacji i uruchomieniu przykładu w lewym górnym rogu okienka pojawi się czerwony trójkącik. Zobacz zrzut ekranu 4. Trójkąciki na zrzutach z ekranu zostały powiększone, w stosunku do kodu źródłowego, tak aby były bardziej widoczne.



Zrzut ekranu 4. Trójkącik

Klasa `TrianglePic` jest obrazkiem czerwonego trójkącika. Należy zwrócić uwagę, że według modelu prototypu jest ona częścią systemu graficznego, a więc w definicji jej metod nie można odwoływać się do stanu gry. Zainteresowanych szczegółami bibliotek języka odsyłam do [8].

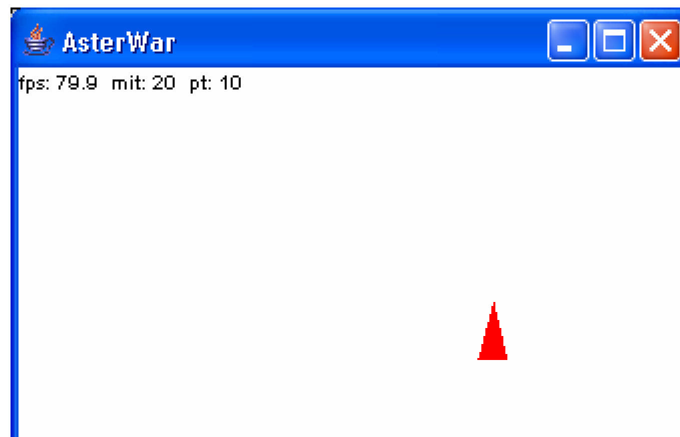
Klasa `ScmManager` jest obiektem szkieletu. Klasa `Surface` to powierzchnia, którą obserwuje gracz. System graficzny pobiera z niej drzewo duszków. Klasa `Clock` to zegar logiczny. W metodzie `start` najpierw ustalamy widoczny obszar powierzchni, następnie tworzymy duszka z obrazkiem czerwonego trójkącika i dodajemy go do powierzchni.

Krok 2. Poruszanie duszkiem

Do metody `start` należy dodać obsługę kontrolera, a do metody `initLocalGame` wstawić inicjowanie kontrolera:

```
void move(Sprite sprite) {
     Trans pos = sprite.addTrans();
     Controller ctrl = mgr.getController(1);
     for (;;) {
          if (ctrl.isKeyDown(0)) pos.translate(0, -0.1);
          if (ctrl.isKeyDown(1)) pos.translate(0, 0.1);
          if (ctrl.isKeyDown(2)) pos.translate(-0.1, 0);
          if (ctrl.isKeyDown(3)) pos.translate(0.1, 0);
          mgr.getClock().sleep(10);
     }
}

public void start(ScmManager mgr, int playerCount) {
     ...
     move(sprite);
}

public void initLocalGame(ScmManager mgr) {
     mgr.createLocalController(1, new int[] {
               KeyEvent.VK_UP, KeyEvent.VK_DOWN,
               KeyEvent.VK_LEFT, KeyEvent.VK_RIGHT,
               KeyEvent.VK_CONTROL });
}
```

Metoda `initLocalGame` tworzy kontroler składający się z klawiszy strzałek i klawisza `Ctrl`. Metoda `move` tworzy transformację duszka, a następnie w nieskończonej pętli bada stan klawiszy kontrolera, odpowiednio zmienia transformację oraz wykonuje czekanie na zegarze logicznym. Dzięki temu po uruchomieniu aplikacji za pomocą klawiszy kursora możemy poruszać duszkiem w ośmiu kierunkach – czterech za pomocą pojedynczych klawiszy i czterech za pomocą ich kombinacji.



Zrzut ekranu 5. Poruszony trójkącik

Krok 3. Obsługa bezwładności

Dodajmy teraz obsługę bezwładności:

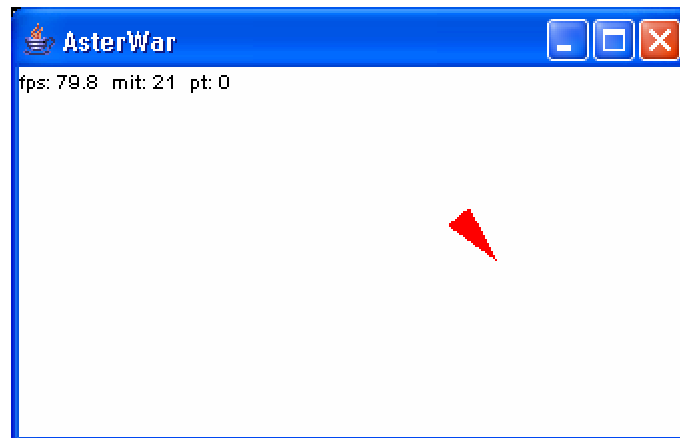
```
double wrap(double x, double total) {
     x = x % total;
     if (x < 0) x += total;
     return x;
}
```

```
class Ship {
    Trans pos;
    Controller ctrl;
    double x, y, vx, vy, dir;

    void service() {
        for (;;) {
            pos.clear();
            pos.translate(x, y);
            pos.rotate(dir);
            handleKeys();
            x = wrap(x + 0.2 * vx, 100);
            y = wrap(y + 0.2 * vy, 75);
            double v = Math.sqrt(vx * vx + vy * vy);
            if (v > 1) {
                vx = vx / v;
                vy = vy / v;
            }
            clock.sleep(10);
        }
    } // Ship.service

    void handleKeys() {
        final double S = 0.005;
        if (ctrl.isKeyDown(0)) {
            vx += S * Math.sin(dir * Math.PI / 180);
            vy -= S * Math.cos(dir * Math.PI / 180);
        }
        if (ctrl.isKeyDown(1)) {
            vx -= S * Math.sin(dir * Math.PI / 180);
            vy += S * Math.cos(dir * Math.PI / 180);
        }
        if (ctrl.isKeyDown(2))
            dir -= 1;
        if (ctrl.isKeyDown(3))
            dir += 1;
    } // Ship.handleKeys()
} // class Ship

void move(Sprite sprite) {
    Ship ship = new Ship();
    ship.ctrl = mgr.getController(1);
    ship.pos = sprite.addTrans();
    ship.service();
}
```

Nazwijmy naszego duszka trójkąta statkiem kosmicznym. Chcielibyśmy aby nasz statek poruszał się jak prawdziwy statek kosmiczny. Z tego powodu zmienimy jego reakcje na klawisze. Stan statku będzie określała jego pozycja `(x, y)`, wektor prędkości `(vx, vy)` oraz kierunek `dir` wyrażony w stopniach. Metoda `service` służy za modyfikację położenia zgodnie z wektorem prędkości oraz za normalizację prędkości, jeśli przekraczałaby dozwoloną wartość. Metoda `service` woła metodę `handleKeys`, która obsługuje klawisze. Klawisze góra/dół są odpowiedzialne za zmianę prędkości zgodnie z bieżącym kierunkiem, a klawisze lewo/prawo za zmianę kierunku.

Metoda `wrap` to naprawione modulo. Niestety w językach programowania została przyjęta bardzo dziwna definicja modulo dla liczb ujemnych, która czyni tą operację zupełnie nieprzydatną. Metoda `wrap` działa zgodnie z matematyczną definicją. Będziemy jej używali do zawijania świata. W poprzednim przykładzie, gdy duszek wyszedł poza ekran to znikał. Teraz po przejściu przez krawędź ekranu będzie pojawiał się po przeciwległej stronie.

Stan statku zamknęliśmy w klasę `Ship`, co zmniejszy liczbę zmian wymaganych w przyszłości. Jeszcze trochę elementarnej matematyki i nasz statek porusza się jak prawdziwy statek kosmiczny.



Zrzut ekranu 6. Poruszający się statek

Krok 4. Strzały

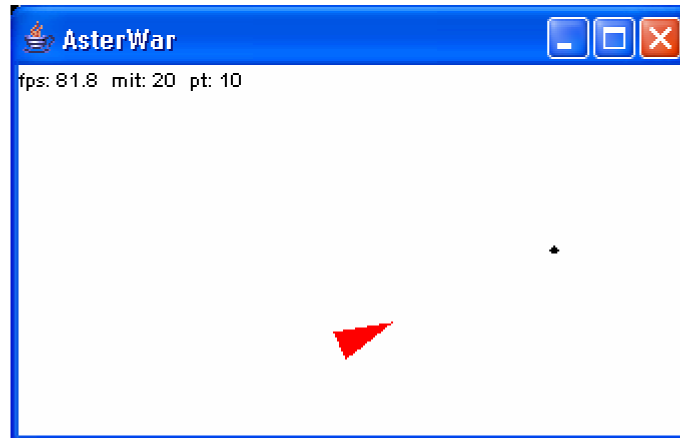
Do pliku `Aster.java` dodajmy definicję klasy `CirclePic`, metody `fireMissle` oraz zmodyfikujmy metodę `handleKeys` w klasie wewnętrznej `Ship`:

```
class CirclePic implements Paintable {
     public void paint(Graphics2D g) {
          g.setColor(Color.BLACK);
          g.fill(new Ellipse2D.Double(-0.3, -0.3, 0.6, 0.6));
     }
}

class Ship {
     ...
     void handleKeys() {
          ...
          if (ctrl.isKeyDown(4)) {
               fireMissle(x, y, dir);
          }
     } // Ship.handleKeys()
} // class Ship

void fireMissle(double x, double y, double dir) {
     Sprite sprite = new Sprite(new CirclePic());
     surf.addPic(sprite);
     Trans t = sprite.addTrans();
     double S = 1.1;
     for (int i = 0; i < 50; ++i) {
          x = wrap(x + S * Math.sin(dir * Math.PI / 180), 100);
          y = wrap(y - S * Math.cos(dir * Math.PI / 180), 75);
          t.clear();
          t.translate(x, y);
          clock.sleep(10);
     }
     surf.removePic(sprite);
}
```

Klasa `CirclePic` to rysunek kółka, analogiczny do klasy `TrianglePic`. Klasa ta jest częścią systemu graficznego. Metoda `handleKeys` w reakcji na klawisz `Ctrl` (numer 4 w metodzie `initLocalGame`) wywołuje metodę `fireMissle` przekazując jej pozycję oraz kierunek statku. Metoda ta tworzy duszka

z obrazkiem kółka, tworzy dla niego transformację i dodaje go do powierzchni. Następnie przez pół sekundy animuje ruch kółka, a na końcu usuwa je z powierzchni.

Po uruchomieniu gry i naciśnięciu przycisku `Ctrl` widzimy wylatujący i po chwili znikający pocisk. Jest pięknie. Niestety efekt pryska, gdy wprowadzimy nasz pojazd w ruch. Wydanie strzału w takim przypadku powoduje, że pojazd na czas strzału zatrzymuje się i rusza dopiero gdy strzał zniknie. Niestety jest to zgodne z tym co napisaliśmy w kodzie.



Zrzut ekranu 7. Strzał

Krok 5. Jednoczesny ruch

Jak naprawić zaistniałą sytuację? Jak sprawić aby statek po wystrzeleniu pocisku nie zatrzymywał się? Rozwiązaniem może być rozszerzenie stanu statku o opcjonalny stan pocisku:

```
class Ship {
    ...
    boolean missleActive;
    int missleLife;
    double mx, my, mdir;
    Trans missleTrans;
    Sprite missleSprite;

    void service() {
        for (;;) {
            ...
            moveMissle();
            clock.sleep(10);
        }
    }

    void moveMissle() {
        if (!missleActive)
            return;
        if (missleLife == 0) {
            surf.removePic(missleSprite);
            missleActive = false;
            return;
        }
        --missleLife;
        mx = wrap(mx + 1.1 * Math.sin(dir * Math.PI / 180), 100);
        my = wrap(my - 1.1 * Math.cos(dir * Math.PI / 180), 75);
        missleTrans.clear();
        missleTrans.translate(mx, my);
    }

    void handleKeys() {
```

```
                ...
                if (ctrl.isKeyDown(4) && !missleActive) {
                    missleActive = true;
                    missleLife = 50;
                    mx = x;
                    my = y;
                    mdir = dir;
                    missleSprite = new Sprite(new CirclePic());
                    missleTrans = missleSprite.addTrans();
                    surf.addPic(missleSprite);
                }
            }
        }
```

To przestaje być proste. Aby dodać pocisk, wystarczyło dodać jedną metodę, niestety nasz statek się zatrzymywał. Teraz, aby obsłużyć jednoczesne poruszanie się statku i pocisku, musimy rozszerzać kod w trzech miejscach. Doszły dodatkowe deklaracje, a logika strzału została podzielona na prolog, krok animacji i epilog, przy czym prolog trafił do metody `handleKeys`, a krok animacji i epilog zostały w dziwny sposób przemieszane w `moveMissle`. Doświadczeni programiści od razu wychwycą niebezpieczeństwo jakie wiąże się z rozdzieleniem w kodzie miejsca dodawania duszka do powierzchni oraz jego usuwania. Rzeczywiście – pisząc ten przykład popełniłem niewielki błąd polegający na braku sprawdzenia `missleActive` przy odpalaniu pocisku. Spowodowało to pozostawanie pocisków, jeśli kolejny został odpowiednio szybko wystrzelony. Jako że klawisze są testowane sto razy w ciągu sekundy, zostawało naprawdę dużo nieruchomych pocisków.

Wycofajmy się z tego podejścia i wróćmy do postaci pliku `Aster.java`, jaką mieliśmy po kroku 4.

Właściwy krok 5. Mikrowątki

Jak naprawić zaistniałą sytuację? Jak sprawić aby statek po wystrzeleniu pocisku nie zatrzymywał się? Tym razem po prostu uruchomimy obsługę pocisku w nowym mikrowątku.

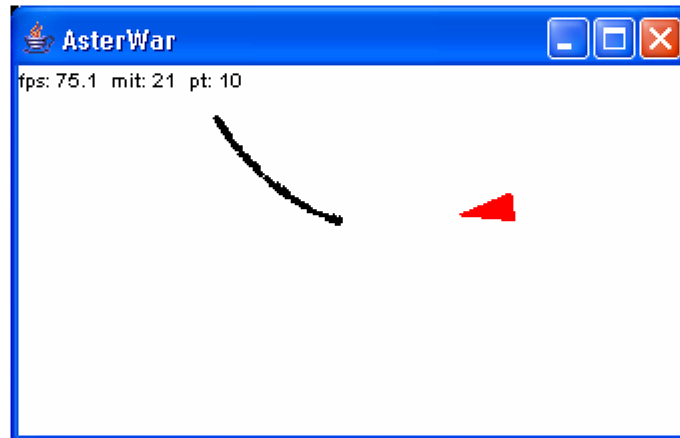
```
                if (ctrl.isKeyDown(4)) {
                    clock.startThread(new Runnable() {
                        public void run() {
                            fireMissle(x, y, dir);
                        }
                    });
                }
```

Po uruchomieniu okazuje się, że wszystko działa bardzo dobrze, a pocisk mamy dostępny nie jeden, a dowolną ich liczbę. Cała magia polega na uruchomieniu nowego mikrowątku dla pocisku. Wewnątrz tego mikrowątku możemy opisywać zachowanie pocisku z perspektywy tego pocisku. Jest to bardzo wygodny i naturalny sposób programowania. Należy zwrócić uwagę, że nie ma problemów z synchronizacją, a komunikacja między mikrowątkami jest bardzo prosta. Pokażemy ją w kolejnych krokach.

Konstrukcja:

```
clock.startThread(new Runnable() { public void run() { /*KOD*/ }});
```

To idiom uruchamiania nowego mikrowątku w prototypie. Istotne jest to, że w kodzie umieszczonym bezpośrednio w miejscu `/*KOD*/` można odwoływać się tylko do zmiennych lokalnych posiadających modyfikator `final`. Można natomiast bez ograniczeń odwoływać się do pól obiektów oraz wywoływać dowolne metody.



Zrzut ekranu 8. Salwa z poruszającego się statku

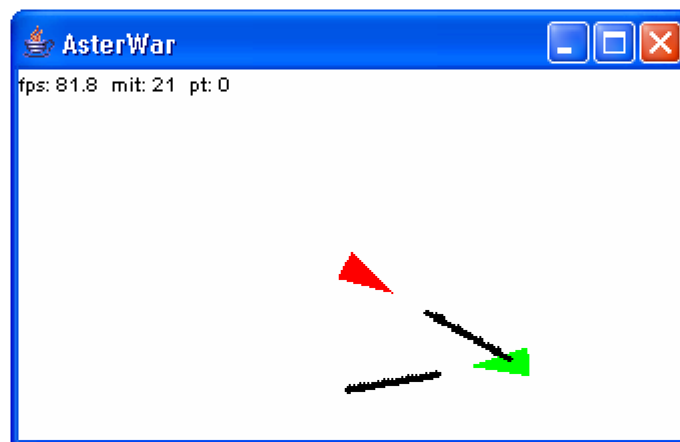
Krok 6. Dwóch graczy

Idzie nam bardzo dobrze. Teraz należy obsłużyć wielu graczy. W tym celu zmodyfikujmy klasę `TrianglePic`, metody `start`, `initLocalGame` i `main` oraz usuńmy metodę `move`.

```
class TrianglePic implements Paintable {
     int color;

     TrianglePic(int color) {
          this.color = color;
     }

     public void paint(Graphics2D g) {
          Color c = new Color(color % 2 * 255,
               color / 2 % 2 * 255, color / 4 % 2 * 255);
          g.setColor(c);
          g.fillPolygon(new int[] { 0, 1, -1 }, new int[] { -2, 2, 2 }, 3);
     }
}

public void start(ScmManager mgr, int playerCount) {
     this.mgr = mgr;
     this.surf = mgr.getSurface();
     this.clock = mgr.getClock();
     surf.setVirtualRect(0, 0, 100, 75);
     for (int i = 1; i <= playerCount; ++i) {
          final Ship ship = new Ship();
          Sprite sprite = new Sprite(new TrianglePic(i));
          surf.addPic(sprite);
          ship.ctrl = mgr.getController(i);
          ship.pos = sprite.addTrans();
          clock.startThread(new Runnable() {
               public void run() {
                    ship.service();
               }
          });
     }
}

public void initLocalGame(ScmManager mgr) {
     mgr.createLocalController(1, new int[] {
               KeyEvent.VK_UP, KeyEvent.VK_DOWN,
               KeyEvent.VK_LEFT, KeyEvent.VK_RIGHT,
               KeyEvent.VK_CONTROL });
     mgr.createLocalController(2, new int[] {
               KeyEvent.VK_W, KeyEvent.VK_S,
               KeyEvent.VK_A, KeyEvent.VK_D,
```

```
                KeyEvent.VK_SHIFT });
    }

    public static void main(String[] args) {
        Game.startLocalGame(Aster.class, "AsterWar", 2);
    }
```

W metodzie `main` przekazujemy liczbę `2` do metody `startLocalGame`. Liczba ta oznacza liczbę graczy. W `initLocalGame` tworzymy dwa kontrolery. Tak jak poprzednio, pierwszy kontroler będzie używał klawiszy kursora do sterowania statkiem oraz klawisza Ctrl jako strzału. Drugi kontroler będzie używał klawiszy liter A, S, D, W oraz klawisza Shift. Obrazek trójkąta sparametryzowaliśmy numerem koloru. Metoda `paint` w ciekawy sposób przelicza ten numer na właściwy kolor. Zastosowana formuła umożliwia stworzenie 6 podstawowych kolorów i koloru czarnego. Ósmym kolorem będzie niestety kolor biały. W metodzie `start` zastosowaliśmy mikrowątki dla każdego ze statków graczy. To pozwoliło na pozostawienie ich logiki całkowicie niezmienionej.



Zrzut ekranu 9. Dwóch strzelających graczy

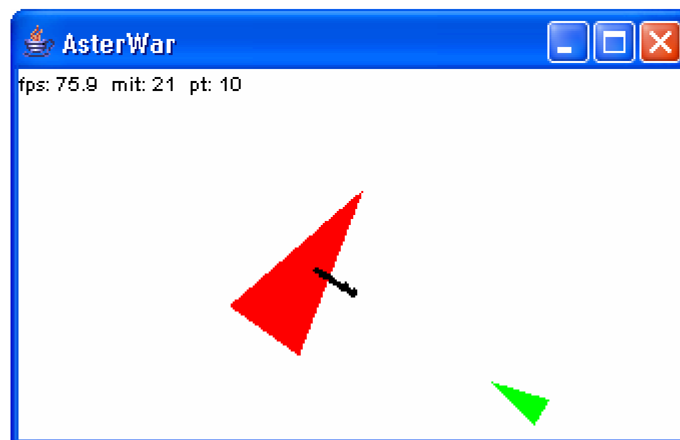
Po uruchomieniu pojawią się dwa statki w lewym górnym rogu ekranu, którymi można niezależnie sterować.

Krok 7. Trafienia

Do tej pory nasza gra była bardzo pokojowa. Latały pociski, ale nie robiły nikomu krzywdy. Aby gra miała sens, trafienia statków muszą być wizualizowane:

```
Ship[] ships;

class Ship {
    ...
    void doHit() {
        for (int i = 0; i < 30; ++i) {
            pos.scale(1.05);
            clock.sleep(10);
        }
        for (int i = 0; i < 70; ++i) {
            pos.scale(0.9);
            clock.sleep(10);
        }
    } // Ship.doHit()
    ...
} // class Ship

void fireMissle(double x, double y, double dir) {
    ...
```

```
    for (int i = 0; i < 50; ++i) {
        x = wrap(x + S * Math.sin(dir * Math.PI / 180), 100);
        y = wrap(y - S * Math.cos(dir * Math.PI / 180), 75);
        if (i > 3) {
            if (checkCollision(x, y))
                break;
        }
        t.clear();
        t.translate(x, y);
        clock.sleep(10);
    }
    ...
}

boolean checkCollision(double mx, double my) {
    for (int i = 0; i < ships.length; ++i) {
        final Ship ship = ships[i];
        double dx = mx - ship.x;
        double dy = my - ship.y;
        double d = Math.sqrt(dx * dx + dy * dy);
        if (d <= 1.5) {
            clock.startThread(new Runnable() {
                public void run() {
                    ship.doHit();
                }
            });
            return true;
        }
    }
    return false;
}

public void start(ScmManager mgr, int playerCount) {
    ...
    ships = new Ship[playerCount];
    for (int i = 1; i <= playerCount; ++i) {
        final Ship ship = new Ship();
        ships[i - 1] = ship;
        ...
    }
}
```



Zrzut ekranu 10. Trafiony gracz

Aby obsługiwać trafienia musimy nawiązać komunikację między mikrowątkami obsługującymi ruch statku, a mikrowątkami obsługującymi ruch pocisków. Osiągamy to poprzez zadeklarowanie pola `ships` w klasie `Aster`. Pole to przechowuje tablicę stanów statków. Jego inicjowanie odbywa się w metodzie start. W metodzie obsługującej ruch pocisków wywołujemy metodę `checkCollision` sprawdzającą czy w pobliżu danej pozycji nie znajduje się statek. Metoda ta podejmuje stosowane

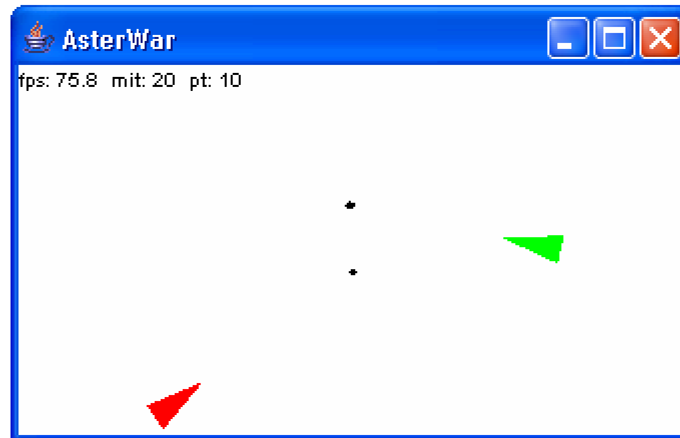
akcje w przypadku napotkania statku. Warunek `i>3` definiuje czas potrzebny na „uzbrojenie” pocisku. Bez tego warunku pocisk niszczyłby pojazd, który go wystrzeliwuje. W przypadku trafienia metoda `checkCollision` wywołuje metodę `doHit` statku. Metoda ta animuje trafienie poprzez powiększenie i pomniejszenie statku. Zauważmy, że stan tej animacji nie jest przechowywany w stanie statku.

Ważne jest to, że nie potrzebujemy żadnego kodu do synchronizacji mikrowątków, a mimo to nasz kod jest poprawny. Jest to spowodowane tym, że przełączenia między mikrowątkami mogą następować tylko w przypadku wywołania metody `Clock.sleep`. Komunikacja mikrowątków jest więc bardzo prosta.

Krok 8. Poprawienie grywalności

Nasza gra jest już prawie gotowa, jednak należy popracować nad jej rzeczywistym sensem. Obecnie trzymanie wciśniętego klawisza strzału powoduje ciągłą salwę, a pojazd raz trafiony traci wszelkie możliwości obrony. Ograniczmy liczbę możliwych do oddania strzałów w przedziale czasowym oraz ustawiajmy trafiony statek w losowym miejscu planszy, aby mógł powrócić do rozgrywki:

```
class Ship {
     ...
     boolean hit;
     double nextFireTime;

     void reset() {
          x = 100 * mgr.rand();
          y = 75 * mgr.rand();
          dir = 360 * mgr.rand();
          vx = 0;
          vy = 0;
     }

     void doHit() {
          if (hit) return;
          hit = true;
          ...
          hit = false;
     }

     void service() {
          reset();
          for (;;) {
               if (hit) {
                    clock.sleep(1000);
                    reset();
               }
               ...
          }
     }

     void handleKeys() {
          ...
          if (ctrl.isKeyDown(4) && clock.getTime() >= nextFireTime) {
               nextFireTime = clock.getTime() + 1000;
               ...
          }
     }
     ...
} // class Ship
```



Zrzut ekranu 11. Ograniczona częstotliwość strzałów

Nasza gra nadaje się do przeprowadzenia rozgrywki.

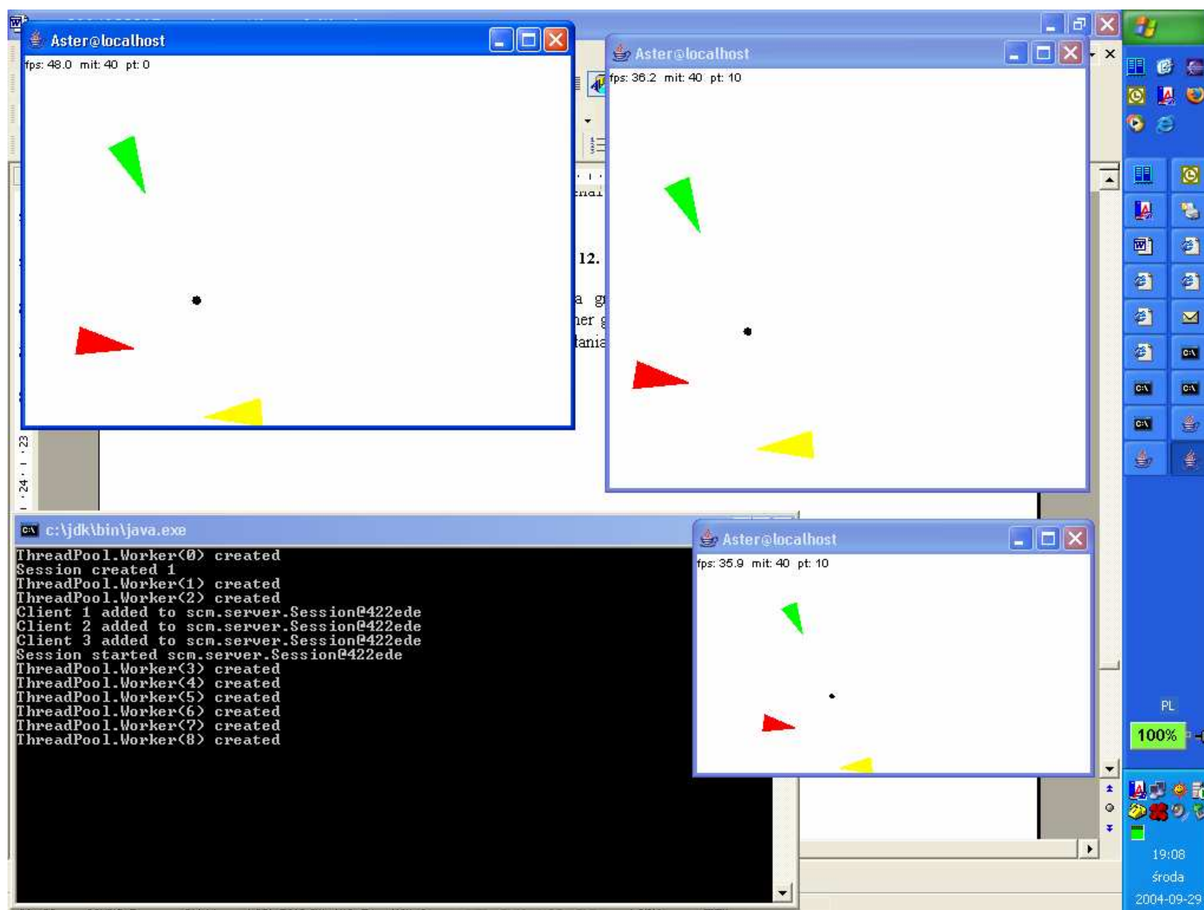
Krok 9. Sieciowość

Stworzyliśmy grę dla dwóch graczy, ale chcieliśmy stworzyć grę sieciową. Zabierzmy się do pracy. Wystarczy dodać metodę:

```
public void initNetworkGame(
     ScmManager mgr,
     int playerNumber,
     int playerCount) {
     mgr.createLocalController(playerNumber, new int[] {
               KeyEvent.VK_UP, KeyEvent.VK_DOWN,
               KeyEvent.VK_LEFT, KeyEvent.VK_RIGHT,
               KeyEvent.VK_CONTROL });
}
```

I kod naszej gry nadaje się do uruchomienia po sieci. Metoda `initNetworkGame` ma za zadanie stworzenie kontrolera dla gracza lokalnego w rozgrywce sieciowej. Kontroler tworzymy dokładnie tak samo, jak w przypadku gry lokalnej. Aby uruchomić naszą grę należy najpierw uruchomić serwer, a później klientów:

```
start java -cp scm.jar scm.server.ScmServerMain 3
start java -cp .;scm.jar scm.game.GameMain Aster localhost 1 3
start java -cp .;scm.jar scm.game.GameMain Aster localhost 2 3
start java -cp .;scm.jar scm.game.GameMain Aster localhost 3 3
```



Zrzut ekranu 12. Rozgrywka sieciowa trzech graczy

Parametrem polecenia `ScmServerMain` jest liczba graczy, a parametry polecenia `GameMain` to odpowiednio nazwa klasy z grą, adres serwera, numer gracza i liczba graczy. Należy zwrócić uwagę, aby liczba graczy była zgodna we wszystkich wywołaniach. Na zrzucie z ekranu 12 widać rozgrywkę sieciową trzech graczy oraz działający serwer.

Pełny kod gry znajduje się w dodatku A.

Wnioski

Ten scenariusz pokazuje, że tworzenie logiki gry może być bardzo proste. W każdym kroku dodawaliśmy po kilka wierszy kodu, które dokładnie odzwierciedlał nasze intencje. Krok 5 pokazuje przewagę mikrowątków nad programowanie jednowątkowym. Dzięki zastosowaniu mikrowątków mogliśmy opisać logikę statku i logikę pocisku zupełnie niezależnie. Logika pocisku została opisana z perspektywy pocisku, analogicznie logika statku – z perspektywy statku. W dalszych krokach pokazaliśmy, że mikrowątki pozwalają na dodawanie komunikacji między duszkami sukcesywnie, w miarę potrzeby. Dzięki mikrowątkom animacje zostały opisane zupełnie niezależnie od logiki gry. Krok 9 pokazał, że dzięki wirtualizacji kontrolerów, możemy niemal bez pisania kodu uzyskać grę sieciową.

Ten scenariusz pokazuje również narzut technologiczny związany z językiem Java oraz niedojrzałością prototypu. Proste obiekty graficzne musieliśmy implementować za pomocą bibliotek Javy, uruchamianie mikrowątków wymaga pisania dużej ilości zbędnego kodu oraz posiada dziwne składniowe ograniczenia, takie jak wymaganie modyfikatora `final` przy zmiennych. Są to oczywiście niewielkie problemy, które powinny zostać usunięte przy tworzeniu produktu platformy.

Rozdział 7

Podsumowanie

Zaprezentowany prototyp platformy jest bardzo elastyczny i pozwala na tworzenie ciekawych gier przy jednoczesnym zachowaniu bardzo dużej prostoty całego procesu. W porównaniu z innymi popularnymi platformami, takimi jak Java J2ME czy Macromedia Flash, udostępnia nowatorski sposób programowania logiki gier oraz umożliwia tworzenie gier sieciowych, czego wspomniane platformy nie wspierają. Dopracowanie platformy umożliwiłoby zastosowanie jej w edukacji, co mogłoby wpłynąć na upowszechnienie nauki programowania.

Mikrowątki i gry sieciowe

Prototyp oraz zaimplementowane na jego podstawie gry pokazują, że mikrowątki w połączeniu z zegarem logicznym są bardzo wygodnym narzędziem do programowania logiki gry. Pozwalają na wyrażenie intencji w bardzo naturalny sposób i są wolne od głównych wad wątków. Komunikacja między mikrowątkami odbywa się przez zmienne dzielone i nie wymaga innych mechanizmów synchronizacyjnych niż czekanie na zegarze logicznym.

W modelu prototypu zastosowano automatyczną synchronizację sieciową. Umożliwia ona tworzenie gier sieciowych bez konieczności pisania i testowania kodu odpowiedzialnego za komunikację sieciową. Aby w pełni wykorzystać możliwości tego mechanizmu konieczne jest zaimplementowanie utrwalania stanu logiki gry, które umożliwi spekulatywne wykonywanie programu logiki gry. W celu zwiększenia zakresu zastosowań należy rozszerzyć model o możliwość pokazywania informacji specyficznych dla gracza lokalnego.

Platforma jako produkt

W celu wdrożenia platformy do zastosowań edukacyjnych niezbędne jest stworzenie na jej podstawie produktu. Prototyp pokazuje model programowania oraz model automatycznej synchronizacji sieciowej, który z powodzeniem może stanowić podstawę platformy. Korzystanie z prototypu wymaga jednak dobrej znajomości języka Java oraz posługiwania się powłoką systemu operacyjnego. Aby platforma stała się produktem wymagane są prace, które zdejmą cały niepotrzebny narzut technologiczy.

Dopracowanie modelu programowania i bibliotek. Należy stworzyć prosty język programowania logiki gier lub dopracować sposób wykorzystania języka Java, maksymalnie pozbywając się narzutu z nim związanego. Kluczowe jest również dopracowanie bibliotek do obsługi grafiki, dźwięku i kontrolerów.

Opracowanie prostego środowiska programistycznego. Środowisko udostępnia zarządzanie źródłami i zasobami w prosty sposób oraz umożliwia edycję, kompilację, uruchamianie oraz udostępnianie gier.

Opracowanie sposobu udostępniania stworzonych gier. Warto rozważyć możliwość tworzenia plików wykonywalnych lub stworzenie specjalnego formatu pliku gry. W drugim przypadku należy stworzyć pakiet instalacyjny dla systemu uruchomieniowego oraz zapewnić pełne bezpieczeństwo oraz przenośność wykonywania plików gier.

Opracowanie portalu gier. Bardzo istotnym elementem nauczania jest wspieranie wymiany doświadczeń między uczniami. W przypadku platformy taką funkcję mógłby pełnić internetowy portal gier. Byłby on częścią platformy służącą do udostępniania swoich gier na stronach WWW, umożliwiałby zestawianie połączeń dla gier sieciowych oraz służyłby jako serwer synchronizacyjny. W portalu należy udostępnić forum dyskusyjne, które pozwoli na wymianę doświadczeń między twórcami gier. Pożądaną funkcjonalnością jest możliwość oceny gier na podstawie częstotliwości ich używania przez graczy.

Opracowanie scenariuszy lekcji. Równie ważne co opracowanie platformy jest opracowanie sposobu jej wykorzystania. Należy opracować przykładowe programy, zadania, a na ich podstawie stworzyć scenariusze lekcji programowania.

Osobiście jestem przekonany, że nauka programowania będzie powszechna. Jestem również przekonany, że będzie to lubiany przez uczniów przedmiot, dzięki któremu myślenie abstrakcyjne i logiczne stanie się bardziej zrozumiałe dla ogółu społeczeństwa. Mam nadzieję, że moja pracy przyczyni się do osiągnięcia tego celu, co najmniej przez wskazanie pewnych kierunków rozwoju nauczania informatyki.

Dodatek A. Listing gry Aster

```
 1: import java.awt.*;
 2: import java.awt.event.*;
 3: import java.awt.geom.*;
 4: import scm.game.*;
 5: import scm.graph.*;
 6:
 7: public class Aster extends Game {
 8:     class TrianglePic implements Paintable {
 9:         int color;
10:
11:         TrianglePic(int color) {
12:             this.color = color;
13:         }
14:
15:         public void paint(Graphics2D g) {
16:             Color c = new Color(color % 2 * 255,
17:                 color / 2 % 2 * 255, color / 4 % 2 * 255);
18:             g.setColor(c);
19:             g.fillPolygon(new int[] { 0, 1, -1 },
20:                           new int[] { -2, 2, 2 }, 3);
21:         }
22:     }
23:
24:     class CirclePic implements Paintable {
25:         public void paint(Graphics2D g) {
26:             g.setColor(Color.BLACK);
27:             g.fill(new Ellipse2D.Double(-0.3, -0.3, 0.6, 0.6));
28:         }
29:     }
30:
31:     ScmManager mgr;
32:     Surface surf;
33:     Clock clock;
34:
35:     Ship[] ships;
36:
37:     double wrap(double x, double total) {
38:         x = x % total;
39:         if (x < 0)
40:             x += total;
41:         return x;
42:     }
43:
44:     class Ship {
45:         Trans pos;
46:         Controller ctrl;
47:         double x, y, vx, vy, dir;
48:         boolean hit;
49:         double nextFireTime;
50:
51:         void reset() {
52:             x = 100 * mgr.rand();
53:             y = 75 * mgr.rand();
54:             dir = 360 * mgr.rand();
55:             vx = 0;
56:             vy = 0;
57:         }
58:
59:         void doHit() {
60:             if (hit)
61:                 return;
62:             hit = true;
63:             for (int i = 0; i < 30; ++i) {
64:                 pos.scale(1.05);
65:                 clock.sleep(10);
```

```
 66:             }
 67:             for (int i = 0; i < 70; ++i) {
 68:                 pos.scale(0.9);
 69:                 clock.sleep(10);
 70:             }
 71:             hit = false;
 72:         }
 73:
 74:         void service() {
 75:             reset();
 76:             for (;;) {
 77:                 if (hit) {
 78:                     clock.sleep(1000);
 79:                     reset();
 80:                 }
 81:                 pos.clear();
 82:                 pos.translate(x, y);
 83:                 pos.rotate(dir);
 84:                 handleKeys();
 85:                 x = wrap(x + 0.2 * vx, 100);
 86:                 y = wrap(y + 0.2 * vy, 75);
 87:                 double v = Math.sqrt(vx * vx + vy * vy);
 88:                 if (v > 1) {
 89:                     vx = vx / v;
 90:                     vy = vy / v;
 91:                 }
 92:                 clock.sleep(10);
 93:             }
 94:         }
 95:
 96:         void handleKeys() {
 97:             final double S = 0.005;
 98:             if (ctrl.isKeyDown(0)) {
 99:                 vx += S * Math.sin(dir * Math.PI / 180);
100:                 vy -= S * Math.cos(dir * Math.PI / 180);
101:             }
102:             if (ctrl.isKeyDown(1)) {
103:                 vx -= S * Math.sin(dir * Math.PI / 180);
104:                 vy += S * Math.cos(dir * Math.PI / 180);
105:             }
106:             if (ctrl.isKeyDown(2))
107:                 dir -= 1;
108:             if (ctrl.isKeyDown(3))
109:                 dir += 1;
110:             if (ctrl.isKeyDown(4) &&
111:                     clock.getTime() >= nextFireTime) {
112:                 nextFireTime = clock.getTime() + 1000;
113:                 clock.startThread(new Runnable() {
114:                     public void run() {
115:                         fireMissle(x, y, dir);
116:                     }
117:                 });
118:             }
119:         }
120:     }
121:
122:     void fireMissle(double x, double y, double dir) {
123:         Sprite sprite = new Sprite(new CirclePic());
124:         surf.addPic(sprite);
125:         Trans t = sprite.addTrans();
126:         double S = 1.1;
127:         for (int i = 0; i < 50; ++i) {
128:             x = wrap(x + S * Math.sin(dir*Math.PI/180), 100);
129:             y = wrap(y - S * Math.cos(dir*Math.PI/180), 75);
130:             if (i > 3) {
131:                 if (checkCollision(x, y))
132:                     break;
133:             }
134:             t.clear();
135:             t.translate(x, y);
136:             clock.sleep(10);
137:         }
138:         surf.removePic(sprite);
139:     }
140:
141:     boolean checkCollision(double mx, double my) {
142:         for (int i = 0; i < ships.length; ++i) {
```

```
143:             final Ship ship = ships[i];
144:             double dx = mx - ship.x;
145:             double dy = my - ship.y;
146:             double d = Math.sqrt(dx * dx + dy * dy);
147:             if (d <= 1.5) {
148:                 clock.startThread(new Runnable() {
149:                     public void run() {
150:                         ship.doHit();
151:                     }
152:                 });
153:                 return true;
154:             }
155:         }
156:         return false;
157:     }
158: 
159:     public void start(ScmManager mgr, int playerCount) {
160:         this.mgr = mgr;
161:         this.surf = mgr.getSurface();
162:         this.clock = mgr.getClock();
163:         surf.setVirtualRect(0, 0, 100, 75);
164:         ships = new Ship[playerCount];
165:         for (int i = 1; i <= playerCount; ++i) {
166:             final Ship ship = new Ship();
167:             ships[i - 1] = ship;
168:             Sprite sprite = new Sprite(new TrianglePic(i));
169:             surf.addPic(sprite);
170:             ship.ctrl = mgr.getController(i);
171:             ship.pos = sprite.addTrans();
172:             clock.startThread(new Runnable() {
173:                 public void run() {
174:                     ship.service();
175:                 }
176:             });
177:         }
178:     }
179: 
180:     public void initNetworkGame(ScmManager mgr,
181:             int playerNumber, int playerCount) {
182:         mgr.createLocalController(playerNumber, new int[] {
183:                 KeyEvent.VK_UP, KeyEvent.VK_DOWN,
184:                 KeyEvent.VK_LEFT, KeyEvent.VK_RIGHT,
185:                 KeyEvent.VK_CONTROL });
186:     }
187: 
188:     public void initLocalGame(ScmManager mgr) {
189:         mgr.createLocalController(1, new int[] {
190:                 KeyEvent.VK_UP, KeyEvent.VK_DOWN,
191:                 KeyEvent.VK_LEFT, KeyEvent.VK_RIGHT,
192:                 KeyEvent.VK_CONTROL });
193:         mgr.createLocalController(2, new int[] {
194:                 KeyEvent.VK_W, KeyEvent.VK_S,
195:                 KeyEvent.VK_A, KeyEvent.VK_D,
196:                 KeyEvent.VK_SHIFT });
197:     }
198: 
199:     public static void main(String[] args) {
200:         Game.startLocalGame(Aster.class, "AsterWar", 2);
201:     }
202: }
```

Dodatek B. Zawartość płyty CD

ags_praca_mgr.pdf	Praca magisterska w formacie PDF.
scm.jar	Skompilowana biblioteka zawierająca prototyp platformy.
scmlib024\	Źródła prototypu platformy.
scmgames\	Przykładowe gry: Dynia, Literki, Aster7Net.
scmgames\literki.bat	Uruchamia grę Literki na komputerze z Windows i zainstalowaną Javą.
scmgames\dyna.bat	Uruchamia grę Dynia.
scmgames\src	Źródła przykładowych gier.
scmgames\resources	Zasoby dla przykładowych gier, obrazki i dźwięki.
casestudy\	Źródła dla studium przypadku.
scm.war	Aplikacja Web gotowa do uruchomienia w kontenerze serwletów. Testowana w Apache Tomcat 4.1. Pozwala na uruchamianie gier Literki, Dynia, Aster7Net z poziomu przeglądarki WWW.

Bibliografia

[1] Mark DeLoura , „Perełki programowania gier. Vademecum profesjonalisty. Tom 2", Helion, 2002.

[2] Derek Franklin, Jobe Makar, „Macromedia Flash MX 2004 ActionScript : Training from the Source", Macromedia Press, 2003.

[3] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, „Java Language Specification, Second Edition", The Sun Microsystems Press, 2000.

[4] Joe Hoffert, Kenneth Goldman, „Microthread, An Object Behavioral Pattern for Managing Object", The 1998 Pattern Languages of Programs Conference, Illinois, USA 1998.

[5] Tim Lindholm, Frank Yellin, „The Java Virtual Machine Specification", Second Edition, Addison-Wesley, 2000.

[6] Macromedia, „Flash Documentation", http://www.macromedia.com/support/documentation/en/flash/, 2004

[7] Jordi Martin Perez, „J2MEGL: Java 2 Micro Edition Game Library", http://j2megl.sourceforge.net/, 2004

[8] Sun Microsystems, „JavaTM 2 Platform Standard Edition v1.4.2 API", http://java.sun.com/j2se/1.4.2/docs/index.html, 2003.

[9] Sun Microsystems, „Java 2 Platform, Micro Edition (J2ME)" , http://java.sun.com/j2me/index.jsp, 2004

[10] Christian Tismer, "Stackless Python. A Python Implementation That Does Not Use The C Stack", http://www.stackless.com, 1999-2004.