Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Michał Słapa
Nr albumu: 201095

Architektura, projekt i realizacja sieciowej gry akcji czasu rzeczywistego

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dr. Janusza Jabłonowskiego
Instytut Informatyki

grudzień 2007

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

W toku pracy magisterskiej miałem okazję spróbować własnych sił w przedsięwzięciu, jakim jest tworzenie od postaw własnej gry komputerowej. Projekt dał mi okazję do przyjrzenia się bliżej programistycznym aspektom elektronicznej rozrywki i zmierzenia się ze stawianymi przez nie problemami. Poniższa praca powstała jako podsumowanie mojego projektu i ma na na celu przedstawienie zarówno ogólnej problematyki tworzenia i programowania gier komputerowych jak i dokładniejszą analizę wybranych zagadnień z nimi związanych. W całej pracy teoretycznej odnoszę się nie tylko do doświadczeń nabytych przy magisterskim projekcie realizowanym w ramach tej pracy, ale również do doświadczeń zawodowych, gdyż na co dzień jestem programistą gier, dotychczas pracującym prawie dwa lata nad niedawno zakończonym projektem "Wiedźmin: Gra Komputerowa".

Słowa kluczowe

gra komputerowa, gra dla wielu graczy, serwer

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

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

Tytuł pracy w języku angielskim

Architecture, design and implementation of multiplayer action game

Spis treści

Wprowadzenie

Moja praca magisterska podzielona jest na dwie części. Praktyczną, w zakresie której opracowałem sieciową grę czasu rzeczywistego, skupiając się przede wszystkim na opracowaniu uniwersalnego i skalowalnego silnika gry. Część teoretyczną pracy stanowi natomiast niniejszy dokument, w którym skupiam się na opisie zagadnień związanych z tworzeniem gier komputerowych, z którymi spotkałem się przede wszystkim w toku prac nad projektem. Jednocześnie konfrontuję tu zastosowane przez mnie rozwiązania z tymi odnalezionymi w literaturze, lub z którymi miałem do czynienia, w trakcie pracy zawodowej nad niedawno wydanym tytułem "Wiedźmin: Gra Komputerowa" [TW07].

0.1. Cel

Zadaniem, które przed sobą postawiłem, pod kątem pracy praktycznej, było stworzenie prototypu sieciowej gry komputerowej, w którym miałem zamiar zastosować własne, myślę, że ciekawe rozwiązania w kwestii architektury aplikacji. Zależało mi szczególnie, na stworzeniu wydajnej architektury klient serwer o dosyć nietypowym sposobie synchronizacji symulacji po obu stronach. Problem spójności świata gry na kilku połączonych maszynach nie jest bynajmniej banalny i pierwsze szeroko stosowane rozwiązania zupełnie nie skalowały się do gier, w których grać może więcej niż maksymalnie kilkunastu graczy. Rozwiązania, które przyjąłem mają zapewnić wydajność i bezpieczeństwo aplikacji serwera, umożliwić grę możliwe dużej liczby graczy, a sama architektura aplikacji ma ułatwić rozbudowę i zmiany projektu.

Jednocześnie w teoretycznej części pracy, główny nacisk chciałem położyć na analizę wybranych, najpopularniejszych aspektów programowania gier. Zastanawiam się tu nad rozwiązaniami algorytmicznymi i projektowymi, opierając się na własnych przemyśleniach, literaturze i zaobserwowanych w praktyce rozwiązaniach.

0.2. Następne rozdziały

Praca składa się z czterech rozdziałów i dodatku. W rozdziale 1 przybliżam ogólnie problematykę i dziedzinę projektu. Wymagania i opis projektu od strony końcowego użytkownika przedstawiam w rozdziale 2. Sercem mojej pracy jest rozdział 3, w którym skupiam się na zagadnieniach implementacyjnych konkretnych modułów gry, zarówno w kontekście mojej pracy, jak i przy szerszym spojrzeniu na problem. Na końcu w rozdziale 4 opisuję jak widzę ewentualny rozwój mojej aplikacji oraz próbuję wyciągnąć z prac nad nią końcowe wnioski. Zamieszczam w nim również wyniki przeprowadzonych testów wydajnościowych mojej aplikacji.

Rozdział 1

Tematyka projektu

Moim projektem magisterskim jest prosta gra komputerowa[1]. Nie miałem w intencji porywania się z motyką na słońce i tworzenia kompletnego oraz rozległego projektu. Pisanie gier wymaga implementacji wielu różnych mechanizmów i by stworzyć produkt, przykuwający odbiorcę na długie godziny konieczne jest wykonanie nie tylko wielu prac programistycznych nad samą grą. Potrzebne jest stworzenie zestawu narzędzi do implementacji mechaniki gry, jej projekt i implementacja, stworzenie grafiki, dźwięku i ostatecznie koordynacja wszystkich tych elementów. Mogłem spróbować pracować nad wszystkimi tymi elementami jednakowo i stworzyć mały, zamknięty projekcik o niewielkich możliwościach rozbudowy. Wolałem jednak stworzyć pewien szkielet gry, szczególnie dopracowany od strony programistycznej. Interesuje się grami jako programista, dlatego najbardziej zależało mi na implementacji mocnego silnika[2] gry, szczególnie skupiając się na aspektach sieciowych oraz wydajnościowych aplikacji serwera. Oczywiście w mojej pracy mamy zarówno trójwymiarową grafikę jak i przyjemną, choć prostą rozgrywkę sieciową, jednak nie na to położony był główny nacisk w projekcie.

1.1. Gra komputerowa

Skoro mówimy o grze komputerowej, czym ona jest z punktu widzenia programisty?

Pisanie gry jest tworzeniem pewnej symulacji rzeczywistości. Jeśli popatrzymy na grę jako program, to jej podstawowym zadaniem jest symulowanie pewnej wyspecyfikowanej abstrakcji świata. Wraz z postępem techniki, abstrakcja ta może być coraz bardziej zbliżona, zarówno do świata rzeczywistego, jak i fantastycznych światów powstałych w umysłach twórców. Oczywiście nigdy nie będzie możliwe zupełne odtworzenie świata i mechanizmów nim kierujących w grze, dlatego konieczne będą zawsze pewne uproszczenia i przybliżenia.

Choć symulacja nigdy nie będzie "dokładna", można zastanowić się, czy miarą "jakości" gry jest możliwie dokładne odwzorowanie przez nią rzeczywistości. Na pewno nadaje to grze realizmu i umożliwia większe zaangażowanie się gracza w przedstawiany świat, ale moim zdaniem nie to jest kwintesencją dobrego produktu. Chodzi jednak o zabawę, odciągającą na kilka chwil od rzeczywistości. To co pokazujemy na ekranie powinno być nie tyle realistyczne, co po prostu ładne, by nie razić gracza przyzwyczajonego do wysokiej jakości grafiki.

[1] Gra komputerowa - rekreacyjna aplikacja, będąca źródłem rozrywki. Z powodu osobistego przyzwyczajenia, w toku pracy nadużywam tego sformułowania odnoszącego się z definicji do gier działających na komputerach osobistych. Prawidłowo powinienem używać określenia gra wideo. Pierwsze gry wideo, to powstała w 1958 "Tennis for Two" i w 1962 "Spacewar!".

[2] Silnik gry - implementacja kluczowej mechaniki gry wideo, bądź innej interaktywnej aplikacji graficznej czasu rzeczywistego. Może obejmować kluczowe mechanizmy gry, w tym symulację świata, wyświetlanie, obsługę dźwięku i fizyki, a także mechanizmy zarządzania zasobami oraz narzędzia do ich edycji.

Jeżeli więc popatrzymy na grę jako na symulację zauważymy, że naszym zadaniem przy jej tworzeniu, będzie próba naśladowania pewnych mechanizmów przedstawianego świata. Sposób naszego naśladowania tych mechanizmów tylko w niektórych przypadkach będzie oparty na modelu fizycznym. Zwykle będziemy opierać się na zupełnie niezależnych od niego rozwiązaniach i obserwacjach. Dobrym przykładem jest na przykład model oświetlenia w grafice komputerowej. Próba mimiki praw rządzących światłem w programach komputerowych wymaga zbyt dużej mocy obliczeniowej by było to wykonywane w czasie rzeczywistym. Istnieją techniki śledzenia promieni (Ray Tracing), ale korzysta się z nich wyłącznie na potrzeby tworzenia statycznych obrazów, bądź animacji, których generowanie pochłania zbyt dużo czasu, by mogłyby być tworzone na bieżąco. Trójwymiarowa grafika komputerowa, wyświetlana w czasie rzeczywistym, od początku powstawała w oparciu o uproszczone modele oświetlenia (Lamberta, model Ponga) i techniki, które dobrze sprawdzały się w praktyce, jednak nie bazowały bezpośrednio na rzeczywistych prawach fizycznych.

Świat mojej gry jest bardzo umowny, ale nie oceniałem mojego silnika pod względem tego jak realistyczna będzie oparta na nim gra. By umożliwić stworzenie przyjemnej w odbiorze gry w moim silniku skupiłem się na własnościach takich, jak udział możliwie dużej liczby graczy, przy jednoczesnym zachowaniu modelu zręcznościowej gry akcji.

1.1.1. Rynek gier komputerowych

Gry komputerowe kojarzą nam się wszystkim głównie z zabawą, trzeba więc zastanowić się, czy można zajmować się nimi na poważnie. Początki rynku gier komputerowych datuje się na późne lata siedemdziesiąte i był to wówczas niszowy rynek hobbistyczny, zyskujący popularność wraz z dostępnością komputerów osobistych.

Jeszcze we wczesnych latach dziewięćdziesiątych, największe hity niezbyt jeszcze upowszechnionej, komputerowej rozrywki powstawały często rękami samotnych twórców. W tych też czasach, powstawały pierwsze studia zajmujące się profesjonalnie tworzeniem gier przeznaczonych na komputery osobiste oraz swoją działalność rozpoczynali pierwsi wydawcy. Rynek rozwijał się wraz z rozwojem sprzętu komputerowego, w dużej mierze zresztą sam był motorem tego rozwoju. Wciąż wykazuje on dużą niestabilność. Ciągle powstają nowi producenci gier, często na zgliszczach tych, którym się nie udało. Rynek jednak powoli osiąga dojrzałość, a dochody jakie notuje pozwalają mu już konkurować z przemysłem fonograficznym.

Elektroniczna rozrywka zresztą już od swoich początków miała bardzo duży wpływ na rozwój innych dziedzin przemysłu komputerowego, szczególnie sprzętu. Gry jako aplikacje wykorzystujące pełną moc komputerów osobistych wymuszały ich rozwój, z którego następnie korzyści czerpały inne gałęzie przemysłu. Poza siłą procesorów, miały szczególny wpływ na rozwój kart i akceleratorów graficznych, kart dźwiękowych, czy napędów CD, choć w wypadku tych ostatnich wciąż kluczową rolę odgrywał przemysł muzyczny, a następnie filmowy.

Z doświadczenia wiem, że przynajmniej polski rynek gier komputerowych bardzo potrzebuje dobrych programistów. Średnia płaca jest niższa od tej jaką oferuje sektor bankowy, a do tego niestabilność rynku odbija się negatywnie na stabilności zatrudnienia. Z uwagi na wciąż rosnące koszty produkcji deweloperzy są coraz bardziej uzależnieni od wydawców i często nawet sukcesy przy wydaniu kilku tytułów mogą nie uchronić firmy przed upadkiem w następnych latach. Dlatego często ludzie pracujący nad grami zmieniają firmy. Rynek jest wciąż wyraźnie nienasycony, więc na szczęście nie ma problemów ze znalezieniem ponownego zatrudnienia.

Kolejnym problemem jest niejednolitość zespołów produkcyjnych. Tylko niewielką ich część stanowią programiści. W "Wiedźminie" na ponad siedemdziesięcioosobowy zespół, programistów zatrudnionych do pracy nad grą było około dziesięciu. O zbliżonych proporcjach

słyszałem również w odniesieniu do innych projektów. Dlatego na rynku gier, choć inżynieria programowania nie wymaga od programisty umiejętności dialogu z końcowym klientem, musi on potrafić znaleźć wspólny język we własnym zespole, w którym równorzędnymi partnerami są dla niego ludzie niezwiązani z programowaniem.

Warunki pracy programisty na rynku elektronicznej rozrywki są prawdopodobnie powodem, dla którego wielu ludzi wybiera sektor finansowy albo usługowy. Rynek jednak istnieje i wykazuje rosnące zapotrzebowaniem na nowych ludzi. Oferuje pracę bynajmniej nie lekką, ale dla wielu, w tym dla mnie, ciekawą i wymagającą stawienia czoła szerokiemu wachlarzowi problemów programistycznych.

1.1.2. Tworzenie gier - zespół

Jak już wspominałem skala przedsięwzięcia jakim jest tworzenie gry komputerowej wraz z rozwojem rynku diametralnie się zwiększyła. Hity początku lat dziewięćdziesiątych takie jak nieśmiertelny "Prince of Persia" (1989), czy "Another World" (1991) stworzone były przez samotnych programistów. Istniały już pierwsze zespoły deweloperskie, ale były złożony z kilku osób. Wraz z rozwojem komputerów osobistych rozwinął się rynek, wraz z rynkiem zapotrzebowanie na komputerową rozrywkę. Wraz z zapotrzebowaniem - zespoły powiększyły się i wyspecjalizowały.

W chwili obecnej liczebność zespołu pracującego nad jedną grą jest różna w zależności od klasy powstającej produkcji. Istnieją gry stworzone w ciągu kilku miesięcy, na gotowym silniku. W ich powstanie bezpośrednio zaangażowanych było kilku projektantów poziomów, grafików i pojedynczy programista, właściwie realizujący tylko bieżące zapotrzebowania zespołu. Nad największymi produkcjami zaangażowane są rzeczywiście duże zespoły, na przykład nad grą "GTA 4" w Rockstar Games pracuje około stu pięćdziesięciu osób. Przy "Wiedźminie" prace koncepcyjne rozpoczęły się w kilkuosobowym zespole, aby po latach w decydującej fazie, nad projektem było zatrudnionych ponad sześćdziesiąt osób. W ostatnim okres prac, przed wydaniem gry, zespół jeszcze się powiększył do wspomnianych wyżej siedemdziesięciu osób. Dodatkowo była prowadzona współpraca z zewnętrznymi działami kontroli jakości. Oddzielne firmy też tworzyły część trójwymiarowych modeli postaci oraz obiektów otoczenia, a także animowane sekwencje filmowe rozpoczynające i kończące grę.

Akurat w tym projekcie liczba dziesięciu programistów była lekko niewystarczająca. Stanowiła problem w końcowej fazie, gdy z uwagi na długi czas akomodacji w projekcie, nie można było zaciągnąć nowych pracowników. Poniżej znajduje się skrócony opis kompetencji pozostałych działów, z którymi bezpośrednio współpracują programiści przy produkcji gry.

Projektanci

Dział odpowiedzialny za projekt gry. Zarówno od strony mechaniki jak i fabuły oraz projektu konkretnych lokacji[3] czy poziomów gry. Oczywiście dział ten bardzo różni się wewnętrznie, w zależności od rodzaju tworzonej gry.

Na pewno można w nim wyróżnić dział mechaniki. Są to ludzie, którzy na początku projektują mechanikę gry[4], następnie są odpowiedzialni za implementację i zbalansowanie rozgrywki na postawie dostarczonych przez programistów narzędzi.

[3] Używam terminu "lokacja" w odniesieniu do wirtualnych, spójnych obszarów świata gry, które są tworzone w trakcie prac nad grą jako jedna całość.

[4] Mechanika gry (ang. *gameplay*). Pod tym terminem rozumiem wszystkie aspekty interakcji gracza z systemem gry.

Kolejnym elementem jest dział implementacji, przyjmujący różne formy w zależności od rodzaju tworzonej produkcji. Bardzo popularne wśród polskich deweloperów gry FPP[5] zatrudniają tu zwykle kilku projektantów poziomów. Przy projekcie "Wiedźmina" dział ten był przede wszystkim odpowiedzialny za wymyślenie szczegółowej fabuły gry oraz jej implementację na przygotowanych przez dział techniczny lokacjach gry.

Dział graficzny

Na ten dział składają się artyści, graficy dwuwymiarowej i trójwymiarowej grafiki, animatorzy, dźwiękowcy oraz technicy odpowiedzialni za przygotowanie elementów gry od strony graficznej i technicznej. Ważny jest kontakt programistów z grafikami. Trzeba pamiętać, że nie znają oni dokładnej mechaniki działania gry i że naturalnie oceniają produkt finalny po tym jak wygląda. Ich praca może mieć bardzo duży wpływ na wymagania pamięciowe gry i jej prędkość działania. Ważne jest by im to uświadomić. Dobrą praktyką jest tworzenie dla nich narzędzi pomagających określać koszt wydajnościowy i pamięciowy tworzonych przez nich zasobów.

Dział kontroli jakości

W produkcji gier nie jesteśmy w stanie generować automatycznych testów aplikacji, przynajmniej nie w pełnym zakresie. Dlatego konieczny jest sprawny dział testów. Na początku prac jest on prawie zbędny. W miarę zbliżania się do końca projektu nabiera znaczenia. Półtora roku przed zakończeniem Wiedźmina testerów było dwóch. Przed samym wydaniem gry wewnętrzny dział kontroli jakości Wiedźmina liczył prawie dwadzieścia osób, z czego część została zatrudniona tylko do czasu premiery. Do tego współpracowaliśmy z działami kontroli jakości wydawcy Atari oraz ze współzależną firmą CD Projekt Localization Center, która również wspierała nas w testowaniu gry.

1.1.3. Tworzenie gier - produkcja

Gdy mamy już kompletny zespół, czas zabrać się za produkcję. Tworzenie gier jest procesem iteracyjnym. Ponieważ na produkt wynikowy pracuje jednocześnie kilka zależnych od siebie zespołów, dosyć szybko potrzebna jest pierwsza działająca wersja. Bardzo dobrą praktyką jest zaczynanie od prototypu - prostego szkieletu gry stworzonego możliwe małym nakładem sił, a sprawdzającego potencjał projektu. Na podstawie prototypu można przed rozpoczęciem rzeczywistej implementacji ocenić i zmodyfikować założoną mechanikę gry.

Prace koncepcyjne potrzebują zaangażowania dużo mniejszego zespołu niż końcowa implementacja projektu. Powstający przez trzy lata hit tego sezonu - gra *Assassins Creed* wyprodukowana przez Ubisoft Montreal w początkowej fazie produkcji była tworzona przez 8 osób. Ten zespół tworzył koncepcję gry oraz rozpoczął pierwsze prace nad silnikiem. W dalszych etapach prac zespół bardzo szybko się rozszerzał, by pod koniec produkcji urosnąć do 158 osób pracujących nad projektem[6].

Dokumentacja projektowa w branży rozrywki elektronicznej różni się i jest słabiej rozwinięta niż w oprogramowaniu biznesowym. Rolę klienta może stanowić dystrybutor. W rozmowach z nim nie są szerzej stosowane diagramy UML, przynajmniej według mojej wiedzy.

[5]Gra FPP (First Person Perspective). Rodzina trójwymiarowych gier akcji, które łączy prezentacja świata z perspektywy pierwszej osoby. Gracz widzi grę oczami sterowanego bohatera. Prekursorami gatunku były Wolfenstein 3-D (1992) i Doom (1993).

[6]Dane zaczerpnąłem z wywiadu z producentką gry Jade Raymond.

Oczywiście w projekcie znalazłyby zastosowania dokumenty wizji. Ważne jest przygotowanie diagramów klas. Od strony projektu gry głównym dokumentem jest Game Doc, bardzo szczegółowo opisujący całą mechanikę gry. Często jego rolę spełnia zbiór pomniejszych dokumentów opisujących poszczególne elementy projektu. Relacje pomiędzy producentem gry a wydawcą różnią się od tych jakie panują między producentem a klientem biznesowym. Dokumenacja pełni w grach dużo mniejszą rolę i jest tworzona raczej na wewnętrzne potrzeby zespołu produkcyjnego.

Kolejną istotną rzeczą są narzędzia. Będą potrzebne innym zespołom, dlatego muszą powstawać szybko, w początkowych fazach implementacji aby wraz z tworzeniem mechanizmów gry, powstały już narzędzia umożliwiające wykorzystanie ich wykorzystanie i ich rozwój przez grafików czy projektantów.

Jak już wspomniałem, pisanie gry to proces iteracyjny. Po wstępnych projektach, możliwie szybko powinna powstać działająca wersja gry. Wraz z nią podstawowe mechanizmy, na których oparta będzie rozgrywka, by umożliwić prace projektantom. Początkowe etapy produkcji mogą być pominięte, jeżeli producent zdecyduje się na wykorzystaniu gotowego silnika gry. Przy ustalaniu harmonogramu prac programistycznych należy brać pod uwagę pozostałe zespoły i zależność ich pracy od stopnia implementacji. Choć to programiści jako pierwsi rozpoczynają prace implementacyjne nad projektem, ich prace mogą trwać do samego jego końca.

1.2. Moja gra

Jak już napisałem powyżej gry komputerowe są często bardzo dużymi przedsięwzięciami, angażującymi grupę której tylko pomniejszą część stanowią programiści. Przy pisaniu pracy magisterskiej moją intencją było więc stworzenie pewnego szkieletu gry. Próba zmierzenia się ze standardowymi problemami programistycznymi pisania gier oraz stworzenie oryginalnej architektury, odpowiadającej specyfice mojego projektu. Nie chciałem skupiać się na tworzeniu zasobów gry i ustawianiu rozgrywki. Chciałem stworzyć silnik gry sieciowej, zobaczyć jak sprawdzą się w praktyce moje własne, oryginalne rozwiązania w kwestii mechanizmów komunikacyjnych i podstawowej symulacji świata gry po stronie serwera.

1.3. Gry sieciowe

Skoro moją pracę stanowi gra sieciowa, pora powiedzieć troszkę o tej gałęzi komputerowej rozrywki. Pod powyższym terminem rozumie wszystkie gry działające pomiędzy kilkoma komputerami, połączonymi za pośrednictwem zarówno sieci lokalnej (LAN) jak i globalnej (WAN). Stworzenie wydajnej architektury gry sieciowej było głównym celem mojej pracy. Możliwości połączenia komputerów i wymiany informacji pomiędzy nimi, w czasie rzeczywistym, dosyć szybko zostały dostrzeżone przez producentów gier. Zanim obsługa sieci LAN stała się standardem komputerów osobistych, gry często umożliwiały połączenie z innym komputerem za pośrednictwem portów szeregowych. Gry sieciowe rozwijały się wraz z rozwojem popularności sieci lokalnych, internetu oraz wzrostem szybkości transmisji. Dzięki coraz większej dostępności łączy szerokopasmowych w rodzaju DSL/ADSL gry sieciowe pozwalają na zabawę coraz większej liczbie graczy równolegle. Największe produkcje, najbardziej renomowanych wydawców, jeżeli nawet nie są grami stricte dla wielu graczy, to przynajmniej pozwalają na wspólną grę za pośrednictwem internetu.

Technologia pisania aplikacji sieciowych wciąż się rozwija. Na rynek wchodzą produkcje, których powstanie jeszcze niedawno było niemożliwe, nie tylko ze względu na szybkość

transmisji, ale również ze względu na techniki tworzenia samych gier.

1.3.1. Kierunki rozwoju rozrywki sieciowej

Najczęściej gra sieciowa stworzona jest w architekturze klient - serwer (niewielka część gier dla kilku graczy stosowała model równorzędny, w którym wszyscy uczestnicy rozgrywki byli wspólnie odpowiedzialni za synchronizację). Serwer może założyć jeden z uczestników rozgrywki lub mogą to być dedykowane maszyny. Większość gier umożliwia także rejestrowanie się serwerów gry w oficjalnych serwisach wyszukujących. Wówczas gracze mają w grze możliwość wyszukania dostępnych w danej chwili serwerów gry. Nawet na dedykowanych maszynach tylko nieliczne gry są w stanie obsłużyć liczby rzędu, na przykład, stu połączeń. Większość gier nie zezwala na połączenie większej liczby niż kilku lub kilkunastu graczy, nawet posiadających szerokopasmowy dostęp do internetu.

Od wielu lat następuje gwałtowny rozwój nowej gałęzi gier dla wielu graczy. Massive Multiplayer Online (MMO) to gry pozwalające na jednoczesną zabawę nawet tysięcy użytkowników. Ich początków możemy dopatrywać się w tak zwanych *Modach*, czyli tekstowych grach RPG[7] , pozbawionych grafiki i obsługiwanych najczęściej za pośrednictwem protokołu telnet. Pierwsze[8] gry MMO umożliwiały zabawę tysięcy graczy we wspólnym wirtualnym świecie. Był to bardzo duży krok dla branży elektronicznej rozrywki. Nowe gry tworzyły całą społeczność graczy. Dystrybutor, jako że wymaga utrzymywania oficjalnych serwerów, pobiera najczęściej miesięczne opłaty od użytkowników. Jednocześnie aspekt istnienia społeczności gry i możliwość rozwoju statusu gracza w świecie gry - tradycyjna dla gatunku gier RPG (najpopularniej używanego w produkcjach massive) zaowocowały bardzo dużym i długotrwałym przywiązaniem graczy do poszczególnych tytułów z gatunku MMO. Sukces komercyjny pierwszych projektów spowodował rozwój gatunku i powstanie całej lawiny kolejnych tytułów.

Zwykle dystrybutor gry MMO udostępnia zestaw oficjalnych serwerów do gry. Często mamy do czynienia z rozproszonym zbiorem serwerów obsługujących poszczególne części świata gry. Są też produkcje takie jak Guild Wars[9], w której pojedynczy serwer jest w stanie obsłużyć wiele egzemplarzy świata w których odbywa się symulacja gry. Cechą wspólną nowych produkcji MMO jest uproszczona mechanika gry, pozwalająca na asynchroniczną obsługę klientów, jednocześnie w bardzo atrakcyjny i spektakularny sposób wizualizowana po stronie klienta.

Wiele gier MMO miało bardzo duże problemy z bezpieczeństwem rozgrywki. Oszczędny protokół komunikacji zbyt dużą część mechaniki gry realizował po stronie klienta (na przykład poruszanie się i walkę), co było wykorzystywane przez graczy poprzez stosowanie własnych lub przerobionych aplikacji klienta. Wielu producentów wykorzystywało w walce z oszustwami w grze rozwiązania w rodzaju GameGuard. Była to aplikacja podobna w działaniu do programów antywirusowych, dołączana do gry. Z założenia miała uniemożliwiać klientowi

[7]RPG - Roleplaying game. Rodzina gier powstała na podstawie "papierowych" gier fabularnych.

[8]Pierwszą wysoko budżetową, komercyjną grą MMORPG (Multiplayer Massive Online Roleplaying Game) była wydana w 1997 roku Ultima Online.

[9]Wydana w 2005 roku gra ArenaNet, na początku opisywana jako świeże podejście do gatunku MMORPG. W praktyce grę ciężko sklasyfikować, rozgrywka jest bardzo szybka i w dużym stopniu zręcznościowa. Toczy się w oddzielnych egzemplarzach świata gry, w których występować może od jednego do kilkunastu zdalnych graczy. Gracze mogą się jednak spotykać licznie w obszarach "miejskich", w których nie występują elementy zręcznościowe. Tam łączą się w grupy i rozpoczynają właściwą rozgrywkę. Producent chwali się, że technologia sieciowa użyta w silniku jest wyjątkowo oryginalna i oszczędna w transmisji, co umożliwiło znaczne obniżenie kosztów utrzymania serwerów i zrezygnowanie z opłat abonamentowych. Jest ona owiana mgłą tajemnicy. Pewne rozwiązania zastosowane w mojej pracy czerpałem z rozważań nad implementacją Guild Wars.

wykorzystanie znanych sposobów nadużycia systemu. By to osiągnąć podejmowała działania w rodzaju skanowania plików, kontrolę działających procesów czy blokowanie wywołań określonych funkcji DirectX czy Windows API. Osobiście uważam możliwość dokonania takich nadużyć za słabość architektury gry i projektując swoją grę, miałem na celu zapewnienie bezpieczeństwa rozgrywki już na poziomie głównych założeń projektowych.

Gry sieciowe uważam za przyszłość branży. Wzrost mocy komputerów niedawno jeszcze wydawał się bardzo spowolniony, dopiero popularyzacja maszyn wielordzeniowych spowodowała pewien powiew świeżości na rynku komputerów osobistych. Natomiast rozwój szybkości transmisji i infrastruktury internetu wciąż postępuje i jest daleki od osiągnięcia granic możliwości rozwoju technologicznego.

Swoją grę chciałem umiejscowić gdzieś pomiędzy grami MMO, a standardowymi grami sieciowymi. Chciałem dokonać pewnego eksperymentu i na własnej skórze sprawdzić jak wielu graczy może obsłużyć gra akcji. Starałem się stworzyć bardzo wydajną aplikację serwera umożliwiającą przy szybkim łączu i dobrej, wielordzeniowej maszynie, by jednocześnie w świecie gry mogła się spotkać rzeczywiście duża liczba graczy.

Rozdział 2

Projekt

2.1. Założenia projektu

- Planowałem stworzyć prostą i przyjemną internetową grę akcji.
- Model sterowania wzorowałem na klasykach takich jak „Asteroids”[1] czy kilku innych grach zaliczanych do „kosmicznych” gier zręcznościowych. Miał być prosty, intuicyjny i przyjemny.
- Gra miała umożliwiać zabawę możliwie dużej liczby graczy.
- Gracz miał wcielać się w pilota kosmicznego myśliwca, realizującego różnorodne misje w pasie asteroidów.
- Realizując powierzone mu zadania gracz miał zarabiać wirtualne pieniądze.
- Uczestnicy gry mieli mieć możliwość podejmowania się różnych misji. Ich zadania miały się od siebie różnić. Czasami ich realizacja miała wymagać od nich współpracy ramię w ramię, a czasami prowadzić do nieuchronnej konfrontacji.
- Pomiędzy realizacją misji gracze mieli możliwość rozmowy między sobą, rozbudowy statków za zarobione wirtualne pieniądze oraz poszukiwania zleceń na nowe misje w pasie asteroidów.
- Rozgrywka miała być wizualizowana z wykorzystanie grafiki trójwymiarowej, choć prosty model lotu zakładał dwuwymiarową mechanikę gry.

2.2. Maelstorm Online

Moją grę nazwałem Maelstorm Online. Stworzyłem zgodny z zaplanowanymi wytycznymi, silnik sieciowej gry akcji. Sama koncepcja gry w trakcie pisania pracy uległa pewnym zmianom. Zadanie tworzenia silnika gry pochłonęło więcej wysiłku i pracy niż na początku zakładałem. Uważam go w bieżącej chwili za kompletny, choć z powodu ograniczeń czasowych pewnemu uproszczeniu uległy właściwości samej rozgrywki.

Tak w tej chwili, w punktach, mogę przedstawić cechy mojej gry.

[1] Asteroids – gra wydana w 1979 roku przez Atari Inc. Polegała na lataniu statkiem w dwuwymiarowej przestrzeni kosmicznej ograniczonej rozmiarem do ekranu i niszczeniu kolejnych fal asteroid.

Rysunek 2.1: Disasteroids3D. Wersja klasycznej gry akcji, będąca jedną z moich inspiracji.

- Podstawowa forma, koncepcja i sposób kontroli oraz prezentacji gry zostały zrealizowane zgodnie z założeniami.
- Model sterowania jest zgodny z założeniami. Obiekty poruszają się z zachowaniem kierunku ruchu i pędu. Mogą obracać się dowolnie i zmieniać trajektorię ruchu, przyśpieszając w kierunku w którym są zwrócone. Na ruch wpływa też siła tarcia[2].
- Rozgrywka toczy się w pasie asteroidów. W świecie cały czas występuje zbliżona liczba asteroidów. Na miejsce zniszczonych z czasem pojawiają się nowe.
- Po zalogowaniu się do gry, gracz wybiera jeden z trzech dostępnych statków.
- Do dyspozycji gracza są trzy rodzaje broni.
- Zarówno statki gracza jak i asteroidy mogą zostać uszkodzone w wyniku kolizji bądź trafienia pociskiem. Statki przed bezpośrednimi uszkodzeniami broni pole energetyczne. Słabnie ono gdy blokuje trafienia, ale odnawia się powoli wraz z upływem czasu.
- Gracze współzawodniczą na serwerze dążąc do zdobycia największej liczby punktów.
- Punkty przyznawane są w małej liczbie za niszczenie asteroidów. Większą liczbę punktów można zdobyć niszcząc statki innych graczy.
- Gdy statek gracza zostanie zniszczony, może on ponownie wejść do gry w nowo wybranym statku, ale kosztuje go to określoną liczbę punktów.
- Gracze wiedzą który z nich dysponuje największą liczbą punktów, dlatego utrzymanie przez niego prowadzenia w pasie asteroidów jest dodatkowo utrudnione.

[2]Mimo że w próżni tarcie nie występuje, uwzględniłem je w moim modelu poruszania, gdyż zwiększa ono atrakcyjność rozgrywki. Powoduje że gracze nie muszą obawiać się zbytniego nabierania prędkości a sama rozgrywka nie jest tak chaotyczna i przypadkowa.

2.3. Wpływ wymagań projektu na architekturę gry

Wymagania projektu były decydującym czynnikiem na etapie projektowania aplikacji i silnika gry.

- **W grze bierze udział możliwie wielu graczy** — Komunikacja musi być oszczędna, a aplikacja serwera bardzo wydajna.
- **Gra toczy się w przestrzeni kosmicznej "rzadko" wypełnionej obiektami** — To wymaganie wpływa na architekturę w następujących kwestiach:
  - Możemy ograniczyć wiedzę graczy o świecie do ich najbliższego otoczenia, a to wpływa na wydajność i bezpieczeństwo aplikacji.
  - Nie trzeba implementować mechanizmu "wyszukiwania ścieżek".
  - Zdarzenia świata gry mają wpływ zwykle na niewielkie grupy obiektów. Jeżeli przyjdą z opóźnieniem, do ich poprawnego obsłużenia powinna wystarczyć wsteczna symulacja dotycząca wyłącznie obiektów bezpośrednio związanych ze zdarzeniem.
  - System AI w Maelstorm można oprzeć o algorytm stada, przedstawione w rodziale 3.7
- **Interfejs gry jest prosty, a sama rozgrywka „łatwa i przyjemna"** — To wymagało umożliwienia wyświetlania ładnej grafiki trójwymiarowej, modeli i efektów świetlnych. Świat gry jest dwuwymiarowy, ale grafika musi się przystępnie prezentować. Z drugiej strony aplikacja klienta nie jest specjalnie rozbudowana a dzięki swojej prostocie nie musiałem się skupiać nad jej optymalizacją.
- **W modelu gry intuicja i taktyka mają być istotniejsze od czystej zręczności i refleksu** — To w sumie jest cecha gry wynikła z nałożonych na nią wymagań. Jej implementacja wymaga:
  - Czysty refleks nie może dominować rozgrywki, ponieważ dyskwalifikuje graczy o większych opóźnieniach.
  - Akcje gracza wykonują się krótki czas po rzeczywistym wydaniu przez gracza polecenia. Ten czas jest potrzebny by informacja o akcji doszła na serwer i by serwer rozesłał ją do zainteresowanych graczy.
- **Obiekty są w ciągłym ruchu, tylko ich niewielka część jest w spoczynku** — To wymusza bardzo wydajną architekturę symulacji poruszania się obiektów. Dla oszczędności komunikacji symulacja poruszania musi być niezależna po stronie serwera i klientów.
- **Gracze muszą mieć przyjemność z zabawy w miarę niezależnie od opóźnienia w komunikacji z serwerem.**
  - Odporność na opóźnienia.
  - W momencie nadejścia opóźnionej wiadomości z serwera musimy móc "cofnąć" w czasie związane z nim obiekty i ponownie je zasymulować po reakcji na zdarzenie.
- **Gra ma być w pełni „uczciwa" i należy uniemożliwić oszukiwanie serwera.**

– "Serwer ma zawsze racje" — symulacja świata po stronie klienta jest odbiciem symulacji świata na serwerze.
– Gracz wysyła do serwera wyłącznie swoje sterowanie oraz dodatkowe zapytania odnośnie widzianych przez niego elementów symulacji.

By nie tracić skalowalności i uniwersalności projektu, istotne było zachowanie właściwej modularyzacji kodu, umożliwiającej łatwą modyfikację i zamianę poszczególnych części silnika. Dlatego starałem się w miarę możliwości nie doprowadzać do głębokich zależności pomiędzy architekturą aplikacji a następującymi właściwościami gry:

- **Dwuwymiarowy świat gry.** Powinna być możliwość stworzenia proporcjonalnie małym kosztem zupełnie nowego świata gry.
- **Użyte protokoły komunikacyjne.**
- **Sposób graficznej prezentacji świata.**

2.4. Szersza koncepcja gry

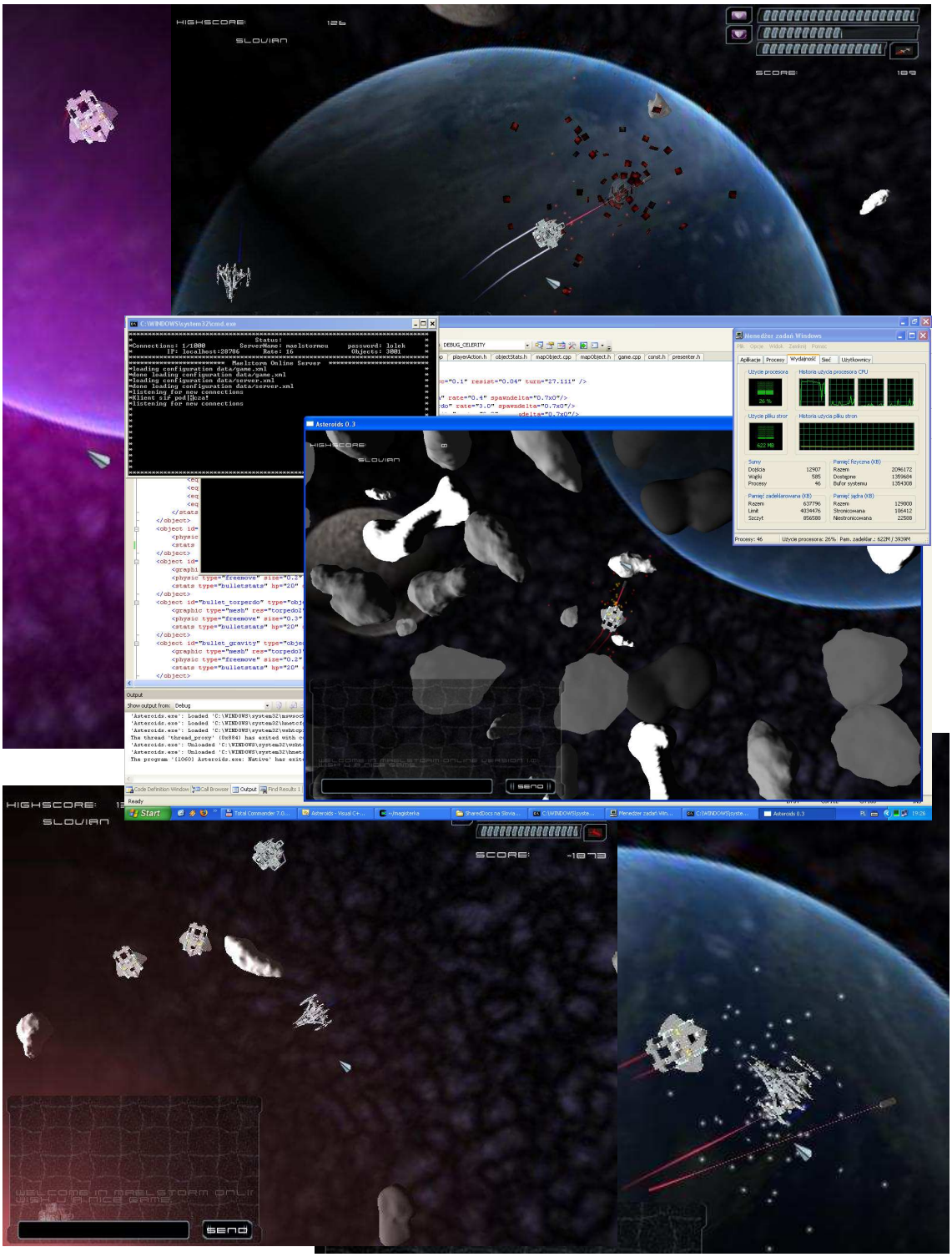
Tworząc Maelstorm chciałem spróbować swoich sił w implementacji gry dla wielu graczy dostępnej za pośrednictwem internetu. Tak naprawdę miałem w zamyśle dużo szerszą koncepcję rozgrywki. Chciałem by Maelstrom, w postaci zbliżonej do obecnej, był elementem większego, rozproszonego systemu. Zakładałem, że gra albo będzie mogła obsłużyć idącą w setki liczbę graczy, albo obsługa mniejszej ich liczby będzie na tyle mało kosztowała aplikację serwera, że będzie można na pojedynczym komputerze uruchomić kilka jej instancji.

Wówczas można by Maelstorm rozbudować o dodatkową aplikację, tworzącą spójny świat gry w którym kluczowym elementem byłaby społeczność graczy. Ta "gra społeczna" symulowałaby rozległy świat stacji kosmicznych, w których gracze przebywaliby pomiędzy misjami wypełnionymi akcją. Mogłaby być bardzo uproszczona i pozwalać graczom na wydawanie zarobionych pieniędzy, interakcję z innymi graczami oraz podejmowanie nowych wyzwań realizowanych na serwerze gry akcji (czyli Maelstrom w jej aktualnej postaci). Główna aplikacja nie potrzebowałaby trójwymiarowej grafiki czy efektownego modelu akcji. Nie miałaby elementów zręcznościowych i po stronie serwera byłaby zrealizowana jako aplikacja bazodanowa. Serwer mógłby więc obsługiwać całą wirtualną społeczność graczy.

Byłaby to właściwie zupełnie nowa gra, o bardzo dużej możliwości rozbudowy, będąca spoiwem zapewniającym całej rozgrywce atrakcyjność. Wraz z grą akcji, której silnik zaimplementowałem, umożliwiłaby stworzenie żywej społeczności graczy. Jej możliwość rozbudowy i własności zależałyby wyłącznie od możliwości implementacyjnych. Mogłaby umożliwiać stworzenie ekonomii świata na który wpływ miałyby akcje realizowane przez graczy i który pozwalałyby graczom na zabawę w handel wirtualnymi dobrami. Można by też zaimplementować system organizacji w których skład mogliby wchodzić gracze. Organizacje oczywiście prowadziłyby własną politykę finansów, sojuszy i wojen. To byłby bardzo mocny mechanizm społeczny, który pozwalałby na nadanie dodatkowego wymiaru zmaganiom graczy w przestrzeni kosmicznej. Całość takiej aplikacji, jak już mówiłem, ograniczają tylko możliwości implementacyjne ewentualnego autora i wyobraźnia. W wyniku jej powstania cała gra stałaby się rzeczywiście produkcją *Massive Multiplayer Online*. Wątpię by istniała realna szansa na taką rozbudowę projektu, ale chciałem wyrazić swoją wizję i marzenia stanowiące o jego koncepcji.

Pisząc Maelstorm zakładałem tak naprawdę, że tworzę element takiej właśnie rozproszonej aplikacji. Stąd na przykład pojawiły się pomysły wczytywania całej konfiguracji rozgrywki,

łącznie z identyfikatorami graczy, z plików xml, które mogłyby być wysyłane na serwer gry akcji z nadrzędnego serwera gry społecznej. Możliwość wyboru serwera, swobodnego logowania się do gry oraz zasady rozgrywki (niszczenie asteroidów i innych graczy) zostały dodane w końcowym etapie implementacji, by praca nabrała pełnego kształtu. Były to jednak elementy do których nie podchodziłem z taką pieczołowitością i sercem jak do podstawowego silnika gry.



Rysunek 2.2: Maelstorm Online.

Rozdział 3

Implementacja

W toku prac nad Maelstorm Online miałem możliwość zmierzenia się z dużą liczbą zagadnień związanych z tworzeniem gier. Poniżej opisałem rozwiązania użyte przeze mnie w toku prac nad projektem. Mocny wpływ miała na nie specyfika mechaniki mojej gry. Poza czystym opisem prac implementacyjnych, moją intencją była dyskusja możliwych sposobów podejścia do omawianych problemów. Swoje inspiracje czerpałem z toku prac nad projektem, literatury, pracy zawodowej i zaobserwowanych w praktyce rozwiązań.

3.1. Projekt i środowisko

3.1.1. Historia prac

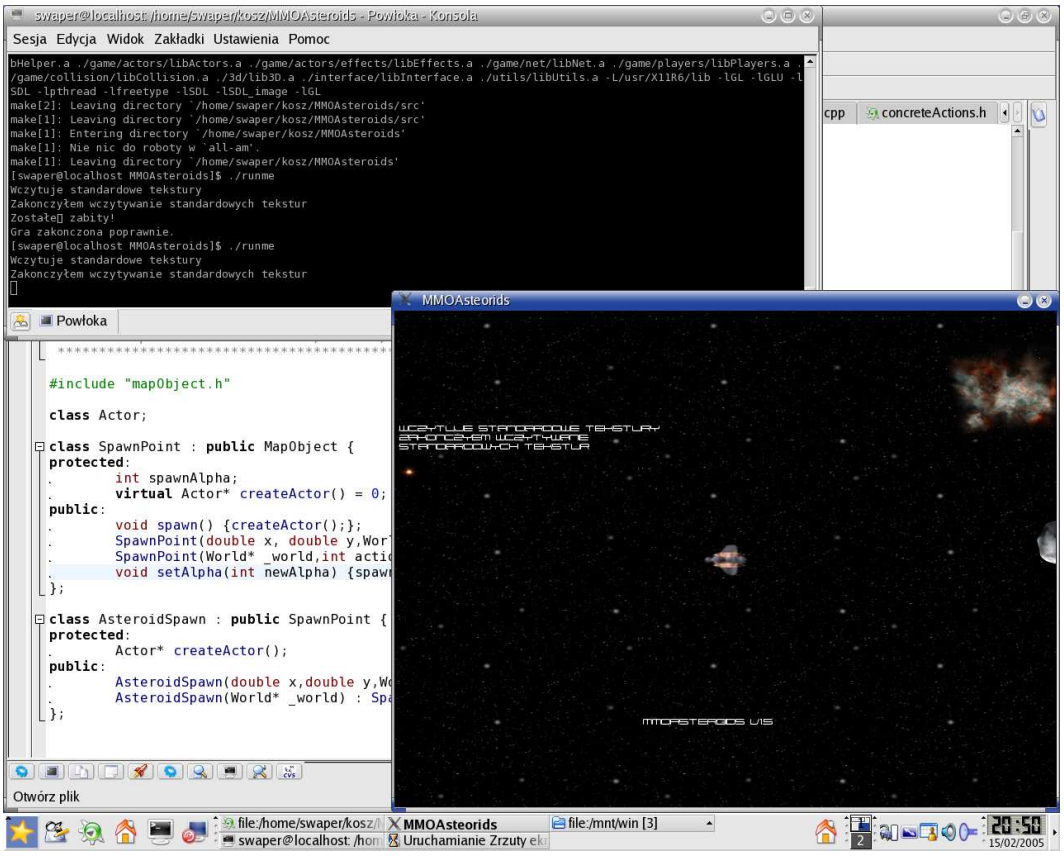
Prace nad projektem zacząłem w roku 2004, będąc na czwartym roku studiów w ramach przedmiotu "Programowanie Praktyczne" prowadzonego przez dr. Roberta Marona. Miałem w ciągu roku zaprojektować i zaimplementować kompletną aplikację - i było to moje pierwsze podejście do tej konkretnej gry sieciowej.

Do projektu zabrałem się w dosyć nietypowy sposób - zaczynając od stworzenia gry na jednego gracza, zupełnie nie dzieląc kodu na aplikację serwera i klienta. Mogłem sobie na to pozwolić dzięki zastosowaniu specyficznego protokołu komunikacji i architektury, ale o tym napiszę w dalszej części pracy. W ciągu połowy roku otrzymałem prosty silnik gry dla jednego gracza, by w ciągu następnego semestru wykonać prosty protokół komunikacji i uruchomić grę na dwóch komputerach. Gra jednak była wyraźnie niekompletna - poza samą konfiguracją rozgrywki, zabrakło bardziej zaawansowanej możliwości tworzenia gry sieciowej. Istniał właściwie szkielet gry, który czekał na wypełnienie danymi oraz zamianę części modułów na końcową implementację.

Postanowiłem uczynić Maelstorm Online moją pracą magisterską. Bogatszy o doświadczenia roku prac nad projektem, zdecydowałem się na bardzo duże zmiany w istniejących już mechanizmach. Dzięki modularnej konstrukcji mogłem zupełnie podmienić dużą część modułów gry, w tym poruszanie się, wykrywanie kolizji i wyświetlanie grafiki. W pierwszej wersji Maelstrom wyświetlałem grafikę wykorzystując bibliotekę OpenGL, jednak zdecydowałem się na wykorzystanie w tym celu gotowej biblioteki Ogre3d i całkowicie podmieniłem moduł graficzny. To samo spotkało napisany zupełnie na nowo moduł poruszania się. Zmiany były związane z nowym mechanizmem cofania ruchów, koniecznym do implementacji mojego protokołu komunikacyjnego. Zmianie uległo też samo serce symulacji oraz większość mechanizmów. Znowu jednak, by umożliwić szybsze stworzenie prototypu gry, zrezygnowałem ze wspierania architektury sieciowej.

W trakcie prac nad projektem na początku piątego roku studiów rozpocząłem pracę zawodową, co było przyczyną poważnego opóźnienia, ale jednocześnie pozwoliło mi na konfrontację własnych rozwiązań i pomysłów z tymi, stosowanymi w komercyjnych projektach.

W sierpniu 2007 roku, ponownie utworzyłem silnik gry dla jednego gracza, bez podziału na aplikację klienta i serwer. Polegałem jednak na uprzednio dobrze sprawdzonym sposobie podziału aplikacji. Rozwój praktycznej części pracy magisterskiej trwał do listopada.



Rysunek 3.1: Wczesna wersja projektu. Działająca w środowisku Linux i oparta o interfejs OpenGL.

3.1.2. Środowisko i platforma

Maelstorm został napisany w całości w języku C++. Choć nie uważałem tego za priorytet, grę pisałem od początku z założeniem przenośności kodu pomiędzy platformami Linux i Windows. Nie wymagało to bynajmniej specjalnych nakładów pracy. Jedynymi ograniczeniami dla mnie, była konieczność używania przenośnych bibliotek oraz nie używanie funkcji systemowych bezpośrednio w kodzie głównej aplikacji. Grę przez dwa lata tworzyłem pod Linuxem, korzystając z darmowej wtyczki C++ do edytora Eclipse oraz zestawu skryptów i programów automake/autoconf do automatycznej konfiguracji kompilacji. Jednocześnie niezależnie poznałem środowisko .Net i skuszony jego prostotą, funkcjonalnością i wygodą użytkowania postanowiłem przenieść projekt na platformę Windows. Zajęło to jeden dzień i jedynym problemem dla mnie była kompilacja użytej w grze biblioteki XercesC w środowisku .Net. To dobra lekcja, bo pokazuje jak łatwo pisać przenośny kod. Większość gier wideo na kompu-

tery osobiste powstaje wyłącznie na platformę Windows, ewentualnie również na konsole. Z punktu marketingowego tylko na tych platformach opłaca się sprzedaż gier, jednak jak się okazuje nakłady związane z wydaniem gry na inną platformę mogą być praktycznie pomijalne - przy założeniu, że korzystamy z przenośnych bibliotek. I tu wychodzi podstawowy problem gier dla systemów Unix, którym jest brak dostępnego tylko na platformie Windows pakietu DirectX. Oferuje on bardzo duże możliwości w wyświetlaniu grafiki 3D. Producenci przedkładają komfort pracy związany z używaniem DirectX nad utrzymywaniem wieloplatformowości kodu. Za mnie na szczęście ten problem załatwiała biblioteka Ogre3d odpowiedzialna za grafikę, a potrafiąca obsługiwać zarówno interfejs DirectX jak i OpenGL.

Zastosowałem nietypowe podejście do rozłącznych aplikacji klienta i serwera. Nie tworzyłem osobnych klas dla obu aplikacji, a jedynie zaznaczałem w kodzie różnice za pomocą dyrektyw kompilatora #if #ifdef. Obie aplikacje chodziły więc w oparciu o ten sam kod, wymagana więc była dwukrotna kompilacja tego samego projektu w różnych konfiguracjach. Moje rozwiązanie tego problemu pod Linuxem było bardzo niefunkcjonalne. W środowisku .Net po prostu stworzyłem nowe, kompilowane do osobnych katalogów, konfiguracje projektu.

3.1.3. Użyte biblioteki

Moją intencją było skupienie się na implementacji silnika mechaniki gry sieciowej. W miarę możliwości implementację pobocznych modułów pozostawiłem w gestii zewnętrznych bibliotek.

- Biblioteki narzędziowe — W pracach nad projektem wykorzystywałem biblioteki **STL** i **BOOST** (*http://www.boost.org*). Są one mocno zintegrowane z kodem aplikacji. STL wykorzystywałem głównie do reprezentacji struktur danych - rozszerzalnych tablic, zbiorów, słowników i kolejek priorytetowych. Dodatkowo wykorzystywałem zaimplementowane w STL sortowanie i obsługę strumieni. Biblioteka BOOST składa się z wielu niezależnych modułów. Ja wykorzystywałem głównie moduł narzędziowy *BIND*[1] i moduł do obsługi wielowątkowości thread. Szczególnie przypadł mi do gustu interfejs tego modułu. Jest on mocno oparty o szablony i w większości wypadków opiera się na inicjalizacji obiektów w konstruktorze. Na przykład mutex jest podnoszony i zwalniany w konstruktorze i destruktorze co znacznie upraszcza kod aplikacji i zapobiega standardowym błędom.

- Biblioteka graficzna — Z początku planowałem samemu stworzyć moduł wyświetlania aplikacji klienta, oparty o **OpenGL** i bibliotekę **SDL** [bSDL] (Simple Direct Media Layer *http://www.libsdl.org*). Pierwsza wersja modułu wyświetlania choć trójwymiarowa, reprezentowała obiekty za pomocą dwuwymiarowych, obracanych obrazów. Napisałem prosty moduł wczytywania modeli z formatu MD2[2]. Gdy uczyniłem projekt moją pracą magisterską uznałem, że ilość pracy, którą musiałbym włożyć w moduł graficzny, jest zbyt duża i całe wyświetlanie oparłem na silniku graficznym **Ogre 3D** [bOGR](*http://www.ogre3d.org*). Ogre okazał się być potężną biblioteką graficzną, która w bardzo prosty sposób umożliwiła mi reprezentację obiektów gry w postaci trójwymiarowych modeli, dodawanie efektów specjalnych, zarządzanie zasobami graficznymi i wyświetlanie. Świat gry w Ogre jest reprezentowany w postaci drzewa. Wszyst-

[1]Moduł *BIND* biblioteki *BOOST* wspiera programowanie funkcyjne w C++. Udostępnia szablony, które umożliwiają tworzenie wskaźników do metod, które mogą być traktowane jak wskaźniki do zwykłych funkcji. Dzięki temu można ich używać w argumentach wywołania funkcji przyjmujących wskaźniki do funkcji.

[2]Format modelu 3D użyty w grze Quake 2. Jest to jedna z kultowych gier FPP stworzona prze id Software. Wydana w grudniu 1997 roku przez Activision.

kie obiekty graficzne są więc przypisane do jego wierzchołków. Zmieniając parametry wierzchołków, przesuwając, obracając albo skalując je, możemy wpływać na zachowanie całej hierarchii obiektów. Do wyświetlania i obsługi GUI (Graphical User Interface) użyłem biblioteki **CEGUI** [bGUI] (Crazy Eddie's Gui System *http://www.cegui.org.uk*). Jest ona polecana do użytku w projektach opartych o bibliotekę Ogre i bardzo dobrze się z nią integruje. Jednocześnie pozostaje zupełnie niezależna i można jej użyć w połączeniu, z praktycznie dowolnym innym, systemem wyświetlania. Użyte w niej podejście i rozwiązania dotyczące tworzenia i konfiguracji interfejsu użytkownika bardzo dobrze pasują do projektu gry. Niestety biblioteka CEGUI jest w trakcie rozwoju. Standardowe schematy graficzne i narzędzia dołączone do bibliotki wymagają jeszcze rozwinięcia. Aplikacja serwera ma wyłączone wyświetlanie grafiki 3D. By zrobić dla niej prosty interfejs użytkownika użyłem biblioteki **pdCurses** (Public Domain Curses *http://pdcurses.sourceforge.net*), czyli wieloplatformowej wersji, znanej spod Linuxa biblioteki *ncurses* do niskopoziomowej obsługi wyświetlania tekstowego w konsoli.

- Biblioteka sieciowa — Poszukiwałem w miarę niskopoziomowej, wieloplatformowej biblioteki sieciowej, obsługującej protokół TCP. Najpierw wybrałem bibliotekę **SDLnet**, jako że cały projekt opierałem na bibliotece SDL. Biblioteka miała pewne braki, ale wydawało się, że jest w trakcie rozwoju. Projekt jednak był już martwy i gdy ponownie zabrałem się za tworzenie warstwy sieciowej użyłem tym razem biblioteki **ASIO** [bAIO] (*http://asio.sourceforge.net*). Nazwę biblioteki można tłumaczyć jako *"asynchronous input output"*. Sama biblioteka jest bardzo mocno zintegrowana z biblioteką BOOST. Jest bardzo wygodna w zastosowaniu, cały jej kod znajduje się w plikach nagłówkowych, dlatego ich dodanie jest jedyną czynnością wymaganą do dołączenia jej do projektu. Oparta jest o szablony i jej skalowalna i uniwersalna obsługa komunikacji sieciowej w bardzo ciekawy sposób przesłania niskopoziomową implementację obsługi protokołów TCP i UDP. Dodatkowo biblioteka w bardzo wygodny sposób obsługuje asynchroniczny odbiór danych wspierając się metodami programowania funkcyjnego. Asynchroniczny odczyt albo zapis pakietu związany jest z wywołaniem powiązanej z operacją funkcji. Bardzo eleganckim rozwiązaniem tej biblioteki jest klasa *ioservice* przesłaniająca obsługę zbioru gniazd sieciowych i ułatwiająca wielowątkowe przetwarzanie ich żądań.

- Biblioteka XML — Postanowiłem wczytywać konfigurację gry za pośrednictwem plików XML. Do obsługi tego formatu wykorzystałem bibliotekę **XercesC** (http://xerces.apache.org). To rozległa i bardzo funkcjonalna biblioteka obsługująca zarówno modele DOM[3] i SAX[4], pozwalająca także na walidację dokumentów XML. Ja wykorzystałem wyłącznie funkcjonalność do obsługi modelu SAX, chociaż w praktyce, odczytując dokument za pomocą interfejs SAX, tworzyłem własne drzewo reprezentacji.

[3]DOM – Document Object Model. Obiektowy, uniwersalny interfejs programistyczny do przeglądania dokumentów XML w całości wczytywanych do pamięci.

[4]SAX – Simple API for XML. Model zdarzeniowy odczytu dokumentów XML.

3.2. Architektura aplikacji

W całym projekcie najważniejszym elementem było stworzenie bardzo wydajnej architektury gry sieciowej. Był to dla mnie rzeczywiście priorytet i w te elementy aplikacji włożyłem chyba najwięcej serca. Silnik gry jest możliwie uniwersalny, mimo mocnego oparcia o założenia projektu.

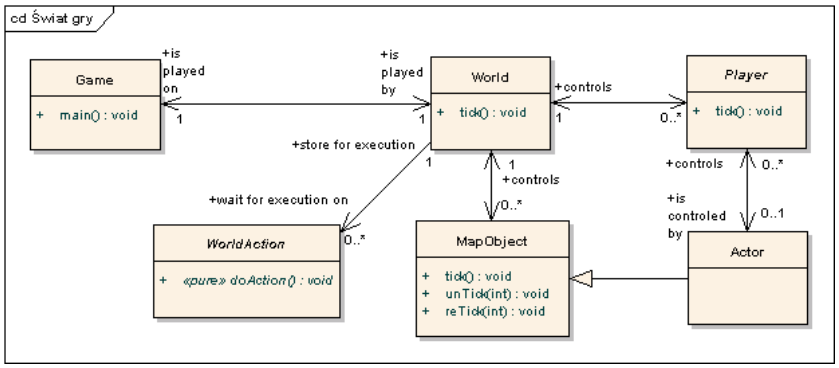
3.2.1. Definicja problemu

Przy projektowaniu Maelstorm Online, przede wszystkim kierowałem się założeniem, by umożliwić stworzenie sieciowej gry o bardzo wydajnej komunikacji pomiędzy serwerem i klientem. Liczbę przesyłanych w obie strony informacji należało więc ograniczyć do minimum. Jednocześnie, ponieważ tworzyłem grę zręcznościową, szczególnie istotna była odporność na opóźnienia transmisji sieci w aplikacji klienta i serwera.

Kolejną kwestią była skalowalność projektu. Maelstorm w chwili obecnej jest malutką grą akcji, chciałem jednak stworzyć infrastrukturę na której można by było oprzeć dużo bardziej rozbudowaną i zróżnicowaną grę. Starałem się by projekt był elastyczny w rozbudowie oraz umożliwiał łatwą zamianę użytych rozwiązań dzięki dużej modularyzacji kodu.

3.2.2. Świat gry

Na całokształt dynamicznego świata gry składają się obiekty gry[5] *MapObject* i gracze *Player*. Gracz to abstrakcyjny obiekt reprezentujący obiekt decyzyjny, sterujący obiektem gry i zbierający o nim informacje. *Player* może na aplikacji klienckiej reprezentować gracza lokalnego. Może być graczem zdalnym, z którym jesteśmy bezpośrednio połączeni (ta sytuacja występuje na aplikacji serwera), bądź jedynie abstrakcją gracza, z którą nie możemy się bezpośrednio komunikować (inni gracze na aplikacji klienckiej). Może to też być gracz AI sterujący obiektem. W silniku zostawiłem też furtkę, by zaimplementować sterowanie wielu obiektów gry przez pojedynczy obiekt gracza, co uprości tworzenie zorganizowanej grupy prostych obiektów. Oczywiście nie każdy obiekt gry musi być sterowany. Proste obiekty nie analizujące otaczającego je środowiska, jak pociski czy szczątki asteroidy nie mają przypisanego im gracza.



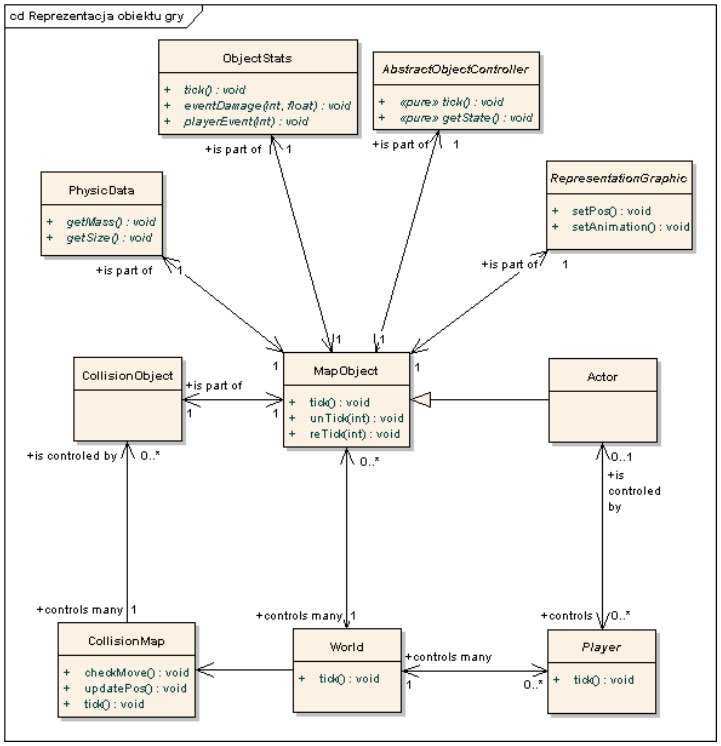
Rysunek 3.2: "Świat gry"

[5]Terminu obiekt gry będę używał w dalszej części pracy w odniesieniu do podstawowego elementu biorącego udział w symulacji świata gry. W zależności od gry "obiektem gry" mogę nazwać biorące w niej udział postacie, reaktywne elementy otoczenia czy na przykład pojazdy.

Akcje świata gry

Klasa *WorldAction* reprezentuje akcje świata gry. Są to bardzo ulotne obiekty, o krótkim czasie istnienia. Akcje są trzymane w kolejce priorytetowej, w której panuje porządek zupełny, to znaczy akcje nawet w jednej dyskretnej jednostce czasu rozpatrywane są w zadanej kolejności. Każda akcja potrafi się "wykonać". Każde istotne zdarzenie gry, które zależy od interakcji pomiędzy obiektami albo reakcji gracza, a nie jest tylko naturalnym następstwem wewnętrznego stanu obiektu, jest reprezentowane przez akcję świata. Poniżej opiszę ten proces dokładniej. Akcje gry są jedynymi obiektami gry bezpośrednio przesyłanymi przez sieć, po uruchomieniu pełnej komunikacji pomiędzy klientem a serwerem.

Ponieważ operacje tworzenia i usuwania obiektów akcji potencjalnie mogły być wąskim gardłem aplikacji stworzyłem własny cykliczny alokator dla nich. Ponieważ są to obiekty o krótkim czasie życia, alokator może użyć dopasowanej do ich sposobu użycia strategii przydzielania i zwalniania ich pamięci. Szczegóły na temat konstrukcji alokatora cyklicznego znajdują się w rozdziale 3.8.1.



Rysunek 3.3: Diagram klas dla hierarchii reprezentującej obiekt gry.

Obiekt gry

Chciałem uniknąć trzymania nadmiarowych informacji w obiektach gry, przy jednoczesnym zachowaniu skalowalności projektu. W oczywisty sposób wiązało się to z mechanizmami obiektowymi, głównie dziedziczeniem. Tak więc obiekt posiadał tylko potrzebne mu dane, przynajmniej jeżeli chodzi o informacje, które mogły być przesyłane w komunikacji klient - serwer. To oczywiście zagrażało skalowalności. Przypuśćmy, że będziemy mieli klasy nieruchomych obiektów (których stan opisuje tylko pozycja) i poruszających się oraz klasę obiektów powiedzmy

strzelających i nieaktywnych. Wówczas jeżeli będziemy wszystkie te informacje chcieli zapisać w jednej klasie, posiadającej funkcjonalność obiektu strzelającego i nieruchomego, naturalnym rozwiązaniem będzie wielodziedziczenie lub kopiowanie kodu. Wielodziedziczenie w takich sytuacjach jest skomplikowane i osłabia skalowalność projektu. Moim rozwiązaniem było rozbicie obiektu gry na zestaw połączonych obiektów opisujących kolejne jego cechy. Poniżej opisuję klasy obiektów ściśle związanych z obiektem klasy *MapObject*.

- *Stats* — Nazwa klasy jest zaszłością z długiej historii projektu. Jest to klasa obiektów odpowiedzialnych za reakcję na zdarzenia gry. W praktyce to one odpowiadają za interakcje z innymi obiektami gry, czyli w konsekwencji za zachowanie obiektu gry od strony mechaniki rozgrywki.

- *Controler* — Klasa obiektów odpowiedzialnych za ruch. Konkretne, oparte na szablonach, podklasy kontrolera pozwalają na spamiętywanie określonej liczby ruchów w tył, co będzie istotne przy synchronizacji klient - serwer. Kontroler sprawuje pieczę nad modyfikacją stanu fizycznego - *PhysicState*, który z kolei przechowuje wszystkie potrzebne informacje o wykonywanym przez obiekt ruchu. Sam ruch jest wykonywany przez podklasy *PhysicAction*, która jest szablonem, wykonującym przekształcenia na obiekcie zadanej klasy stanu.

- *PhysicData* — Klasa, której obiekty przechowują stałe fizyczne obiektu gry. Jego rozmiar, masę, czy wartości tarcia potrzebne do określenia parametrów ruchu. Koncepcyjnie klasa ta miała tworzyć na podstawie swoich parametrów akcje przemieszczania *PhysicAction*, jednak taka funkcjonalność nie była potrzeba w trakcie implementacji gry, choć być może będzie należało ją zaimplementować.

- *Player* — Ewentualny wskaźnik do obiektu gracza, jeżeli taki jest potrzebny. Tylko obiekty klasy *Actor*, będącej podklasą *MapObject*, mogą być kontrolowane przez gracza.

- *Graphic* — Po stronie serwera klasa ta reprezentuje identyfikator zasobu graficznego. W aplikacji klienta ta klasa służy za warstwę kontrolera pomiędzy modułem graficznym a modelem gry i stanowi uchwyt do obiektu graficznego. Może on kontrolować pozycję obiektu gry oraz reagować na zdarzenia animacyjne gry.

- *CollisionProxy* — Obiekt tej klasy jest reprezentacją obiektu gry w systemie kolizji. Nie jest serializowany i jest tworzony niezależnie na komputerze klienta i serwera, a jego działanie jest zdeterminowane przez obiekt *Stats* i ewentualnie kontrolującego gracza. W tej chwili system kolizji działa w aplikacji serwera i w aplikacji klienckiej, choć w grze nie jest zaimplementowany system kontroli kolizji po stronie klienta. Nie jest on potrzebny, choć mógłby się przydać do testowania synchronizacji, jeżeli kolizje po stronie serwera i klienta by się nie pokrywały, w oczywisty sposób dowodziłoby to problemów z synchronizacją.

Taka reprezentacja obiektu gry zwiększyła skalowalność i uniwersalność całego projektu. Jako przykład podam zmiany, których dokonałem po pierwszej iteracji projektu, gdy postanowiłem uczynić go moją pracą magisterską. Udało mi się podmienić sam tylko moduł fizyki, na nowy, oparty o obiekt kontrolera. To samo spotkało cały silnik graficzny i mechanizm kolizji. Wszystkie te zmiany tylko w niewielkiej części wymagały modyfikacji w innych modułach gry. Klasy reprezentują pewną zamkniętą funkcjonalność, definiującą konkretny aspekt funkcjonowania obiektu.

Kolejną korzyścią wynikającą z powyższej reprezentacji jest możliwość dokonywania bardzo oszczędnej serializacji obiektu. By zapisać obiekt do bufora danych, wystarczy umieścić

w nim kolejne jego obiekty składowe. Każdy z nich, jako że jest reprezentantem konkretnej klasy, wie dokładnie jakie dane będą potrzebne, by odtworzyć lub zaktualizować jego stan przy odczycie z bufora.

Również definiowanie obiektów od strony danych gry jest bardzo uproszczone dzięki powyższej reprezentacji. Każda składowa odpowiada za pewien aspekt funkcjonowania obiektu. Definiując więc obiekt musimy oddzielnie określić - w jaki sposób obiekt się porusza, jak reaguje na zdarzenia interakcji z innymi obiektami i ewentualnie, kto jest odpowiedzialny za jego kontrolę. Pozwala nam to na:

- Użycie różnych sposobów poruszania dla obiektów o tym samym zachowaniu. Dla przykładu pozwala nam to na stworzenie obiektu reprezentującego standardowy statek kontrolowany przez gracza, a następnie dostosowanie sposobu jego poruszania do napędu konkretnego pojazdu.

- Rozdzielenie logiki poruszania od jego implementacji. W tej kwestii to dyskusyjne rozwiązanie, wymagające przemyślanego interfejsu modułu fizyki.

- Uwzględnienie aspektów sieciowych w projektowaniu obiektów. Fizyka jest w całości poprawnie symulowana w aplikacji klienckiej i jedynie zdarzenia kontroli mają wpływ na przebieg symulacji. Obiekty reakcji na zdarzenia w większości wypadków wymagają przesyłu wiadomości w odpowiedzi na zdarzenia gry. Inaczej ujmując, obiekty fizyki, statystyk czy kolizji reprezentują funkcjonalności wymagające różnego traktowania w mechanizmie synchronizacji.

3.2.3. Symulacja

Serwer dokonuje symulacji gry używając dyskretnej jednostki czasu. Ze stałą częstotliwością dokonuje kroku symulacji świata gry wykonując kolejno:

1. Ruchu wszystkich obiektów gry w zadanej kolejności.

2. Ruch wszystkich graczy, co wiąże się z dodaniem nowych akcji do kolejki gry.

3. Rozstrzygnięcia wszystkich kolizji, co wiąże się z dodaniem nowych akcji gry do kolejki priorytetowej.

4. Wykonania akcji, które są zaplanowane na ten stan zegara gry.

W trakcie ruchu poszczególnych obiektów oddzielnie rozpatrywany jest ruch (przez obiekt kontrolera), symulacja upływu czasu obiektu reakcji (tura obiektu *Stats*) oraz inne wewnętrzne mechanizmy kontroli obiektu (jak akcje wewnętrzne obiektu). Jest to istotne ponieważ większość obiektów jest w ciągłym ruchu, w dużym stopniu niezależnym od innych podejmowanych przez nie akcji.

Nie mając doświadczenia w szacowaniu kosztu obsługi transmisji sieciowej i wykonywania głównej pętli gry, przez cały projekt zachowałem możliwość konfiguracji częstotliwości czasu trwania jednej tury gry, jednak ten sam czas musiała mieć tura gry na serwerze i na kliencie. Istotne jest, że serwer właściwie z każdą swoją turą musi zmieścić się w założonym czasie. Gdy w czasie kolejnych tur przestanie się mieścić w danej jednostce czasu synchronizacja może zupełnie zawieść. Dobrym rozwiązaniem ewentualnego problemu, może być implementacja mechanizmów zmniejszających szczegółowość symulacji świata w momencie gdy serwer zaczyna być przeciążony (rzadkie testy kolizji, ewentualnie wstrzymanie przetwarzania niektórych zapytań sieciowych, uniemożliwienie logowania się nowych klientów, wyłączenie AI).

Drugim mechanizmem który powinienem tu zaimplementować jest czasowe wstrzymywanie symulacji, rozłączanie nadmiarowych graczy i wznawianie symulacji po rozwiązaniu problemu. Jako, że są to bardzo uciążliwe w implementacji i testowaniu mechanizmy, ramy czasowe mojej pracy magisterskiej nie pozwoliły mi na ich implementację.

Klient synchronizuje się z aplikacją serwera i czas płynie u niego możliwie tak samo. Bardzo obawiałem się tej pełnej synchronizacji czasu, czy nie będzie to przyczyną ewentualnej klęski tego protokołu komunikacyjnego. Tury gry na kliencie są podzielone taką samą dyskretną jednostką czasu. W trakcie obrotu głównej pętli gry moduł graficzny interpoluje stan obiektów gry pomiędzy aktualną turą a poprzednią i na jego podstawie je wyświetla.

Na powyższy model symulacji szczególny wpływ miała specyfika mojego projektu. W ustaleniu modelu symulacji istotny bardzo był determinizm. Właściwy porządek symulacji zapewniał, że mimo opóźnień w przybyciu akcji, które niszczą determinizm świata, przy użyciu kilku dodatkowych mechanizmów można zapewnić dostateczną synchronizację gry pomiędzy klientem a serwerem. Cała symulacja musi odbywać się w określonej kolejności identycznej po stronie klienta i serwera. To właśnie wymaganie wymusiło na przykład opóźnienie rozstrzygania efektów kolizji do czasu obsługi zdarzeń gry.

W trakcie projektowania systemu największą jednak uwagę poświęciłem zagadnieniom sieciowym i to ich aspekty miały kluczowy wpływ na proces symulacji.

Aplikacja klienta i serwera - różnice w symulacji

Symulacja gry po stronie klienta i serwera różnią się od siebie w kilku kwestiach. Rozwijając aplikacje serwera możemy ukrywać przed klientem pewne właściwości obiektów symulacji. Często nie tylko nie potrzebuje on wszystkich danych dotyczących tych obiektów, ale mógłby on je wykorzystać do nadmiarowej wiedzy, o której nie powinien wiedzieć gracz. W aktualnej implementacji Maelstorm nie skorzystałem z tej funkcjonalności, ale pierwsza wersja gry zawierała po stronie klienta dużo mniejszą liczbę informacji o obiektach reakcji (*Stats*).

Z drugiej strony mamy aplikację klienta. Zajmuje się ona tylko niewielkim podzbiorem obiektów gry, dlatego z punktu widzenia wydajnościowego, mogę sobie pozwolić na dużo większą rozrzutność. Zdarzenia gry po stronie klienta mogą wywoływać zdarzenia animacji obiektów, które przekładają się na wizualny efekt. Mogą też być generowane specjalne obiekty gry, znajdujące się wyłącznie po stronie klienta a reprezentujące na przykład efekty graficzne.

3.2.4. Synchronizacja

Efektem, który chcieliśmy uzyskać jest zapewnienie, że wszyscy podłączeni klienci będą dysponowali u siebie ograniczonym obrazem tego samego, wirtualnego świata, którego pełna reprezentacja jest trzymana przez aplikację serwera. Było wiele zagrożeń, które brałem pod uwagę rozpoczynając projektowanie systemu synchronizacji. Po pierwsze, widziane przez klienta obiekty mogą wchodzić w interakcję z obiektami przez niego nie widzianymi. Po drugie, jeżeli dowiemy się o zmianie stanu danego obiektu z opóźnieniem, to okaże się, że dostajemy wiedzę na temat prawdopodobnie nieaktualnego już stanu obiektu, z przed kilku jednostek czasu. Na tej podstawie będziemy musieli próbować wywnioskować jaki jest aktualny stan, by nie obciążać serwera niepotrzebnymi prośbami o aktualizację danych. Kolejnym ewentualnym problemem mogły być kwestie bezpieczeństwa, tak by już na poziomie projektowym zapewnić, że żadna aplikacja podająca się za klienta gry, nie będzie w stanie wykonać akcji niedostępnych zwykłym użytkownikom i zaburzających rozgrywkę.

Zaimplementowałem funkcjonalność umożliwiającą przesyłanie obiektów gry przez sieć. Jednak wysyłanie całego stanu wewnętrznego obiektu przeczyło założeniom stworzenia wy-

dajnej komunikacji i możliwości obsługi bardzo dużej liczby graczy.

By zapewnić efektywność komunikacji istotne było zminimalizowanie przesyłu danych, potrzebnych do zapewnienia spójności świata gry po stronie aplikacji serwera i klienta. Ponieważ gra jest symulowana po obu stronach w dokładnie ten sam sposób, teoretycznie przynajmniej, wpływ na jej przebieg mogły mieć wyłącznie zdarzenia wywołane akcją użytkownika, bądź interakcją obiektów.

Synchronizacja na podstawie zdarzeń

Istotą komunikacji jest więc synchronizacja na poziomie kolejki priorytetowej zdarzeń (obiektów klasy *WorldAction*). Zdarzenia dzielą się między innymi na lokalne i współdzielone za pośrednictwem sieci. Każde zdarzenie może dotyczyć maksymalnie kilku obiektów gry. Współdzielone zdarzenia zostają rozesłane z serwera do wszystkich klientów, których reprezentacja na serwerze (obiekt klasy *UserPlayer* posiadający przypisany obiekt klasy *Actor*) "widzi" jeden z obiektów gry, którego dotyczy zdarzenie. Ten prosty i czytelny mechanizm zapewnia, że klient jest informowany o zdarzeniach dotyczących widzianego przez siebie wycinka świata i jednocześnie nie dostaje nadmiarowych informacji o obiektach gry nie biorących udziału w symulacji po jego stronie.

W drugą stronę aplikacja kliencka może serwerowi wysyłać wyłącznie zdarzenia określające akcje użytkownika. Takie podejście samo w sobie wpływa na bezpieczeństwo serwera, ponieważ klient nie może wykonać w nim akcji, na którą nie zezwala interfejs udostępniony przez klasę gracza (wszystkie akcje użytkownika wyłącznie odwołują się do interfejsu klasy *UserPlayer*).

Oczywiście na podstawie samych zdarzeń nigdy nie zapewnimy w stu procentach synchronizacji świata bez użycia dodatkowych mechanizmów. Zawsze nasze akcje mogą po prostu nie przejść, albo przybyć do klienta z parosekundowym opóźnieniem. Takich sytuacji może nie da się uniknąć, ale moim nadrzędnym celem było zapewnienie, by sama architektura aplikacji pomagała w poradzeniu sobie z problemem spójności świata i synchronizacji.

Cofanie ruchów

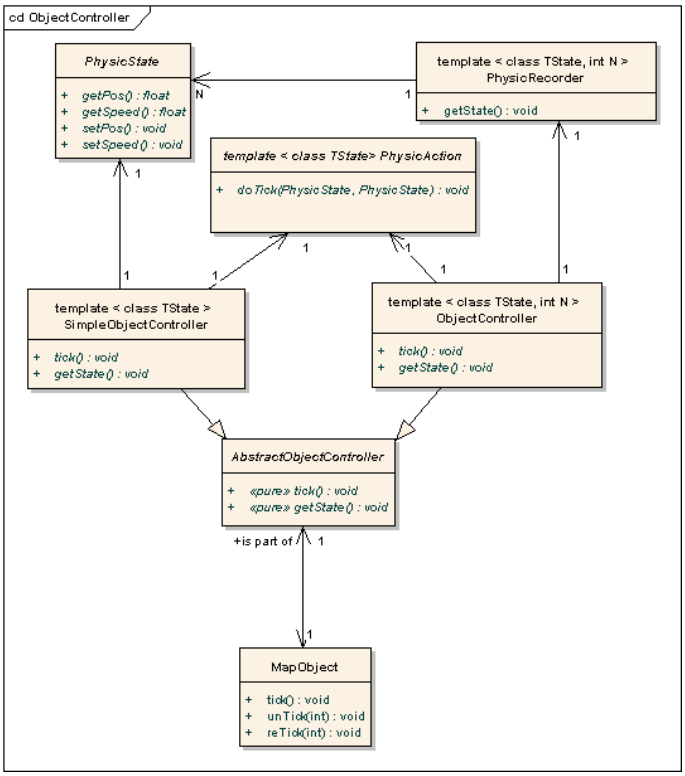
Podstawowym problemem w synchronizacji świata po obu stronach są opóźnienia w komunikacji oraz niedokładność w synchronizacji kolejki gry. By zminimalizować problemy z tym związane rzeczywisty ruch klienta wykonywany jest określony czas po wciśnięciu przez gracza przycisku. Czas ten powinien wystarczyć na przesłanie informacji o akcji gracza na serwer, a następnie rozesłanie jej do innych, obserwujących nas graczy. Nigdy jednak nie mamy pewności, że komunikaty zostaną wysłane w wymaganym czasie.

Rozwiązaniem, które zastosowałem, by poradzić sobie z zaistniałą sytuacją, jest cofanie ruchów. W pierwszej implementacji systemu stworzyłem model odwrotnej kinematyki, czyli każda akcja poruszania potrafiła się również wycofywać. Niestety to rozwiązanie okazało się problematyczne. Wbrew pozorom, błędy zaokrągleń miały bardzo duży wpływ na synchronizację. Ponieważ w symulacji ruchu wykorzystywałem model fizyki newtonowskiej wykorzystującej tarcie, prędkość w trakcie przyśpieszania miała ograniczenie górne, do którego powoli zbiegała. Cofając ruch przyśpieszający, który był już blisko granicy zbieżności, z uwagi na niedomiar wartości zmiennoprzecinkowych, zupełnie traciliśmy spójność świata po stronie klienta i serwera. Był to pierwszy problem tego rodzaju, na który się natknąłem, ale pokazywał on już słabą stronę tej metody. Postanowiłem więc przepisać moduł fizyki na nowo. Tym razem pojedynczą klasę fizyki zastąpiłem kontrolerem, tablicą pamiętanych stanów fizycznych i akcją fizyczną określającą metodę i parametry ruchu. Hierarchia klas

związana z obsługą poruszania jest przedstawiona w diagramie 3.4. W użytym rozwiązaniu obiekt spamiętuje pewną liczbę ruchów w tył. Jeżeli zażądamy od niego cofnięcia się dalej, gra zażąda aktualizacji pozycji po stronie klienta (zarówno jeżeli cofaliśmy obiekt jednej albo drugiej aplikacji).

Jeżeli więc do klienta przychodzi akcja, która powinna być wykonana w jednej z poprzednich tur gry, klient cofa ruch powiązanych z nią obiektów, następnie wykonuje akcje i ponownie symuluje fizykę obiektów. Jeżeli obiekty nie mogą zostać wycofane do danego momentu, wówczas klient prosi serwer o ich aktualizację.

Serwer otrzymuje wyłącznie akcje ruchu gracza. Uznałem za bezpieczne rozwiązanie, w którym serwer w momencie dotarcia do niego o kilka tur spóźnionej informacji o ruchu gracza sam aktualizuje jego stan u siebie. Jeżeli jednak informacja będzie zbyt spóźniona - wyśle klientowi żądanie anulowania wykonanego ruchu i aktualizacji stanu obiektu.



Rysunek 3.4: Obsługa poruszania się obiektów.

Niezawodność

Powyższy system, choć zapewnia dosyć dobrą synchronizację, nie gwarantuje oczywiście pełnej niezawodności sieci. Cofanie ruchów i opóźnienia w obsłudze wiadomości niszczą determinizm symulacji i w konsekwencji ich efekty są bardzo trudne do przewidzenia. Moim celem było takie zaprojektowanie systemu, by możliwe rzadko trzeba było dokonywać pełnej aktualizacji obiektów. W praktyce jednak rozwiązanie jest w aktualnej wersji gry absolutnie wystarczające. Grając i testując aplikację z grupą znajomych nie spotkałem się z poważniejszymi problemami synchronizacji. Oczywiście musiałem stworzyć mechanizmy, które zapewniałyby możliwość odzyskania synchronizacji na wypadek jej braku. Jest gotowe rozwiązanie

w którym gracze możliwe rzadko mogą otrzymywać aktualizację obiektów, symulowanych od relatywnie długiego czasu. Nie jest one w obecnej wersji aplikacji konieczne. Gdyby jednak trzeba było kiedyś je zastosować, troszkę lepszym rozwiązaniem, ale wymagającym implementacji bardziej skomplikowanego mechanizmu, byłoby proszenie serwer o aktualizację obiektów w momencie zauważenia rozspójnienia gry. Można by to stwierdzać na przykład analizując pokrywanie się kolizji po stronie klienta i serwera. To ciekawe rozwiązanie do implementacji w kolejnych wersjach projektu.

Dla branży gier typowe jest, że nie jesteśmy w stanie tworzyć automatycznych testów większości funkcjonalności. Nie możemy też w formalny sposób udowodnić poprawności wielu mechanizmów, zwłaszcza że sami często świadomie decydujemy się na małe oszustwa i przybliżenia. Jedynie wyczerpujące testy oraz opinie zadowolenia graczy mogą potwierdzić poprawność zastosowanych rozwiązań.

3.2.5. Wnioski

W Maelstorm aplikacja klienta ma przed graczem ukryte wyłącznie dane, których nie powinien on znać, albo które są mu zupełnie zbędne. Świat gry po jego stronie w istotnych dla niego kwestiach jest odzwierciedleniem świata gry na serwerze. Od standardowego podejścia różni się to tym, że klient jest lustrzanym odbiciem aplikacji serwera, zamiast pozostawać jedynie programem wizualizującym. Jest jeszcze inne podejście, stosowane w grach MMO, gdzie mamy rozdział na mechanikę wykonywaną po stronie serwera i po stronie klienta. Wymaga ono tytanicznej pracy by zapewnić przynajmniej w podstawowym zakresie uczciwą dla wszystkich graczy rozgrywkę. Istnieje wiele zalet mojego podejścia do roli klienta w grze.

Dla przykładu, w silniku Aurora, na którym bazował Wiedźmin, serwer bezpośrednio rozkazywał obiektowi odtwarzać dane animacje oraz dokonywać ustalonego ruchu. Problemem w tym rozwiązaniu jest to, że obiekty po stronie klienta tak naprawdę nie wiedzą jaką czynność właściwie wykonują, mają wiedzę tylko o tym jak tą czynność przedstawić. Może to między innymi prowadzić do wysyłania aplikacji klienckiej dodatkowych, nadmiarowych informacji. Stosując to podejście, wyobraźmy sobie dla przykładu sytuację drwala rąbiącego drzewo w świecie gry. By poinformować o tym aplikacje kliencką musimy wysyłać oddzielnie informacje dotyczące konieczności wyekwipowania drwala w siekierę, następnie ustawić mu pozycję, kierunek i animację. Teraz ustawiamy odpowiednią animację na ścinanym drzewie. No i zaraz potem kolejna seria informacji gdy drwal ścina drzewo... Drzewo upada... Drwal zbiera rozrzucone drewno... Wreszcie wrzuca je na taczkę, drapie się po głowie i idzie do domu... Wszystkie te czynności drwala można było przewidzieć, znając tylko sposób w jakim będzie się zachowywał.

Obsługa akcji obiektu poprzez bezpośrednią kontrolę serwera nakłada też bardzo duże ograniczenie na programistów. Pisząc aplikację klienta muszą oni określać stan obiektu na podstawie takich własności jak, dla przykładu, jego animacja. Pomyłka i brak pełnej korelacji pomiędzy stanem rzeczywistym obiektu, a własnością na podstawie której próbujemy go określić, prowadzi do bardzo ciężkich w wyśledzeniu i naprawie błędów. W Maelstorm, ponieważ klient prowadzi właściwie taką samą symulację świata, jaka dzieje się na serwerze, mamy na nim dostęp do większości informacji i funkcjonalności związanych z mechaniką gry. Dużo trudniej tu o błędne określenie stanu obiektu. Jeżeli taki błąd wystąpi, oznacza on brak spójności danych gry i jest to dużo bardziej jednoznaczne i klarowne w odnalezieniu i naprawie.

Chciałem też, by dzięki temu, że mamy do czynienia z jednym silnikiem gry po obu stronach sieci, ułatwione było programowanie nowych funkcjonalności. Implementując nowe własności gry tworzylibyśmy je w pewnym oderwaniu od mechanizmów sieciowych. Nie

jest to niestety takie proste. Choć mechanizmy, które tworzymy nie musza odwoływać się bezpośrednio do sieci, ciągle trzeba mieć na uwadze sposób w jaki zostanie zapewniona ich synchronizacja. Architektura zapewnia nam mocną barierę abstrakcji pomiędzy tworzeniem nowych funkcjonalności a obsługą transmisji danych i zdarzeń z nią związanych, jednak na każdym etapie implementacji musimy pamiętać, że tworzymy grę sieciową i każda modyfikacja symulacji gry będzie miała swoje następstwa w procesie komunikacji i zapewnienia synchronizacji. W praktyce dyskusyjne jest czy nie łatwiejszym w tej kwestii podejściem jest standardowe rozwiązanie, w którym to serwer jest odpowiedzialny za realizacje mechaniki a klient wyłącznie za wyświetlanie. W tym konkretnym aspekcie, standardowe podejście może być prostsze, ponieważ projektując nową funkcjonalność po stronie mechaniki gry, istnieje mniejsza szansa na to, że wpłynie ona na aspekty związane z synchronizacją aplikacji.

Kolejną, pozytywną cechą projektu jest skalowalność. Właściwie w każdym jego aspekcie uważam, że jest dobrze przystosowany do ewentualnej rozbudowy czy wprowadzania nawet gruntownych zmian. Próbując dla przykładu stworzyć na podstawie silnika Maelstorm grę dla jednego gracza, wystarczy tylko wyłączyć obsługę sieci i podpiąć moduł wyświetlający grafikę do aplikacji serwera. Skalowalność architektury udowodniłem częściowo podmieniając w trakcie prac nad projektem całe kluczowe moduły gry (fizykę, kontrolę obiektów, obsługę sieci oraz kilkukrotnie moduł graficzny). Uzyskałem ją dzięki stosowaniu wirtualnego interfejsu dla używanych bibliotek zewnętrznych oraz w miarę trafnemu podziałowi obiektu gry na elementy składowe, decydujące o różnych aspektach rozgrywki.

Dyskusyjnym aspektem technicznym, jest kompilacja zarówno aplikacji serwera i klienta na podstawie tego samego kodu. Wymaga to stosowania dyrektyw kompilatora do oddzielenia fragmentów dotyczących tylko jednej z aplikacji. Zmniejsza to czytelność kodu, zwłaszcza głównej aplikacji gry. Mógłbym ten proces znacząco uprościć tworząc wspólny interfejs głównych klas sterujących przebiegiem gry (*Game* i *World*) i implementując go oddzielnie po stronie serwera i klienta. Moje rozwiązanie zastosowałem by przyśpieszyć prace nad projektem, poprawić wydajność, by uniknąć nadmiarowego kopiowania kodu oraz dla automatycznego zapewnienia, że symulacja po obu stronach przebiega zgodnie.

3.3. Implementacja warstwy sieciowej

Implementacja protokołu komunikacji była dla mnie bardzo ciekawym doświadczeniem. Osobiście po raz pierwszy zmierzyłem się z zagadnieniami implementacji pełnej obsługi sieci w grze komputerowej. Decydując się na wybór określonych protokołów komunikacyjnych i technologii, opierałem się na opiniach znalezionych w artykułach na które natrafiłem w internecie i literaturze. W poniższym podrozdziale chciałem podzielić się z czytelnikiem doświadczeniami i przemyśleniami związanymi z tworzeniem warstwy sieciowej gry Maelstorm.

3.3.1. Wybór protokołu komunikacyjnego

W czasach modemów o transmisji 56 Kb/s standardem było, że gry akcji wykorzystywały do komunikacji protokół UDP[6]. To wymuszało implementację własnych mechanizmów zapewnienia niezawodności sieci. Dopasowana do mechaniki gry implementacja tych mechanizmów mogła być wydajniejsza niż stosowana w TCP[7], a z uwagi na małą przepustowość internetu należało je maksymalnie wykorzystać. Minęło jednak trochę czasu. Z uwagi na popularność, protokół TCP już nie tylko gwarantuje niezawodność transmisji. Ponieważ jest najpopularniejszym protokołem internetowym ma bardzo mocne wsparcie od strony architektury sieciowej, serwerów, routerów i obsługi ścian ogniowych. Właściwie we wszystkich opracowaniach dotyczących oprogramowania sieci w grach słyszałem podobne argumenty przemawiające za wyborem protokołu TCP zamiast UDP. Jako ciekawostkę podam fakt, że moje wątpliwości dodatkowo pomógł rozwiać fakt, że Guild Wars, wspominana uprzednio gra MMO o prawdopodobnie najwydajniejszej komunikacji sieciowej, również korzysta z protokołu TCP.

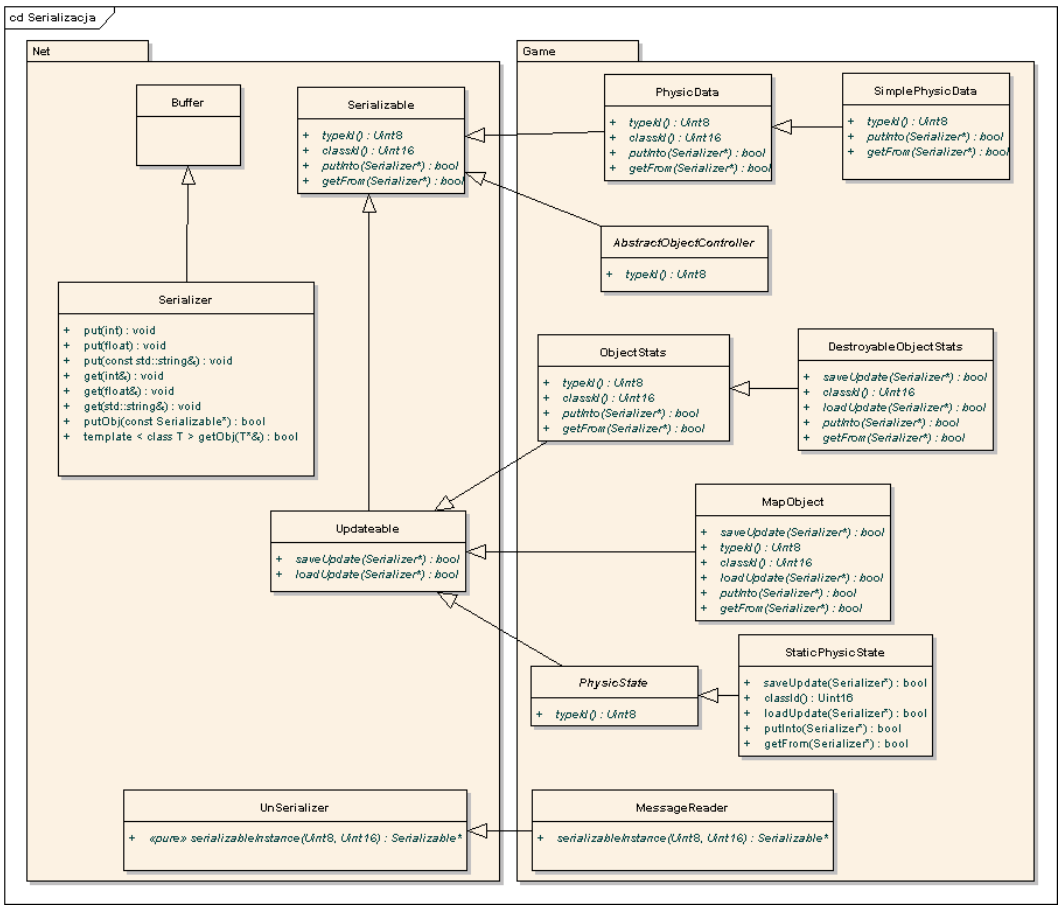
3.3.2. Serializacja

Możliwość serializacji i deserializacji obiektów była podstawową funkcjonalnością, która umożliwiła mi stworzenie bariery abstrakcji pomiędzy obiektowym modelem gry a obsługą sieci opartą o strumieniowe przesyłanie pakietów danych. Wzorowałem projekt systemu na rozdziale Jasona Beardsleya, który znalazłem w Perełkach Programowania Gier [DLo3].

Przedstawiona na diagramie 3.5 klasa *Serializer* służy do zapisu i odczytu obiektów przed wysłaniem i po odebraniu ich z sieci. Przeciążam w niej metody *put()* i *get()*, do obsługi wszystkich podstawowych typów wykorzystywanych przez obiekty gry. Dzięki temu serializacja obiektu sprowadzała się do wywołania tej samej, przeciążonej metody, na przesyłanych przez sieć atrybutach. Klasa *Serializable* reprezentuje obiekty gry, które ostatecznie chcemy przenosić. Ponieważ do implementacji jednolitego interfejsu do zapisu i odczytu tych obiektów przez obiekty *Serializer* używam szablonów, nie mogę użyć dla nich przeciążonych operatorów *put()* i *get()*. Wywołanie metod zapisu i odczytu obiektów automatycznie dobierze właściwe parametry dla szablonów, więc całość jest równie prosta w obsłudze. Obiekty są odczytywane przez Serializer za pomocą klasy gry, implementującej interfejs *UnSerializer*. Na podstawie identyfikatora typu (oznaczającego rodzinę obiektów dziedziczących niekoniecznie bezpośrednio po tej samej nadklasie gry) oraz klasy (identyfikującego konkretną klasę danego typu) jest tworzony obiekt będący czystą instancją danej klasy. Następnie obiekt wczytuje swoje

[6]User Datagram Protocol. Wydajny protokół warstwy transportu wysyłający wiadomości w formie datagramów. Nie gwarantuje tego, że wiadomości zostaną odczytane w kolejności wysłania, mogą też zostać odczytane podwójnie lub w ogóle nie dotrzeć do celu.

[7]Transmission Control Protocol. Protokół warstwy transportowej wysyłający dane w postaci strumieniowej. Na ile to możliwe zapewnia dostarczenie danych oraz gwarantuje, że zostaną one odczytane w kolejności wysłania.



Rysunek 3.5: Interfejs klas odpowiedzialnych za realizację serializacji obiektów gry.

atrybuty z bufora danych, reprezentowanego przez obiekt klasy *Serializer*. Jest możliwe obsługiwanie po stronie aplikacji serwera i klienta odczytu wyłącznie tych klas obiektów, które dana aplikacja rzeczywiście może odebrać z sieci, co dodatkowo zwiększa bezpieczeństwo komunikacji (ograniczamy jakie klasy obiektów mogą zostać wczytane za pośrednictwem sieci w aplikacji serwera lub klienta).

Powyższy mechanizm serializacji ma prosty i skąpy interfejs. Za jego pośrednictwem mogę przenosić złożone obiekty pomiędzy aplikacją serwera i kliencką. Mechanizm wykorzystywałem również w jeden, dyskusyjny sposób. Ponieważ zwykle wiem po której stronie obiekt jest zapisywany a po której odczytywany, kilkakrotnie pozwoliłem sobie korzystać z tej wiedzy. Część obiektów serwera przy zapisie do bufora nie zapisuje części danych, których nie chcemy udostępnić klientowi. To rozwiązanie ogranicza troszkę skalowalność projektu, ale jego ewentualna podmiana na ogólniejsze jest prosta.

System serializacji sprawdził się w praktyce znakomicie. Działa bez zarzutu, jest bardzo prosty w rozwoju oraz wyszukiwaniu błędów. Był obecny w pierwszej iteracji projektu i właściwie od tego czasu niewiele się zmienił. Funkcje zapisu i odczytu tworzy się bardzo szybko i dzięki prostocie w stosowaniu błędy w nich zdarzają się rzadko.

3.3.3. Obsługa połączeń

Decydując się na tworzeniu warstwy obsługi sieci musiałem zdecydować się co do sposobu obsługi połączeń TCP. Ponieważ gra może w założeniu obsługiwać powyżej setki połączeń, nie mogłem sobie pozwolić na ich synchroniczną obsługę - co wymusza tworzenie osobnego wątku do obsługi każdego połączenia. Z kolei bałem się jak poradzi sobie pojedynczy zbiór gniazd z obsługą rzeczywiście dużej liczby połączeń. Znalazłem artykuł w Perełkach Programowania Gier [AKi4] dotyczące obsługi serwera w grach dla bardzo dużej liczby graczy. Autor sugerował w nim, że gdy musimy w grze utrzymywać rzeczywiście duże liczby połączeń, najlepszym rozwiązaniem jest asynchroniczne przetwarzanie żądań przez kilka niezależnych wątków. Dla liczby pięciuset do tysiąca połączeń mówił tu o pięciu wątkach. Nie mając doświadczenia w tej kwestii oczywiście zastosowałem dokładnie to rozwiązanie. Połączenia są trzymane przez obiekty klasy *ConnectionHandler*, będące abstrakcją dla procesu obsługującego ich żądania. Gdy tworzymy nowe połączenie sprawdzamy czy wszystkie istniejące procesy obsługują już pewną minimalną liczbę połączeń. Jeżeli tak jest oraz liczba procesów jest mniejsza od maksymalnie ustalonej, wówczas tworzony jest nowy obiekt *ConnectionHandler*, uruchamiający nowy proces przetwarzania połączeń. W innym przypadku nowe połączenia dystrybuowane są pomiędzy procesami tak by zrównoważyć ich rozkład.

3.4. Dynamiczne wykrywanie kolizji

Potencjalnie, wąskim gardłem mechaniki gry może być systemu poruszania i wykrywania kolizji obiektów. Zwykle to ruch jest najczęściej wykonywaną przez obiekty akcją i zwykle związany on być musi właśnie z wykrywaniem kolizji. W projekcie Maelstorm mogłem się ograniczyć wyłącznie do wykrywaniem kolizji pomiędzy obiektami gry oraz testowaniem widoczności obiektów gry przez graczy. Zwykle jednak na wykrywanie kolizji składa się cały zbiór mechanizmów, ściśle zależnych od świata gry. Implementacja tych modułów ma kluczowe znaczenie dla aspektów wydajnościowych mechaniki gry.

3.4.1. Kolizje obiektów

W wypadku testów kolizji pomiędzy obiektami, naszym zadaniem jest stwierdzenie, że w danym punkcie lub na danej linii obiekt nie koliduje z innymi. Zwykle sam test kolizji dwóch obiektów jest w miarę prosty w implementacji, ponieważ obiekty reprezentować możemy w przybliżony sposób. Postacie mogą być w trójwymiarowym świecie walcami o podstawie koła. Jeśli potrzebujemy dokładnych kolizji, dowolnych trójwymiarowych obiektów, zbudowanych z siatki trójkątów, również dla przyśpieszenia podstawowego testu, przybliżamy je najpierw za pomocą prostej, symetrycznej bryły, takiej jak sześcian albo kula. Jeśli bryły obiektów będą kolidować ze sobą, szczegółową kolizję sprawdzamy szukając przecięć trójkątów z których zbudowane są obiekty.

W tej części chciałem skupić się raczej na sposobach ograniczenia liczby obiektów testowych. Jeżeli dla każdego obiektu będziemy przeprowadzać test kolizji ze wszystkimi pozostałymi, czas samego sprawdzania kolizji w turze gry będzie zależny co najmniej kwadratowo od liczby obiektów.

Do testów kolizji obiektów gry można podejść dwojako, w zależności od tego jakie wymagania funkcjonalne nakładamy na silnik naszej gry. Możemy liczyć je wszystkie raz na turę gry w konkretnym miejscu pętli. To utrudnia (ale nie uniemożliwia) zintegrowanie systemu kolizji z systemem sztucznej inteligencji i wykonywanych akcji obiektów, ale upraszcza wymuszenie determinizmu w naszej symulacji[8]. Obiekty gry mogą też sprawdzać kolizje we własnym zakresie, co jest dobrym rozwiązaniem, gdy po prostu nie chcemy wpuścić obiektów w swoją przestrzeń osobistą i nie chcemy by były sprawdzane kolizje dla obiektów pozostających w spoczynku.

W Wiedźminie zastosowane zostało to drugie podejście. Obiekty poruszając się sprawdzały czy nikt nie stoi im na drodze. Ewentualna kolizja powodowała próbę ominięcia przeszkody bądź zatrzymanie się. W Maelstorm połączyłem oba rozwiązania. Obiekty wykrywają kolizje w swojej turze ruchu. Wykryta kolizja jest odkładana do tablicy kolizji, o ile nie spowodował jej wcześniej obiekt z którym kolidujemy. Powtarzanie się kolizji jest sprawdzane na podstawie tablicy mieszającej, biorącej pod uwagę wartości funkcji mieszającej obu obiektów oraz turę gry (czyli tablica automatycznie resetuje się co turę gry). Wszystkie kolizje rozpatrywane są kolejno w określonej części tury gry.

Sortowanie obiektów

Prostym rozwiązaniem o ograniczonej skuteczności jest trzymanie obiektów gry na posortowanej względem konkretnej współrzędnej liście. To rozwiązanie zostało użyte w Wiedźminie,

[8]Determinizm w grze jest istotną własnością, jeżeli zdecydujemy się ją zapewnić istotne jest uwzględnienie tego na etapie projektowania podstawowego silnika gry. Determinizm może być istotny w systemach symulacji fizyki, grach sieciowych takich jak Maelstorm, w grach umożliwiających zapisywanie przebiegu rozgrywki, czy cofanie czasu gry albo inne modyfikacje jego przebiegu.

gdzie liczba obiektów gry zwykle nie przekraczała stu jednocześnie się poruszających. Oczywiste są ograniczenia tego rozwiązania. Najczęściej jako listy używamy posortowanej tablicy, przez co dodawanie i usuwanie elementów tablicy odbywa się w czasie liniowym. W zamian za to inne operacje zyskują na wydajności (wyszukiwanie elementu i przeglądanie posortowanej tablicy działa szybciej od, dla przykładu, implementacji zbiorów *ang. set* z biblioteki STL). Aktualizacja położenia w tablicy odbywa się w oczekiwanym czasie logarytmicznym, choć nie można wykluczyć, że w pesymistycznym przypadku czas ten będzie liniowy. To rozwiązanie przedstawiłem z uwagi na jego prostotę, niewielki koszt pamięciowy i w wypadku wielu projektów, wystarczającą skalę spodziewanej optymalizacji.

Zastosowanie siatki

By zminimalizować zbiór, branych pod uwagę przy testowaniu kolizji obiektów, możemy zastosować podział świata gry na siatkę (dla uproszczenia przyjmijmy, że jest ona dwuwymiarowa) kwadratowych, jednakowych pól. W każdej komórce siatki znajduje się lista obiektów mająca w niej swój środek. Rozmiar komórek siatki jest tak dobrany do rozmiaru obiektów gry, że test kolizji zadanego obiektu, wymaga przejścia wyłącznie list znajdujących się w jego komórce oraz komórkach przyległych.

To bardzo skuteczne rozwiązanie, problemem może być w nim tylko reprezentacja większych obiektów. Można to zrobić, zmuszając je by zajmowały one większą liczbę kratek, co ostatecznie może spowolnić zarówno aktualizację jak i wyszukiwanie widoczności. Jest to dobre rozwiązanie dla gier, w których obiektów większych od standardowego będzie z założenia bardzo mało.

Drugim sposobem na obsługę obiektów różnych rozmiarów jest zastosowanie różnych poziomów szczegółowości siatki kolizji. Najlepiej jest przyjąć, że każdy poziom szczegółowości ma dwukrotnie dłuższą krawędź podziałki[9]. Wówczas obiekt jest zaznaczany na siatce odpowiadającej jego wielkości i wszystkich mniej szczegółowych (z większą podziałką). Obiekt testując kolizję sprawdza ją na swoim poziomie szczegółowości, po czym testuje ją na wyższych poziomach, z obiektami o rozmiarach im odpowiadających.

Tego rozwiązania użyłem przy implementacji systemu kolizji w grze Maelstorm. Istnieje w niej zadana liczba poziomów szczegółowości siatki. Obiekt poruszając się sprawdza kolizje z obiektami występującymi w swoim i przyległych polach siatki. Rozwiązanie wydaje się działać bardzo skutecznie. Podobnego mechanizmu użyłem do sprawdzania widoczności. Implementacja siatki widoczności różni się małymi szczegółami. W poszczególnych komórkach siatki trzymam obiekty w innej strukturze danych niż w siatce kolizji (w zbiorze zamiast w tablicy). Drugą różnicą implementacji systemu widoczności, jest funkcjonalność, w której obiekty nie zmieniają swojego położenia w siatce, jeżeli tylko minimalnie przechodzą na inną komórkę (z uwagi na dłuższy czas dodawania i usuwania elementów z wypełnionych obiektami zbiorów). Minusem tego rozwiązania jest duże zapotrzebowanie na pamięć. Szczególnie tyczy się to sytuacji, gdy próbujemy zastosować małą podziałkę siatki, dla dużego świata gry. Węzeł siatki zwykle niekoniecznie jest prostą strukturą i z przyczyn wydajnościowych powinien mieć zaalokowaną pewną pamięć na ewentualne obiekty w nim przebywające.

Rekurencyjne grupowanie wymiarami

Wrócimy teraz jeszcze na chwilę do pomysłu sortowania obiektów. Opiera się na nim algorytm rekurencyjnego grupowania wymiarami (RDC *Recursive Dimensional Clustering*). Dla każ-

[9]Ogólnie w odniesieniu do siatki kolizji istotne jest by jej krawędź była potęgą dwójki. Dzięki temu wyznaczenie pola siatki dla danej pozycji sprowadza się do prostych operacji przesunięcia bitowego.

dego wymiaru w jakim rozpatrujemy zderzenia będziemy dynamicznie tworzyli posortowaną tablicę. Tablica będzie zawierała minimalne i maksymalne współrzędne na jakich rozpostarte są rozpatrywane obiekty gry. Konstrukcja tej tablicy odbywa się w czasie liniowym, sortowanie w czasie $O(nlog(n))$. Po utworzeniu przeglądamy tablicę próbując wyszczególnić w niej grupy - czyli zbiory obiektów których początki i końce na siebie nachodzą. Ten krok może być wykonany w czasie liniowym. Następnie dla każdej grupy rekurencyjnie wykonywany jest krok RDC, sortujący obiekty tym razem według innej współrzędnej. Algorytm zostaje przerwany, jeżeli rozpatrywana grupa jest jednoelementowa, bądź nie została podzielona na mniejsze względem żadnej ze współrzędnych.

Algorytm RDC (Recursive Dimensional Clustering) dla przestrzeni dwuwymiarowej działa w następujący sposób:

1. Na wejściu zbiór wszystkich obiektów traktujemy jako grupę.
2. Tworzymy listę granic obiektów według aktualnie rozpatrywanej współrzędnej.
3. Sortujemy listę granic.
4. Na podstawie listy granic dzielimy rozpatrywaną grupę na mniejsze.
5. Dla każdej odnalezionej grupy rekurencyjnie załączamy algorytm analizując kolejną współrzędną. Jeżeli nie dokonaliśmy nowego podziału, analizujemy według kolejnej współrzędnej grupę wejściową. Jeżeli nie udało nam się rozpatrywanej grupy podzielić według żadnej, kolejno rozpatrywanej współrzędnej kończymy działanie algorytmu.

Oczekiwany czas działania algorytmu, dla n obiektów, jest zbliżony do $O(nlog(n))$ – jeżeli założymy, że możemy ograniczyć z góry średnią głębokość na jaką algorytm się zagłębia. W pesymistycznym przypadku może wzrosnąć do $O(n^2 log(n))$, jednak dotyczy to sytuacji, które nie powinny mieć miejsca w rzeczywistych zastosowaniach.

Algorytm RDC radzi sobie dobrze w sytuacjach, w których obiekty są w miarę równomiernie rozrzucone po świecie gry. Można go też użyć w odniesieniu do obiektów statycznych, by grupować je, co może być następnie użyte w tworzeniu struktur wspomagających sprawdzanie z nimi kolizji. Algorytm źle radzi sobie w momencie gdy wszystkie obiekty kolidują ze sobą wzajemnie. Wówczas rekurencja może schodzić bardzo głęboko a algorytm nie jest w stanie rozdzielić obiektów na grupy. Nie miałem praktycznej styczności z powyższym algorytmem a na jego dokładne opracowanie natknąłem się w *Perełkach Programowania Gier* [DLo2][10]. W internecie znalazłem bardzo różnorodne opinie na jego temat (od w miarę pozytywnych, do bardzo negatywnych), dlatego wspominam o nim głównie by pokazać alternatywę w podejściu do problemu wyszukiwania kolizji. Myślę, że jest dobrym rozwiązaniem dla projektów, w których obiekty są w ciągłym ruchu a kolizje wykrywamy raz na turę w określonej procedurze. Przykładem dobrym do wykorzystania tego podejścia może być silnik fizyczny[11].

3.4.2. Kolizje z geometrią świata

Istotnym elementem omawianego modułu jest funkcjonalność testowania kolizji obiektów z geometrią świata. Szczęśliwie w Maelstorm uniknąłem implementacji tego mechanizmu, jednak miałem z nim sporo styczności w trakcie prac nad Wiedźminem. Trzeba pamiętać, że

[10] Inne opracowanie algorytmu można znaleźć na stronie: *http://lab.polygonal.de/articles/recursive-dimensional-clustering*

[11] Obok silników graficznych, fizyka jest modułem gry najczęściej importowanym od zewnętrznych dystrybutorów. Na rynku komercyjnym najbardziej znanymi są silniki *Havok Physics* oraz *PhysX* firmy *AGEIA*.

geometria świata gry jest tworzona zwykle z myślą o wyświetlaniu jej i w praktyce może być bardzo skomplikowana. Świat gry składa się zwykle z olbrzymiej liczby opisujących go trójkątów, które tylko w niewielkim stopniu są dla prowadzonego przez gracza bohatera podłogą albo ścianą. Testowanie kolizji z tak skomplikowaną geometrią może być stanowczo zbyt kosztowne. Dlatego w grach popularnie wprowadza się dodatkową geometrię, generowaną bądź tworzoną przez grafików. Najczęściej ma ona postać siatki nawigacyjnej. Jest to geometria złożona z trójkątów, która wyznacza powierzchnię po której mogą poruszać się obiekty gry. Może ją wyznaczać dokładnie, tak że pozycja znajdujących się na niej obiektów jest ustalana wyłącznie na podstawie geometrii nawigacyjnej. Wysokość może być też pobierana z z bardziej szczegółowej geometrii świata. To rozwiązanie jest preferowane, ponieważ pozwala na uproszczenie geometrii nawigacyjnej. W konsekwencji prowadzi to do przyśpieszenia pozostałych operacji wykonywanych na niej oraz ułatwia jej wyznaczanie przez projektantów gry.

Dla każdego trójkąta musimy móc podać w czasie liniowym trójkąty z nim sąsiadujące. Wówczas, dzięki zastosowaniu geometrii nawigacyjnej kolizje postaci z krawędziami świata można liczyć, biorąc trójkąt siatki nawigacyjnej, na którym znajduje się postać. Następnie przeszukuje się wszerz trójkąty siatki nawigacyjnej leżące pod obrysem postaci. Jeżeli natrafimy na "dziurę"(brak sąsiada), bądź trójkąt po którym nie powinniśmy chodzić, oznacza to, że wychodzimy poza świat. Koszt czasowy takiego rozwiązania jest rzędu $O(log(n) + mlog(m))$, gdzie n to liczba trójkątów siatki nawigacyjnej (koszt wyszukania startowego trójkąta), a m to liczba trójkątów leżących pod testowanym obszarem (koszt przeglądania wszerz algorytmem Dijkstry).

Zastosowanie geometrii nawigacyjnej to oczywiście pewna dodatkowa praca, którą musi wykonać dział przygotowujący lokacje, na których toczy się gra. Jednak dzięki niej poza przyśpieszeniem wykrywania kolizji, uzyskujemy możliwość kontrolowania obszaru po jakim zezwalamy poruszać się graczowi oraz unikamy problemów związanych z niewłaściwą interpretacją geometrii świata przez silnik gry.

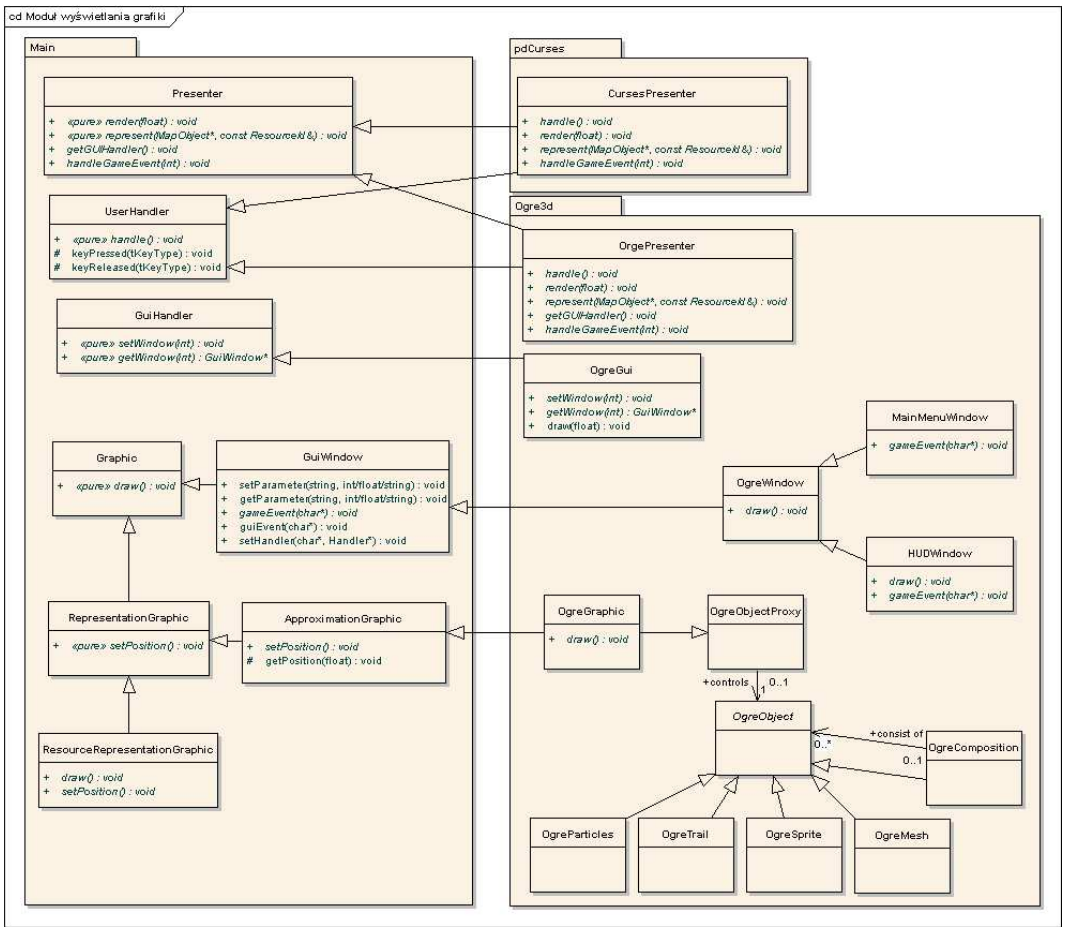
W Wiedźminie siatka nawigacyjna była tworzona przez grafików i określała obszar po którym można się poruszać. Wysokość postaci w danym punkcie brana była jednak z geometrii kolizyjnej świata, a jej odległość od geometrii nawigacyjnej w pionie mogła wynosić co najwyżej pół metra (w metryce świata gry).

3.4.3. Wnioski

Nie omówiłem bynajmniej wszystkich aspektów związanych z wykrywaniem kolizji. Starałem się przybliżyć parę podstawowych, popularnie stosowanych rozwiązań. Implementacja sprawnego, dostosowanego do świata gry i wymagań projektu, systemu kolizji jest bardzo istotna. Gdy użyjemy narzędzi profilujących, pozwalających śledzić czas wykonania konkretnych funkcji w czasie działania gry, zobaczymy że wydajność modułów wyszukiwania ścieżek i poruszania jest zwykle ściśle zależna od systemu kolizji z którego korzystają. System kolizji jest nierozerwalnie powiązany z reprezentacją świata gry w kodzie. Dlatego tworząc ambitny projekt istotne jest by oprzeć go na mocnych podstawach, a w ich skład wchodzi właśnie odpowiednio wydajny system wykrywania kolizji.

3.5. Wyświetlanie grafiki

Moduł wyświetlania grafiki Maelstorm Online nie był dla mnie najistotniejszym elementem gry. Nie siliłem się na optymalne wykorzystanie mocy obliczeniowej procesora oraz dostępnej pamięci operacyjnej. Pisząc go starałem się by był możliwie skalowalny i niezależny od pozostałej części kodu. Starałem się uczynić interfejs pomiędzy mechaniką gry a jej wyświetlaniem możliwie prosty i jednoznaczny, jednocześnie umożliwiając jego podmianę oraz rozbudowę.



Rysunek 3.6: Interfejs pomiędzy grą a zewnętrznymi bibliotekami.

3.5.1. Interfejs modułu graficznego

Założeniem, którego przestrzegałem w implementacji Maelstorm było maksymalne ograniczenie interfejsu zewnętrznych bibliotek z kodem aplikacji. Wyjątek stanowiły biblioteki narzędziowe (BOOST i STL) i biblioteka sieciowa asio. Importując pozostałe biblioteki tworzyłem klasę gry mającą maksymalnie rozbudowaną uniwersalną funkcjonalność i pozostawiającą wirtualny interfejs, który powinien być zdefiniowany przez konkretną bibliotekę. W kodzie gry odnosiłem się wyłącznie do abstrakcyjnej klasy przesłaniającej funkcjonalność opartą o zewnętrzne biblioteki. Diagram klas 3.6 przedstawia metodologię implementacji bibliotek którą zastosowałem w kodzie aplikacji.

Ta architektura sprawdziła się już wielokrotnie. Proporcjonalnie bardzo małym nakładem

sił podmieniałem już cały moduł wyświetlania. Najpierw pierwszą implementację systemu wyświetlania na moduł *Ogre3d*. Później z przyczyn wydajnościowych zredukowałem wyświetlanie stanu gry na serwerze, do tekstowego wyświetlania w konsoli za pomocą biblioteki *pdCurses*.

3.5.2. Animacja

System animacji w Maelstorm powstał w końcowym okresie prac nad projektem. Projektując go miałem na uwadze błędy leżące u podstaw analogicznego systemu w grze Wiedźmin. Był on pozostałością z Aurory, starego już silnika który stanowił bazę gry. Cały mechanizm wyświetlania w Wiedźminie został napisany na nowo, lecz pewne elementy starej architektury pozostały. Jednym z nich był właśnie system animacji. Aurora była silnikiem Neverwinter Nights, chyba pierwszej w pełni trójwymiarowej gry RPG. Była to gra oferująca możliwość gry wieloosobowej, tak więc w kodzie mieliśmy aplikację napisaną w architekturze klienta i serwera. Ten rozdział został zachowany w Wiedźminie. Z początku, by umożliwić ewentualne stworzenie gry wieloosobowej. Później gdy projekt urósł, było zbyt późno na modyfikacje tak fundamentalnych elementów silnika. Dzięki temu, mimo że nie pracowałem nad Wiedźminem od początku, miałem styczność z rozwiązaniami kwestii gry dla wielu graczy użytymi w Neverwinter Nights. System animacji był projektowany z uwzględnieniem konieczności obsługi wydajnej komunikacji sieciowej.

Chciałem więc uniknąć błędów projektowych systemu animacji, z którymi miałem styczność w pracy zawodowej. Pierwszym z nich było ustawianie animacji obiektów klienta przez serwer. Tą kwestię omówiłem dokładniej w sekcji 3.2. Kolejnym problemem występującym w Wiedźminie była sama reprezentacja animacji. Zdarzenie animacji wysyłane siecią miało postać liczby całkowitej, identyfikującej konkretną animację. Niestety nie stworzyliśmy, żadnego spójnego systemu pobierania dodatkowych danych dotyczących animacji, zadanej w postaci identyfikatora. To prowadziło do tego, że sprawdzenie dodatkowych informacji o animacji, określonej liczbą całkowitą, implikuje wywołanie jednej z funkcji wypełnionych albo gigantycznym blokiem *switch* albo ciągiem niepotrzebnych porównań. Ponieważ pewne animacje musiały być traktowane w sposób wyjątkowy, funkcje ustawiające je po stronie klienta były niesłychanie skomplikowane, wypełnione potężną liczbą porównań, często po części już niepotrzebnych. W systemie nie sposób było odróżnić martwy, nieużywany kod od rzeczywistych rozwiązań rzadko występujących błędów. Jakiekolwiek zmiany w ogólnym systemie animacji były niemożliwe, ponieważ nikt nie panował nad jego kodem. Każda modyfikacja groziła zawsze poważnymi błędami, mogącymi wystąpić w zupełnie wyjątkowych sytuacjach. System animacji był tu na granicy skalowalności, ewentualne próby jego nawet najmniejszej rozbudowy były bardzo trudne do przeprowadzenia.

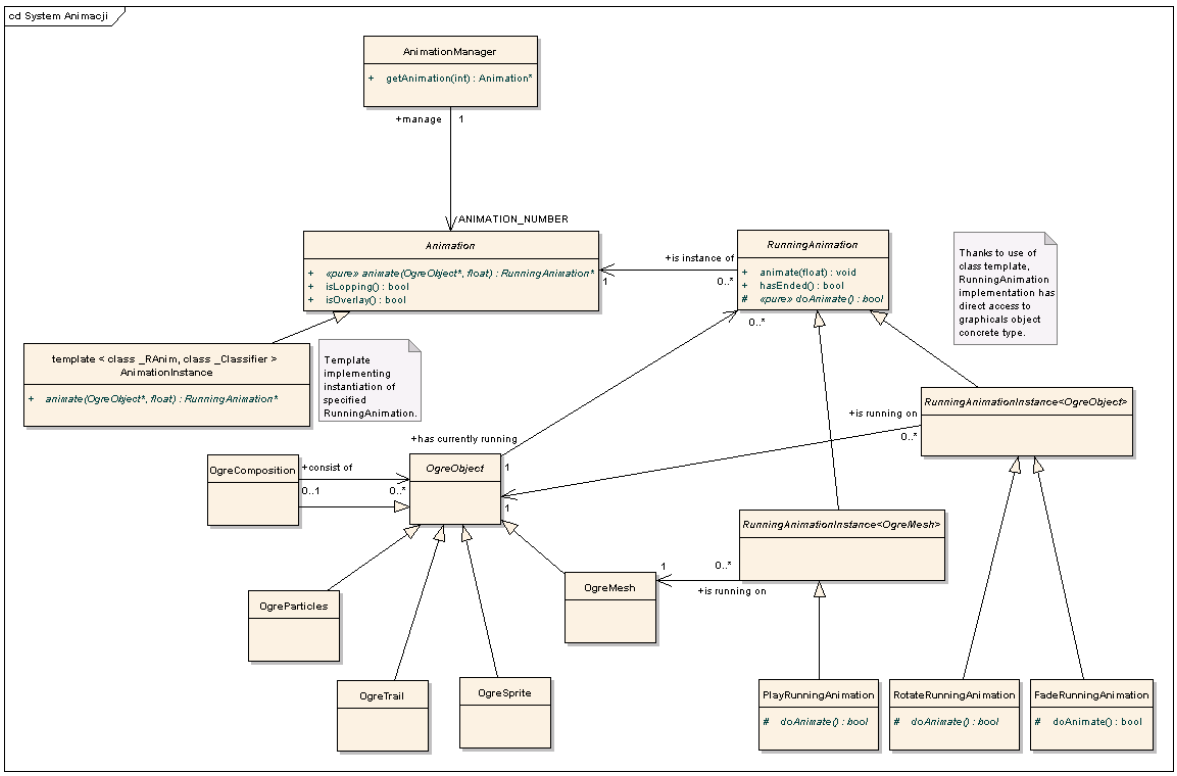
Wymagania systemu animacji

Projektując mój system chciałem więc by był on:

- Jak najbardziej czytelny i skalowalny.
- Chciałem by od strony mechaniki gry jego obsługa była wyjątkowo prosta i jednoznaczna.
- Jednocześnie obciążałem ten system dużą odpowiedzialnością odnoszącą się do graficznej strony mojej gry. Interfejs animacji miał umożliwić uzyskanie w jednolity sposób bardzo różnorodnych efektów graficznych. Od standardowej obsługi animacji modeli,

przez przezroczystość dowolnych obiektów graficznych aż do wyświetlania efektów cząsteczkowych w rodzaju eksplozji.

- Chciałem też, by na obiekcie mogły działać na raz różne animacje, a jednocześnie niektóre z nich powinny się wykluczać.



Rysunek 3.7: Diagram klas przedstawiający system animacji Maelstorm.

Realizacja

W Maelstorm Online animacje są identyfikowane za pomocą liczby całkowitej. Diagram na ilustracji 3.7 przedstawia podstawową hierarchię klas odpowiedzialną za implementację systemu animacji. Obiekty *Animation* są zgrupowane w tablicy animacji a identyfikator animacji jest w niej indeksem. Potrafią one tworzyć konkretną reprezentację animacji, działającej na obiekcie. Przechowują też ogólne dane dotyczące reprezentowanego typu animacji. Obiekty *RunningAnimation* natomiast są instancjami animacji na obiekcie graficznym. Potrafią na niego dowolnie wpływać, nawet tworząc inne obiekty graficzne. Dla przykładu, animacja efektu cząsteczkowego tworzy obiekt efektu w miejscu w którym znajduje się posiadający ją obiekt graficzny.

Korzyści jakie płyną z powyższej architektury:

- Dzięki obiektowej strukturze i zastosowaniu dziedziczenia można bardzo łatwo obsłużyć w specjalny sposób nietypowe animacje, wymagające specjalnego traktowania.

- By uprościć interfejs modułu animacji obiekty gry wpływają na animację informując swoją reprezentację graficzną o zdarzeniu animacyjnym. Może być ono obsłużone przez

odpowiadający mu obiekt graficzny. To zawęża interfejs animacji do wywołania jednej tylko funkcji.

- Animacje mogą wywoływać się na dowolnych obiektach graficznych. Dzięki zastosowaniu szablonów mogą też korzystać z ich specyficznej, nie wirtualnej funkcjonalności.
- Na obiekcie może na raz działać wiele animacji. Jednocześnie animacja może posiadać referencję do zbioru animacji wykluczanych, które będą wówczas zdejmowane wraz z jej aplikacją do obiektu.

3.6. Pathfinding - wyszukiwanie ścieżek

Wyszukiwanie ścieżek to istotny mechanizm, z którym muszą uporać się programiści większości gier. W Maelstorm Online światem gry jest otwarta przestrzeń kosmiczna, w której brak większych przeszkód "terenowych", a wzajemne omijanie się obiektów oparte jest na prostych mechanizmach, o których napiszę w dalszej części pracy. W ten sposób ominęła mnie konieczność implementacji modułu wyszukiwania ścieżek, będąca często bardzo pracochłonna i kosztowna.

Jednak w projekcie Wiedźmin [TW07] osobiście odpowiedzialny byłem szczególnie za system wyszukiwania ścieżek. W tym rozdziale chciałem więc podzielić się wiedzą i przemyśleniami zdobytymi w trakcie pracy zawodowej oraz w toku własnych poszukiwań dotyczących całościowego rozwiązania omawianego problemu.

3.6.1. Definicja problemu

Stajemy przed problemem wyszukania ścieżki dla danego obiektu gry, pomiędzy dwoma punktami w przestrzeni, lub stwierdzenia, że ścieżka jest niemożliwa do uzyskania. Trzeba tu jednak pamiętać, że światy gry różnią się pomiędzy sobą, w dodatku cechuje je zwykle dynamika. W dodatku zupełnie innym problemem w praktyce może okazać się ominięcie stojącego nam na drodze stołu, a innym, przeprowadzenie postaci pomiędzy dwoma punktami na przeciwległych krańcach wirtualnej lokacji. Inaczej ujmując system wyszukiwania ścieżek może składać się z kilku różnych mechanizmów, skutecznych w różnych sytuacjach, przy różnych długościach i stopniach komplikacji ścieżki wynikowej.

Często z uwagi na złożoność świata gry system wyszukiwania ścieżek musi być mocno zintegrowany z systemem sztucznej inteligencji i konfigurowalny za pomocą narzędzi i skryptów gry. O ewentualnej potrzebie konfiguracji systemu napiszę w trakcie rozważań nad aspektami oceny rozwiązań, tu już jednak chciałbym zaznaczyć, że wyszukiwanie ścieżek nie jest wyłącznie problemem algorytmicznym, ale często związane jest z dodatkową organizacją procesu pracy nad lokacjami świata gry i implementacją fabuły.

Aspekty oceny rozwiązania - realizm

Podstawowym kryterium, w jakim system wyszukiwania ścieżek oceniać będzie końcowy użytkownik jest intuicyjność znalezionych przez niego rozwiązań. Gdy gracz gdzieś "*kliknie*", to liczy, że postać pójdzie do tego miejsca w ten sam sposób jak on sam by to zrobił. Będzie to od programisty wymagało wzięcia pod uwagę wielu różnych czynników. Po pierwsze ścieżka musi być intuicyjnie wygładzona i pozbawiona nagłych, nienaturalnych zwrotów. O wygładzaniu ścieżek napiszę poniżej kilka słów. Kolejnym istotnym elementem będzie na pewno ukształtowania terenu, czy nawierzchni. W Wiedźminie na przykład problemem, który wyszedł w trakcie prac, była woda. W silniku była ona wyłącznie efektem graficznym, nie mającym wpływu na mechanikę gry. Brodzenie po wodzie przez korowody postaci wyglądało jednak wyraźnie nienaturalnie. Trzeba było to uwzględnić i postacie, na ile to było możliwe, chodziły po stałym gruncie, wybierając wodną lokację tylko, gdy mogło to wyraźnie skrócić drogę. Czynników takich w każdej produkcji może znaleźć się bardzo dużo. Mogą to być na przykład drogi, których postacie powinny się trzymać, mimo że mogą one nie mieć konkretnego wpływu na poruszanie się postaci. Mogą to też być przeszkody, których pokonanie nie tylko zabiera dodatkowy czas, ale intuicyjnie dla gracza związane jest z wysiłkiem. Tu dla przykładu również kłania się Wiedźmin. W grze można się było wspinać na niewielkie wzniesienia. Animacja wspinania była szybka, jednak zbyt częste wspinanie się czy zeskakiwanie wygląd-

dało po prostu nierealistycznie. Dlatego trzeba ich było unikać bardziej, niż wynikałoby to z samego spowolnienia marszu.

Ważne może być też wiele dodatkowych mechanizmów. Jako przykład podam efekt poruszania się postaci prawą stroną drogi czy chodnika. Zastosowanie takiego rozwiązania w miastach nie tylko wpływa na wrażenie realizmu świata gry, ale również zmniejsza częstotliwość omijania i kolizji postaci, co w perspektywie może mieć wpływ na poprawę wydajności gry.

Aspekty oceny rozwiązania - konfiguracja

Zastanówmy się teraz nad dynamiką świata gry. Stwarza ona problemy, które w większości powinny zostać przewidziane i rozwiązane jeszcze w trakcie projektowania systemu wyszukiwania ścieżek. Na przykład przy wyszukiwaniu ścieżek na większy od kilku kroków dystans, nie powinniśmy zupełnie brać pod uwagę innych postaci - gdyż gdy znajdziemy się w ich sąsiedztwie one z dużym prawdopodobieństwem będą w zupełnie innym położeniu. Skoro, nie bierzemy dynamicznie poruszających się postaci pod uwagę, to pojawia się kolejny nietrywialny problem do rozwiązania, a mianowicie stworzenie mechanizmu omijania. To pewien dodatkowy element, istotny szczególnie, gdy przestrzeń gry jest gęsto wypełniona postaciami. Ale to nie jedyny problem związany z dynamiczną naturą świata gry.

By opisać bardzo istotny w wielu produkcjach aspekt problemu, przyjrzyjmy się abstrakcyjnej, hipotetycznej sytuacji. Wyobraźmy sobie, że jesteśmy w budynku, na parterze. Chcemy dostać się do pokoju na piętrze. Jednak klatka schodowa jest zamknięta, i jedynym sposobem na dostanie się na kolejne piętro jest winda. A może jest właśnie na odwrót... Rozwiązanie problemu wydaje się proste - nasz system wyszukiwania ścieżek powinien wiedzieć jak działa winda i dokąd może nas przenieść oraz brać pod uwagę fakt, że drzwi mogą być zamknięte. A co jeśli winda jest zepsuta? Oczywiście system i z tym musi sobie poradzić. Elementów podobnych w działaniu do naszej windy może być w grze bardzo dużo i wiele z nich może być zaimplementowanych przez projektantów poziomów na podstawie dużo bardziej ogólnych mechanizmów. Działanie takich obiektów może opierać się na skryptach, a wówczas ich efekt ciężko przewidzieć w kodzie gry. Może ono także powodować trójwymiarowy ruch obiektów, którego konsekwencje też mogą być praktycznie niemożliwe do przewidzenia. W praktyce zdarza się, że mając w kodzie sam obiekt windy, nie tylko nie będzie trywialne określenie czy ta winda działa, ale także zorientowanie się jak ona działa, gdzie może nas przenieść i czy w ogóle powinna być brana pod uwagę w systemie wyszukiwania ścieżek. Praktyczny przykład powyższej sytuacji można znaleźć w silniku *Unreal Engine*[12]. Obiekty interaktywne są w nim reprezentowane jako ruchoma geometria, która pod wpływem zdarzeń gry porusza się w dowolny sposób w trójwymiarowym środowisku. Ich ruch może być spowodowany akcją dowolnego, wywoływanego w grze skryptu. Nie dość, że w takim systemie ciężko jest stwierdzić co właściwie robi obiekt windy (trzeba wpaść na to, że należy wejść do jej środka), to jeszcze właściwie nierozwiązywalnym problemem jest analiza współzależności obiektów i skryptów gry.

Ważnym aspektem oceny systemu wyszukiwania ścieżek jest możliwość jego konfiguracji za pośrednictwem narzędzi. W gruncie rzeczy, jest to podstawowe rozwiązanie postawionego wyżej problemu. Po pierwsze może nam to umożliwić specjalne traktowanie obiektów takich jak wspomniana przez nas winda i zamknięte drzwi. Dobrze napisany system wyszukiwania ścieżek powinien umożliwić projektantowi poziomu wpływanie na swoje działanie z uwzględ-

[12]Unreal Engine — silnik gry, produkcji *Epic Games*. Kolejne generacje *Unreal Engie* powstają na bieżąco i stanowią awangardę nowych technologii w dziedzinie komputerowej rozrywki. Aktualnie jest dostępna na rynku trzecia generacja silnika - *Unreal Engine 3*.

nieniem sytuacji fabularnych oraz mechaniką funkcjonowania obiektów etapu - oczywiście jeżeli takie możliwości oferuje nasza gra.

Jest jeszcze jeden problem, który wynika z braku możliwości konfiguracji systemu. System jako taki będzie musiał działać w całej grze, a do programisty nadchodzić będą błędy dotyczące specyficznych miejsc świata gry i sytuacji. Poprawienie tych błędów spowoduje zmiany globalne, dlatego możliwe jest pojawienie się nowych problemów. Błędy zresztą mogą nie leżeć po stronie programisty. Możliwe, że błędne są na przykład dane geometryczne lokacji gry, ale jeżeli projektanci i testerzy nie dostaną narzędzi pomagających im konfigurować i obserwować działanie systemu ścieżek - błędy za każdym razem będą przechodzić przez programistę.

Nie dotyczy to jednak wszystkich produkcji i specyfika systemu wyszukiwania ścieżek oraz użyte rozwiązania powinny być dopasowane do świata gry oraz mechaniki rozgrywki. Konfiguracja systemu ścieżek jest potrzebna w grach, w których poruszanie postaci jest kluczowym elementem, i w których świat gry jest ściśle zależny od projektantów lokacji gry, a nie jest tak prosto konfigurowalny przez gracza. Szczególnie w grach, w których poruszamy się po otwartych przestrzeniach i mamy dużą swobodą ruchu, konfiguracja systemu wyszukiwania ścieżek jest dyskusyjna. Nakładanie takiego obowiązku na projektantów i implementacja narzędzi do tego celu wydają się wówczas zupełnie niepotrzebne.

3.6.2. Wydajność

Wyszukiwanie ścieżek potencjalnie może być wąskim gardłem wydajności aplikacji gry. Jeżeli dużo obiektów porusza się po lokacji gry jednocześnie, każdy co jakiś czas będzie wymagał uwagi systemu wyszukiwania ścieżek. Dodatkowo, tej uwagi obiekty mogą często wymagać właśnie jednocześnie - gdy jakaś grupa obiektów jest w podobnej sytuacji i na jej podstawie decyduje się podjąć akcję związaną z poruszaniem. Ważne jest więc, by zapewnić mechanizmy równoważące obciążenie obliczeniowe wywołane działaniem systemu wyszukiwania ścieżek. Dwa podstawowe rozwiązania, z którymi miałem w tej kwestii styczność to:

1. **Wielowątkowość** — system wyszukiwania ścieżek działa we własnym wątku. Dostaje od obiektów gry zlecenia na wyszukiwanie ścieżek i realizuje je niezależnie od głównej pętli gry. Poza prostotą, dodatkową zaletą tego rozwiązania, szczególnie teraz, gdy standardem stają się maszyny wielordzeniowe, jest możliwość wykorzystania dodatkowych jednostek obliczeniowych. Na niekorzyść tego rozwiązania przemawia zaś to, że bez implementacji dodatkowych mechanizmów, to system operacyjny będzie decydował, ile czasu nasza aplikacja poświęci na wyszukiwanie ścieżek.

2. **Ograniczony czas ruchu** — to rozwiązanie było użyte w Wiedźminie. Obiekt gry dostaje ograniczoną jednostkę czasu na wykonanie ruchu. Jeżeli wyszukiwanie ścieżek nie zmieści się w nim, stan wyszukiwania jest zapamiętywany i obiekt rozpocznie wyszukiwanie od tego momentu w następnej ramce. Minusem tego rozwiązania jest fakt, że jest ono szczególnie zależne od prędkości komputera i na słabszym sprzęcie przydzielany czas może pozwalać na mniejszą liczbę kroków algorytmu, co przy mniejszej liczbie klatek powoduje czasami olbrzymi wzrost długości czasu wyszukiwania ścieżek.

3.6.3. A*

Podstawą większości systemów wyszukiwania ścieżek jest algorytm A*, sprowadzający się do wyszukiwanie wszerz algorytmem Dijkstry, gdzie wierzchołkami grafu wyszukiwania są punkty, bądź obszary przestrzeni gry. Algorytm działa często na kilku różnych poziomach w zależności od tego jakie odległości rozpatrujemy. O metodzie reprezentacji, tworzenia i

konfiguracji grafu wyszukiwania na większe odległości napiszę w następnych podrozdziałach. Tu natomiast chciałbym wspomnieć jeszcze kilka słów na temat wyszukiwania ścieżki na stosunkowo małych odległościach, gdy musimy odnaleźć dokładną drogę, którą musi przebyć postać z uwzględnieniem wszystkich przeszkód terenowych.

Dla uproszczenia przyjmijmy, że wyszukujemy ścieżkę na płaszczyźnie dwuwymiarowej, do czego zresztą, zwykle problem jest sprowadzany. Jako węzły wyszukiwania możemy użyć stałego i równomiernego podziału płaszczyzny, na przykład siatki kwadratowej, lub nawigacyjnej (patrz rozdział 3.4.2). Innym rozwiązaniem jest utworzenie logicznej, leniwej[13] przestrzeni dla danego wyszukiwania (w której reprezentacja węzłów grafu wyszukiwania powstaje dopiero w momencie wzięcia ich pod uwagę przez algorytm przeszukiwania, jako ewentualne następniki). Najprościej jest wówczas przyjąć punkt początkowy i końcowy wyszukiwanej ścieżki jako punkty odniesienia dla tworzonej przestrzeni. Wierzchołki grafu wyszukiwania A* będą logicznie przyporządkowane punktom terenu znajdującym się na kratownicy o zadanej podziałce. Wierzchołki mogą mieć leniwą reprezentację, to znaczy mogą powstawać dopiero w momencie dojścia do nich algorytmu wyszukiwania. Pomiędzy wierzchołkami grafu wyszukiwania będzie istniała krawędź, jeśli będziemy w stanie przeprowadzić pomiędzy reprezentującymi je punktami obiekt dla którego wyszukujemy ścieżkę. Wielkość podziałki można dostosować do wymaganej szerokości drogi, odległości punktów krańcowych czy ukształtowania terenu. Jest to bardzo ważny wybór, ponieważ to od wielkości podziałki zależy rozmiar przestrzeni wyszukiwania. Stosując zbyt duży krok podziałki algorytm może nie trafić na właściwe rozwiązanie. Problem pojawia się, gdy świat gry jest zróżnicowany i mamy do czynienia zarówno z wąskimi przejściami, ciasnymi pomieszczeniami oraz rozległymi otwartymi przestrzeniami. Bardzo prostym, a skutecznym rozwiązaniem, które zastosowałem w Wiedźminie, była dynamiczna zmiana wielkości podziałki. Użyłem prostej heurystyki, którą oczywiście można, myślę, zastąpić celniejszą. Za każdym razem, gdy algorytm odniósł sukces zwiększałem dwukrotnie podziałkę. Gdy algorytm nie znalazł ścieżki podziałka zmniejszana była dwukrotnie, aż do ustalonego limitu, a szukanie było powtarzane. Ponieważ każdy krok zwiększał, bądź zmniejszał wykładniczo przestrzeń wyszukiwania czas wyszukiwania rozwiązania przy większej podziałce był pomijalny w stosunku do dokładnego wyszukiwania. Heurystyka nie była idealna, bo nie uwzględniała na przykład sytuacji, w których postać nie mogła dojść do celu gdyż podziałka była za mała (był limit liczby odwiedzonych przez przeszukiwanie węzłów), ale ogólnie dobrze sprawdzała się w praktyce. Dodatkowym problemem jaki czasami powodował bardzo duży krok podziałki były nienaturalne ścieżki. Przy dużej podziałce algorytm mógł pominąć bardziej naturalną drogę do celu. Póki nie dotyczyło to postaci gracza nie było to problemem, a używanie w miarę możliwości dużych podziałek znacznie przyśpieszało wyszukiwanie ścieżek.

3.6.4. Automatyczny podział płaszczyzny

Czas zająć się wyszukiwaniem ścieżki na większe odległości. Dla wielu projektów najlepszym rozwiązaniem, będzie automatyczny podział świata gry na rejony, pomiędzy którymi przeliczane są (leniwie bądź przy wczytywaniu) połączenia. Szukając drogi najpierw sprawdzamy przez jakie rejony kolejno przejdziemy, a dopiero później wyszukujemy konkretne ścieżki na przejście w ich obrębie. Takie podejście nadaje się do stosowania szczególnie w grach, w których świat gry jest bardzo dynamiczny (zmienia się w trakcie działania gry) lub geometria świata, bądź model poruszania są proste. Plusem tego rozwiązania jest to, że nie wymaga

[13]Leniwe obliczanie to technika programistyczna odwlekania obliczeń do czasu, aż ich wynik będzie potrzebny. Do korzyści wynikających z leniwego obliczania należy zaliczyć: zwiększenie wydajności, dzięki uniknięciu niepotrzebnych obliczeń i możliwość tworzenia nieskończonych struktur danych.

żadnej pracy ze strony innych osób poza programistą. Utrudnia ono jednak możliwości zewnętrznej konfiguracji systemu oraz jego silną integrację z systemem sztucznej inteligencji. W praktyce jest to dobre rozwiązanie w projektach, w których z założenia na systemie wyszukiwania ścieżek nie spoczywa większa odpowiedzialność. Jeżeli nie nakładamy ograniczeń na geometrie świata, rozwiązanie to może być bardzo problematyczne. Można sobie wyobrazić wiele sytuacji, których automatyczny podział świata nie będzie mógł obsłużyć bez dodatkowych, skomplikowanych mechanizmów. Dużo trudniej też jest zachować realizm i intuicyjność ścieżek. Trzeba też pamiętać, że podział płaszczyzny ma kluczowe znaczenie dla wydajności wyszukiwania ścieżek na niższym poziomie - a przy przypadkowym podziale nie mamy na to wpływu.

W silniku gry Neverwinter Nights[14] użyte było właśnie to podejście. Wyszukiwanie ścieżek odbywało się na trzech poziomach. Najpierw pomiędzy "lokacjami" (ang. *Area*) gry, odpowiadającymi jednemu spójnemu etapowi. Dla gracza przejście pomiędzy "lokacjami" wymagało przeładowania, stanowiły więc one duże obiekty. Na drugim poziomie wyszukiwanie przechodziło przez "kafelki" (ang. *Tile*), będące kwadratowymi obszarami o zadanej wielkości, na które podzielona była lokacja. Następnie na najniższym poziomie ścieżka liczona była już pomiędzy konkretnymi "kafelkami".

3.6.5. Pokoje

Zamiast automatycznego podziału płaszczyzny możemy zastosować podział ręczny na "pokoje". Pociąga to za sobą konieczność stworzenia narzędzia do ich ustawiania oraz ręcznego ich wyznaczenia w lokacjach gry. Znając mechanizm działania systemu wyszukiwania ścieżek uzyskujemy wpływ na poprawny wybór przez postacie wyszukiwanej drogi. W wyniku tego rozwiązania możemy uzyskać znaczące przyśpieszenie wyszukiwania ścieżek[15] a ścieżki wyliczone będą dużo logiczniejsze i lepiej dopasowane do geometrii i specyfiki lokacji. Z tym podejściem są związane jednak pewne problemy. Po pierwsze wymaga one nakładu pracy proporcjonalnie dużego, w porównaniu do pozostałych, prezentowanych przeze mnie rozwiązań. Po drugie jest nie intuicyjne i wyrobienie zasad podziału na pokoje wymaga nauki. Po trzecie - zmiana w systemie wyszukiwania ścieżek może wymusić zmianę ustawienia pokoi lokacji w grze... Wszystkich.

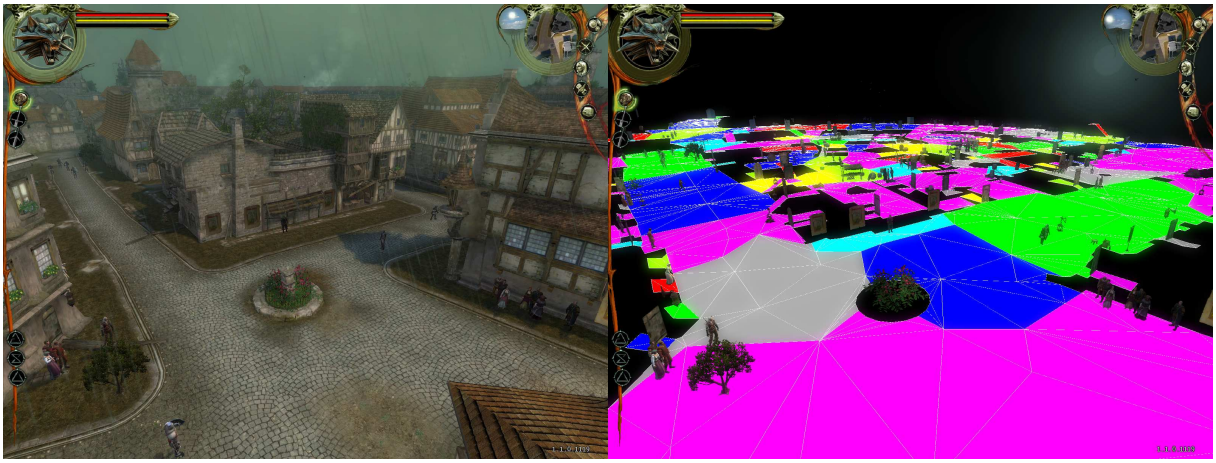
To rozwiązanie zostało użyte w Wiedźminie. Niestety przyniosło sporo problemów. Podstawowe z nich leżą nie po stronie samego algorytmicznego rozwiązania, lecz procesu pracy nad lokacją gry. System wyszukiwania ścieżek jest konfigurowany poprzez podział przestrzeni poruszania na pokoje. Jest on robiony w programie 3D Studio Max przez grafika. Na tym etapie prac mamy lokację gry, ale nie jesteśmy w stanie powiedzieć jak rozmieszczone będą w niej obiekty i przeszkody. By poradzić sobie ze statycznymi przeszkodami umieszczanymi w lokacji, trzeba było napisać skomplikowany system sprawdzający spójność pokoi oraz połączeń pomiędzy nimi, a w szczególnych przypadkach, gdy całe lokacje gry pokryte były barykadami, specjalnie ustawiać "na oko" krawędzie pokojów by dopasować je do przeszkód. Projektanci implementujący fabułę nie mieli dostępu do konfiguracji systemu ścieżek, w praktyce więc nie rozumieli go, nie znali jego słabości i nie wykorzystywali jego możliwości. To nie jedyne

[14] Jedna z pierwszych komputerowych gier RPG (roleplaying game) prezentowana z perspektywy trzeciej osoby, w pełni trójwymiarowym świecie. Napisana przez Bioware Interactive i wydana w połowie 2002 roku przez Infogames (dzisiaj występujące jako Atari). Silnik Aurora, na którym była oparta, był podstawą do stworzenia polskiego "Wiedźmina".

[15] Wiemy, pomiędzy którymi punktami będzie wyszukiwana ścieżka. Możemy zapewnić, że w większości przypadków wierzchołki, pomiędzy którymi ścieżka przez dane pokoje będzie wyszukiwana będą połączone prostą linią. To z kolei może kończyć pracę algorytmu w pierwszym kroku, bez potrzeby odpalania algorytmu A*.

problemy jakie sprawiał ten podział. W praktyce w Wiedźminie system wyszukiwania działa myślę przyzwoicie, jednak zostało to osiągnięte zupełnie nieproporcjonalnym nakładem sił. Poniżej omawiam rozwiązanie opierające się na zastosowaniu punktów nawigacyjnych. Choć algorytmicznie sprowadza się do omawianego, to z punktu widzenia procesu pracy nad grą jest myślę dużo lepsze.

Przykład użycia podziału na pokoje oraz siatki nawigacyjnej można zaobserwować na ilustracji 3.8.



Rysunek 3.8: Świat Wiedźmina i jego siatka nawigacyjna z zaznaczonym kolorem podziałem na pokoje

3.6.6. Graf widoczności

Najskuteczniejszym rozwiązaniem, o którym słyszałem, automatycznie tworzącym reprezentację świata w systemie wyszukiwania ścieżek, jest graf widoczności. Zastosowanie opartego o tą strukturę algorytmu zapewnia bardzo efektywne wyszukiwanie ścieżki zarówno na małe jak i duże odległości, choć wymaga dokonania pewnej prekalkulacji lokacji świata gry w której toczyć się będzie rozgrywka. Jest to doskonałe rozwiązanie dla skomplikowanej geometrii. Choć nie miałem z nim bezpośredniej styczności, sprawia ono wrażanie trudnego w implementacji ale nie wymaga dodatkowej pracy od reszty zespołu projektowego.

Spójną część świata gry możemy abstrakcyjnie reprezentować jako wielokąt dowolnego kształtu posiadający w sobie puste obszary, których brzegi również stanowią wielokąty o dowolnym kształcie. Wówczas węzłami grafu widoczności mogą być wierzchołki leżące na krawędzi wielokąta świata i pustych obszarów w jego wnętrzu. Dwa wierzchołki tego grafu są połączone, jeżeli istnieje pomiędzy nimi prosta linia nie przechodząca przez wnętrze pustego obszaru i nie przecinająca krawędzi świata. Mając utworzony taki graf jesteśmy w stanie w bardzo szybkim czasie znaleźć najkrótszą łamaną łączącą dwa dowolne punkty świata. Opierając się na wiedzy z wykładu *Geometria Obliczeniowa* prowadzonego na naszym wydziale przez doktora Mirosława Kowaluka, wiem że graf widzialności jesteśmy w stanie stworzyć w pesymistycznym czasie $O(n^2)$, gdzie n jest liczbą krawędzi świata. Jest to jednocześnie pesymistycznym rozmiar struktury. Wyszukiwanie w nim drogi pomiędzy dwoma punktami odbywa się w pesymistycznym czasie $O(nlog(n) + m)$, gdzie m to liczba krawędzi grafu widoczności. W praktyce przy odpowiednim dostosowaniu algorytmu do specyfiki projektu możemy otrzymać bardzo uproszczoną reprezentację przeszukiwanej przestrzeni.

Wierzchołki grafu widoczności można wyznaczyć automatycznie w inny sposób, umieszczając je wyłącznie przy brzegach spójnych obszarów świata. To minimalizuje przestrzeń wyszukiwania. Podstawowe wprowadzenie do zastosowania grafu widoczności w systemie wyszukiwania ścieżek znalazłem w rozdziale napisanym przez *Stevea Robina* w pierwszym tomie *Perełek Programowania Gier* [DLo1].

Z zastosowaniem tej metody wiążą się jednak pewne trudności.

- **Reprezentacja dynamicznych przeszkód** – czyli postaci (co zwykle i tak sprowadza się do odrębnego problemu) i ruchomej geometrii oraz obiektów statycznych. By reprezentować dynamiczne obiekty w tym systemie wyszukiwania możemy w trakcie działania tworzyć ich reprezentację w grafie wyszukiwania ścieżek i modyfikować graf w zależności od niej. Ewentualnie, jeżeli jesteśmy w stanie określić wszystkie możliwe stany tych obiektów (np. drzwi mogą być otwarte lub zamknięte), albo przynajmniej je sparametryzować, można użyć parametrycznej reprezentacji połączeń grafu, które będą aktywne w zależności od stanu dynamicznej przeszkody.

- **Rozmiar obiektów** – połączenia są testowana przez prowadzenie prostej linii pomiędzy punktami. Uwzględnienie rozmiaru obiektów w wyszukiwaniu jest przy tym podejściu nie trywialne.

- **Zależność od geometrii** – czasami, jeżeli nie zastosujemy dodatkowych, heurystycznych optymalizacji i uproszczeń geometrii nawigacyjnej, rozmiar grafu może zupełnie niepotrzebnie urosnąć. Wyobraźmy sobie kwadratowy pokój, na środku którego jest bardzo dokładnie wymodelowana okrągła kolumna - w rzeczywistości będąca n-kątem foremnym, dla wysokiego n. Liczba wypukłych obszarów w tym świecie będzie równa liczbie wierzchołków kolumny. Wraz z ich liczbą będzie rosła wielkość grafu widoczności.

Rozszerzona geometria

W grafie widoczności połączenia mamy wyznaczone w formie linii prostych, co nie uwzględnia kształtu i rozmiaru obiektów. Przybliżmy kształt obszaru kolizji obiektu pewną figurą geometryczną. Nazwijmy rozszerzoną geometrią sumę Minkowskiego[16] geometrii świata i kształtu przybliżającego. Graf widoczności rozszerzonej geometrii świata reprezentuje możliwość przemieszczania kształtu przybliżającego. Nie bierzemy tu pod uwagę obrotów, należy więc przybliżyć obiekt gry możliwie prostą, symetryczną figurą. Najczęściej możemy sobie pozwolić by obiekty gry reprezentować w dwuwymiarowej przestrzeni wyszukiwania ścieżek jako koło. Jako nasz kształt budujący rozszerzoną geometrię najlepiej nadawać się będzie opisany na okręgu kolizji kwadrat, ewentualnie ośmiokąt foremny, ale liczba wierzchołków grafu widoczności jest liniowo zależna od liczby krawędzi kształtu przybliżającego. Stosując rozszerzoną geometrię dostajemy graf widoczności, który dobrze opisuje przestrzeń wyszukiwania obiektu o zadanym rozmiarze. Możemy zastosować parametryczną reprezentację rozszerzonej geometrii by wygenerować przestrzeń wyszukiwania ścieżek dla różnego rozmiaru obiektów.

[16]Definicja Sumy Minkowskiego. Rozpatrzmy wielokąty A i B jako zbiory wektorów o współrzędnych odpowiadających współrzędnym punktów należących do tych wielokątów. Wtedy Sumę Minkowskiego możemy określić następująco.

$$A + B = \{x + y : x \in A \wedge y \in B\}$$

Uproszczony graf wyszukiwania

Metoda tworzenia grafu wyszukiwania, opisana na początku może zostać znacząco zoptymalizowana, pod kątem minimalizacji grafu. Interesuje nas graf punktów przestrzeni taki, że z każdego miejsca w świecie widzimy co najmniej jeden z tych punktów a obszary punktów, które możemy przypisać do danego wierzchołka są spójne. Wbrew pozorom problem ten nie jest problemem galerii sztuki[17], choć prawdopodobnie z rozwiązań tego problemu można wyciągnąć wiele wniosków dotyczących optymalizacji grafu widoczności w grze komputerowej. Implementując graf widoczności istotą optymalizacji będzie minimalizacja wierzchołków oraz krawędzi grafu. Trzeba mieć na względzie, że praktyczne zastosowanie tego algorytmu będzie wiązało się z koniecznością wprowadzenia wielu mechanizmów optymalizacyjnych.

W rozważaniach na temat grafu wyszukiwania opierałem się na rozdziałach autorstwa *Thomasa Younga Perełek Programowania Gier* [DLo2]. Znalazłem tam szczegółowo przedstawione metody optymalizacji grafu widoczności oraz opis algorytmów związanych z zastosowaniem rozszerzonej geometrii i tworzeniu sumy Minkowskiego.

3.6.7. Punkty nawigacyjne

Punkty nawigacyjne możemy uznać za równoważne grafowi widoczności, w którym ręcznie wstawiamy wierzchołki grafu. Są one jednocześnie bardzo zbliżone do rozwiązania opartego o "pokoje". W lokacji gry zostają ręcznie rozstawione punkty nawigacyjne. Mogą one być obiektami gry, reagującymi na zdarzenia i kontrolowanymi przez ogólne mechanizmy gry oraz na przykład polecenia języka skryptowego obsługiwanego przez grę. Punkty nawigacyjne tworzą graf, po którym wyszukiwane są ścieżki. Pomiędzy dwoma wierzchołkami grafu istnieje połączenie, jeżeli pomiędzy reprezentującymi je punktami nawigacyjnymi istnieje przejście w linii prostej. W praktyce wstawiając punkty nawigacyjne w edytorze możemy zobaczyć jak wpływają one na system wyszukiwania ścieżek. Każdy z tych punktów możemy opisać zbiorem parametrów pozwalającym nam kontrolować ich działanie.

- Kto może, a kto nie używać tego punktu w wyszukiwaniu ścieżek.
- Możemy prosto obserwować lub "ręcznie" modyfikować wartościowanie węzłów grafu wyszukiwania ścieżek.
- Mamy pełną kontrolę nad "aktywnością" punktu nawigacyjnego w systemie wyszukiwania ścieżek.
- Silna integracja z systemem AI. Możemy zdefiniować wartość strategiczną danego punktu. Może być ona również heurystycznie wyliczona. Punkty nawigacyjne mogą określać miejsca, w których warto wypatrywać przeciwnika, przygotowywać zasadzkę, kryć się czy do których nie można dopuścić innych postaci.
- Punkt nawigacyjny może być powiązany z obiektami gry których stan może decydować o jego aktywności.
- Możemy z łatwością tworzyć różne, specjalne rodzaje przejść jak np. drogi o określonej szerokości (po których prowadzi ścieżka, nawet jeśli wygładzanie pozwoliłoby ją skrócić), drogi jednostronne (pisałem już wyżej o korzyściach w utrzymaniu prawostronnego ruchu na drodze), przejścia wymagające użycia obiektów otoczenia (winda).

[17]Galeria sztuki to znany NP-trudny problem geometrii obliczeniowej. Celem jest znalezienie minimalnego zbioru obserwatorów potrzebnych by pokryć "wzrokiem" całą powierzchnię wirtualnej galerii sztuki.

W praktyce rozwiązanie to, przy dobrej implementacji, sprowadza się w działaniu do systemu pokoi. Potrzebuje większego wsparcia od strony narzędzi i paru dodatkowych elementów w implementacji. Jest za to dużo wszechstronniejsze, skalowalne, bardziej intuicyjne, prostsze w edycji i wymagające od reszty zespołu mniejszego nakładu pracy. Jednocześnie jest to wykorzystanie grafu pokoi stworzonego przez projektantów. Być może jest to wspaniała szansa dla stworzenia hybrydowego systemu wyszukiwania ścieżek. Możemy korzystać z algorytmów automatycznych, gdy nie mamy opisanej przestrzeni lub tworzyć automatycznie konfiguracje systemu, która następnie mogłaby być ręcznie modyfikowana.

3.6.8. Dodatkowe elementy systemu wyszukiwania ścieżek

Chciałem jeszcze wspomnieć o dwóch elementach systemu wyszukiwania ścieżek. Nie mają one większego wpływu na architekturę systemu jako całości, ale są podstawowymi mechanizmami, których implementacja ma bardzo duży wpływ na grę.

Wygładzanie

Wygładzanie ścieżek jest bardzo istotną kwestią. Dobrze napisane ściąga wiele odpowiedzialności z podstawowego systemu wyszukiwania. W wyniku wyszukiwania A* dostajemy zwykle listę punktów, które łączą się w ścieżkę pełną kątów prostych, nagłych zwrotów, często małych nielogiczności. Wygładzanie działa na tych punktach zapewniając dwie podstawowe rzeczy. Na ile to możliwe zmniejsza liczbę węzłów, łącząc te, pomiędzy którymi możemy przeprowadzić prostą. Drugim zadaniem wygładzania jest dopilnowanie by w miarę możliwości na ścieżce nie było ostrych, nienaturalnych zwrotów - to z kolei wymaga zwykle dodania dodatkowych wierzchołków.

Implementacja wygładzania w Wiedźminie wygląda następująco. Najpierw na ścieżce wstawiane są w równych odstępach dodatkowe wierzchołki - by umożliwić skrócenie ścieżki. Następnie algorytm przechodzi po wierzchołkach ścieżki. Dla badanego wierzchołka wybiera następnik, ułożony możliwie najdalej na ścieżce, a do którego można dojść z niego w prostej linii. Wszystkie wierzchołki pomiędzy badanym a jego następnikiem są odrzucane, a algorytm wykonuje się ponownie dla następnika, aż dotrze do końca ścieżki. Potem, jest dokonywane wygładzanie ostrych krawędzi poprzez dodawanie dodatkowych wierzchołków oraz przesuwanie istniejących. Nie jestem autorem tego mechanizmu. Osobiście do systemu wygładzania dodałem tylko dodatkowy mechanizm łączący dwie gładkie ścieżki. Potrzebny był on, gdy dołączano do aktualnie wyliczonej ścieżki, kolejny jej fragment, przechodzący przez następny pokój.

Koszt czasowy wygładzania jest właściwie pomijalny w stosunku do czasu odnalezienia ścieżki. Efekt jest jednak bardzo dobry. Ścieżki są naturalnie proste, pozbawione ostrych zwrotów, a jednocześnie optymalnie blisko przechodzą gładkim łukiem przy przeszkodach. Efekt użycia wyszukiwania i jego wpływ na wyliczone ścieżki przedstawiam na ilustracji 3.9.

Omijanie obiektów

Ponieważ środowisko gry jest dynamiczne, jest duża szansa, że wyliczona przez nas ścieżka zostanie zablokowana przez inny poruszający się obiekt. Rozwiązanie tego problemu jest bardzo istotne i często nie trywialne. Dla gracza bardzo irytujące może być, gdy nie będzie mógł przedostać się przez tłum szarych przechodniów, przypadkowa postać zablokuje mu wąskie przejście lub zobaczy dwóch nie mogących się wyminąć przechodniów.

Rysunek 3.9: Wpływ wygładzania na ścieżkę

Powyższy problem można rozwiązać na wiele różnych sposobów i w praktyce, najlepsze jest właśnie równoległe wykorzystanie wielu z nich, co nie powoduje u gracza wrażenia powtarzalności i pozwala obsłużyć bardzo ciężkie do rozwiązania sytuacje.

- **A*** — Jednym z podstawowych rozwiązań jest wyszukanie ścieżki obchodzącej przeszkodę. Wynajdujemy bezpieczną pozycję na naszej drodze, w której nie kolidujemy z żadną dynamiczną przeszkodą, a następnie włączamy wyszukiwanie A*. Zależnie od tego jak ustawimy wyszukiwanie A*, metoda ta może wiązać się z pewnymi problemami. Po pierwsze wyszukiwanie potencjalnie może być dosyć drogie, a jednocześnie powinno być liczone w jednej ramce gry. Z drugiej strony, jeśli ograniczymy je do jednej ramki i przeznaczymy na nie ograniczony czas, bądź liczbę wierzchołków, jego zasięg i użyteczność mogę ucierpieć. Kolejnym potencjalnym problemem jest wielkość podziałki. Przy mniejszej podziałce nie ma problemu z zauważaniem węższej drogi, natomiast większa podziałka znacząco zwiększa zasięg przeszukiwania.

- **Przewidywanie kolizji** — Na kolizje powinniśmy reagować zanim się pojawią. Istnieją bardziej skomplikowane algorytmy, ale w Wiedźminie zastosowaliśmy prosty i dosyć skuteczny mechanizm. Idąca postać testowała, czy na jej ścieżce, bezpośrednio przed nią, nie znajdują się inne postacie. Jeśli tak, włączała wyszukiwanie A* by ominąć je. Wyszukiwanie było ograniczone czasem, ale było powtarzane w kolejnych ramkach. Oczywiście takie podejście nie załatwi wielu sytuacji, gdy ktoś na przykład nagle wbiegnie nam na drogę, ale w praktyce sprawdzało się dostatecznie dobrze i w połączeniu z innymi mechanizmami omijania dawało komfort poruszania.

- **Przewidywanie ruchu** — Problemem obu powyższych rozwiązań jest fakt, że działały

dla statycznego środowiska. Wyszukując ścieżkę, która okrążała poruszającą się postać bardzo często mogło się zdarzyć, że już po chwili znowu z nią kolidowaliśmy. Ważną rzeczą jest więc by zdawać sobie sprawę z ruchu obiektów przy liczeniu drogi je omijającej. Istnieją zaawansowane techniki uwzględniające przy liczeniu ścieżki omijającej wartość czasu i pozycję obiektów w danej jednostce czasu, jednak nie miałem z nimi większej styczności i nie byłem w stanie zgromadzić wiarygodnych materiałów na ich temat. Bardzo prostym rozwiązaniem, a dającym wymiernie dobre rezultaty, jest wyłączenie z wyszukiwania A* krótkiego wycinka ścieżki omijanych obiektów. Jego długość może być zdefiniowana odgórnie, zależna od odległości od gracza, bądź prędkości poruszania się obiektu. Rezultat jest zauważalny i przyzwoity, a koszt działania tej metody w grze oraz czas jej implementacji są nieznaczne.

- **Algorytmy stada** — Dobrym podejściem jest wykorzystanie do omijania rozwiązań podobnych jak te użyte w algorytmie stada.[18] Gdy obiekty poruszają się dostatecznie blisko siebie, na ich ruch zaczyna oddziaływać siła odpychająca je od innych, zależna od ich bliskości. W efekcie w bardzo naturalny i intuicyjny sposób grupy postaci mogą niejako wymijać się wzajemnie nawet w rozległym tłumie.

- **Integracja z grą i AI** — Pamiętajmy, że jeżeli odpowiednio wizualnie i mechanicznie to zamodelujemy, może spowodować, że postacie będą się wzajemnie przepuszczać, przepychać czy czekać parę chwil aż droga będzie wolna. Możemy pokusić się o integrację omijania z systemem AI i tworzenie rozwiązań, w których postacie będą wzajemnie na siebie reagować. Jeżeli dobrze zamodelujemy takie rozwiązania, mogą one tchnąć dużo życia w świat gry, spowodować, że będzie wyglądał bardziej naturalnie. Tu znowu posłużę się przykładem Wiedźmina. Jednym z gorszych mechanizmów omijania, stosowanym w przypadkach, gdy nie mogliśmy wykorzystać innych, było przepychanie postaci. Postać, która stała na drodze innej, wyszukiwała sobie bezpieczne miejsce obok ścieżki i poruszała się do niego. Druga postać po paru chwilach kontynuowała ruch. Wyglądało to nienaturalnie, dlatego rozwiązanie to było wykorzystywane bardzo rzadko, gdy inne sposoby rozwiązania problemu zawiodły. Wystarczało jednak troszkę mocniej zintegrować ten mechanizm z grą. Został on delikatnie zmodyfikowany. Postać odpychająca miała włączaną animację ręki, którą przesuwała drugą osobę. Ta z kolei również wykonywała specjalną animację "bycia odepchniętą". Do tego dodałem małe smaczki takie jak prosto definiowany parametr "godności", świadczący o tym kto kogo i w jakich sytuacjach może w ten sposób odepchnąć. Implementacja tego była naprawdę szybka i prosta. Poprawiło to omijanie, ponieważ był to mocny mechanizm wyglądający na tyle ładnie, że starałem się rozsądnie często wykorzystywać ten efekt. W dodatku pozwoliło to na dodanie wielu prostych, a ładnych smaczków do gry, takich jak na przykład postacie o wysokim priorytecie, dla przykładu bogaci mieszczanie czy rycerstwo, nie wymijające pospólstwa i biedoty, tylko brutalnie rozpychający wchodzących im w drogę.

- **Oszustwa** — Gra to nie aplikacja bankowa i tu czasami programista może troszkę oszukać. Warto z tego korzystać z pewnym umiarem. Jeżeli gracze nie widzą bezpośrednio zdarzenia, omijanie możemy sprowadzić do czystej teleportacji. Warto o tym pamiętać, bo może nam to wyraźnie wpłynąć na wydajność całego mechanizmu. Kolejną okazją do oszustw są sytuacje, w których postacie, zgodnie z mechaniką gry po prostu nie są w stanie się wyminąć. Warto wtedy pomyśleć, czy czasami nie warto nagiąć wówczas

[18] Algorytm stada(ang. *flock*) - dokładne informacje o nim znajdują się w rodziale 3.7.

zasady gry i na przykład pozwolić postaciom na wzajemne przenikanie. Takie rozwiązania powinny być stosowane wyłącznie w ostateczności, gdy wszystkie inne zawiodą. Jeżeli nie będziemy mogli poświęcić większego czasu na stworzenie rozwiązania takiej sytuacji, opartego o system sztucznej inteligencji albo specjalne zachowanie postaci, wówczas warto rozważyć wprowadzenie małego oszustwa.

- **Niedeterminizm** — Dobrą praktyką przy wszystkich powyższych podejściach jest zastosowanie pewnego niedeterminizmu. Szczególnie tyczy się to rozwiązań opartych o zachowania, sztuczną inteligencję i animacje. Jeżeli gracz przyzwyczai się do tego, że pewne sytuacje są zawsze jednakowo rozwiązywane, może to prowadzić do łatwego spowszednienia tych mechanizmów i w konsekwencji osłabić efekt, który wywrą one na graczu. Jeżeli zasady doboru metody omijania będą bardzo płynne, intuicyjne ale ciężkie do pełnego wyśledzenia, bądź po prostu zależeć będą w ograniczony sposób od przypadku, może to wpłynąć pozytywnie na odbiór przez gracza zachowania się postaci w grze.

Rysunek 3.10: Omijanie. Po prawej: przykład użycia przewidywania kolizji (niebieskie linie). Po lewej: ciężka sytuacja rozwiązana przez interakcje obiektów (przepychanie).

3.6.9. Wnioski

Najlepsze wyszukiwanie ścieżek, to takie którego końcowy użytkownik nie zauważy. To takie też, które nie będzie zauważalne w kontekście rozważania wydajności aplikacji oraz które zafunduje najmniejszą ilość pracy zespołowi pracującemu nad grą. System musi być zaprojektowany w odniesieniu do gry, funkcjonalności jaką od niego oczekujemy oraz odpowiedzialności jaką na niego chcielibyśmy złożyć. Bardzo często moduł wyszukiwania ścieżek stanowi integralną część systemu sztucznej inteligencji. Już w trakcie projektowania należy przewidzieć jak wpłynie on na cykl pracy nad lokacjami i fabułą gry. Zaprezentowałem kilka znanych rozwiązań. Do problemu wyszukiwania ścieżek można podejść jednak jeszcze na wiele innych sposobów. Szczególnie delikatnie potraktowałem systemy automatycznego podziału przestrzeni, jako że w tej kwestii mam mniejsze doświadczenie oraz w ich przypadku, implementacja

modułu sprowadza się do rozwiązania algorytmicznego. Zestawiając ze sobą zaprezentowane rozwiązania, chciałem uczulić czytelnika na aspekty, które wymagają uwagi już na etapie projektowania systemu.

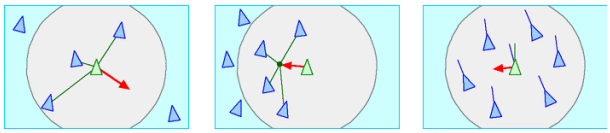
3.7. Algorytm stada i sztuczna inteligencja

Algorytm stada został zaprezentowany w 1987 roku przez Craiga Reynoldsa w referacie "Flocks, Herds, and Schools: A Distributed Behavior Model"[19]. Przedstawiał on model reprezentacji obiektów symulacji. W jego podejściu obiekty były autonomicznymi agentami, znających wyłącznie swoje dynamiczne, lokalne otoczenie. Agenci przestrzegając kilku podstawowych, intuicyjnych zasad wykazywali zbiorowe zachowania symulujące stada ptaków, owadów, czy ławicę ryb. Te zasady to:

- **Separacja** — zapobieganie zbytniej bliskości obiektów z naszego otoczenia.
- **Wyrównanie** — obiekt dopasowuje się do średniego kierunku i tempa poruszania obiektów lokalnej grupy.
- **Spójność** — obiekt jest sterowany w kierunku centrum lokalnej grupy.
- **Unikanie** — dodana później zasada. Agent stara się unikać przeszkód oraz innych, szczególnie wrogich, obiektów.

Agenci wchodzą w interakcję wyłącznie z obiektami znajdującymi się w ich bezpośrednim otoczeniu. Nie komunikują się, nie planują, nie przechowują dodatkowych informacji o stadzie czy środowisku w którym się znajdują. Ich działanie oparte jest wyłącznie o percepcję.

Do zasad na których oparł się Craig Reynolds można dodać oczywiście własne. Przypisując im priorytety w bardzo intuicyjny sposób możemy modelować zachowanie obiektów.



Rysunek 3.11: Ilustracje Craiga Reynoldsa dotyczące kolejno separacji, wyrównania i spójności.

3.7.1. Implementacja

Dla uproszczenia przyjmijmy, że nasz obiekt znajduje się w trójwymiarowej przestrzeni. W danej jednostce czasu jest skierowany w pewnym konkretnym kierunku oraz posiada skierowany w tą stronę aktualny wektor ruchu.

Algorytm w każdym kroku modyfikuje wektor ruchu, poprzez dodanie do niego wektora modyfikacji. Wektor modyfikacji tworzymy sumując wektory potrzeb, których długość jest mnożona proporcjonalnie do priorytetu potrzeby (specyficznego dla danego obiektu). Wektory potrzeb z kolei są implementacją stosowanych zasad stadnych (np. separacji, wyrównania i spójności).

Na wejście dla algorytmu będziemy musieli zdefiniować mechanizm widoczności. Obiekt musi widzieć znajdujące się wkoło obiekty gry. Powinien móc określić najbliższy z nich oraz uśrednione położenie widzianych przez niego członków stada.

Potrzebę separacji możemy zaimplementować sprawdzając odległość między rozpatrywanym obiektem i najbliższym, widzianym członkiem stada. Jeżeli odległość ta jest mniejsza od pewnej założonej, tworzymy wektor potrzeby separacji. Skierowany on będzie od widzianego

[19]Dostępny pod adresem *http://www.red3d.com/cwr/papers/1987/boids.html*

obiektu do obiektu rozpatrywanego i jego długość jest zależna od współczynnika bliskości obiektów (czym bliżej tym mocniejsze oddziaływanie odpychające). Jednocześnie możemy też przyciągać obiekt do najbliższego, jeżeli odległość jest większa od pewnej założonej, choć przynajmniej ja nie widziałem dla tego oddziaływania dobrego zastosowania.

Wektor potrzeby wyrównania możemy z kolei utworzyć, biorąc po prostu znormalizowany wektor ruchu najbliższego obiektu.

Wektor spójności może być reprezentowany jako znormalizowany wektor skierowany od rozpatrywanego obiektu, do środka stada.

Unikanie obiektów możemy zaimplementować na wzór fizycznej siły odpychania, kwadratowo przybierającej na sile wraz ze zmniejszaniem się odległości.

Wszystkie powyższe wektory mnożymy przez priorytety potrzeb jakie reprezentują. Następnie sumujemy je, a otrzymany wektor możemy poddać dodatkowym przekształceniom - na przykład ograniczyć jego maksymalną długość. Następnie wektor zostaje użyty w sterowaniu i wpływa na zmianę aktualnego wektora ruchu obiektu.

3.7.2. Wydajność

Podstawowa implementacja algorytmu wymaga by każdy członek stada wykonał co najmniej liniową liczbę czynności związanych z jego obsługą. Głównym problemem jest tu zapewnienie widoczności. Ponieważ często algorytm stada jest stosowany do implementacji prostych obiektów, będących raczej tłem gry i nie wpływających na podstawową mechanikę rozgrywki, szkoda czasem poświęcać mu zbyt wiele zasobów gry. Podstawową optymalizacją jest uproszczenie mechanizmów widoczności. Możemy stworzyć przedefiniowane stada, w których obiekty widzą się bez potrzeby wykonywania dodatkowych testów. Może to bardzo przyśpieszyć algorytm, ponieważ również testy widoczności z innymi obiektami stado może wykonywać jako całość. Niestety takie stada stracą swoją podstawową funkcjonalność, jaką jest autonomiczność agentów. Przestaną się one łączyć, dzielić, oddziaływać wzajemnie na siebie, a całość straci na wiarygodności. Kolejną możliwością jest uproszczenie mechanizmów widoczności. Podzielmy na przykład świat na kratki i powiedzmy, że wszystkie obiekty stadne w danej kratce się widzą. To również proste i skuteczne rozwiązanie, sprawiające jednak, że obiekty stadne czasami będą wobec siebie ślepe, co łatwo dostrzeże uważny obserwator. Gdy więc rozpoczynamy implementację algorytmu warto dostosować ją do zadań jakie mu powierzamy i ich priorytetu w projekcie.

3.7.3. Modyfikacje algorytmu

Na stronie twórcy algorytmu *Craiga Reynoldsa*[20] [CR] znajdziemy odnośniki do wielu przykładów jej wykorzystania i modyfikacji. Stosując omawianą metodologię i ustawiając odpowiednie zasady sterujące modelem można uzyskać bardzo różnorodne modele sterowania obiektami. Zastosowane w algorytmie rozwiązania są często tylko bazą do tworzenia nowych rozwiązań zupełnie różnych problemów. Przykładem może być tutaj zupełnie nieprzypadkowe odniesienie do algorytmu stada przy okazji omawiania mechanizmów omijania obiektów w sekcji 3.6.

3.7.4. Zastosowania

Algorytm znajduje szczególne zastosowanie w grach. W wielu produkcjach był używany do reprezentowania małych żyjątek wchodzących w skład środowiska. Owadów, ptaków ryb czy

[20] Aktualny adres: *http://www.red3d.com/cwr/*

innych prostych organizmów reprezentujących raczej tło wydarzeń. Może być jednak używany do reprezentacji bardziej skomplikowanych i istotnych elementów rozgrywki jak tłum czy oddziały postaci. W Wiedźminie algorytm stada został użyty do opisania zachowania małych zwierzątek budujących nastrój i dodających realizmu świata gry, ale nie mających żadnego wpływu na rozgrywkę. Dzięki temu, że konfiguracja zachowania zwierzątek wczytywana była z zewnętrznych skryptów LUA[21], były one ustawiane przez projektantów, bez bezpośredniego udziału programisty. Możliwość parametryzacji zachować umożliwiła oparcie na tym mechanizmie zwierząt takich jak: ptaki (czasami obsiadające gniazda, bądź zwłoki i odlatujące gdy ktoś się zbliżał), żywy inwentarz (na przykład kury i gęsi) lub inne małe stworzenia (żaby, myszy i koty), czy nawet niekoniecznie należące do królestwa zwierząt, psotliwe gnomy. Szczególnie ciekawe były jednak nietoperze, które na widok wiedźmina wzbijały się do lotu, i przelatywały obok niego, wyrzucane następnie siłą odpychającą leciały daleko w kierunku z którego przyszliśmy, by już nie powrócić. Wszystkie te zachowania implementował jeden, w miarę spójny, mechanizm.

Rysunek 3.12: Zwierzątka oparte na algorytmie stada w grze Wiedźmin. Czy wypatrzysz przyczajonego w trawie kota?

3.7.5. Maelstorm

Na algorytmie stada planowałem oprzeć sztuczną inteligencję w Maelstorm. Algorytm jest bardzo prosty koncepcyjnie oraz implementacyjnie. Jego zastosowanie jest bardzo intuicyjne i szczególnie nadaje się do otwartego świata w jakim toczy się moje gra. Jedyną rzeczą jaka stanęła mi na drodze był czas realizacji całego projektu. Niestety nie zdążyłem rozpocząć implementacji sztucznej inteligencji w Maelstorm i będzie to jedna z pierwszych rzeczy którą zaimplementuję w przyszłych pracach nad projektem.

[21]Obiektowy i funkcyjny język programowania, stosowany często w grach jako język skryptowy lub do definiowania GUI. Więcej informacji można znaleźć na stronie projektu: *http://www.lua.org/*

Nowością byłaby próba zamodelowania w oparciu o algorytm stada bardziej skomplikowanych akcji, jak atakowanie, unikanie wystrzałów czy realizacja celów. Wymagałoby to oczywiście w pewnych sytuacjach odejścia od bezstanowego charakteru algorytmu. Kolejnym problemem byłaby implementacja zachowania stadnego obiektów, które nie mogą bezpośrednio zmieniać swojego wektora ruchu i dla których taka zmiana wymusza zastosowanie serii akcji (na przykład obrót statku w konkretnym kierunku i włączenie silników/hamulców). Te problemy jednak tym bardziej motywują do próby eksperymentowania z rozwiązaniem opartym o algorytm stada i podjęciem się stworzenia ciekawej logiki rozmytej, kontrolującej statki kosmiczne komputerowego gracza.

3.7.6. Dołączony program demonstracyjny

Ponieważ nie udało mi się zawrzeć implementacji algorytmu stada w Maelstorm, dołączyłem do pracy magisterskiej inny, niewielki program mojej produkcji. Demonstracja "Park" to prosty program symulujący pewne zamknięte środowisko. Kontrola większości obiektów symulacji oparta jest na algorytmie stada. W świecie gry "żyją":

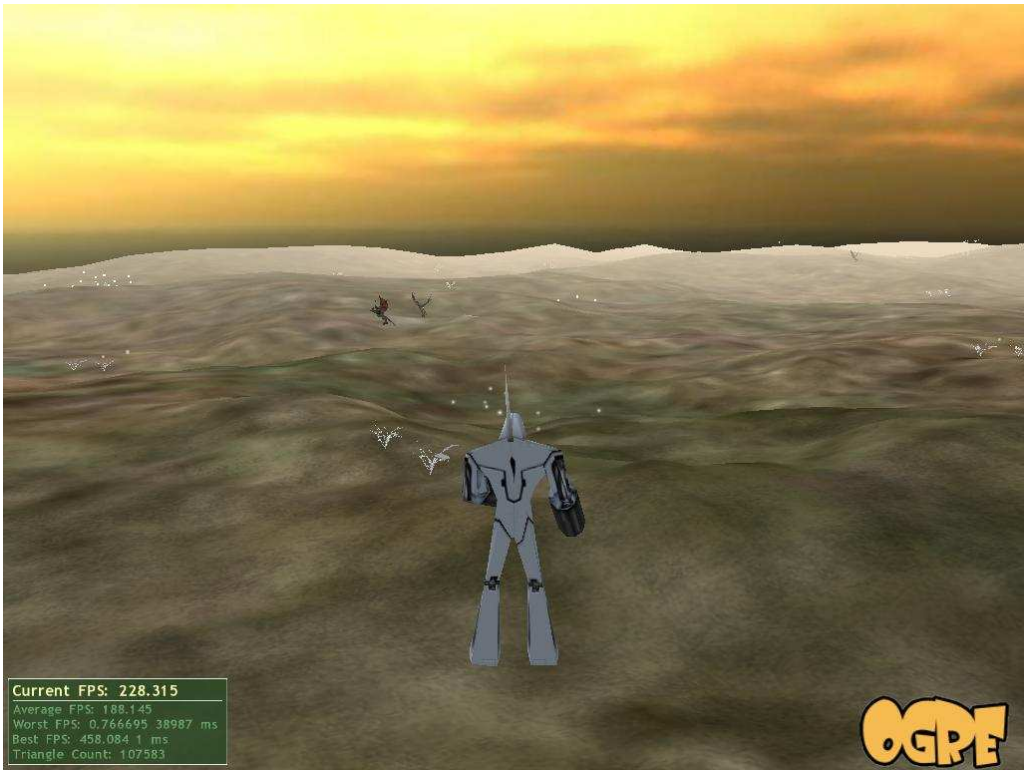
- **Robot** — obiekt gracza. Swobodnie poruszamy się po parku i obserwujemy co się dzieje na około.
- **Świetliki** — małe pasożyty. Trzymają się w stadach. Rozmnażają się między sobą bezpłciowo i żerują na rosnących w parku roślinach. Uciekają przed drapieżnymi sępami.
- **Sępy** — ptaki. Nie lubią swojego towarzystwa i jeżeli akurat nie mają ochoty się rozmnażać, wolą nie wchodzić sobie w strefę łowiecką. Boją się też robota. Żerują na świetlikach.
- **Rośliny** — rosną i powolutku się rozmnażają. Są zjadane przez stada świetlików.

By uzyskać wiarygodne zachowanie świetlików i sępów użyłem następujących zasad:

- **Utrzymuj wysokość** — obiekty starają się lecieć na konkretnej wysokości. Świetliki latają delikatnie nad ziemią by nie przegapić roślinek i oszczędzać siły. Sępy natomiast latają wysoko by lepiej widzieć z wysokości swoje ofiary. Gdy są głodne nurkują w dół na ofiarę.
- **Unikaj łowców** — obiekty uciekają od łowców. Stada świetlików rozpierzchają się we wszystkie strony gdy wpadnie w nie sęp, a ten z kolei musi być krytycznie głodny by zbliżyć się do stada świetlików pilnowanego przez robota.
- **Poluj** — gdy obiekt poczuje głód, zaczyna coraz bardziej pragnąć zbliżyć się do swojego pożywienia.
- **Szukaj partnera** — co jakiś czas obiekty mają możliwość przedłużyć swój gatunek. Niespełnione pragnienia oczywiście się potęgują.
- **Poruszaj się** — pewną, lekką potrzebą było poruszanie się z określoną prędkością w aktualnym kierunku. Dzięki temu świat nabrał dynamiki. Sępy patrolowały okolicę, a stada świetlików szukały nowych źródeł pożywienia.
- **Utrzymuj dystans** — podstawowe założenie separacji z pierwszych implementacji Craiga Reynoldsa. Utrzymujemy odpowiednią odległość od najbliższego członka stada.

- **Utrzymuj kierunek** — założenie wyrównania. Tutaj realizowane przez naśladowanie prędkości i kierunku najbliższego członka stada.
- **Podążaj za stadem** — założenie spójności. Świetliki dążą do środka ciężkości widzianego przez nie stada. Sępy go unikają.

W powyższym opisie chciałem zwrócić uwagę na intuicyjność stosowanych zasad. Intensywność z jaką zasada wpływa na ruch jest oczywiście zależna od natężenia danego pragnienia i jego priorytetu. Program Park pisałem przez półtora tygodnia, ale algorytm stada zadziałał właściwie od razu. Pierwsza działająca wersja symulacji z zaimplementowanym algorytmem stada, w której poprawione były podstawowe błędy programistyczne, miała poprawnie działający system sztucznej inteligencji. Prostota tego podejścia umożliwia bardzo łatwą konfigurację systemu. Ponieważ pracujemy wśród wartości ciągłych konfiguracja umożliwia intuicyjne tworzenie dosyć zróżnicowanych zachowań na podstawie jednego systemu. Gdybym miał dalej rozwijać demonstrację "Park", myślałem by eksperymentalnie zróżnicować w małym zakresie obiekty danego gatunku (lekko różne priorytety potrzeb) i wprowadzić możliwość dziedziczenia cech przez potomków. W bogatym, stabilnym środowisku powinno to prowadzić do ewolucyjnej konfiguracji obiektów.



Rysunek 3.13: Demonstracja "Park".

3.8. Techniki programistyczne

3.8.1. Alokacja pamięci

Alokacja pamięci jest bardzo istotną kwestią w odniesieniu do gier komputerowych. Po pierwsze gry komputerowe potencjalnie są aplikacjami o bardzo dużym zapotrzebowaniu na pamięć, z uwagi na konieczność przydziału zasobów potrzebnych do wyświetlania grafiki. Drugą kwestią jest iteracyjny charakter aplikacji, która przetwarza obiekty wirtualnego świata wielokrotnie w trakcie sekundy. Jakiekolwiek błędy z gubieniem alokowanych wskaźników nasilają się lawinowo i prowadzą do utraty stabilności. Dodatkowo, jeżeli w trakcie tury gry będziemy dokonywali dużej liczby alokacji i zwalniania pamięci, poza oczywistym spowolnieniem aplikacji, pojawią się problemy z fragmentacją dostępnej pamięci operacyjnej. W trzydziesto dwu bitowym systemie Windows XP pojedynczy proces ma do dyspozycji nie więcej jak dwa gigabajty pamięci operacyjnej. W Wiedźminie były bardzo duże problemy z występowaniem braku pamięci. Oczywiście wszystkie kończyły się błędem krytycznym i zamknięciem aplikacji.

Gra często nie wykorzystywała więcej jak gigabajt pamięci, ale system nie mógł już przydzielić jej spójnego bloku większej długości. Powodem tego była właśnie fragmentacja. Ją z kolei powodowała zbyt częsta alokacja małych bloków pamięci w ciągu przeciętnej tury gry. Usunięcie tego problemu wymagało bardzo dużej pracy i sukces został osiągnięty tylko częściowo, ponieważ gra w chwili wydania miała wciąż pewne problemy ze stabilnością.

Jednym z rozwiązań części problemów z zarządzaniem pamięcią jest stosowanie własnych lub zewnętrznych funkcji alokacji. Szczególnie tyczy się to sytuacji gdy projekt jest w zaawansowanym stadium a my mamy problemy ze stabilnością lub wydajnością. Przykładem popularnie stosowanego, zewnętrznego alokatora jest na przykład biblioteka *pool* wchodząca w skład dystrybucji omawianej już biblioteki *BOOST*. Często własne funkcje alokacji są w pewnym stopniu ograniczone w stosunku do standardowej implementacji *malloc*, ale z drugiej strony mogą działać dużo szybciej lub przeciwdziałać fragmentacji.

By obiekty klasy C++ przy alokacji korzystały ze specjalnych metod alokacji należy przedefiniować dla nich operatory *new()* i *delete()*. Oto przykład:

```
class WorldAction : public Serializable {
public:
    void* operator new(size_t sz) { return g_ActionAllocator.AllocGeneric(sz); }
    void operator delete(void* p) { g_ActionAllocator.FreeGeneric(p); }
    virtual void doAction() = 0;
    ...
};
class WorldActionAddObject : public WorldAction {
public:
    void* operator new(size_t sz)
        { return g_ActionAllocator.Alloc< WorldActionAddObject >(); }
    void operator delete(void* p)
       { g_ActionAllocator.Free< WorldActionAddObject >((WorldActionAddObject*)p); }
    void doAction();
    ...
};
```

Nasz operator *new()* wywoła się za każdym razem gdy będziemy tworzyć obiekt danej klasy. W powyższym przykładzie jednak, jeżeli będziemy niszczyć obiekt wirtualnej klasy *WorldAction*,

należący w rzeczywistości do *WorldActionAddObject*, wówczas jeżeli destruktor klasy bazowej nie będzie zadeklarowany jako wirtualny, to jej operator zwalniania zostanie wywołany. Inaczej ujmując trzeba pamiętać, że operator zwalniania jest wywoływany przez destruktor klasy.

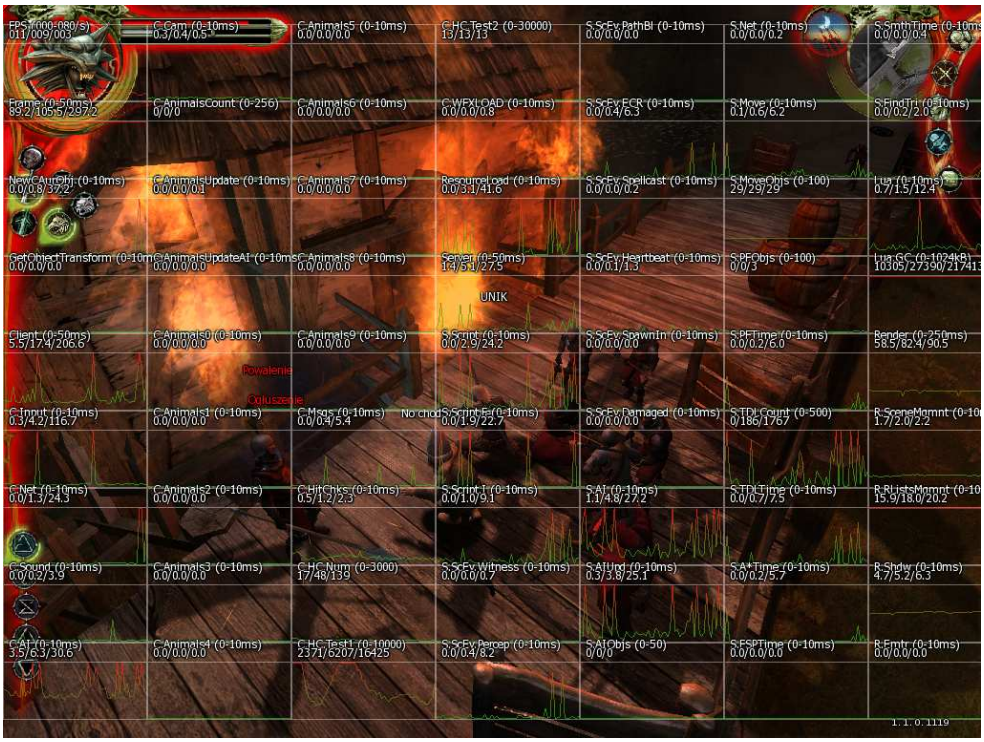
W Maelstorm na obecnym etapie projektu, nie musiałem przejmować się problemami alokacji. Dla własnej satysfakcji i nauki napisałem jednak swój alokator i podpiąłem go do gry by przydzielał pamięć dla obiektów akcji gry. To dobre rozwiązanie, ponieważ ich alokacja była potencjalnym wąskim gardłem aplikacji. Ilość powstających akcji była zależna od liczby obiektów gry i graczy, co mogło prowadzić do tego, że mocno obciążona gra miałaby po pierwsze olbrzymie problemy z fragmentacją pamięci, a po drugie sama alokacja zajmowała by bardzo dużą część tury gry. Rozwiązaniem które zastosowałem było napisanie cyklicznego alokatora. Pomysł pewnie nie jest oryginalny, ale był w całości mój. Alokator posiada tablicę pozycji które może przydzielić. Rozmiar jednostki alokacji jest dopasowany do średniej wielkości najczęściej alokowanych obiektów akcji. Jeżeli obiekt wymaga większej ilości pamięci, rezerwuje sobie kilka jednostek alokacji. Pamięć przydzielana jest "po kolej". Przechodzę zaalokowany na początku obszar pamięci przydzielając kolejne jednostki alokacji i odnotowując w dodatkowej tablicy które z nich są zajęte. Gdy dojdę do końca tego obszaru, zaczynam przydzielanie znowu od jego początku. Ponieważ akcje są obiektami o zwykle dosyć krótkim czasie życia, zakładam że przy ponownym przechodzeniu tablicy większość pozycji będzie już ponownie wolna. Opisałem to rozwiązanie nie z powodu jego wyjątkowości, ale raczej żeby pokazać, że indywidualna alokacja pamięci powinna być dostosowana do specyfiki zarządzania obiektami.

3.8.2. Profilowanie kodu

Nie wcześniej niż jakieś pół roku przed wydaniem gry, pracujący nad Wiedźminem zespół zorientował się, że gra ma zdecydowanie zbyt wysokie wymagania sprzętowe. Ponieważ dystrybutor nalegał, zresztą słusznie, na konieczność obniżenia minimalnych wymagań gry, mały zespół programistów, w tym również ja, został dedykowany na krótki okres do prac wyłącznie na rzecz optymalizacji gry. Bardzo istotnym elementem tej pracy było znalezienie elementów mających decydujący wpływ na wydajność całej aplikacji. Wiele funkcji, mimo ewidentnej rozrzutności obliczeniowej okazywało się wywołanymi na tyle rzadko, że nie warto było poświęcać czasu na ich poprawę.

Pierwszym użytym do tego celu mechanizmem było zastosowanie wykresów wydajności. Został zdefiniowany zbiór makrodefinicji które włączały i wyłączały pomiar czasu dla zdefiniowanego mechanizmu oraz prosty zestaw komend pozwalający włączyć i wyłączyć wyświetlanie wykresów w grze. Wyniki zbierane były co rundę gry i prezentowane na ekranie w formie zestawu prostych wykresów (patrz ilustracja 3.14). Dzięki zastosowaniu makrodefinicji kompilatora C++ możemy usunąć zupełnie funkcjonalność wykresów wydajności w wynikowym kodzie aplikacji. Uzyskane za pomocą tej metody wyniki określają czas działania kluczowych mechanizmów gry. Można na jej podstawie wyprowadzić pewne ogólniejsze wnioski. Brakuje jednak szczegółowej wiedzy, co dokładnie stanowiło o koszcie danych mechanizmów.

W znalezieniu słabych punktów aplikacji podstawową pomocą był program profilujący, umożliwiający zebranie danych statystycznych, na temat czasu i częstotliwości wywołań funkcji zdefiniowanych w kodzie gry. Przy korzystaniu z programu profilującego istotne było by sam proces profilowania gry i zbierania danych nie zaburzał wyników. Wielokrotnie spowalniał on wykonanie kodu gry. Uruchamiany na normalnej konfiguracji gry, powodował, że działała ona z częstotliwością klatki na kilka sekund. Wyniki wówczas opisywały zupełnie ekstremal-



Rysunek 3.14: Wykresy wydajności w grze Wiedźmin.

ną sytuację, która nigdy nie miałaby miejsca w rzeczywistości. By profilować Wiedźmina tworzona była kompilacja, w której funkcja licząca przepływ czasu była kilkudziesięciokrotnie spowalniana. Bez tej prostej funkcjonalności, przeprowadzone testy pozbawione byłby jakiejkolwiek użyteczności.

Kolejnym istotnym elementem była konieczność odpowiednio dobranego momentu przeprowadzania testu. Program profilujący zaczynał działać dopiero jakiś czas po włączeniu rozgrywki oraz przejściu kilku pierwszych kroków, tak by funkcje doczytujące elementy lokacji gry nie zdominowały i nie zaburzyły wyników testu. W trakcie jego działania staraliśmy się sterować postacią gry, omijając elementy rozgrywki, które mogłyby zaburzyć wyniki testu. Na przykład istotne było by nie rozpoczynać dialogów, których wystąpienie miało globalny wpływ na symulację mechaniki gry oraz wyświetlanie.

Wyniki testów dawały nam bardzo dokładną wiedzę na temat każdej funkcji w kodzie gry. Mieliśmy informacje na temat tego ile razy była wywołana. Wiedzieliśmy ile czasu aplikacja poświęciła na wykonanie jej kodu oraz wywołanie każdej z funkcji w niej zagnieżdżonych z osobna. Na podstawie takich danych mogliśmy już nie tylko stwierdzić jakie mechanizmy miały największy negatywny wpływ na wydajność gry, ale również znaleźć tego przyczyny i prawdziwe wąskie gardła aplikacji.

Rozdział 4

Podsumowanie

Moja praca magisterska był dla mnie okazją do zdobycia dodatkowego doświadczenia oraz własnych poszukiwań i przemyśleń dotyczących programowania a w szczególności tworzenia gier komputerowych. W toku prac nad projektem Maelstorm Online, które rozciągały się na przestrzeni trzech lat, jako programista starałem się rozwijać indywidualnie i zawodowo. Sam projekt był mi możliwością do wielu eksperymentów programistycznych i kontaktu z nowymi dla mnie problemami.

W części teoretycznej pracy chciałem pokrótce przedstawić tematykę tworzenia gier oraz przybliżyć kilka, wybranych aspektów z nią związanych. W prezentowanych w rozdziale 3 rozwiązaniach popularnych problemów programistycznych tworzenia gier, starałem się przedstawić własne rozwiązania tych kwestii oraz próbowałem przeprowadzić dyskusję różnorodnych sposobów podejścia do problemu. Starałem się przy tym pokazać jak bardzo istotny jest świadomy dobór użytych rozwiązań na podstawie specyfiki tworzonej gry, przy możliwym zachowaniu ich uniwersalności.

Gry komputerowe są dosyć specyficznym działem programowania. O tym, że można się nimi zajmować zawodowo przekonywałem w rozdziale 1. Osobiście uważam tworzenie gier za szczególnie interesujące z programistycznego punktu widzenia, z uwagi na bardzo szeroki wachlarz problemów jaki stawiają przed programistą. Wciąż dla większości z nich nie ma idealnych i zawsze skutecznych rozwiązań. Istnieje jeszcze bardzo duże pole manewru do tworzenia zupełnie nowych technik programistycznych, wzorców projektowych i algorytmów. Choć tworzenie gier sprowadza się zwykle do podobnego procesu, nie można sobie pozwolić na powtarzanie utartych schematów. Kolejne produkcje zarówno pod względem projektu jak i implementacji po prostu muszą się zmieniać. Tak, by za każdym razem móc przyciągać uwagę graczy kusząc ich nową jakością i zaskakując nowymi elementami rozgrywki.

4.1. Wnioski dotyczące projektu

Maelstorm Online w chwili obecnej jest małą i prostą grą akcji dla wielu graczy. Możliwa jest daleko idącą rozbudowa projektu i stworzenie na jego silniku dużo bardziej skomplikowanej rozgrywki. Założenia projektu, które przyjąłem na początku, zarówno te dotyczące wizji funkcjonalności projektu oraz architektury, wciąż uważam za aktualne. Mechanizm synchronizacji w oparciu o zdarzenia gry w dużym stopniu sprawdził się w praktyce. Implementacja wydajnej i zapewniającej synchronizację komunikacji była za pośrednictwem tej metody dużo prostsza w porównaniu do standardowego podejścia implementacji sieci. Oczywiście Maelstorm Online od strony mechanizmów rozgrywki był bardzo prostym projektem, dlatego pozostaje otwartym pytanie jak rozwiązanie to będzie się skalować do systemów w których obiekty wchodzą

w dużo bardziej skomplikowaną interakcję między sobą.

4.2. Testy aplikacji

Aplikacja Maelstorm Online była testowana w konfiguracji, która została oddana z pracą magisterską. Poza próbą gry przez internet była testowana wydajność silnika gry aplikacji serwera. Chciałem odnaleźć zależności oraz limity dotyczące komplikacji symulowanego świata. Ilość obiektów, które może on obsłużyć i zależność wydajności od rozmiaru prezentowanego świata.

4.2.1. Warunki techniczne przeprowadzonych testów

Gra testowana była wyłącznie pod systemem Windows XP. Najsłabszą konfiguracją na jakiej była uruchamiana aplikacja kliencka, był komputer klasy Pentium IV 2.8 MHz wyposażony w 512 MB pamięci RAM oraz kartę graficzną ATI Radeon 9550. Aplikacja działała na tej konfiguracji sprzętowej bez zarzutów i wydaje się, że powinna działać również na słabszych komputerach. W trakcie testów aplikacja kliencka działała na komputerze klasy Intel Core 2 Quad Q6600 2.4 MHz wyposażonym w 2 GB pamięci RAM. Serwer podłączony był z internetem za pośrednictwem routera sieci lokalnej. Łącze z internetem było łączem ADSL udostępnianym w ramach usługi Neostrada z pasmem odbioru około 2 Mb/s, a wysyłaniem 256 Kb/s. Inne komputery łączyły się z serwerem za pośrednictwem sieci lokalnej oraz za pośrednictwem internetu. Łączące się za pośrednictwem internetu komputery dysponowały łączem ADSL o prędkość odbioru co najmniej 1 Mb/s, choć tu również wydaje się, że do płynnej gry i wydajnej synchronizacji starczyć powinno dowolne łącze klasy DSL/ADSL bądź szybsze.

4.2.2. Wyniki testów

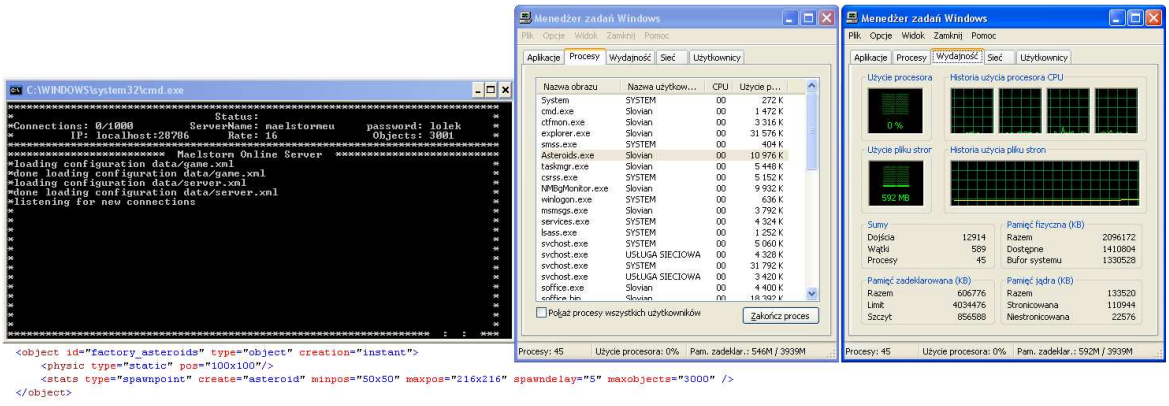
Podstawowe testy polegały na grze pomiędzy trójką, bądź dwójką graczy połączonych za pośrednictwem internetu. Testy oceniam dosyć pozytywnie, w normalnych warunkach opóźnienia spowodowane komunikacją nie wpływały na rozgrywkę. Występowały jednak sporadyczne problemy związane z synchronizacją zegara. W ich wyniku akcje przychodziły z aplikacji klienckiej z kilkusekundowym opóźnieniem co danemu graczowi uniemożliwiało grę. Pozostałe, pomniejsze błędy z synchronizacją zostały wyłapane i poprawione, w większości jeszcze w trakcie prowadzonych testów.

Testy wykazały poprawność zastosowanego mechanizmu. W trakcie prac nad grą regularnie była ona sprawdzana w działaniu przez sieć lokalną. Nie byłem pewien jak gra zachowa się pod wpływem większych i bardziej przypadkowych opóźnień. Testy w tej formie niestety nie odpowiadają na pytanie, odnośnie potencjału i skuteczności zastosowanych rozwiązań. Potrzebne są testy aplikacji przy dużym obciążeniu komunikacyjnym z sieci internet. Należałoby przeprowadzić testy aplikacji serwera w sytuacji gdy podłączona będzie do niej bardzo duża ilość zdalnych komputerów. Takie testy są dosyć trudne do zaplanowania i przeprowadzenia. Niestety nie zdołałem ich wykonać przed oddaniem niniejszej pracy.

Drugim rodzajem przeprowadzonych testów były testy wydajności mechaniki gry. Ich rezultaty były bardzo zadowalające. Na serwerze gry były regularnie, co pięć klatek tworzone nowe obiekty asteroid, poruszające się z losową prędkością w losowym kierunku. Był ustalony na ich liczbę odgórny limit trzech tysięcy. Świat gry był nie duży, tak by częściej dochodziło do interakcji asteroid (obiekty cały czas zderzały się ze sobą). Aplikacja serwera wykonywała szesnaście kroków symulacji na sekundę. To myślę jest optymalna częstotliwość symulacji i

przy takiej też pracowała w trakcie testów przez sieć. Przy trzech tysiącach obiektów aplikacja serwera działa bez zarzutu właściwie nie obciążając systemu. Działała też bardzo równo i ani razu nie zdarzyło jej się wyjść poza przypisany na klatkę serwera czas. Za pierwszym razem uruchomiłem ją w normalnej konfiguracji gry. W wyniku kolizji, które powodowały uszkodzenia i w konsekwencji niszczenie obiektów gry, ich ilość przestała przyrastać około poziomu 1900 asteroid. Mimo ciągłej destrukcji i tworzenia obiektów co świadczyło o dosyć mocnym obciążeniu mechaniki gry rozpatrywaniem kolizji, aplikacja działała swobodnie wykorzystując bardzo małą ilość zasobów systemowych. Około 10 MB pamięci operacyjnej oraz minimalnie wykorzystując siłę obliczeniową procesorów, mniej niż wiele innych działających procesów. Następnie włączyłem zadawanie uszkodzeń i jeszcze raz rozpocząłem test. Przy trzech tysiącach obiektów zalogowałem się na serwer włączając aplikację kliencką. Obiekty były bardzo ciasno ułożone. Niektóre powoli się poruszały ale od razu prowadziło to do zderzeń. Niestety test był miarodajny tylko w ograniczonym zakresie. Wiele potencjalnie kosztownych mechanizmów działa wyłącznie w odniesieniu do zdalnych graczy połączonych z serwerem. Przykładem jest tu sprawdzanie widoczności oraz kontrola rozsyłania akcji wyłącznie do widzących je graczy.

Wyniki testów skłaniają do wniosku, że przynajmniej jedno z założeń wydajnościowych, stawianych przed grą może być spełnione. Nawet jeżeli, nie będzie ona w stanie obsłużyć ponad stu połączeń, wygląda na to, że zasoby komputera potrzebne do obsługi mniejszej ilości graczy, będą pozwalały na uruchomienie wielu jej egzemplarzy.



Rysunek 4.1: Aplikacja serwera Maelstorm w trakcie testów wydajnościowych mechaniki.

4.2.3. Horyzonty rozbudowy projektu

Jest wiele elementów które nie zostały zaimplementowane z uwagi na ograniczenia czasowe projektu. Większość z nich na obecnym etapie nie była do funkcjonowania gry niezbędna. Przy rozwoju projektu w pierwszej kolejności należałoby się zająć:

- Można poprawić wydajność komunikacji. Przykładem jest przesyłanie małych obiektów, takich jak pociski, przez sieć. Część obiektów może być wczytywana do gry zarówno po stronie klienta jak i serwera.

- Należy poprawić bezpieczeństwo aplikacji serwera na wypadek prób połączenia się programów klienckich wykorzystujących słabości jego implementacji.

- Brakuje inteligentnych obiektów niezależnych. Żałuję, że nie miałem możliwości implementacji sztucznej inteligencji.
- Gracze powinni móc tworzyć statki z dużo większą dowolnością. Gracz powinien móc konfigurować każdy wchodzący w skład myśliwca podsystem. Podsystemy powinny mieć widoczny wpływ na wygląd statku.
- Potrzeby jest rozwój modułu wyświetlania, nowe efekty wizualne i animacje oraz przyjemniejsza dla oka szata graficzna GUI.
- Kod powinien zostać tak zorganizowany by rozdział na aplikację kliencką i serwera był dużo prostszy i czytelniejszy, jednocześnie zachowując pozytywne aspekty obecnego rozwiązania. Aktualne rozwiązanie z zastosowaniem makrodefinicji może nie skalować się do większego projektu.
- Słabym punktem mojego silnika może być pełna synchronizacja zegara po stronie aplikacji klienckiej z zegarem serwera. Różnice w pomiarze czasu pomiędzy komputerami mogą w konsekwencji prowadzić do utraty synchronizacji gry. Natknąłem się już na ciekawe rozwiązania tego problemu. Między innymi kompletne opracowanie dotyczące tej kwestii można znaleźć w artykule *Propagation of Visual Entity Properties Under Bandwidth Constraints*, napisanym przez *Oliviera Cado*, opartym na doświadczeniach w pracy nad komercyjnymi grami MMORPG. Artykuł jest dostępny na serwisie gamasutra [GS] pod adresem http://www.gamasutra.com/view/feature/1421/propagation_of_visual_entity_.php.

Jak już wspominałem w rozdziale 2.4 tworząc moją pracę magisterską miałem na uwadze dużo większy projekt, którego elementem składowym byłaby gra Maelstorm w formie zbliżonej do obecnej. Projektując aplikację chciałem by mogła ona stać się częścią rozproszonego systemu gry dla bardzo dużej liczby graczy (*Massive Multiplayer Online*). Zakładałem, że jeżeli w Maelstorm będzie mogło na raz spotkać się powyżej stu graczy, lub jeżeli na pojedynczym komputerze, będzie można uruchomić kilka aplikacji serwera obsługujących różne rozgrywki dla mniejszej liczby graczy, wówczas gra mogłaby stworzyć architekturę rozproszoną. Centralnym punktem tej architektury byłby serwer gry "społecznej". Byłaby to zupełnie nowa aplikacja, pozbawiona elementów zręcznościowych i w swoim działaniu dużo bardziej przypominająca serwis internetowy oparty na bazie danych. Aplikacja "społeczna" symulowałaby funkcjonowanie rozmieszczonych w pasie asteroidów stacji kosmicznych. Gracze za jej pośrednictwem mogliby szukać nowych, wypełnionych akcją zadań oraz wydawać zarobione w trakcie ich realizacji pieniądze. Opowiadać o szerszej koncepcji projektu można by długo. Chciałem po prostu zaznaczyć, że projekt Maelstorm Online w swojej obecnej postaci nie powinien być traktowany jako ostateczny produkt. To kompletny silnik umożliwiający implementację dużo bardziej rozbudowanej rozgrywki. Ma perspektywy rozwoju, których realizacja umożliwiłaby stworzenie naprawdę atrakcyjnej gry.

Dodatek A

Materiały dostarczone wraz z pracą magisterską

Oprócz niniejszego teoretycznego opracowania, do mojej pracy magisterskiej dołączona jest płyta CD. Zawiera ona między innymi realizację praktycznej części pracy magisterskiej w postaci kodu źródłowego oraz binarnej, skompilowanej w systemie Windows. Oto spis zawartości dołączonej płyty z grą:

- */maelstorm/source/* — Kod źródłowy gry Maelstorm Online. Wraz z kodem dołączone są pliki projektu Visual Studio 2005. W tym środowisku zalecam ewentualną kompilację kodu.
- */maelstorm/binary/* — Archiwum zawierające skompilowaną i działającą na platformie Windows grę Maelstorm Online. By ją uruchomić, należy uprzednio rozpakować archiwum na nośnik o możliwości zapisu i odczytu (najlepiej dysk twardy). Szczegóły dotyczące kwestii użytkowania gry znaleźć można w pliku *readme.txt*.
- */documentation/praca magisterska.pdf* — Niniejsza praca magisterska w wersji elektronicznej.
- */documentation/source/* — Katalog zawiera pliki źródłowe w formacie LaTeX, na podstawie których został wygenerowany dokument pracy magisterskiej.
- */additional/Wizja MMOAsteroids.pdf* — Dokument wizji, który powstał w 2004 roku i dotyczył projektu mojej przyszłej pracy magisterskiej.
- */additional/Prezentacja 27.V.2007.ppt* — Prezentacja pracy magisterskiej dotycząca jej stanu z dnia 27.V.2007 roku.
- */additional/illustrations/* — Katalog zawierający wysokiej rozdzielczości ilustracje do pracy magisterskiej.
- */additional/flocking/source/* — Kod źródłowy aplikacji symulującej zachowanie stada, o której wspominałem w rozdziale 3.7.
- */additional/flocking/binary/* — Skompilowana wersja demonstracyjnego programu symulującego zachowanie stada.

Bibliografia

[DLo1] Mark DeLoura, *Perełki Programowania Gier tom I*, 2002 Helion.

[DLo2] Mark DeLoura, *Perełki Programowania Gier tom II*, 2002 Helion.

[DLo3] Mark DeLoura, *Perełki Programowania Gier tom III*, 2003 Helion.

[AKi4] Andrew Kirmse, *Game Programming Gems 4*, 2004 Charles River Media.

[TW07] *Wiedźmin: Gra Komputerowa* (ang. *The Witcher: Role - Playing Game*), 2007 CD Projekt RED. http://www.thewitcher.com

[GS] Serwis internetowy Gamasutra http://www.gamasutra.com

[bOGR] Ogre 3D, biblioteka graficzna http://www.ogre3d.org

[bGUI] CEGUI Crazy Eddie's Gui System, biblioteka interfejsu użytkownika http://www.cegui.org.uk

[bAIO] ASIO, Asynchronous Input Output, biblioteka sieciowa http://www.libsdl.org

[bSDL] SDL Simple Direct Media Layer, biblioteka multimedialna http://www.libsdl.org

[CR] Craig Raynolds, strona poświęcona algorytmowi stada http://www.red3d.com/cwr/