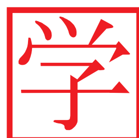
POLSKO-JAPOŃSKA
WYŻSZA SZKOŁA
TECHNIK KOMPUTEROWYCH

Wojciech Borczyk, Jacek Kujawski,
Krzysztof Marzec

Architektura i programowanie wieloplatformowych silników gier video



WYDAWNICTWO
PJWSTK

Notki biograficzne

Wojciech Borczyk (mgr inż.), absolwent kierunku Informatyka na Politechnice Śląskiej. Napisał doktorat z zakresu syntezy fotorealistycznych obrazów z wykorzystaniem modelu oświetlenia globalnego na wydziale Informatyki Politechniki Śląskiej. Prowadzi wykłady i zajęcia na Politechnice Śląskiej oraz PJWSTK. Wieloletni pasjonat gier wideo, a od 2005 roku związany zawodowo z przemysłem produkcji gier wideo. Aktualnie pracuje jako project leader oraz assistant producer, prowadząc katowicki oddział City Interactive. Brał udział przy tworzeniu ponad 10 projektów na platformy PlayStation3, Xbox360, PC, Nintendo Wii oraz Nintendo DS.

Jacek Kujawski (mgr inż.), absolwent kierunku Informatyka na Politechnice Śląskiej w Gliwicach. Od dzieciństwa marzył o zostaniu programistą gier video; od 4 lat zajmuje się zawodowo ich tworzeniem. Obecnie pracuje na stanowisku engine programmer w katowickim oddziale City Interactive, biorąc udział w projektach na takie platformy jak: PlayStation3, Xbox360, PC, Nintendo Wii, Nintendo DS. Oprócz gier interesuje się motocyklami i motoryzacją zabytkową.

Krzysztof Marzec (mgr inż.), absolwent kierunku Informatyka na Politechnice Śląskiej oraz Coventry University. Z przemysłem gier wideo związany zawodowo od 2007 roku, przy pracy nad projektami na platformy PlayStation3, Xbox360, PC, Nintendo Wii oraz Nintendo DS . Obecnie pracuje jako senior engine programmer w Katowickim oddziale City Interactive. Specjalizuje się w programowaniu systemów niskopoziomowych oraz projektowaniu architektury kodu gry i silnika gier wideo.

Streszczenie

Książka omawia najważniejsze zagadnienia z zakresu projektowania i programowania wieloplatformowych silników gier wideo. Czytelnik dowie się o zastosowaniach tego typu silników i założeniach które leżą u ich podstaw. Książka przedstawia architekturę silnika i opisuje zagadnienia charakterystyczne związane z wieloplatformowoscią. Szczegółowo omówiono aspekty wirtualizacji, zarządzania pamięcią oraz narzędzi diagnostycznych.

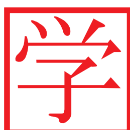
Seria: Podręczniki akademickie

Edytor serii: Leonard Bolc
Tom serii: 57

POLSKO-JAPOŃSKA
WYŻSZA SZKOŁA
TECHNIK KOMPUTEROWYCH

Wojciech Borczyk, Jacek Kujawski,
Krzysztof Marzec

Architektura i programowanie wieloplatformowych silników gier video



WYDAWNICTWO
PJWSTK

**© Copyright by Wojciech Borczyk, Jacek Kujawski,
Krzysztof Marzec
Warszawa 2011**

**© Copyright by Wydawnictwo PJWSTK
Warszawa 2011**

Wszystkie nazwy produktów są zastrzeżonymi nazwami handlowymi lub znakami towarowymi odpowiednich firm. Książki w całości lub w części nie wolno powielać ani przekazywać w żaden sposób, nawet za pomocą nośników mechanicznych i elektronicznych (np. zapis magnetyczny) bez uzyskania pisemnej zgody Wydawnictwa.

Edytor
Leonard Bolc

Kierownik projektu
Prof. dr hab. inż. Konrad Wojciechowski

Korekta
Anna Bittner

Redaktor techniczny
Aneta Ługowska

Komputerowy skład tekstu
Grażyna Domańska–Żurek

Projekt okładki
Andrzej Pilich

**Wydawnictwo
Polsko-Japońskiej Wyższej Szkoły Technik Komputerowych**
ul. Koszykowa 86, 02-008 Warszawa
tel. 22 58 44 526, fax 22 58 44 503
e-mail: oficyna@pjwstk.edu.pl

Oprawa miękka
ISBN 978-83-63103-09-5
nakład: 150 egz.

Wersja elektroniczna
ISBN 978-83-63103-62-0

Projekt „ Nowoczesna kadra dla e-gospodarki – program rozwoju Wydziału Zamiejscowego Informatyki w Bytomiu Polsko- Japońskiej Wyższej Szkoły Technik Komputerowych. Współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego. Poddziałanie 4.1.1 „ Wzmocnienie potencjału dydaktycznego uczelni" Programu Operacyjnego Kapitał Ludzki

This book should be cited as:

Borczyk, W., Kujawski, J. & Marzec, K., 2011. Architektura i programowanie wieloplatformowych silników gier video. Warszawa: Wydawnictwo PJWSTK.

Spis treści

1 Wstęp ... 1
- 1.1 O czym i dla kogo jest ta książka ... 1
- 1.2 Przemysłgier wideo ... 1

2 Wieloplatformowy silnik gier wideo ... 5
- 2.1 Silnik gier wideo ... 5
  - 2.1.1 Proces tworzenia gry wideo ... 5
  - 2.1.2 Idea silnika gier wideo ... 14
- 2.2 Idea wieloplatformowego silnika gier wideo ... 24
  - 2.2.1 Tworzenie gry wideo na urządzenie niebędące komputerem ... 24
  - 2.2.2 Tworzenie gier wideo przeznaczonych na wiele platform 31
  - 2.2.3 Wieloplatformowy silnik gier wideo ... 41

3 Architektura silnika wieloplatformowego ... 53
- 3.1 Wybór języka programowania ... 53
  - 3.1.1 Rodzaje języków programowania ... 53
  - 3.1.2 Język programowania a przeznaczenie silnika gier wideo 56
  - 3.1.3 Cechy języków programowania ... 61
  - 3.1.4 Wybór języka dla celów tej książki ... 63
- 3.2 Organizacja logiczna ... 63
  - 3.2.1 Podziałfunkcjonalny elementów silnika ... 63
  - 3.2.2 Struktura logiczna silnika ... 67
  - 3.2.3 Przepływ sterowania w silniku ... 70
- 3.3 Organizacja projektu silnika ... 73
  - 3.3.1 Struktura podprojektów ... 73
  - 3.3.2 Struktura plików i katalogów ... 75

3.4 Organizacja kodu 77
3.4.1 Przydatne Mechanizmy C++ 77
3.4.2 Separacja kodu poszczególnych platform 80

4 Problemy związane z wieloplatformowością 91
4.1 Wyzwania dla wieloplatformowego silnika 91
4.2 Różnice sprzętowe 92
4.3 Wirtualizacja sprzętu 95
4.4 Przykłady wirtualizacji 96
4.4.1 Wirtualizacja kontrolerów wejścia 96
4.4.2 Wirtualizacja pamięci operacyjnych 99
4.4.3 Wirtualizacja pamięci masowych 100
4.4.4 Wirtualizacja urządzeń graficznych 103
4.5 Wykorzystanie specyficznych jednostek w procesorach 105
4.6 Różnice formatów danych 109
4.7 Różnice w implementacji gry 111

5 Zarządzanie pamięcią operacyjną 115
5.1 Pamięć operacyjna 115
5.2 Alokacja oraz dealokacja pamięci 116
5.2.1 Sterta vs Stos 116
5.2.2 Alokator małych obiektów 118
5.2.3 Pamięci o różnym czasie dostępu 119
5.2.4 Implementacja alokatorów 119
5.2.5 Alokatory a wielowątkowość 126
5.2.6 Podsumowanie alokatorów 126
5.3 Menadżer zasobów 127
5.3.1 Idea menadżera zasobów 127
5.3.2 Uchwyty, klucze 128
5.3.3 Zliczanie referencji 135
5.3.4 Buforowanie i cache'owanie zasobów 138
5.3.5 Współpraca menadżera zasobów z alokatorem 139
5.4 Wielowątkowość a zasoby plikowe 140

6 Narzędzia diagnostyczne 141
6.1 Błędy 141
6.1.1 Przyczyny błędów 141
6.1.2 Typy błędów 142
6.1.3 Wyszukiwanie błędów 143

6.2 Debugger 144
6.3 Wykorzystanie kompilacji warunkowej do tworzenia wersji testowych 146
6.4 Asercje 147
6.5 Konsola diagnostyczna 149
6.6 Śledzenie wycieków pamięci 150
6.7 Profiler 152
6.8 Monitor zasobów 155

7 Podsumowanie 159
7.1 Współczesne silniki komercyjne 159
7.2 Współczesne silniki opensource'owe 163

Literatura 165

1

Wstęp

1.1 O czym i dla kogo jest ta książka

Książka ta przedstawia proces powstawania gier wideo oraz cel istnienia i sposób wykorzystania wieloplatformowych silników gier wideo w ich implementacji. W pracy przedstawiona jest także podstawowa architektura takich systemów oraz elementów, z których są zbudowane, jak również problemy, jakie mogą wyniknąć przy ich projektowaniu i implementacji na różnych platformach sprzętowych oraz programowych. Celem tej publikacji jest przedstawienie różnych rozwiązań stosowanych przy rozwoju silników gier wideo w celu lepszej oceny przy wyborze istniejącego silnika, lub też pomocy w projektowaniu własnego systemu.

1.2 Przemysł gier wideo

Rynek gier wideo na świecie

Przemysł gier wideo jest branżą, która posiada już kilkudziesięcioletnią tradycję. Pierwsze, bardzo proste gry wideo zaczęły powstawać już w latach 40-tych XX wieku. Nie były to programy, lecz samodzielne urządzenia zaprojektowane z wykorzystaniem elektronicznych układów analogowych, nierzadko wykorzystujące laboratoryjne oscyloskopy do graficznej prezentacji przebiegu rozgrywki. Wraz z powstaniem pierwszych komputerów cyfrowych pojawiły się także pierwsze gry wideo będące programami komputerowymi. Programy te w tamtym okresie nie były niczym więcej niż ciekawostkami, tworzonymi w wolnym czasie przez zapaleńców i hobbystów.

Początek komercyjnego rynku gier wideo miał miejsce w 1971 roku, wraz z premierą automatu „Computer Space”. Był to pierwszy automat stworzony w celach zarobkowych. Został on wyprodukowany w liczbie 1500 egzemplarzy i umożliwiał rozgrywkę po wrzuceniu monety. Maszyna ta nie odniosła wiel-

kiego sukcesu (była trudna w obsłudze), jednak niewątpliwie zapisała się w historii jako początek istnienia tej branży.

Dzisiaj branża gier wideo, obok przemysłu filmowego oraz muzycznego jest jedną z największych gałęzi przemysłu rozrywkowego. Według gsales[1] przewidywana wartość rynku gier wideo w 2012 roku wyniesie 68 mld dolarów i będzie najszybciej rozwijającą się gałęzią przemysłu rozrywkowego. Jeszcze kilkanaście lat temu studia produkujące gry wideo były relatywnie niewielkimi przedsiębiorstwami, posiadały niewielkie budżety i często kilku pracowników. Dzisiaj rozmach oraz budżet przy produkcji największych światowych hitów porównywalny może być tylko z przemysłem filmowym[2], zaś olbrzymi marketing sprawia, że o produkcjach tych głośno jest we wszystkich możliwych środkach masowego przekazu. Wysokobudżetowe produkcje zgarniają oczywiście ogromną część dochodów z tego rynku, jednakże największą sprzedażą mogą obecnie pochwalić się producenci gier wideo dla tak zwanych odbiorców casualowych. Odbiorcy ci nie zwracają uwagi na najwyższej jakości grafikę czy inne najnowsze trendy technologiczne w grach, lecz oczekują prostej i przyjemnej rozrywki na co dzień, najczęściej w gronie rodziny lub znajomych. Wraz z olbrzymim wzrostem dostępności wszelakich urządzeń, dla których wydawane są gry wideo, odbiorcy ci stali się największą grupą w tej branży. Przeglądając zaś statystyki sprzedaży na portalu VGChartz[3], możemy zaobserwować dominację w sprzedaży produktów z gatunku gier rekreacyjnych dla konsol Nintendo Wii oraz Nintendo DS nad innymi grami dla bardziej wymagających graczy, przeznaczonych chociażby dla popularnych konsol Xbox360 i PlayStation 3. Dominacja ta wynika z kierunku, jaki obrało Nintendo dla swoich produktów, ale także z ogromnego wpływu marki, którą firma ta budowała przez lata jako dostawca gier przyjaznych dla użytkowników w każdym możliwym wieku.

Dzięki temu, że gry casualowe zazwyczaj nie wymagają tak dużego nakładu środków i pracy w ich wytworzeniu, można zaistnieć w tej branży bez dysponowania wielomilionowym budżetem. Ostatnimi czasy szansa ta stała się większa niż kiedykolwiek. Ogromna ilość różnego rodzaju urządzeń, takich jak telefony komórkowe, netbooki, tablety, konsole przenośne oraz popularność „Flashowych” gier na stronach internetowych spowodowała wielkie zapotrzebowanie na proste gry, w których tak samo jak w początkach branży dobry pomysł i wykonanie liczy się o wiele więcej niż dostępny budżet. Nie bez znaczenia jest tutaj także sposób dystrybucji, który w takich urządzeniach realizowany jest najczęściej przez Internet i różnego rodzaju usługi online. Brak konieczności produkcji wersji „pudełkowych” znacznie ogranicza koszty oraz ułatwia start niewielkim firmom. Z drugiej strony ogromnie zwiększa konkurencję. Na rynku obecnych jest bardzo dużo niewielkich deweloperów oraz wydanych przez nich

[1] http://vgsales.wikia.com

[2] Dla przykładu budżet hitu firmy Rockstar z 2008 roku - GTA IV, zamknął się kwotą w przybliżeniu 100 mln dolarów

[3] http:/www.vgchartz.com

gier, w związku z czym przebicie się z własnym produktem nie musi być wcale prostą sprawą.

Silniki gier wideo

Większość produkowanych gier komputerowych zawiera w sobie wiele wspólnych elementów takich jak np. mechanizmy renderowania grafiki, odgrywania dźwięków czy obsługi fizyki. Spowodowało to wyodrębnienie się pewnych systemów programowych zawierających ich obsługę oraz ułatwiających tworzenie gier z ich wykorzystaniem - silników gier wideo.

Silnik wieloplatformowy w dzisiejszych czasach jest już właściwie wymogiem. Spowodowane jest to obecnością na rynku tak wielu platform do gier, począwszy od komputerów osobistych wyposażonych w różne systemy operacyjne, poprzez konsole do gier (stacjonarne i przenośne), do telefonów komórkowych. Tworzenie gier na podobne platformy w oparciu o taki silnik jest opłacalne, gdyż trafia ona do większej rzeszy odbiorców, niż gdyby była dedykowana na jedną tylko konkretną platformę. Tworząc grę wieloplatformową, w oparciu o pewne założenia, otrzymujemy kilka produktów korzystających z bardzo wielu wspólnych elementów. Gra powinna zostać zaprojektowana w taki sposób, aby posiadała maksymalnie dużo wspólnych elementów, niezależnych od platformy, robiąc tylko niewielkie wyjątki dla części, które w lepszy sposób wykorzystują dane specyficzne cechy w konkretnej platformie. W oparciu o dobry projekt gry możliwe jest stworzenie modeli trójwymiarowych o wystarczająco dużym poziomie szczegółowości dla najmocniejszej platformy, grafik dwuwymiarowych w najwyższej możliwej rozdzielczości jak również dźwięków i muzyki w najlepszej jakości. Dla słabszych platform wystarczy „odchudzić” wyżej wymienione zasoby. Większość tłumaczeń podczas lokalizacji gry powinna być również wspólna. Przy dobrze zaprojektowanym silniku gry, kod logiki gry może właściwie się nie różnić zależnie od platformy. Dzięki silnikowi wieloplatformowemu wykorzystuje się w kilku produktach pracę wykonaną przez projektantów, grafików, dźwiękowców, muzyków, tłumaczy, programistów i wielu innych ludzi zaangażowanych w projekt.

Wiele firm deweloperskich rozwija własny silnik gier na potrzeby tworzenia swoich produktów. Stworzenie wydajnego, dojrzałego i nowoczesnego silnika jest jednak procesem bardzo złożonym i w związku z tym drogim, nierzadko zbyt drogim, aby silnik taki służył jedynie na własne potrzeby. Częstą praktyką stosowaną przez deweloperów jest odsprzedawanie licencji na wykorzystanie silnika tym firmom, które nie zdecydowały się, albo nie mają środków na pracę nad własnym, lub wręcz skoncentrowanie się tylko na rozwoju silnika na sprzedaż, bez produkcji własnych gier.

W przypadku małej firmy, której celem jest wejście na rynek gier wideo, powstaje pewien dylemat - kupić licencję na silnik, czy rozwijać swoje własne rozwiązanie. Wykorzystanie gotowego silnika jest godne rozważenia - możemy od razu rozpocząć tworzenie gry i nie przejmować się szczegółami sprzętowymi

konsoli lub innego urządzenia. W zależności od wybranego silnika przyjdzie nam jednak mniej lub więcej zapłacić lub wręcz podzielić się zyskami ze sprzedaży wydanej gry wideo. Silnik taki może bowiem nie obsługiwać wybranych przez nas platform jak też nie wspierać w wystarczający sposób mechanizmów, z których mamy zamiar skorzystać. Dlatego czasem warto zastanowić się nad rozwojem własnego silnika. Istnieje też trzecia możliwość, w której korzystamy z gotowego silnika i rozbudowujemy go o niezbędne dla nas elementy. To tak naprawdę najczęstszy scenariusz zarówno w małych studiach deweloperskich, jak i u dużych producentów gier wideo.

W każdym z wymienionych powyżej przypadków przydatna będzie znajomość podstawowej architektury silników gier wideo i elementów, z których się składają. Pomoże nam to w sposób przemyślany dokonać właściwego wyboru spośród wielu dostępnych na rynku lub z większą wiedzą zabrać się do projektowania w przypadku własnej implementacji.

2

Wieloplatformowy silnik gier wideo

2.1 Silnik gier wideo

2.1.1 Proces tworzenia gry wideo

Poziom skomplikowania przedsięwzięcia

Wielu spośród tych, którzy próbują profesjonalnie lub hobbystycznie zgłębiać tajniki programowania w wybranym języku programowania, prędzej czy później stawia przed sobą wyzwanie napisania bardziej lub mniej skomplikowanej gry komputerowej. Praktycznie każdy, kto miał kiedyś kontakt z komputerem, zagrał w swoim życiu w przynajmniej jedną grę, na przykład sławny już w biurach „Pasjans". Stworzenie aplikacji, jaką jest gra komputerowa, ma kilka konsekwencji. Przede wszystkim, będziemy tworzyli produkt, który będzie miał za zadanie wywołać u odbiorcy emocje. Wiele emocji będzie też udziałem samego autora, gdyż sam proces twórczy będzie nimi naznaczony. Ponadto będziemy musieli zmierzyć się z całą gamą wyzwań typowych jedynie dla gier. Wszystko to powoduje, iż programowanie gier jest znacząco różne od programowania aplikacji, ale dla wielu osób różne w pozytywnym aspekcie - ciekawsze, stanowiące wyzwanie i dające znacznie więcej satysfakcji.

Podczas pisania pierwszych gier zauważamy pewien bardzo istotny fakt - mianowicie, jest to proces bardzo czasochłonny. W zasadzie tworzenie gier wideo jest jednym z najbardziej złożonych i czasochłonnych procesów ze wszystkich rodzajów oprogramowania. Pisząc nasze pierwsze gry, które są najczęściej klonami jednej z prostych gier typu „Arkanoid", „Tetris", czy „Warcaby", możemy spędzić nad nimi kilka tygodni lub miesięcy, a nadal znalazłoby się coś, co można by dopracować. Oczywiście wraz z rozwojem naszych umiejętności będziemy potrafili tworzyć coraz sprawniej, lecz niewiele ratuje to sytuację, bo każda gra, której napisania się podejmiemy, będzie coraz bardziej skomplikowana. Będziemy wykorzystywać coraz więcej zagadnień z wszelkich dziedzin, zależnie od tematyki naszego projektu: matematyki, fizyki, grafiki komputerowej czy sztucznej inteligencji. Decydując się na każdy większy projekt, docho-

dzimy do nieuchronnego wniosku - jeżeli nie jest to nasz hobbystyczny pomysł z długoterminowym, kilkuletnim czasem realizacji, to potrzebujemy więcej programistów.

Architektura „Monolit"

W obecnych czasach większość komercyjnych gier wideo pisana jest przez zespoły kilku - kilkunastu programistów. Odległy jest już pionierski okres, kiedy gry powstawały „w garażu", a jeden pasjonat będący jednocześnie programistą, grafikiem i muzykiem nierzadko potrafił stworzyć produkt, który był konkurencyjny na ówczesnym rynku. Dzisiaj, gdy budżety głośnych gier wideo porównywalne są z budżetami w przemyśle filmowym, taka sytuacja jest rzadka i o wiele trudniejsza do osiągnięcia.

Praca nad wspólnym kodem jest kluczowym zagadnieniem inżynierii programowania i podczas pierwszych prób często przychodzi z trudem. Jednym z głównych powodów tego stanu jest fakt, że pierwsze programy młodych programistów (i to niekoniecznie gry komputerowe, ale programy w ogóle) często przypominają tak zwany „monolit".

Monolit jest to program, w którym nie można w łatwy sposób wydzielić sekcji odpowiedzialnych za poszczególne zadania, np. algorytmika, obsługa interfejsu użytkownika. Kod powstaje w miejscu, w którym akurat jest potrzebny, z wykorzystaniem wielu zmiennych globalnych. Każda funkcja w programie może odwoływać się do każdej innej funkcji, w związku z czym powstaje skomplikowana sieć połączeń, która wiąże w nierozerwalny sposób kod powstającego programu. Najczęściej kod znajduje się w jednym pliku źródłowym.

Monolit utrudnia tworzenie większych projektów, ponieważ:

- *Jest trudny w zarządzaniu.* Po pewnym czasie połączenia i zależności w programie stają się bardzo skomplikowane. W tym momencie każda modyfikacja lub rozbudowa funkcjonalności powoduje niezamierzone efekty w zupełnie innej części programu, prowadzące do bardzo trudnych w odnalezieniu błędów. Zależności w takim programie po pewnym czasie uniemożliwiają wręcz wprowadzenie jakichkolwiek modyfikacji bez przebudowy znacznej części kodu.
- *Bardzo utrudnione ponowne użycie kodu.* Opracowane przez nas rozwiązania i algorytmy powinny być użyte wszędzie tam, gdzie jest to tylko możliwe, zamiast być implementowane na nowo wtedy, gdy zajdzie potrzeba ich użycia. Dzięki temu oszczędzamy naprawdę wiele czasu. Dotyczy to zarówno wykorzystania pewnych funkcjonalności w tym projekcie, jak też stworzenia biblioteki z algorytmami dla przyszłych naszych projektów. Niestety, jest to niemożliwe, jeśli te rozwiązania są „zaszyte" gdzieś w głębi naszego kodu.
- *Trudna diagnostyka.* W każdym programie prędzej czy później pojawią się błędy. Monolit skutecznie utrudnia odnalezienie źródła błędnych danych powodujących błędną pracę programu. Kiedy ten błąd już odnajdziemy,

to ze względu na brak wielokrotnego użycia kodu bardzo prawdopodobne jest, że będziemy musieli poprawiać kod w wielu miejscach programu.

- *Niemożliwa współpraca.* Pisanie kodu w postaci monolitu przypomina budowanie domu z kart. Dodanie każdej kolejnej karty jest coraz trudniejsze i zwiększa prawdopodobieństwo widowiskowej katastrofy. Budowanie takiego domu przez więcej niż jedną osobę jednocześnie bardziej przypomina sztukę cyrkową, jeśli w ogóle jest wykonalne. Z tych samych powodów niemożliwa jest jednoczesna współpraca nad kodem, w którym „wszystko zależy od wszystkiego", i w dodatku kod całego programu znajduje się w jednym pliku źródłowym.

Pisanie pierwszych projektów zawsze jest pewnego rodzaju eksperymentem, gdzie chcemy sprawdzić jakieś nowe rozwiązania lub czego się ostatnio nauczyliśmy. W wyniku tego pisany kod jest prosty i nastawiony na szybki efekt. Po jego uzyskaniu do gry dopisywane są kolejne elementy i usprawnienia w miejscach gdzie po prostu jest to możliwe. W ten sposób bardzo szybko powstaje bardzo zawiły i nienajlepszy kod.

Powstawanie monolitów w początkowych etapach nauki wydaje się jednak nieuniknione. Większość podręczników do nauki programowania w dowolnym z języków w ten sposób prezentuje kod przykładowych programów i ćwiczeń. To samo dotyczy różnych samouczków uczących „programowania" z wykorzystaniem bibliotek jak OpenGL[1] czy DirectX[2]. W ten sposób łatwiej nauczyć się narzędzia, jakim jest język lub biblioteka, gdyż kod skupia się tylko na aspektach związanych z tym narzędziem.

Programowanie większych systemów jest jednak czymś więcej. Jest to sztuka wykorzystania tego narzędzia w celu uzyskania pożądanego efektu i utrzymania projektu w ryzach niezależnie od jego wielkości i liczby programistów nad nim jednocześnie pracujących.

Żeby uzyskać taki efekt, program musi być bardziej przemyślany. Przed przystąpieniem do implementacji warto poświęcić trochę czasu na jego zaprojektowanie. Włożona praca, mimo pozornie dodatkowego czasu, który musimy poświęcić, szybko zwróci się z nawiązką przy każdej późniejszej zmianie w kodzie czy dodawaniu nowej funkcjonalności. Modyfikacje staną się o wiele prostsze do wykonania i o wiele mniej inwazyjne w stosunku do pozostałych części programu.

Proces projektowania gry wideo

Każdy złożony system powinien być zaprojektowany przed rozpoczęciem jego powstawania. Dotyczy to jak najbardziej oprogramowania i gier wideo, które

[1] OpenGL - uniwersalny interfejs do generowania grafiki implementowany przez biblioteki na większości systemach operacyjnych - http:/www.opengl.org

[2] DirectX - interfejs do generowania grafiki, dźwięku i obsługi urządzeń wejściowych firmy Microsoft implementowany przez biblioteki dla systemów Windows i konsol Xbox 360 - http:/www.opengl.org

mogą być bardzo złożonymi systemami. Projekt jest pewnego rodzaju szablonem nakreślającym ogólną koncepcję i kierunek, w którym będziemy rozwijać nasze oprogramowanie. Szczególnie ważne jest to w trakcie pierwszych projektów, podczas których najprawdopodobniej będziemy badali różne możliwe rozwiązania. Na początku nie możemy określić, na które się zdecydujemy, lub będą one wymyślane na bieżąco podczas pisania kolejnych elementów gry. Ogólny szkielet pozwoli zachować nam porządek i jakiś kształt całości, niezależnie od tego, jaka będzie wersja ostateczna.

W trakcie programowania nieomalże samoistnie będzie się nam narzucać strategia „dziel i zwyciężaj”, z której warto skorzystać.

Definicja 2.1 - Strategia dziel i zwyciężaj

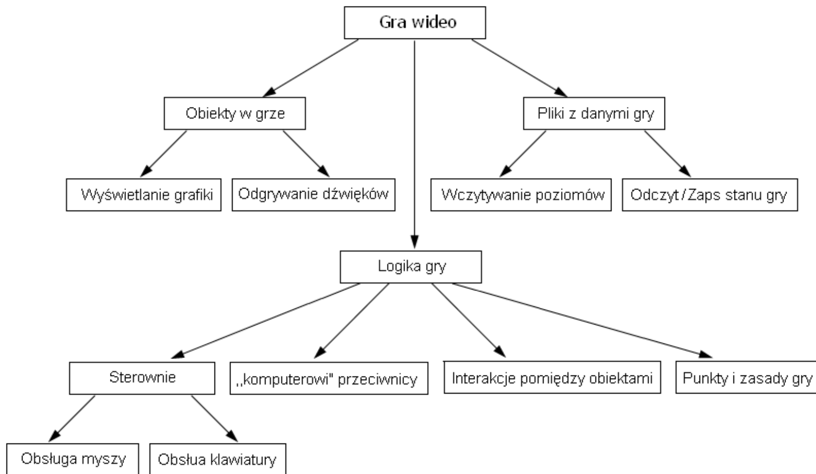
Strategia konstruowania oprogramowania „dziel i zwyciężaj” polega na rekurencyjnym podziale danego problemu na dwa lub więcej podproblemów, aż powstałe zadania będą wystarczająco małe, aby w szybki sposób rozwiązać je bezpośrednio.

W każdym programie możemy wyróżnić co najmniej kilka zupełnie odrębnych modułów funkcjonalnych. W przypadku prostej gry komputerowej mogą to być np.: sterowanie (obsługa klawiatury, myszy, gamepada, itp.), renderowanie[3] grafiki, obsługa dźwięku oraz logika gry. Każdy z tych elementów powinien być projektowany jako niezależny od innego i może zostać z pewnością podzielony na mniejsze fragmenty. W wyniku tych podziałów otrzymamy pewnego rodzaju hierarchię, która pozwoli nam podczas rozwiązywania danego problemu skupić się wyłączenie na nim. W przypadku, gdy będziemy chcieli usprawnić lub zmienić sposób działania któregoś z elementów, po prostu można go wymienić lub zmodyfikować, bez wpływu na inne.

Obecnie ogólnie przyjętym paradygmatem programowania dla większości zastosowań jest projektowanie obiektowe. Jest to podejście możliwie najbliższe opisowi rzeczywistości, którą chcemy za pomocą oprogramowania modelować. Bardzo dobrze sprawdzi się ono również przy projektowaniu gier komputerowych, która często przedstawia jakiś fragment rzeczywistego świata i w sposób praktycznie naturalny przekłada się na obiekty języka programowania. Także takie cechy jak np. enkapsulacja[4] i dziedziczenie ułatwiają stworzenie spójnej, ogólnej architektury naszego programu, niezależnej od faktycznej implementacji poszczególnych elementów, dzięki czemu w łatwy sposób będzie można podzielić pracę pomiędzy poszczególnych programistów.

[3] W grach wideo rendering jest to generowanie dwuwymiarowego obrazu stanowiącego pojedynczą klatkę animacji na podstawie danych opisu sceny takich jak modele znajdujących się na niej, opis oświetlenia itp.

[4] Enkapsulacja - zabronienie dostępu z zewnątrz do pól i metod klasy stanowiących szczegół implementacyjny, nieistotny z punktu widzenia interfejsu, z jakim inne obiekty komunikują się z obiektem tej klasy



Rysunek 2.1. Przykładowa hierarchia funkcjonalności w grze wideo

W pracy tej dla potrzeb projektowania będziemy wykorzystywać model obiektowy. Wygodnym sposobem opisu takiego modelu jest język UML, przy wykorzystaniu którego przedstawiać będziemy diagramy klas oraz inne elementy projektu.

Warstwowość oprogramowania

Projektowanie architektury z wykorzystaniem paradygmatu obiektowego samo w sobie nie gwarantuje nam porządku. Łatwo wyobrazić sobie sytuację, gdy w większym projekcie powstaje duża liczba klas, których powiązania ze sobą coraz bardziej się komplikują, co prowadzi do powstania nierozerwalnej całości i kodu, który trudno współdzielić pomiędzy różnymi projektami. Sposobem, który pomaga uniknąć tych problemów, jest architektura warstwowa.

Strategia „dziel i zwyciężaj" omawiana powyżej ma pewną niedogodność. Zgodnie z którą bardziej skomplikowane klasy, posiadające dużą funkcjonalność nie będą pisane całe „od zera", lecz będą implementowane z wykorzystaniem mniejszych klas, na które tę funkcjonalność podzieliliśmy podczas projektowania. Podział według tej strategii powoduje powstawanie struktury drzewiastej, gdzie liczba klas na poszczególnych poziomach będzie rosła w sposób wykładniczy. Przy większych projektach trudno jest zapanować nad porządkiem w niższych poziomach drzewa, gdzie poszczególne klasy po prostu muszą zawierać to, czego wymagają poziomy wyższe, często zawierając podobne lub wręcz powtarzające się fragmenty kodu.

Warstwowość jest cechą projektowania, która w przypadku gier wideo pojawia się w sposób naturalny. Uzupełnia ona doskonale strategię „dziel i zwyciężaj", skutecznie eliminując wspomnianą wcześniej wadę.

Definicja 2.2 - Oprogramowanie warstwowe

Oprogramowanie warstwowe zbudowane jest z pewnego rodzaju wyspecjalizowanych warstw logicznych, komunikujących się między sobą za pomocą ściśle określonych interfejsów.

Ideą warstwowości jest to, by nie dzielić funkcjonalności każdej klasy z osobna, lecz sporządzić pewną grupę mniejszych klas tak dobranych, aby stanowiły pewnego rodzaju „prefabrykaty”, z których będzie można składać obiekty większe. Taką grupą klas jest właśnie warstwa, która składa się z obiektów o podobnym poziomie abstrakcji i które będą wykorzystane dla tworzenia obiektów o wyższym poziomie abstrakcji - kolejnej warstwy.

Dzięki podziałowi na warstwy, liczba obiektów w niższych partiach drzewa oraz ich zawartość będzie bardziej przewidywalna i uporządkowana. Kolejną zaletą jest to, że funkcjonalności kolejnych warstw będą odseparowane. Każda kolejna warstwa zależąca od samej siebie i warstw niższych jest projektowana i implementowana niezależnie oraz rozwiązuje najczęściej problemy ze sobą powiązane. Wystawia pewien interfejs dla warstwy wyższej, ukrywając przed nią szczegóły implementacyjne.

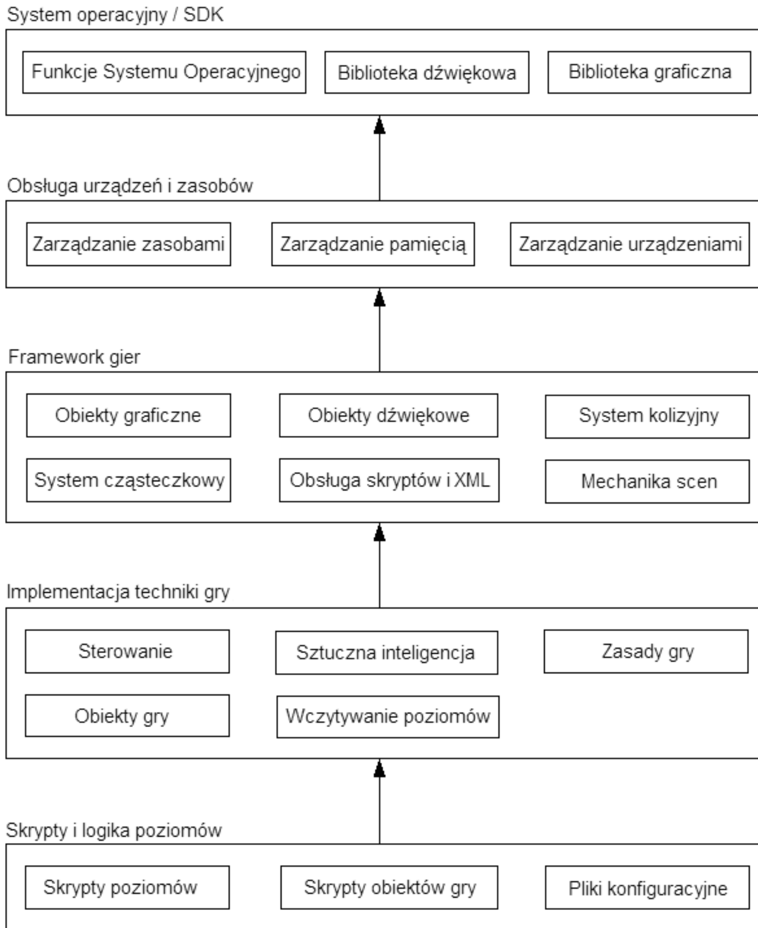
W przypadku gier wideo architektura warstwowa bardzo ułatwia ich projektowanie. Podział funkcjonalności na poszczególne warstwy jest zawsze „płynny” i dyskusyjny, lecz przykładowa architektura warstwowa, która wystarczy do stworzenia zaawansowanego projektu, mogłaby wyglądać w ten sposób:

- *System operacyjny/SDK*[5] - warstwa zawierająca funkcje i obiekty udostępniane przez system operacyjny pod który piszemy, jak również system graficzny (np. DirectX, OpenGL) i dźwiękowy (np. DirectSound[6], OpenAL[7]), których chcemy używać. Z tą warstwą zaczynamy pracę, decydując się na implementację pod daną platformę sprzętową i programową. Najczęściej mamy już ją gotową, lecz w niektórych, skrajnych przypadkach (np. pisanie emulatora innej platformy, implementacja programowego renderera grafiki) będzie trzeba ją odpowiednio dostosować.
- *Obsługa urządzeń i zasobów* - warstwa zarządzająca zasobami graficznymi, dźwiękowymi, pamięcią operacyjną oraz definiująca obiekty obsługi urządzeń wejściowych, pamięci masowej itp.
- *Framework gier* - warstwa zawierająca obiekty, które będą używane przez programistów projektujących i piszących konkretną grę, np. obiekty gra-

[5] SDK (Software Development Kit) - zestaw narządzi programistycznych, dokumentacji i bibliotek wspomagających tworzenie pewnego rodzaju oprogramowania

[6] DirectSound - interfejs służący do odtwarzania i rejestrowania dźwięku, będący częścią składową interfejsu DirectX

[7] OpenAL - uniwersalny interfejs do odtwarzania dźwięku implementowany przez biblioteki na większości dostępnych systemów operacyjnych - http:/www.openal.org



Rysunek 2.2. Przykładowy podział warstwowy gry wideo

ficzne 2D/3D, obiekty dźwiękowe, systemy obsługi kolizji, systemy cząsteczkowe itp.

- *Implementacja mechaniki gry* - pierwsza warstwa, w której implementowana jest konkretna gra wideo. W przypadku pierwszych gier pisanych przez początkujących programistów jest jedyną warstwą, oprócz SO/SDK, która zawiera w sobie wszystkie pozostałe, połączone w jedną całość. Warstwa ta powinna jednak zawierać tylko mechanikę gry. Jeśli w naszej grze poszczególne poziomy potrzebują dodatkowego oprogramowania, to ich implementacja powinna się znaleźć w kolejnej warstwie, natomiast w tej, pozostawić ogólne „reguły gry”.
- *Skrypty i logika poziomów* - warstwa ta zawierać będzie kod logiki poszczególnych poziomów w grze, jeżeli tego wymagają. Może być tworzona w tym

samym języku co kod mechaniki gry, lecz dobrym pomysłem jest użycie odrębnego języka skryptowego. Dzięki temu podczas każdej modyfikacji istniejącego czy po dołożeniu kolejnego poziomu nie będzie potrzebna rekompilacja całego projektu. Poza tym otrzymujemy wyraźną granicę pomiędzy logiką a mechaniką poziomu w grze. Skrypty nie mają praktycznie żadnej możliwości ingerencji w logikę zaimplementowaną w warstwach niższych, otrzymując tylko ściśle określoną funkcjonalność. Jest to bardzo ważna zaleta, która sprawia, że gra wideo staje się o wiele bardziej odporna na błędy popełniane podczas pisania skryptów. Dzięki temu ich implementację możemy bez obaw powierzyć osobie niebędącej programistą.

Specjalizacja programistów

Dzięki prawidłowej architekturze i strukturze warstwowej gra wideo może być pisana przez wielu programistów jednocześnie. Jest elastyczna pod względem wprowadzania zmian w każdej z warstw pod warunkiem tylko braku lub niewielkich zmian w interfejsie pomiędzy nimi.

Tutaj nasuwa się kolejny wniosek. Otóż tworzenie gry wideo jest jak najbardziej procesem wytwórczym, a w każdym takim procesie kluczem do uzyskania optymalnej wydajności i jakości produkcji jest specjalizacja pracowników. Podział warstwowy ułatwia i zarazem definiuje tę specjalizację. Programiści nie są dobierani w sposób dowolny do zadań związanych z implementacją danego projektu, lecz częściej specjalizują się do zadań w ramach którejś warstwy, w zależności od własnych preferencji czy umiejętności.

Pierwsze trzy warstwy są domeną programistów silnikowych (Engine programmers). Można powiedzieć, że nie są oni związani zbyt mocno z tym, jak wygląda tworzona gra wideo oraz z jej mechaniką. Ich zadaniem jest głównie dostarczenie jak najbardziej kompletnej i wydajnej funkcjonalności potrzebnej przy jej tworzeniu. Przy pierwszej warstwie (System Operacyjny/SDK) praca polega głównie na współpracy z pomocą techniczną dostarczyciela sprzętu i ewentualnym wprowadzaniu poprawek i aktualizacji. Kolejne dwie warstwy (Obsługa urządzeń i zasobów oraz Framework gier) mogą podzielić programistów silnikowych na niskopoziomowych i wysokopoziomowych (mowa oczywiście o poziomie abstrakcji, a nie poziomie umiejętności). Pierwsza z nich jest warstwą wewnętrzną, więc główny nacisk jest położony na wydajność oraz funkcjonalność rozwijanych rozwiązań, podczas gdy druga jest warstwą, z której korzystać będzie kolejna specjalizacja programistów, więc bardzo ważna jest także architektura i projekt udostępnionych funkcjonalności.

Drugą specjalizacją są programiści gier (Gameplay programmers). Korzystają z Frameworku i odpowiedzialni są za warstwę mechaniki gry. To od nich zależy, jak ta gra będzie wyglądać, działać i czy będzie miała przyjazne sterowanie (oczywiście przy współpracy z designerami, grafikami i dźwiękowcami). Dla tej grupy programistów ważniejsza jest umiejętność interdyscyplinarnego spojrzenia na projekt i zgrania wszystkich detali niż umiejętność znajdowania

najbardziej wydajnego rozwiązania. Tutaj również bardzo ważna jest architektura, która musi być elastyczna, gdyż jest to warstwa, która będzie najmocniej poddawana modyfikacjom podczas dodawania nowych elementów do gry lub ich usprawniania.

Ostatnią warstwą zajmują się skryptowcy (Scripters). Od nich zależy urozmaicenie i ciekawość poszczególnych poziomów w grze. Zaletą udostępnienia tej warstwy w postaci niezależnego od kodu gry skryptu jest to, że niekoniecznie muszą się nią zajmować programiści. Skrypty są najczęściej na tyle proste, że może je pisać osobna grupa projektantów poziomów, niekoniecznie obeznanych z językami programowania.

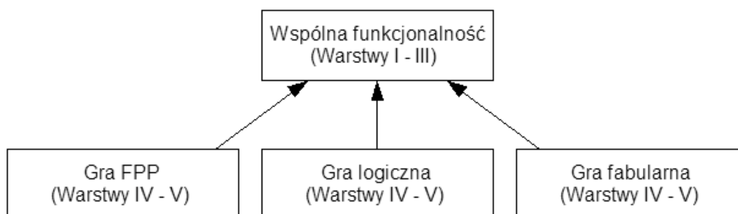
Ponowne wykorzystanie wspólnego kodu

Jednym z głównych założeń firm zajmujących się tworzeniem gier wideo jest wydanie przynajmniej kilku tytułów, a nie zakończenie działalności po pierwszym produkcie. Praca programistów stanowi znaczną część kosztów takiego projektu, dlatego niezależnie, czy strategią firmy jest tworzenie kilku projektów naraz, czy praca nad nimi kolejno, bardzo ważne jest ponowne wykorzystanie kodu.

Praktycznie wszystkie gry wideo, niezależnie od ich gatunku, wielkości czy tematyki będą posiadały znaczną część wspólną. Będzie to użycie pamięci, zasobów graficznych, zasobów dźwiękowych, wykorzystanie urządzenia wejściowego do sterowania czy też renderowanie grafiki 2D lub 3D. Oprócz pewnego zbioru funkcjonalności, które znajdą wykorzystanie w każdej grze wideo, można wydzielić zbiór takich, które będą występowały w projektach dosyć często, zależnie od ich gatunku.

Wspólna funkcjonalność pokrywa się z pierwszymi trzema warstwami zaprezentowanej architektury gry wideo, dlatego prostym sposobem na ponowne wykorzystanie kodu jest wydzielenie tych warstw do osobnego projektu, w ten sposób, żeby wszystkie istniejące projekty gier go współdzieliły. Wspólny projekt może być wykorzystany jako biblioteka dynamiczna, statyczna lub jego kod może być użyty bezpośrednio.

Wydzielenie wspólnej części ma wiele zalet. Każda dodana funkcjonalność lub poprawki błędów są jednocześnie dostępne we wszystkich innych projek-



Rysunek 2.3. Współdzielony projekt przez projekty gier wideo

tach. Część wspólna będzie ewoluowała i rozwijała się na doświadczeniu kolejnych gier wideo, tworzonych z jego wykorzystaniem, dzięki czemu będzie miała coraz mniej błędów, będzie coraz wydajniejsza i bardziej kompletna. Ważną zaletą jest to, że będzie niezmienna dla programistów korzystających z warstwy Framework. Dzięki temu każdy programista gier będzie mógł szybko rozpocząć pracę w dowolnym projekcie, bez czasochłonnego przestawiania się na inną technologię.

2.1.2 Idea silnika gier wideo

Silnik obejmuje drugą i trzecią warstwę architektury (Obsługa urządzeń i zasobów oraz Framework gier) napisanych na podstawie warstwy pierwszej, czyli przyjętego systemu operacyjnego oraz SDK graficznego. Możemy je implementować sami, lecz jeśli znajdziemy gotowy podsystem, którego architektura nam odpowiada, a licencja pozwala nam na jego użycie, możemy zawrzeć go w swoim silniku.

Definicja 2.3 - Silnik gier wideo

Silnik gier wideo jest to system oprogramowania, zaprojektowany w celu wspomagania projektowania i tworzenia gier wideo.

Przykładowy zbiór podsystemów silnika gier wideo może wyglądać tak:

Podsystemy niskopoziomowe (Core)

Alokator pamięci operacyjnej - użycie systemowego alokatora pamięci operacyjnej nie zawsze jest najlepszym rozwiązaniem. Systemowy alokator poprzez ogromną ilość wpisów, jaką musi zarządzać, oraz przez kontrole spójności pamięci, jaką wykonuje podczas jej rezerwacji i zwalniania, jest relatywnie wolny, co może mieć istotny wpływ na wydajność gry wideo, jeśli potrzebujemy alokacji dużej ilości małych bloków pamięci. Innym problemem jest fragmentacja pamięci, która może pojawić się po pewnym czasie działania aplikacji na urządzeniach posiadających tej pamięci niewiele, bez systemu stronicowania. Efektem fragmentacji jest brak możliwości zarezerwowania większych bloków pamięci, co może się po prostu skończyć zawieszeniem naszego programu. Dobry alokator pamięci specjalizowany pod względem szybkości alokacji bądź przeciwdziałający fragmentacji może być kluczowy pod względem stabilności, jak też wydajności całego silnika. Należy jednak pamiętać, że źle napisany własny alokator pamięci będzie zazwyczaj mniej wydajny od alokatora systemowego, jednocześnie stanowiąc potencjalne źródło poważnych błędów.

Menadżer zasobów - zasobami w silniku gier są wszelkie dane odczytywane z pamięci masowej (dysku twardego, DVD), przesyłane przez sieć lub też

generowane dynamicznie, które są potrzebne dla funkcjonowania różnych obiektów silnikowych i mogą być przez nie współdzielone. Mogą to być np. tekstury, dane dźwiękowe, animacje, siatki trójkątów itp. Rolą menadżera zasobów jest dbanie, aby wszelkie zasoby były dostępne w pamięci, kiedy obiekty silnikowe będą ich potrzebowały. Obejmuje to wczytywanie zasobów do pamięci, strumieniowanie dużych zasobów, usuwanie niepotrzebnych zasobów czy też cache'owanie zasobów.

Obsługa pamięci masowej - system obejmuje obsługę blokowego oraz strumieniowego zapisu i odczytu danych z urządzeń pamięci masowych. W zależności od platformy, której dotyczy silnik gier wideo, może to być dysk twardy, napęd DVD, pamięć flash, kartridż.

Obsługa urządzeń wejściowych - system obsługujący podłączanie/odłączanie urządzeń wejściowych i pobieranie danych z nich otrzymywanych. Urządzenia wejściowe obejmują w zależności od platformy mysz, klawiaturę, gamepad, joystick oraz inne akcesoria w postaci kierownicy, pistoletu świetlnego itp.

Obsługa urządzeń dźwiękowych - system odgrywania dźwięków w różnych formatach, obsługa miksera dźwiękowego, DSP, efektów dźwiękowych oraz dźwięku wielokanałowego.

Renderer - w silniku gier wideo obsługuje wszystkie urządzenia renderujące (karty graficzne). Zadaniem renderera jest obsługa kamer w trybach 2D i 3D, lecz przede wszystkim rysowanie wszelkich silnikowych obiektów graficznych, w taki sposób, aby zostały wyświetlone poprawnie. Obejmuje to określanie widoczności obiektów w każdej klatce, rysowanie ich w odpowiedniej kolejności oraz z odpowiednimi ustawieniami urządzenia renderującego.

Podsystemy wysokopoziomowe (Framework)

System kolizyjny - jest podstawowym elementem silnika, dzięki któremu można określić wystąpienie „fizycznych” interakcji pomiędzy obiektami na scenie. System ten pozwala z dowolnie przyjętą dokładnością ustalić, czy dwa dowolne obiekty zachodzą na siebie oraz kiedy ta kolizja nastąpiła. Dzięki niemu jest możliwa implementacja wszelkich uderzeń, trafień pociskami, ale także poruszania się po terenie lądowym i ograniczeń ruchu przez ściany.

System fizyczny - pozwala symulować interakcje pomiędzy obiektami na scenie zgodnie z prawami fizyki. Obiektom przypisuje się cechy, takie jak masa, kształt, prędkość liniowa i kątowa. Dzięki temu mogą na siebie oddziaływać podczas zderzeń. Bazą dla systemu fizycznego jest system kolizyjny.

System sztucznej inteligencji - Jest to zbiór algorytmów i narzędzi służących do podejmowania decyzji przez komputerowego gracza. W ich skład wchodzą algorytmy do wyszukiwania drogi, rozpoznawania kształtów podejmowania decyzji taktycznych. Algorytmy do rozwiązywania tych problemów

stosują technologie takie jak: logika rozmyta, sieci neuronowe, systemy ekspertowe, maszyny stanów, itp.

Parsery[8] i interpretery języków skryptowych i XML - Parser języka XML wspiera obsługę plików danych w tym formacie, które mogą zawierać ustawienia konfiguracyjne gry, obiektów graficznych, materiałów, a nawet dane całych siatek trójkątów obiektów 3D. Dokumenty XML są także wykorzystywane przy przesyłaniu komunikatów poprzez sieć. Interpretery języków skryptowych, takie jak np. LUA wspomagają tworzenie skryptów z zewnętrzną logiką, którą może być na przykład kod poziomu gry.

System cząsteczkowy - system służący do modelowania i renderowania efektów graficznych, takich jak ogień, dym lub opady atmosferyczne. System ten wykorzystuje dużą ilość punktowych cząstek, które ulegają interakcji między sobą, z otoczeniem, a także ulegają sile wirtualnej grawitacji. Renderowane są jako pojedyncze sprite'y[9] i pożądany efekt powstaje poprzez odpowiednie nagromadzenie dużej ich ilości w jednym miejscu ekranu.

Zbiór obiektów graficznych - Jest to zbiór obiektów, które potrafi narysować renderer i z których może korzystać programista w celu utworzenia sceny. W ich skład wchodzą takie obiekty jak: kamery, światła, sprite'y, obiekty tekstowe, modele 3D statyczne i animowane, systemy renderowania chmur itp.

Zbiór obiektów dźwiękowych - Jest to zbiór obiektów korzystających z urządzeń dźwiękowych, przeznaczonych do umieszczenia na scenie przez programistę. Obiekty te symulują fizyczne istnienie źródła dźwięku w przestrzeni sceny zgodnie z prawami fizyki. Na przykład poruszając w wirtualnej przestrzeni takim źródłem dźwięku, automatycznie zostanie zastosowany odpowiednio efekt Dopplera i pozycjonowanie dźwięku, korzystając z głośników.

System obsługi lokalizacji - Większość dzisiejszych gier wideo jest wydawana przynajmniej w kilku wersjach językowych. System lokalizacji zarządza danymi zależnymi od wersji językowej (dane tekstowe, dźwiękowe, graficzne) i dostarcza odpowiednią ich wersję w zależności od aktualnie wybranej konfiguracji języka w grze.

Interakcja z użytkownikiem

Jednym ze sposobów klasyfikacji silników gier jest to, w jaki sposób zrealizowana jest interakcja z użytkownikiem, czyli osobą, która za jego pomocą chce

[8] Parser jest to program analizujący tekst w celu rozpoznania jego struktury i zgodności z pewną gramatyką formalną. Najczęściej stosowany jest do analizy języków programowania lub języków opisu danych takich jak XML.

[9] Sprite - w grach wideo jest to dwuwymiarowy obraz, który po ewentualnych przekształceniach takich jak translacja, skalowanie i rotacja jest przenoszony na ekran, stanowiący podstawowy element graficzny gier 2D lub element interfejsu gier 3D.

stworzyć grę wideo. Użytkownikiem silnika wcale nie muszą być programiści - mogą to być designerzy, graficy lub też cały zespół. Sposób interakcji definiuje, kto nim będzie i do kogo jest skierowany.

Silnik w postaci udostępnionego SDK

Jednym z możliwych podejść jest zaprojektowanie silnika jako bibliotekę lub też zbiór bibliotek jakiegoś języka programowania udostępnionych w postaci SDK, czyli kodu biblioteki, dokumentacji i przykładowych kodów źródłowych. Jest to rozwiązanie bardzo intuicyjne, wywodzące się z wydzielenia wspólnej funkcjonalności gier do osobnej biblioteki. W tym podejściu pracę nad grą wideo rozpoczynamy od stworzenia aplikacji w pewnym języku programowania. Najczęściej jest to ten sam język programowania, w którym został stworzony silnik, jednakże nie jest to konieczne - ważne jest natomiast, by wybrany język potrafił z tych bibliotek skorzystać. Następnie dołączamy biblioteki silnika do projektu i rozpoczynamy implementację gry, w czym pomagają nam zawarte w nich elementy.

Użytkownikami tego typu silników, co jest oczywiste, są prawie wyłącznie programiści, gdyż każdy element tworzonej gry trzeba po prostu zakodować. W przypadku większych projektów jest jak najbardziej wskazane, by udostępnić możliwość użycia zewnętrznych skryptów i plików konfiguracyjnych, lecz funkcjonalność ta musi być zrealizowana po stronie gry.

Silniki tego typu są bardzo elastyczne. To programista decyduje, czy użyje danej funkcjonalności silnika, czy też w danym przypadku może lepiej będzie ją zmodyfikować lub zaimplementować samemu. W bardzo łatwy sposób można też rozbudowywać taki silnik o kolejne moduły, będące po prostu kolejnymi bibliotekami. Dzięki temu podczas tworzenia gry ograniczają nas tylko wyobraźnia, możliwości sprzętu i budżet.

Wadą takiego rozwiązania jest jednak dość duża pracochłonność przy tworzenia projektu - stworzenie nawet prostego prototypu może zająć dużo czasu. Dostosowywanie gry pod względem np. wizualnym powinno angażować tylko grafika, niestety przy silniku w postaci SDK zazwyczaj niezbędny będzie programista. Sytuację poprawi dbanie o udostępnianie narzędzi i plików konfiguracyjnych, lecz to jest kolejny element, który trzeba przewidzieć i napisać. To powoduje, że w projekcie potrzebnych jest wielu programistów, co oczywiście kosztuje.

Środowisko tworzenia gier wideo - silnik edytorowy

Silniki tego typu można by określić mianem edytorów do tworzenia gier. Pracując z nimi, nie będziemy musieli napisać choćby linijki w żadnym natywnym języku programowania. Silniki te składają się z wielu różnego rodzaju edytorów i narzędzi służących do rozkładania i ustawiania parametrów obiektów na

scenie, edycji modeli i materiałów. Logika gry najczęściej realizowana jest poprzez różne języki skryptowe lub „wyklikiwane” schematy blokowe.

Silnik jest właściwie interpreterem danych w postaci skryptów, plików opisu sceny i wszelkiego rodzaju innych zasobów. Z punktu widzenia użytkownika nie jest ważna jego architektura wewnętrzna. Sam silnik jest czarną skrzynką, a jedyne, co użytkownik może zobaczyć, to interfejs, który jest udostępniony przez edytory i narzędzia.

Taki rodzaj silników dosyć jasno określa swoich potencjalnych użytkowników. Będą to designerzy, skrypterzy, graficy, dźwiękowcy, którzy mają pomysł na grę, lecz nie mają możliwości lub nie chcą skupiać się na pracy programistycznej.

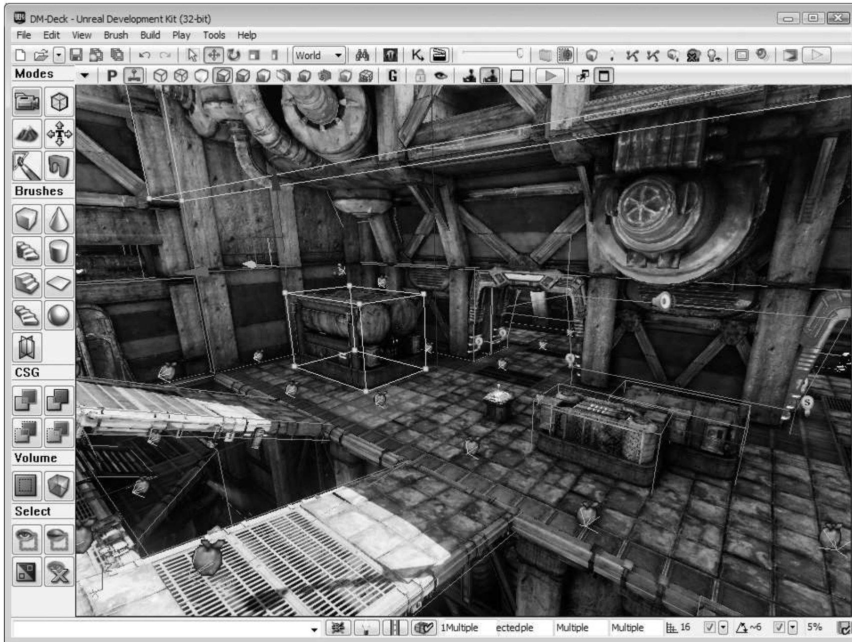
Napisanie tego rodzaju silnika jest jednak sporym wyzwaniem. Musi być on dobrze zaprojektowany, a w szczególności dotyczy to interfejsów edytorów i narzędzi. Nie ma nic bardziej frustrującego, niż brak możliwości wykonania jakiejś prostej czynności lub zrealizowania swojego pomysłu na jakiś element gry tylko dlatego, że projektanci tego nie przewidzieli i jest to po prostu niewykonalne. Jeżeli silnik taki ma być narzędziem uniwersalnym do pracy nad dowolnym gatunkiem gier, to musi być też bardzo rozbudowany. Gry wideo mogą zawierać ogromne ilości różnych elementów, z których każdy może być zrealizowany na wiele sposobów. Silnik taki będzie tym lepszy im więcej będzie ich zawierał, gdyż twórca gry będzie zazwyczaj mógł składać grę tylko z tych elementów, które znajdzie w jego interfejsie.

Niewątpliwą zaletą tego rodzaju silników jest możliwość szybkiego tworzenia, w szczególności bardzo szybkiego tworzenia prototypów. Osoba z nim dobrze zaznajomiona może w prosty sposób „wyklikać” kilka prototypów danego elementu gry, z których najlepszy zostanie wybrany do rozbudowy do wersji finalnej.

Silnik taki zawsze jednak będzie mniej elastyczny niż silnik dostępny w postaci biblioteki programistycznej. Oprócz możliwości sprzętowych przez cały czas ogranicza nas skończony zbiór elementów, który się w nim znalazł. W najbardziej nawet rozbudowanym silniku zawsze jednak czegoś zabraknie, gdyż jego projektanci uznali to za zbędne lub wręcz nieopłacalne. Problemem może być również ich wydajność. Po pierwsze ze względu na logikę gry, która w całości będzie interpretowana, w postaci skryptów lub różnego rodzaju grafów, lecz nie będzie szybkim kodem natywnym. Z tego powodu można także zapomnieć o jakiejś bardziej złożonej algorytmice w tych skryptach, które po prostu zajęłyby zbyt dużo mocy obliczeniowej procesora i trzeba by zdać się na algorytmy umieszczone w silniku. Drugim powodem niższej wydajności może być sama idea takiego silnika. Gra wideo budowana jest tutaj z dużych prefabrykatów, które niestety nie zawsze w najbardziej odpowiedni sposób są do niej dopasowane.

Przykładowym silnikiem gier wideo tego typu jest znany „Unreal Engine”, z wykorzystaniem którego powstała ogromna ilość gier wideo, także tych naj-

głośniejszych tytułów. Do zastosowań niekomercyjnych jest on dostępny nieodpłatnie w postaci UDK (*Unreal Development Kit*[10]), dzięki czemu każdy chcący spróbować swych sił ma możliwość się z nim zmierzyć.



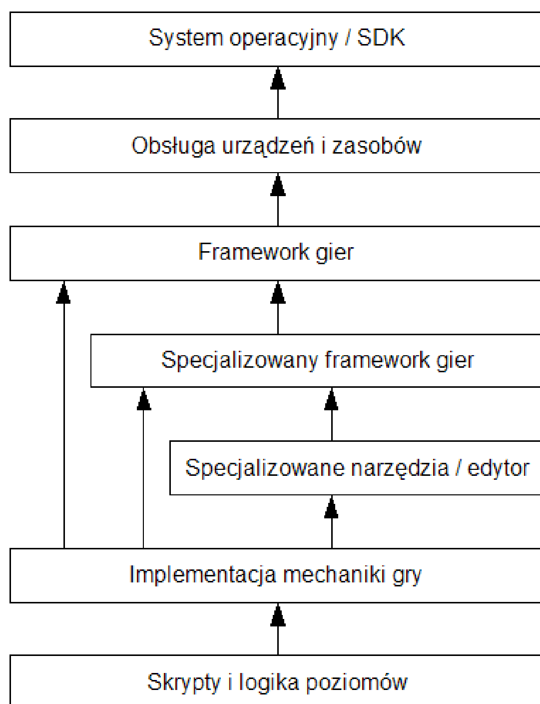
Rysunek 2.4. Przykładowy silnik edytorowy - Unreal Engine 3

UDK jest typowym silnikiem edytorowym, w którym podczas tworzenia gry wideo nie napiszemy ani jednej linii kodu w natywnym języku programowania. Cały proces powstawania projektu zachodzi w rozbudowanym edytorze, posiadającym ogromne możliwości, lecz liczba dostępnych w nim opcji może także przytłoczyć podczas pierwszego podejścia. Za logikę powstającej gry odpowiada język skryptowy „*UnrealScript Language*”, stworzony specjalnie na potrzeby tego silnika.

Podejście mieszane

Jeśli w tej klasyfikacji są tak skrajne rozwiązania, jak powyższe, posiadające przeciwstawne wady oraz zalety, to oczywiste jest, że pojawią się rozwiązania mieszane. Silniki takie starają się łączyć elastyczność silników bibliotecznych z szybkością i łatwością pracy silników edytorowych.

[10] Unreal Engine 3 (Unreal Development Kit) - http:/www.udk.com



Rysunek 2.5. Warstwy silnika specjalizowanego

Rozwiązania realizujące takie podejście mogą być różne. Jednym z przykładów może być silnik, którego podstawa oparta jest na bibliotekach udostępnionych w postaci SDK. Na tej podstawie zaimplementowanych jest wiele narządzi, służących do szybkiego stworzenia prototypu, które wystarczają do najczęściej realizowanych zastosowań. Bardziej rozbudowane i specjalistyczne elementy można dodawać dzięki udostępnianemu SDK jako niezależne moduły lub wtyczki do istniejących narzędzi.

Poziom abstrakcji silnika

Definicja 2.4 - Poziom abstrakcji

Poziom abstrakcji jest to poziom złożoności danego systemu. Im jest niższy, tym więcej detali potrafimy dostrzec i rozróżnić, a im jest wyższy, tym o większych i bardziej złożonych obiektach mówimy, ale nie wgłębiając się w szczegóły.

Poziom abstrakcji silnika gier wideo określa, na jakim poziomie abstrakcji będzie pracował jego użytkownik i z jakimi obiektami będzie miał do czynienia. Silnik na niskim poziomie będzie głównie zawierał obiekty proste, których dopiero programista gry użyje do modelowania bardziej złożonych obiektów. Przykładowo mogłyby to być podstawowe obiekty graficzne, takie jak Tekstura, Sprite, Model 3D. Natomiast silnik na wysokim poziomie będzie zawierał obiekty bardziej złożone, łatwiejsze w użyciu, gdyż będą już w pewien sposób przygotowane. Z drugiej strony odbija się to na ich mniejszej elastyczności. Przykładem mógłby tu być obiekt modelu człowieka z przygotowaną obsługą animacji szkieletowych, systemem poruszania się po terenie oraz możliwością podpięcia dźwięków pod predefiniowane akcje.

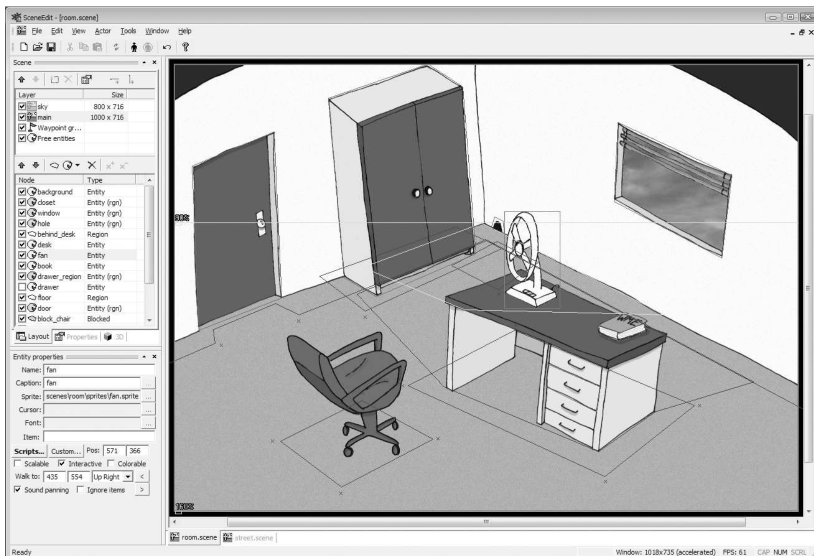
Można z tego wyciągnąć wniosek, że im silnik jest przeznaczony do bardziej ogólnych zastosowań, tym jego poziom abstrakcji jest niższy i analogicznie rośnie wraz z jego ukierunkowaniem pod któryś z gatunków gier wideo. Jest to prawdą, lecz trzeba też zauważyć, że silnik może mieć kilka poziomów abstrakcji. Najłatwiej o to w silniku udostępnianym jako SDK, gdzie poziomy abstrakcji powstają samoistnie wraz z architekturą warstwową oraz mechanizmami dziedziczenia i polimorfizmu. Trudniej o taki wachlarz poziomów w silnikach edytorowych, najczęściej nie ma tam najprostszych obiektów. Wynika to z faktu, że ogromna ich liczba musiałaby być stworzona na scenie, co utrudniłoby zarządzanie nimi z poziomu edytora oraz wpłynęłoby na wydajność takiego silnika, gdyż obsługa każdego obiektu w silniku edytorowym wprowadza dość spory narzut obliczeniowy.

Silniki specjalizowane

Skoro wraz z poziomem abstrakcji silnika gry wideo rośnie jego ukierunkowanie pod konkretny gatunek gier, to stąd jest już tylko krok do silników specjalizowanych. Gry wideo, które klasyfikujemy w ramach jednego gatunku, posiadają wiele cech wspólnych. Wykorzystują podobne obiekty, działają według podobnej mechaniki i w podobny sposób wykorzystują zasoby sprzętowe. Silnik specjalizowany zawiera elementy ułatwiające tworzenie pewnego gatunku gier lub w pełni skupia się na tym zadaniu. Oczywiście najważniejszy cel ich istnienia to przyspieszenie procesu implementacji gry.

Silniki specjalizowane mogą powstać poprzez rozbudowę silnika ogólnego. Po pierwsze sprowadza się to do dopisania do niego kolejnej warstwy rozszerzającej jego funkcjonalność, zawierającej wysokopoziomowe obiekty i algorytmy. Po drugie rozszerzany jest zestaw narzędzi silnikowych, głównie o wyspecjalizowane edytory pracujące przeważnie na obiektach dostarczonych przez tę nową warstwę.

Innym rodzajem specjalizacji są silniki tworzone od początku z myślą o danym gatunku. Są to najczęściej silniki edytorowe. Interfejs udostępniony użytkownikowi to właśnie wysoko wyspecjalizowany edytor, który dzięki temu, że służy wyłącznie do stworzenia konkretnego gatunku, pozwala na bardzo szyb-



Rysunek 2.6. Edytor sceny silnika Wintermute Engine

kie stworzenie gry poprzez „wyklikanie” oraz dostarczenie ewentualnych skryptów zawierających logikę. Sam silnik w tym wypadku jest tylko interpreterem plików danych otrzymanych z tego edytora. Dzięki wąskiej specjalizacji może jednak być bardzo wydajny.

Przykładowy silnik specjalizowany - Wintermute Engine[11]

Przykładem takiego typu silnika jest „*Wintermute Engine Development Kit*”, darmowy silnik służący do tworzenia gier przygodowych w technologii 2D oraz 2.5D (trójwymiarowe postacie oraz dwuwymiarowe tła sceny i obiekty).

Jest on właściwie wysoce wyspecjalizowanym edytorem, stworzonym z myślą o tworzeniu gier przygodowych i to w dodatku według zaledwie jednej mechaniki „Wskaż i kliknij” (Point & Click[12]) - opartej na warstwach złożonych z dwuwymiarowych teł i obiektów interaktywnych z ewentualnymi trójwymiarowymi postaciami. Silne wyspecjalizowanie jednak ogranicza, nie stworzymy w tym silniku gry innego gatunku, lecz nie o to przecież chodzi. Jego największą zaletą jest to, że jest niewielki, dzięki czemu jest prosty w obsłudze i szybki do nauczenia. Powoduje to, że proces tworzenia gry wideo lub jej prototypu jest bardzo szybki, co oczywiście ogranicza koszty jej powstawania.

[11] Wintermute Engine Development Kit - http:/dead-code.org

[12] „Point & Click” jest to typ gry przygodowej, w której sterowanie rozwiązane jest za pomocą urządzenia wskazującego (np. interfejs dotykowy lub mysz). Za jego pomocą wskazuje się punkt na scenie, gdzie powinien przejść bohater gry lub przedmiot, na którym ma wykonać pewną akcję.

Różnego rodzaju akcje na przedmiotach oraz logikę gry w silniku *Wintermute* oprogramowuje się za pomocą języka skryptowego „WME Script”, który oparty jest na języku JavaScript.

Poniżej pokazany jest przykładowy wycinek skryptu opisującego obiekt wentylatora na scenie, definiujący reakcję gry na akcje „Obejrzyj” oraz „Weź” wykonane na tym obiekcie.

```
#include "scripts\base.inc"

global StateRoom;

on "LookAt"
{
  actor.GoToObject(this);
  actor.Talk("It's a fan. I can change its spinning speed.");
}

on "Take"
{
  actor.GoToObject(this);
  Game.Interactive = false;

  // get the fan entity
  var EntFan = Scene.GetNode("fan");

  // set the fan etity sprite depending on the state variable
  // (0-off, 1-normal, 2-false)
  if(StateRoom.FanSpeed==0)
  {
    actor.Talk("It's turned off. I'll turn it on...");
    actor.PlayAnim("actors\molly\ur\take1.sprite");
    EntFan.PlaySound("sounds\fan_start.ogg");
    actor.PlayAnim("actors\molly\ur\take2.sprite");
    Sleep(1400);
    EntFan.SetSprite("scenes\room\sprites\fan.sprite");
    StateRoom.FanSpeed = 1; / save the new fan state
  }

  // ...
}
```

Listing 2.1-1 - Przykładowy skrypt akcji wentylatora w języku WME Script

Narzędzia silnikowe

Różnego rodzaju narzędzia silnikowe są bardzo ważnym elementem obecnych silników gier wideo. Ułatwiają i przyspieszają tworzenie różnych elementów

gier, a jednocześnie umożliwiają podział pracy pomiędzy osoby tworzące grę niebędące programistami.

Pierwszą grupą stanowią narzędzia diagnostyczne, przeznaczone głównie dla programistów, poprawiające kontrolę nad stabilnością, wydajnością i wykorzystaniem zasobów przez grę wideo. Najważniejsze z nich to:

debugger - podstawowe narzędzie diagnostyczne każdego programisty pozwalające na krokową pracę z programem, podgląd w dowolnym momencie zawartości pamięci, zmiennych oraz ścieżki wykonywanego algorytmu,

konsola diagnostyczna - pozwalająca wyprowadzać w postaci tekstowej informacje diagnostyczne z dowolnego miejsca gry,

wykrywacz wycieków pamięci - wykrywa niezwolnione obszary pamięci, o które bardzo ławo w większych projektach,

profiler - narzędzie umożliwiające ocenę wykorzystania mocy obliczeniowej procesora przez określone fragmenty kodu i funkcje, dzięki czemu można wykryć i zoptymalizować najbardziej obciążające system fragmenty,

monitor zasobów - narzędzie służące do monitorowania w dowolnym momencie załadowanych przez silnik zasobów i stopnia ich wykorzystania.

Drugą grupą są narzędzia do edycji zasobów, które będą ładowane przez silnik. Silniki gier wideo posiadają często własne formaty danych dla różnych rodzajów zasobów, dzięki czemu jest możliwe ich wczytanie i optymalne wykorzystanie. Dzięki tym narzędziom osoby niebędące programistami mogą edytować zasoby gry. W skład takich narzędzi mogą wchodzić na przykład edytory modeli 3D, animacji, materiałów, shaderów[13], czcionek itp.

Inną grupą narzędzi są edytory scen i poziomów, dzięki którym w szybki sposób można rozmieścić obiekty na scenie oraz przypisać im skrypty i interakcje. Edytory tego typu najczęściej są elementem silników edytorowych, lecz często są również implementowane dla potrzeb edycji poziomów konkretnej gry wideo.

2.2 Idea wieloplatformowego silnika gier wideo

2.2.1 Tworzenie gry wideo na urządzenie niebędące komputerem

Proces powstawania oprogramowania

Pierwsze podejścia do pisania gier wideo mają przeważnie na celu stworzenie gry działającej na komputerze PC z jego najpopularniejszym obecnie systemem operacyjnym MS Windows. Oczywiście ta konfiguracja nie jest jedyną,

[13] Shader jest to program wykonywany przez procesor graficzny podczas procesu renderingu, obliczający kolor pikseli powstającego podczas tego procesu obrazu dwuwymiarowego na podstawie danych o geometrii, materiałach i oświetleniu.

na którą tworzone są gry wideo. Każdy system operacyjny ma zawsze trochę inny zestaw funkcji, które udostępnia programiście, w związku z czym nawet w przypadku gry wyglądającej identycznie na różnych systemach, ich kod będzie się różnił. Rodzaj i liczba tych różnic będzie w znacznym stopniu zależna od języka i środowiska, które zostały wybrane do tworzenia danej gry wideo. W znakomitej większości przypadków nie będzie możliwe użycie tego samego kodu na różnych systemach bez przynajmniej częściowych modyfikacji.

Ta różnorodność nie kończy się na systemach operacyjnych dla komputerów osobistych. Gry wideo przecież są tworzone także na wiele innych urządzeń, zaczynając od telefonów komórkowych, a na różnego rodzaju konsolach przenośnych czy stacjonarnych kończąc. Każde z tych urządzeń jest obsługiwane przez bardziej lub mniej rozbudowany system operacyjny nim zarządzający. Połączenie sprzętu oraz systemu operacyjnego tworzy odrębną platformę, na którą może powstawać oprogramowanie, w tym gry wideo.

Definicja 2.5 - Oprogramowanie warstwowe

Platforma jest to zestawienie architektury sprzętowej i interfejsu programowego umożliwiające uruchamianie oprogramowania.

Pojawia się tutaj jednak pewien problem. O ile użytkowanie gier wideo jest wykonalne na każdym z tych urządzeń, bo w końcu w takim celu zostały napisane, to sam proces tworzenia gry nie na wszystkich z nich jest w ogóle możliwy. Przykładowo gdy chcemy stworzyć grę komputerową na platformę Linux, najmądrzejszym rozwiązaniem jest pisanie jej na komputerze z takim właśnie systemem operacyjnym. W trakcie procesu powstawania gry będziemy setki razy ją uruchamiać i testować, a praca na platformie, która jednocześnie jest docelowa, pozwoli zrobić to szybko i sprawnie. W przypadku tworzenia gry wideo na konsole sprawa jest jednak trudniejsza. Można by sobie wyobrazić jakieś narządzie lub edytor uruchamiane bezpośrednio na konsoli, pozwalające dopracowywać różnego rodzaju parametry gry lub nawet tworzyć całe poziomy lub ich elementy. Jednak tworzenie całej mechaniki gry, w szczególności jej niższych warstw nie jest możliwe, gdyż byłoby to bardzo trudne do zrealizowania i po prostu niewygodne. W przypadku konsol przenośnych lub telefonów komórkowych tworzenie gier tylko z ich wykorzystaniem jest trudne do wyobrażenia.

Można z tego wyciągnąć jeden wniosek. Praca przy tworzeniu gier wideo praktycznie zawsze odbywa się z wykorzystaniem komputerów, niezależnie od tego, na jaką platformę docelową powstają. Pojawia się tutaj jednak kolejny problem - tworzone oprogramowanie trzeba oczywiście prędzej czy później uruchomić. W przypadku wykorzystania kompilowanego języka programowania potrzebne jest utworzenie pliku wykonywalnego. Kompilator, którego będziemy używać na komputerze, musi mieć możliwość zbudowania binarnego pliku wykonywalnego na platformę, na którą tworzymy, czyli inną niż ta, na

której jest uruchamiany. Taki tryb pracy nazywamy kompilacją skrośną (*cross compilation*) i jest on obecnie wspierany przez wiele różnych kompilatorów. Przykładowo darmowy kompilator GCC[14], który jest dostępny na większość komputerowych systemów operacyjnych, posiada możliwość kompilacji skrośnej na inne systemy, a także na inne procesory i urządzenia.

Uruchamianie produktu na urządzeniu docelowym

Kiedy mamy już plik wykonywalny, możemy go uruchomić na platformie docelowej. Rozwiązań, w jaki sposób można to zrobić, jest kilka, w zależności od tego, jaki sprzęt posiadamy.

- *Przeniesienie na platformę docelową*

  Pierwszy i chyba najbardziej intuicyjny sposób uruchomiania gry wideo tworzonej na innej platformie to przeniesienie pliku wykonywalnego oraz plików danych na urządzenie docelowe. W zależności od urządzenia może to wyglądać w różny sposób. Przykładowo dla aplikacji Java na telefony komórkowe będzie to polegało na utworzeniu pliku stanowiącego „pakiet” instalacyjny (pliki JAR - *Java ARchive*), który możemy utworzyć na komputerze, korzystając z dostarczonych narządzi deweloperskich, następnie przenieść do pamięci telefonu i uruchomić. W przypadku konsol zarówno stacjonarnych, jak i przenośnych, aplikacje oraz gry uruchamia się, przeważnie korzystając z nośników zewnętrznych. Dla konsol stacjonarnych będzie to najczęściej nośnik optyczny w postaci DVD lub Bluray, natomiast dla konsol przenośnych nośnikiem danych są głównie różnego rodzaju kartridże. Korzystając z narzędzi deweloperskich dla danej platformy, tworzony jest obraz danych dysku lub kartridża, który następnie możemy zapisać na nośniku, korzystając z odpowiedniej nagrywarki i uruchomić na konsoli podobnie jak każdą inną grę wideo.

  Sposób ten ma jednak swoje wady. Aplikację podczas pisania powinniśmy uruchamiać dosyć często, aby móc wcześnie wykrywać błędy i kontrolować napisany przez nas kod. Poza tym wiele problemów występujących podczas pisania gry wideo najlepiej rozwiązuje się doświadczalnie, co także wymaga częstego uruchamiania. Przygotowywanie paczki instalacyjnej bądź obrazu, a następnie przenoszenie na konsolę jest bardzo czasochłonne i może się szybko okazać, że ten proces pochłania nam większość naszego czasu. Problemem mogą być także same nośniki - w przypadku konsol szczególnie nośniki optyczne. Nie mogą to zazwyczaj być zwykłe płyty DVDR/ RW, ale drogie specjalne płyty jednorazowego zapisu, więc częste uruchamianie oprócz straty czasu może nas narazić na duży koszt. Praktycznie wszystkie nowe konsole posiadają także zabezpieczenia uniemożliwiające uruchamianie gier z nośników nieoryginalnych. Przygotowanie nośnika, aby

[14] GCC - GNU Compiler Collection - zestaw kompilatorów dla różnych języków programowania http://gcc.gnu.org

uruchomił się na konsoli, może być także bardzo uciążliwe, jeżeli w ogóle będzie w legalny sposób wykonalne.

Jednak najważniejszym problemem przy tym sposobie uruchamiania gry wideo jest brak kontroli nad wykonaniem programu. Do podstawowych narzędzi każdego programisty powinny należeć debugger i profiler, pozwalające na szybkie wykrywanie błędów oraz optymalizację kodu. Niestety, tutaj nie mamy możliwości ich wykorzystania. Często jedyną możliwością posiadania jakiejś kontroli nad kodem będą informacje testowe, które sami umieścimy na ekranie.

- *Urządzenie deweloperskie*

  Problemy związane z powyższym sposobem sprawiają, że jest to bardzo nieefektywny sposób pracy. Dlatego właśnie firmy produkujące i rozwijające konsole zajmują się także dostarczaniem deweloperom specjalnych urządzeń wspomagających tworzenie oprogramowania. Urządzenia te spełniają różne funkcje, zależnie od producenta oraz typu konsoli. Jednym z wariantów jest na przykład specjalna wersja konsoli, umożliwiająca uruchamianie niezabezpieczonych nośników, a poza tym nieróżniąca się niczym od wersji sklepowej. Inne problemy związane z przenoszeniem pozostają, lecz urządzenie takie pozwala chociażby w sprawny sposób dokonać prezentacji gry wideo wtedy, gdy jest ona jeszcze w wersji rozwojowej. Jednak dla programistów najbardziej interesującym urządzeniem będzie tzw. zestaw deweloperski (DEV-KIT), który eliminuje praktycznie wszystkie powyższe problemy. Zestaw taki jest to specjalnie przygotowana wersja konsoli, a co najmniej urządzenie, które tę funkcję konsoli spełnia, gdyż czasami ma ono zupełnie inną budowę i wersji sklepowej nawet nie przypomina. Zestaw taki jest przeznaczony do współpracy z komputerem i narzędziami na nim uruchomionymi, komunikuje się z nim poprzez sieć lub też inne łącza, np. USB. Połączenie z komputerem ma dwa główne cele. Pierwszym z nim jest dostarczanie do konsoli pliku wykonywalnego i danych. Zestaw taki nie posiada napędu nośników zewnętrznych, lecz zamiast tego wszystkie dane są bezpośrednio przesyłane z komputera podczas uruchamiania i umieszczane w pamięci wewnętrznej, zazwyczaj zrealizowanej w formie dysku twardego. Dzięki temu można szybko i sprawnie uruchamiać grę po każdej kompilacji czy też modyfikacji danych. Drugim celem komunikacji z komputerem jest bardzo ważna możliwość jego współpracy z narzędziami deweloperskimi. Dzięki połączeniu możemy użyć debuggera do pracy krokowej, profilera, dokonywać symulacji różnego rodzaju błędów.

  Zestaw deweloperski jest najlepszym narzędziem do tworzenia gry lub aplikacji na konsolę. Posiada jednak jedną dość istotną wadę - cenę. Ceny takich zestawów są bardzo wysokie, mogą przekroczyć koszt sklepowej wersji konsoli nawet kilkadziesiąt razy. Przy grze wideo może pracować od kilku do kilkudziesięciu programistów jednocześnie. Dostarczenie zestawu deweloperskiego dla każdego programisty staje się naprawdę poważnym wydatkiem, na który nie wszystkie firmy są w stanie sobie pozwolić. W takim

przypadku zestawy muszą być współdzielone pomiędzy programistów, co może być uciążliwe, lub trzeba sobie radzić w inny sposób.

- *Emulator*

  Gdy brakuje zestawów deweloperskich, a uruchamianie na sklepowych wersjach konsol jest zbyt uciążliwe, pozostaje jeszcze jeden sposób, mianowicie praca z wykorzystaniem programowego emulatora. Emulator jest to oprogramowanie, które jak sama nazwa wskazuje emuluje działanie innej platformy na komputerze. Praca z nim wygląda więc na bardzo wygodne rozwiązanie, jedyny sprzęt, jakiego potrzebujemy, to komputer, na którym piszemy, kompilujemy i w każdej chwili możemy szybko uruchomić grę, obserwując jej działanie na ekranie komputera.

  Rozwiązanie to jest obiecujące, jednakże ma kilka poważnych niedogodności. Pierwszą z nich jest współpraca z narzędziami takimi jak debugger[15]. Jest ona możliwa praktycznie tylko wtedy, kiedy dostawcą emulatora jest producent konsoli i zarazem narzędzi deweloperskich albo kiedy sami stworzymy taki emulator. W przypadku użycia emulatora dostarczanego przez stronę trzecią możemy o tej współpracy zapomnieć.

  Dla emulatora pewnym problemem może być wygodne odwzorowanie urządzeń wejściowych. Do konsol możemy podłączyć wiele rozmaitych kontrolerów, za pomocą których możliwe jest sterowanie grą. O ile symulowanie gamepada na klawiaturze komputera jest proste do zrealizowania i nawet wygodne w użyciu, to kontroler śledzący swoją pozycję w przestrzeni za pomocą akcelerometrów lub żyroskopów jest w praktyce niemożliwy do poprawnego odwzorowania. Kolejnym sporym problemem przy pracy z emulatorem może być jego wydajność. Komputer musi symulować działanie procesora konsoli, który najprawdopodobniej będzie reprezentował zupełnie inną architekturę oraz wszystkie urządzenia wewnętrzne. Powoduje to, że moc obliczeniowa komputera musi być o wiele większa niż emulowanej konsoli. Tak więc o ile emulacja konsol przenośnych nie stanowi przeważnie większego problemu, to dla konsol stacjonarnych, posiadających najczęściej zbliżoną moc obliczeniową do współczesnych im komputerów takie rozwiązanie jest niemożliwe. Testowana gra wideo lub aplikacja pracowałaby z kilkukrotnie mniejszą prędkością, co wyklucza sensowne użycie takiego emulatora. Jeszcze inną niedogodnością może być jakość, z jaką konsola jest odwzorowana w emulatorze. Konsola czy jakiekolwiek urządzenie, które ma być platformą docelową dla gry wideo, jest bardzo złożonym systemem, zawierającym wiele współpracujących ze sobą układów, działających często asynchronicznie względem siebie. Dokładne odwzorowanie tego sposobu działania może być po prostu niemożliwe do zrealizowania za pomocą oprogramowania. Powoduje to, że emulator nigdy nie odwzorowuje konsoli w sposób doskonały, więc nawet w przypadku, gdy emulacja jest głównym sposobem pracy przy pisaniu gry wideo, to konieczne jest częste urucha-

[15] Debugger - program służący do analizy innych programów, w celu odnalezienia zawartych w nich błędów.

mianie gry na urządzeniu docelowym lub zestawie deweloperskim w celu kontroli jej działania. Uruchamianie takie powinniśmy przeprowadzać tym częściej, im część tego oprogramowania aktualnie rozwijana bardziej związana jest ze sprzętem i SDK danej platformy, więc dotyczy to w szczególności pracy nad silnikiem gier wideo.

Rysunek 2.7. Emulator mobilnego systemu operacyjnego Android[16]

Użycie silnika wieloplatformowego

Uruchamianie na rzeczywistym czy też emulowanym urządzeniu docelowym może być problematyczne, na szczęście istnieje inne rozwiązanie. Programiści silnikowi są nierozerwalnie związani ze sprzętem i SDK danej platformy, natomiast dla każdej wyższej warstwy to powiązanie jest coraz to mniejsze. Jeżeli programista wyższych warstw oprogramowania, czyli tych warstw już bezpośrednio związanych z konkretną grą będzie miał zagwarantowany stały interfejs bibliotek silnikowych, to nie ma dla niego większego znaczenia, na jaką platformę aktualnie pisze, w końcu mechanika gry wideo i jej algorytmy są niezmienne.

Tutaj nasuwa się od razu pewne rozwiązanie - silnik, na podstawie którego pisana jest gra wideo, mógłby też jednocześnie działać na platformie docelo-

[16] Android - system operacyjny dla urządzeń mobilnych, rozwijany przez Google - http://www.android.com.

wej (czyli na przykład jakiejś konsoli) oraz na komputerze, takim samym jaki używany jest przez programistów podczas pracy. Przy zapewnieniu stałego interfejsu programistycznego kod źródłowy tworzonej gry byłby identyczny dla obu platform i uruchomianie na każdej z nich wymagałoby jedynie przekompilowania kodu z użyciem innych bibliotek silnika, używając innego kompilatora bądź innych jego ustawień. Dla programisty gry wideo całe to „zamieszanie" z różnymi platformami jest praktycznie niewidoczne, dzięki czemu powyższe rozwiązanie jest bardzo wygodne. Pisanie kodu, uruchamianie go, profilowanie oraz wszystkie inne czynności związane z jego tworzeniem mogą odbywać się z wykorzystaniem jedynie komputera, bez potrzeby korzystania z innych urządzeń czy oprogramowania w postaci emulatorów. Przyspiesza to znacznie cały proces powstawania kodu gry wideo. Używanie wszelkich narzędzi diagnostycznych jest dużo prostsze na komputerze, chociażby ze względu na ich liczbę, jaka jest dostępna na rynku - nie jesteśmy przecież już ograniczeni do jednego dostawcy narządzi, czyli producenta urządzenia, na które tworzymy. Poza tym programiści korzystający z takiego silnika będą używać tych samych narzędzi bez względu na platformę, na którą tworzone będzie oprogramowanie. Dzięki dobrej znajomości narzędzi kod gry będzie powstawał szybciej i będzie miał znacznie mniej błędów, nie będzie także potrzeby przeprowadzania szkolenia przy każdej zmianie na nową platformę. Bez wątpienia zmniejszy to także zapotrzebowanie na zestawy deweloperskie, a to oznacza konkretne oszczędności finansowe.

Tworzenie gry wideo na komputerze, podczas gdy przeznaczona jest na inny sprzęt ma oczywiście swoje wady. Komputer oraz konsola posiadają różne możliwości i zasoby sprzętowe. Dotyczy to na przykład wydajności procesora, układu graficznego czy pojemności dostępnej pamięci operacyjnej. Programista teoretycznie jest odcięty od wszelkich specyficznych właściwości urządzania, na które będzie przeznaczone oprogramowanie. Zadaniem takiego silnika wieloplatformowego jest w końcu ich maskowanie i jeżeli programista będzie pisał swoją część kodu gry wideo na komputerze, to użyty silnik pozwoli mu w maksymalny sposób wykorzystać wydajność tego komputera. Niestety, oczywiście nie przekłada się to zazwyczaj na wydajność np. konsoli - gra będzie na konsoli działała dużo wolniej i miała problemy z wolną pamięcią. Pomimo posiadania komfortu odcięcia od szczegółów SDK i bibliotek programistycznych danej konsoli, programista musi znać przynajmniej w przybliżeniu jej możliwości, a od czasu do czasu jednak uruchamiać na niej swój kod w celu chociażby kontroli, czy możliwości sprzętu nie zostały przekroczone, lub przeciwnie, czy nie jest dostępny jeszcze jakiś zapas mocy do wykorzystania.

Innym problemem, podobnie jak w przypadku użycia emulatora, jest odwzorowanie urządzeń wejściowych. Wiele urządzeń można z powodzeniem zastąpić kontrolerami dostępnymi dla komputera, lecz dla niektórych, szczególnie specyficznych dla danej konsoli będzie to niemożliwe. Jest to pewne utrudnienie, lecz jego wpływ można ograniczyć. Zależnie oczywiście od typu tworzonej gry system obsługi sterowania pisany będzie przez programistę lub niewielką

grupę programistów. Prawdopodobnie uda im się wydzielić konsolę lub urządzenie deweloperskie, na którym będą mogli ten system przetestować i dokonać niezbędnych ustawień w taki sposób, aby sterowanie to było wygodne. Dla reszty programistów nie będzie to miało większego znaczenia, czy pracują ze sterowaniem, które w 100% odpowiada temu, które wykorzystane będzie w końcowym produkcie, jeśli tylko wszystkie jego funkcje będą w jakiś sposób emulowane.

2.2.2 Tworzenie gier wideo przeznaczonych na wiele platform

Proces powstawania wieloplatformowej gry wideo

Powyższe zastosowanie silnika obsługującego więcej niż jedną platformę jest bardzo pomocne w procesie powstawania gier wideo, lecz oczywiście nie jest to zastosowanie główne, którym jest oczywiście... wydanie produktu na więcej niż jedną platformę.

Proces produkcji oprogramowania, zwłaszcza zaś gier jest procesem bardzo kosztownym, szczególnie jeśli chcemy stworzyć duży, konkurencyjny produkt. Wynika to chociażby z dużego nakładu pracy, jaką muszą wykonać programiści, o czym wspomniano już wcześniej, ale dotyczy to także każdego elementu składającego się na proces jej produkcji, czyli grafiki, muzyki, designu, testowania i tworzenia poziomów. Ten duży koszt wynika z prostego rachunku liczby ludzi pracujących przy tych wszystkich elementach razy czas trwania projektu. Gra wideo, żeby jej powstanie miało jakikolwiek sens ekonomiczny, musi na siebie zarobić. Na rynku z powodzeniem istnieje kilka konsol stacjonarnych, przenośnych oraz różne platformy gier na telefony komórkowe, komputerowe gry stacjonarne, flashowe, a nawet przeglądarkowe. Aby produkt przyniósł jak największe zyski, trzeba go wydać na jak największą liczbę platform, by zmaksymalizować grono odbiorców.

Wydanie gry wideo na wiele platform może być bardzo opłacalne. Z jednej strony pomnażamy rynek dla naszego produktu, z drugiej strony koszty uzyskania wersji na inne urządzenie bądź system nie muszą być bardzo wielkie, szczególnie jeżeli zostanie to przewidziane wcześnie w fazie projektowania gry. Wersje na poszczególne platformy nie muszą być identyczne, lecz im bardziej będą do siebie podobne, tym więcej wspólnych elementów trzeba będzie wykonać tylko raz. Dotyczy to zarówno designu gry, grafiki, danych dźwiękowych, tekstów, ale także i kodu, który może być wspólny, dzięki zastosowaniu silnika wieloplatformowego. Weźmy np. grę wideo, którą chcielibyśmy wydać jednocześnie na dwie popularne konsole stacjonarne - Xboxa 360 i PlayStation 3. Wydajność oraz możliwości tych konsol są bardzo zbliżone, dzięki czemu praktycznie wszystkie dane graficzne, muzyczne, modele 3D będą mogły być wykorzystane bez zmian w obu wersjach. Ponieważ obie konsole należą do tego samego segmentu odbiorców, także całą mechanikę gry prawdopodobnie będziemy chcieli zostawić dokładnie taką samą. W takiej sytuacji

dzięki wykorzystaniu dobrego silnika wieloplatformowego, który skutecznie ukryje przed twórcami szczegóły różniące obie konsole, takie jak obsługa innych kontrolerów czy trybu gry sieciowej, kod gry dla obu tych wersji będzie praktycznie identyczny. Tak więc utworzenie wersji na drugą platformę będzie wymagało jedynie przekompilowania kodu i złożenia danych płyty zgodnie ze standardem konsoli oraz implementacji specyficznych dla platformy wymagań (np. testy TCR/Lotcheck[17] etc.). Deweloper gry poniesie więc relatywnie niewielkie dodatkowe koszty przeniesienia, testowania oraz drobnych poprawek, które zawsze trzeba mieć na uwadze. Inną sprawą jest to, że dobry silnik wieloplatformowy, który umożliwi takie bezbolesne przejście gry wideo pomiędzy platformami, na pewno nie będzie tani, niezależnie od tego, czy firma tworząca grę będzie chciała rozwijać swój własny, czy wykupić licencję na jakiś już istniejący.

Rysunek 2.8. „World of Goo" - gra wydana na platformy Windows, Linux, Mac OS, Nintendo Wii, iOS i nieróżniąca się na nich praktycznie wcale wyglądem czy mechaniką

Proces powstawania wersji gier na poszczególne platformy może być różny, w zależności od przyjętej przez producenta strategii. W idealnym przypadku, kiedy wersje gry wideo są identyczne, a silnik załatwia wszystkie nasze prob-

[17] TCR i Lotcheck - testy gier wideo dla konsol odpowiednio Xbox 360 i Nintendo DS/Wii, w celu sprawdzenia wymagań dopuszczających tytuł do wydania.

lemy, kolejną wersję mamy niejako za darmo, dzięki czemu proces powstawania może iść zupełnie równolegle. Często jednak w wersjach pojawią się większe lub mniejsze różnice, które będą wymagały oczywiście dodatkowej pracy designerów, grafików czy programistów. W takiej sytuacji możliwe są dwa rozwiązania. Pierwsze z nich zakłada, że wszystkie wersje tworzone są jednocześnie. Wymaga to większego zespołu pracowników, którzy jednocześnie będą dostosowywać wszelkie elementy pod poszczególne wersje. Odpowiednio zwielokrotniona ilość pracy będzie również potrzebna w przypadku zmian w projekcie gry wideo, które zawsze pojawiają się w procesie jej powstawania i które będą musiały być wprowadzane we wszystkich wersjach. Do niewątpliwych zalet takiego rozwiązania należy zaliczyć to, że wszystkie wersje gotowe są w stosunkowo krótkim czasie i praktycznie jednocześnie, co może być bardzo ważne ze względów marketingowych.

Drugim podejściem jest tworzenie poszczególnych wersji po kolei. Jest to możliwe do zrealizowania przez mniejszy zespół ludzi, zmniejsza się też koszt ze względu na brak rozwoju „ślepych uliczek" we wszystkich odmianach, a jedynie w pierwszej, która będzie stanowiła wzorzec przy dostosowywaniu później powstających wersji dla innych platform. Do wad tego rozwiązania należy jednak właśnie to, że wersje tworzone są po kolei, co może być problemem marketingowym. Wydawanie ich kolejno w ramach powstawania wymagać będzie powtarzania kampanii reklamowej. Czekanie bowiem z wydaniem, aż wszystkie wersje będą gotowe, spowoduje sztuczne wydłużenie całego procesu. Z drugiej strony, w przypadku braku finansowego sukcesu gry na pierwszej platformie, możemy zmniejszyć stratę przez zarzucenie stworzenia wersji na kolejne platformy.

Często postępuje się tak, iż gra powstaje w podstawowym studiu na podstawowy zestaw podobnych do siebie pod względem możliwości platform sprzętowych (np. PC, PS3, Xbox 360), a przeniesienie jej na platformy znacząco różne (np. Wii, DS, PSP ...) powierza się zewnętrznym deweloperom.

Ostatecznie wybór metody tworzenia wieloplatformowej gry zależy przede wszystkim od możliwości zasobowych i finansowych firmy i od przyjętej strategii postępowania.

Różnice w projekcie wieloplatformowym

Z powyższych rozważań wynika pewien istotny wniosek. Największym problemem przy tworzeniu wersji gier wideo przeznaczonych na wiele platform są różnice występujące w tych wersjach. Wynikają one chociażby z różnic sprzętowych pomiędzy urządzeniami, na które wersje gry chcemy stworzyć. Szczególnie widoczne jest to pomiędzy urządzeniami z odmiennych klas, np. pomiędzy konsolami stacjonarnymi i przenośnymi, ale nawet różnice sprzętowe urządzeń tej samej klasy, lecz innych producentów także mogą być znaczne.

Różnic tych nie da się zupełnie uniknąć, a jedynie można minimalizować ich wpływ. Możemy projektować grę wideo w ten sposób, żeby nie korzystać

lub ograniczać korzystanie z unikalnych właściwości sprzętowych danych urządzeń, a skupić się na ich elementach wspólnych. Podobnie też można postąpić w sprawie wydajności. Grę wideo można zaprojektować w taki sposób, żeby z powodzeniem pracowała na najsłabszym z planowanych urządzeń, a wówczas nie będzie problemów z każdym innym. Te sposoby pozwolą nam szybciej i taniej stworzyć wersje gry wideo dla poszczególnych platform, jednak trzeba postawić sobie pytanie, czy jest to dobra droga. Maksymalne wykorzystanie wydajności urządzeń do gier wideo oraz ich unikalnych właściwości są cechami wybitnych produktów wysokiej jakości, podczas gdy programy średnie toną w morzu podobnych produkcji. Dlatego czasem najlepszym wyjściem może być po prostu stworzenie osobnych wersji tej samej gry z wykorzystaniem specyficznych cech danych klas urządzeń.

Różnice w wersjach gry wideo dla poszczególnych platform mogą być spowodowane przez:

- *Procesor*

  Pierwszym elementem, na który zwracamy uwagę w urządzeniu, na którym możemy uruchamiać gry wideo, jest niewątpliwie procesor, co nie jest oczywiście pozbawione przyczyny. Procesor jest jednym z ważniejszych elementów decydujących o wydajności danego urządzenia. Wprawdzie jest on „tylko” jednym z ważniejszych, jednak cała reszta wyposażenia musi z nim współgrać, przez co ich wydajność jest najczęściej dostosowana oraz proporcjonalna do wydajności procesora. Dzięki temu patrząc na procesor, możemy „z grubsza” ocenić, z jakiej klasy urządzeniem mamy do czynienia.

  Gdy oceniamy procesor, jednym z najważniejszych parametrów jest jego taktowanie określające liczbę operacji wykonywanych na sekundę. Ten parametr może nas jednak łatwo zwieść. Niektóre telefony komórkowe taktowaniami swoich procesorów potrafią obecnie doścignąć słabsze konsole stacjonarne. Nie znaczy to oczywiście, że grę z takiej konsoli stacjonarnej można przenieść na telefon bez obawy o moc obliczeniową, gdyż dla wydajności procesora równie ważna jest jego architektura. O ile zaś architektura procesorów komputerach osobistych musi się trzymać pewnego kompatybilnego standardu, to w świecie konsol, telefonów oraz innych urządzeń, na których można uruchamiać gry wideo, panuje zupełna dowolność. Od architektury zależy między innymi, do jakiej rodziny procesor należy, a przez to jaką listę rozkazów obsługuje. W liście rozkazów może się znaleźć np. grupa rozkazów do operacji macierzowych, dzięki czemu procesor taki będzie się lepiej nadawał do różnego rodzaju symulacji i obliczeń fizycznych. Innymi cechami architektury może być ilość jednostek wykonawczych i pojemność pamięci podręcznej, decydujące o tym, jak efektywnie procesor przetwarza rozkazy kodu maszynowego.

  Odrębnym parametrem procesora mającym ogromy wpływ na sposób pisania oprogramowania jest liczba jego rdzeni, co dla programisty oznacza po prostu liczbę wirtualnych procesorów, jaką ma do dyspozycji. O ile szybsze

PC - Windows

Nintendo DS

Rysunek 2.9. Assassin's Creed w wersji dla „dużych" platform posiada wiele różnic w mechanice rozgrywki w porównaniu z wersją mobilną. Zachowuje jednak bardzo podobny klimat. ©Ubisoft

taktowanie lub lepsza architektura automatycznie sprawia, że oprogramowanie pracuje szybciej, to w przypadku procesora wielordzeniowego musi o to zadbać sam programista. Trzeba wydzielić niezależne zadania, które będą mogły być wykonywane równolegle i będą w najbardziej równomierny sposób obciążać poszczególne rdzenie. Powoduje to, że przenoszenie gry wideo z architektury jednordzeniowej na wielordzeniową, a także odwrotnie nie jest prostą sprawą i wymaga często przeprojektowania znacznej części jej mechaniki.

Niektóre konsole posiadają kilka fizycznych procesorów dostępnych dla programisty, co z jednej strony otwiera nowe możliwości, z drugiej zaś czyni platformę jeszcze bardziej egzotyczną, a jej programowanie staje się coraz bardziej skomplikowane.

- *Pamięć operacyjna*

  Pamięć operacyjną najczęściej definiuje po prostu jej wielkość. Dla danej gry wideo wyznacza ona, ile struktur danych i zasobów będziemy mogli jednocześnie przechowywać, a jakie dane będzie trzeba doczytywać z pamięci masowej. Może powodować to pewne komplikacje, szczególnie podczas tworzenia oprogramowania na urządzenia znacznie różniące się ilością tej pamięci.

  Podczas pisania programów komputerowych przyzwyczailiśmy się, że pamięć operacyjna dostępna jest jako jednolity obszar, dla którego możliwa jest alokacja potrzebnych jej bloków. Niektóre urządzenia mogą też udostępniać pamięć w inny sposób, a mianowicie w postaci banków pamięci. Każdy bank jest tak jakby odrębną pamięcią operacyjną i podczas alokacji musimy brać pod uwagę ilość pozostałego miejsca w każdym z nich. O ile banki są równoważne między sobą, nie ma znaczenia, w którym pamięć zaalokujemy - wtedy zarządzanie nimi najlepiej powierzyć silnikowi gier. Zdarza się natomiast, że podział na banki dzieli pamięć na obszary o szybszym i wolniejszym czasie dostępu. W takim przypadku kontrola, w którym banku alokujemy miejsce na różne struktury danych, może mieć kluczowe znaczenie dla wydajności gry.

- *Układ graficzny*

  Układ graficzny jest obecnie bardzo ważnym elementem urządzeń przeznaczonych dla obsługi gier wideo. Każda konsola, komputer, a obecnie nawet coraz częściej telefony komórkowe posiadają układy specjalizowane w przyspieszaniu generowania grafiki 2D bądź 3D.

  Podstawowym parametrem układu graficznego jest jego wydajność, przekładająca się na dokładność modeli oraz ilość detali, jaką układ potrafi wyrenderować w każdej klatce animacji. Tworzenie gry na platformy z układami graficznymi o znacznej różnicy wydajności spowoduje, że konieczne będzie przygotowanie wielu zestawów danych modeli trójwymiarowych, dostosowanych do każdej platformy. Podobny problem dotyczy ilości pamięci w układzie, który jest dostępny dla tekstur. Mianowicie, różnice w ilości do-

stępnej pamięci spowodują, że potrzebne będzie przygotowanie zestawów tekstur o różnej rozdzielczości dla poszczególnych platform.

Utrudnieniem może być także różnica w samym rodzaju układu graficznego. Podstawowe operacje takie jak wyświetlanie modeli czy nawet pojedynczych prymitywów graficznych mogą być ukryte przed nami przez silnik wieloplatformowy, w ten sposób, że mamy do nich dostęp przez niezależny, jednolity interfejs. Jednak tworzenie kodu shaderów może ściśle zależeć od modelu układu graficznego i istnieje zagrożenie, że będziemy zmuszeni implementować osobny ich zestaw dla każdej platformy.

Ogólna wydajność urządzenia, na które tworzymy oprogramowanie, określa przede wszystkim wielkość gry, jaką może obsłużyć. Często może się zdarzyć sytuacja, że upraszczanie grafiki czy algorytmów nie jest wystarczające w przypadku próby stworzenia gry na platformy o znacznej różnicy wydajności. Jedynym rozwiązaniem jest wtedy upraszczanie, a nawet zmiana mechaniki i założeń gry, co może doprowadzić do tego, że tworzone będą dwie zupełnie odrębne programy, które łączyć będzie tylko wspólny tytuł i tematyka. W takim przypadku trzeba się zastanowić, czy użycie silnika wieloplatformowego ma sens i czy prostszym rozwiązaniem nie będzie tworzenie takich gier niezależnie od siebie.

Wszystkie te cechy sprawiają, że przenosząc grę wideo na inne urządzenie, musimy się dobrze zastanowić, jaką rzeczywiście ma ono wydajność obliczeniową i graficzną i jak się ono sprawdzi w tym konkretnym przypadku. Gdy nie znamy tych parametrów, możemy oczywiście posłużyć się różnego rodzaju benchmarkami dostępnymi w dokumentacjach lub po prostu obejrzeć inne tytuły wydane już na tę platformę i w praktyce sprawdzić, do czego dany sprzęt jest zdolny.

- *Ekran*

  Ekran jest bardzo istotnym elementem urządzeń, dla których tworzy się gry wideo. Wrażenia wizualne, oprócz dźwiękowych jest to główny sposób odbioru gry przez użytkownika. Podstawowym parametrem ekranów, istotnym z punktu widzenia osoby tworzącej grę wideo jest ich rozdzielczość - liczba pikseli w pionie i w poziomie, które potrafią wyświetlić. Rozdzielczość ekranu, po pierwsze, przekłada się na liczbę pikseli, które musi przetworzyć procesor graficzny podczas procesu renderingu, po drugie na ilość szczegółów, jakie ekran potrafi zaprezentować. W związku z tym przy większej rozdzielczości ekranu musimy tworzyć sceny składające się z dokładniejszej geometrii oraz z tekstur o większym rozmiarze. Rozdzielczość ekranu wpływa więc na minimalną ogólną wydajność, jaką dane urządzenie musi się charakteryzować, by w pełni ten ekran wykorzystać. Innym parametrem, który należy wziąć pod uwagę, jest fizyczny rozmiar ekranu, który najczęściej wyrażony jest w postaci długości jego przekątnej w calach. Parametr ten jest szczególnie istotny podczas projektowania gier dla urządzeń przenośnych. Na przykład, dla dwóch urządzeń wyposażonych w ekrany z podobną rozdzielczością, lecz znacznie różniących się rozmiarami fizycz-

nymi, w inny sposób trzeba będzie zaprojektować rozmieszczenie różnych obiektów na ekranie, aby były czytelne dla gracza. Szczególnie ważne jest odpowiednie dopasowanie wielkości czcionek używanych przy prezentacji różnych tekstów na ekranie.

Tym, co może zupełne wywrócić do góry nogami sposób myślenia o prezentacji gry wideo graczowi, jest liczba dostępnych ekranów. W ogromnej większości konsol nadal dominuje wykorzystanie jednego ekranu, wbudowanego dla urządzeń przenośnych i zewnętrznego, najczęściej telewizora dla urządzeń stacjonarnych. Zdarzają się jednak wyjątki, których przykładem może być przenośna konsola „Nintendo DS”, posiadająca dwa ekrany (od tego zresztą pochodzi nazwa konsoli DS - Dual Screen). Dolny wyposażony jest dodatkowo w interfejs dotykowy. Przykłady wykorzystania więcej niż jednego ekranu można znaleźć także dla konsol stacjonarnych oraz komputerów osobistych. Niektóre gry wideo, na przykład symulacje lotnicze lub samochodowe, umożliwiają wykorzystanie dodatkowych monitorów lub telewizorów, umieszczonych z lewej i prawej strony gracza, dla poszerzenia pola widzenia. Niesamowicie potęguje to odczuwane wrażenie realizmu.

Obsługa ekranów w grach wideo musi być także elastyczna. O ile w konsolach przenośnych z wbudowanym ekranem problem jego zmienności nie istnieje, to w przypadku urządzeń stacjonarnych użytkownik może podłączyć ekran o różnych rozdzielczościach i formatach, a gra powinna poprawnie się prezentować we wszystkich tych konfiguracjach.

- *Kontrolery*

  Projektowanie gry wieloplatformowej utrudnia różnorodność dostępnych urządzeń wejściowych - kontrolerów (lub HID - *Human Interface Device*), z którymi różne systemy potrafią współpracować. Każdy typ urządzenia, na którym można uruchomić gry wideo, ma swój domyślny, najczęściej stosowany sposób sterowania, który wynika z jego budowy oraz przeznaczenia. Dla komputerów osobistych będzie to nadal mysz i klawiatura, dla większości konsol stacjonarnych gamepad, natomiast dla tabletów i telefonów nowszej generacji interfejs dotykowy. Przenoszenie gry pomiędzy komputerem osobistym i konsolami z gamepadem często jest relatywnie proste - sprowadza się głównie do przemapowania klawiszy klawiatury na gamepadowe i odwrotnie. Utrudnieniem może być tutaj przeniesienie sterowania myszy na sterowanie analogowymi gałkami oraz dobranie parametrów czułości sterowania, by grało się wygodnie.

  Większy problem występuje w przypadku przenoszenia gry wideo pomiędzy urządzeniami o zupełnie odmiennym typie sterowania, np. klawiaturą i ekranem dotykowym. Często nie ma możliwości bezpośredniego przeniesienia sposobu sterowania przez zmianę klawiszy fizycznych na wirtualne klawisze wyświetlane na urządzeniu dotykowym, gdyż jest to niewygodne lub po prostu się nie sprawdza. W takim przypadku dostosowanie sposobu sterowania może wymagać większych lub mniejszych ingerencji w mecha-

Rysunek 2.10. Forza Motorsport 3 uruchomiona w trybie trójekranowym. ©Microsoft Game Studios & Turn 10

nikę gry wideo, które aby przebiegały jak najbardziej „bezboleśnie”, muszą być przewidziane na wczesnym etapie tworzenia gry.

Oprócz domyślnego sposobu sterowania, konsole i komputery osobiste posiadają możliwość podłączenia całej gamy różnego rodzaju akcesoryjnych kontrolerów, jak chociażby kierownice. Wykorzystanie różnych typów sterowań niewątpliwe podnosi walory gry wideo, a często pozwala też użytkownikowi odkryć dany gatunek gry na nowo, odróżniając go od podobnych produkcji konkurencji.

- *Rodzaj pamięci masowej*

  Rodzaj pamięci masowej jest czynnikiem określającym, w jaki sposób urządzenie odczytuje i zapisuje dane niezbędne dla prawidłowego działania gry wideo. Możemy wyróżnić dwa podstawowe rodzaje dostępnej pamięci masowej. Pierwszą jest pamięć tylko do odczytu. Jest to pamięć, za pomocą której gra jest dostarczana do urządzenia i może być na nim uruchamiana. Przykładami takiej pamięci może być dysk DVD, kartridż lub pamięć wbudowana w postaci dysku twardego albo pamięci Flash. W dwóch ostatnich przypadkach termin „tylko do odczytu” odnosi się do dostępu przez uruchomioną grę wideo, ponieważ modyfikacja tej pamięci jest najczęściej możliwa poprzez system operacyjny urządzenia, w celu chociażby skopiowania do niej danych gry. Drugim rodzajem pamięci jest pamięć zapisu i odczytu,

przeznaczona głównie do przechowywania stanów gry, profili graczy czy ich zdobyczy punktowych. W przypadku komputerów osobistych i nowszych generacji konsol stacjonarnych jeden i drugi rodzaj pamięci może oznaczać ten sam napęd - dysk twardy. Natomiast w innych przypadkach pamięć zapisu/odczytu realizowana jest w postaci kart pamięci. Mogą być to specjalne pamięci, kompatybilne tylko z danym urządzeniem oraz umieszczone w specjalnie wydzielonej pamięci Flash lub EPROM znajdującej się na kartridżu z grą lub dostarczanej w postaci kart pamięci. Mogą to być także klasyczne karty pamięci Flash, np. standardu SD.

Rodzaj pamięci masowej definiuje przede wszystkim ilość miejsca, jaką będziemy mieli do dyspozycji na dane gry wideo lub zapisane stany rozgrywki. Drugim istotnym czynnikiem jest czas dostępu do tej pamięci i prędkość przesyłu danych, które będziemy musieli wziąć pod uwagę, projektując na przykład doczytywanie danych „w locie" do pamięci operacyjnej.

- *Unikalne cechy sprzętowe*

  Duża część różnic w wersjach wieloplatformowych gier wideo wynika z unikalnych cech urządzeń, na które te gry tworzymy. Zaliczyć do nich możemy podzespoły wbudowane w samo urządzenie, takie jak kamera, ekran dotykowy, jak też wszelkiego rodzaju akcesoria i kontrolery, które można z tym urządzeniem wykorzystać. Dobrym przykładem jest tutaj konsola Nintendo Wii, która posiada możliwość współpracy z ogromem różnych kontrolerów i dodatków takich jak specyficzny kontroler wykrywający swoją pozycję w przestrzeni, czy dodatek Wii Balance Board w postaci podstawki do ćwiczeń gimnastycznych. Unikalne cechy urządzeń bardzo często warte są wykorzystania, gdyż podnoszą znacznie atrakcyjność gry wideo, często wnosząc wiele świeżości chociażby przez ciekawy, innowacyjny system sterowania. Wykorzystanie tych urządzeń wprowadza jednak wiele pracy przy obsłudze wieloplatformowości gry wideo. Jeżeli unikalny jest na przykład tylko kontroler, to stosunkowo niewielkim kosztem można dopisać specjalny tryb sterowania dla danej platformy. W przypadku jednak wykorzystania na przykład kamery może być już trudniej, ponieważ jej rozsądne użycie może wymagać nawet dużej zmiany w mechanice gry dla danej platformy.

- *System operacyjny i firmware*

  System operacyjny i firmware określają platformę programową do uruchamiania oprogramowania. Możemy się skupić oczywiście na jednym, najpopularniejszym systemie, lecz rolą silnika wieloplatformowego w takiej sytuacji może być zapewnienie współpracy z więcej niż jednym systemem operacyjnym i jedną wersją firmware.

  W przypadku wielu systemów operacyjnych na urządzenia mobilne i konsole gier wideo, dostawca systemu operacyjnego będzie wymagał, by aplikacja spełniała rygorystyczne wymogi danej platformy, żeby mogła zostać

wydana. Wymogi te będą podlegały sprawdzeniu przez licencjonodawcę oprogramowania i będą musiały być uwzględnione w trakcie pisania gry. Wiele z nich tyczy się bardzo niskopoziomowego dostępu do sprzętu i musi być ujęta na poziomie silnika.

Większość różnic związanych z systemami operacyjnymi urządzeń nie ingeruje jednak w mechanikę gry, dlatego większość różnic na ich poziomie powinna być ukryta przez implementację silnika wieloplatformowego i niewidoczna dla programisty.

- *Różne grupy docelowe produktu*

  W przypadku gier czynnikiem, który w silny sposób decyduje o treści oraz mechanice tworzonej gry wideo, jest jej grupa docelowa. Obecnie gry wideo są rozrywką bawiąca ludzi z wielu różnych grup społecznych. Każda z nich ma inne oczekiwania w stosunku do rozrywki z nimi związanej. Powoduje to, że konkretny produkt nie może być adresowany do wszystkich, lecz zawsze musi się skoncentrować na jakiejś grupie lub grupach odbiorców. Często o tym, jaka jest grupa docelowa, decyduje samo urządzenie, na które gra wideo jest tworzona. Przykładowo w przypadku konsol przenośnych znaczną grupą będą osoby, które używają jej podczas podróży pociągiem lub autobusem. Oczekują oni przede wszystkim rozgrywki podzielonej na krótkie etapy, możliwe do ukończenia w kilka minut i niewymagające ciszy oraz skupienia, o które w takich warunkach raczej trudno.

  Dobrym przykładem jest podział upodobań graczy występujący w przypadku obecnie dostępnych na rynku konsol stacjonarnych. Konsole Xbox 360 i Playstation 3 leżą w kręgu zainteresowań osób silniej związanych z grami wideo, oczekujących dłuższej i bardziej wymagającej rozgrywki. Natomiast Nintendo Wii jest adresowane do osób traktujących grę jako bezstresową i odprężającą rozrywkę, często w gronie rodzinnym lub znajomych.

  Różnice w grupach docelowych są dla projektantów gier wideo bardzo odczuwalne. Mogą one decydować o modyfikacjach mechaniki i treści gry lub nawet o całkowitej ich zmianie w zależności od wersji produktu dla danej platformy.

2.2.3 Wieloplatformowy silnik gier wideo

Idea silnika wieloplatformowego

Definicja 2.6 - Wieloplatformowy silnik gier wideo

Wieloplatformowy silnik gier wideo jest to silnik wspomagający tworzenie gier wideo na różnych platformach sprzętowych i programowych za pomocą jednolitego interfejsu.

Tworzenie gry wideo przeznaczonej na różne platformy sprzętowe może być procesem bardzo złożonym, w zależności od różnic w tych platformach, jak też różnic w samej mechanice gry. Złożoność ta powoduje, że obecnie większość ciężaru obsługi różnic sprzętowych i programowych jest obsługiwana oraz niejako maskowana przez silniki gier wideo, które dzięki temu stają się silnikami wieloplatformowymi. Silnik taki udostępnia swoją funkcjonalność za pomocą wspólnego interfejsu dla wszystkich platform (a przynajmniej jej większość). Dzięki temu twórca gry wideo może skoncentrować swe wysiłki na implementacji mechaniki, po prostu korzystając z tej funkcjonalności i nie martwiąc się, jak została ona zrealizowana na poszczególnych platformach.

Różne silniki wieloplatformowe oczywiście różnie radzą sobie z tym zadaniem. Dlatego właśnie wybór silnika, w przypadku gdy decydujemy się na wykorzystanie istniejącego silnika lub wybór sposobu jego implementacji, kiedy sami taki silnik rozwijamy, jest bardzo ważną decyzją. Decyzję tę niestety najczęściej trzeba podjąć stosunkowo wcześnie, przed rozpoczęciem implementacji gry wideo, kiedy nie znamy jeszcze wielu problemów, które mogą wyniknąć podczas przenoszenia gry na poszczególne platformy. Zmiana silnika w zaawansowanym stadium projektu powoduje najczęściej zupełne wywrócenie tego projektu „do góry nogami", na co praktycznie nigdy nie można sobie pozwolić. Wszelkie niedoskonałości lub też braki silnika będziemy musieli uzupełnić sami albo negocjować naprawę błędów oraz rozwój funkcjonalności z jego producentem.

Jakość radzenia sobie z obsługą wieloplatformowości jest również dość silnie związana z poziomem specjalizacji silnika. Prostym sposobem oceny, jak dobrze sobie silnik z tym radzi, jest obserwacja różnic w wyglądzie i zachowaniu wszelkich obiektów i funkcjonalności udostępnianych przez niego na wszystkich obsługiwanych platformach. Silnik jest oczywiście tym lepszy, im tych różnic jest mniej. W przypadku dużej specjalizacji korzystamy z wysokopoziomowych obiektów posiadających stosunkowo wąską oraz bardzo ukierunkowaną funkcjonalność. Dzięki temu łatwiej jest przetestować większość okoliczności, w których obiekt ten może być użyty oraz wprowadzić pewne poprawki w ich implementacji, żeby zachowywały się identycznie na poszczególnych platformach. Odmienna sytuacja występuje w przypadku silników uniwersalnych. Teoretycznie można by przypuszczać, że skoro udostępniają one prostsze obiekty, to łatwiej jest zaimplementować je w sposób identyczny. Dla podstawowych zastosowań jest to prawdą, lecz to, co jest trudne do opanowania, to ogromna ilość kombinacji, z jakimi obiekty te mogą być użyte oraz wynikające z tego zależności pomiędzy nimi. Spora część różnic w zachowaniu nie jest możliwa do wykrycia przez programistów implementujących silnik, lecz objawia się dopiero podczas jego wykorzystania do tworzenia konkretnej gry wideo. Jest to jeden z wielu argumentów potwierdzających, że komunikacja programistów silnikowych z „gameplayowymi" jest bardzo istotna. Dlatego wybierając dla przykładu silnik komercyjny, warto zwrócić uwagę chociażby na jakość świadczonej pomocy technicznej.

Właściwości silnika wieloplatformowego

Aby dany silnik można było uznać za wieloplatformowy, musi on spełnić kilka warunków:

- *Umożliwia funkcjonowanie gry wideo na różnych platformach*

  Jak sama nazwa wskazuje, silnik wieloplatformowy musi umożliwiać stworzenie gry wideo przeznaczonej do działania dla różnych architektur sprzętowych lub programowych. Przy czym wsparcie dla samego procesu tworzenia gry, w postaci narządzi, kompilatorów, edytorów może być już ograniczone do jednej platformy.
- *Wspólna implementacja mechaniki i logiki gry*

  Celem istnienia silników wieloplatformowych jest maksymalne ograniczenie wszelkich prac, które będziemy musieli wykonać wielokrotnie, odrębnie dla każdej platformy, na którą produkt przygotowujemy. W związku z tym możemy oczekiwać, że implementacja mechaniki i logiki gry wideo będzie dla tych platform wspólna. Implementacja ta może mieć postać kompilowanego kodu źródłowego w przypadku silników SDK, a także kodu skryptów lub różnego rodzaju schematów blokowych przepływu sterowania w przypadku silników edytorowych (sterowanych danymi).
- *Jednolity sposób obsługi danych*

  Ograniczenie czynności, które musimy wykonać wielokrotnie, dotyczy także danych (np. plików graficznych czy dźwiękowych), z których gra wideo korzysta podczas działania. Różne systemy operacyjne, a także różne platformy sprzętowe wymagają, aby te dane były w ich własnym formacie. Niewątpliwie stratą czasu byłoby przygotowywanie tych danych zupełnie odrębnie dla poszczególnych platform, więc silnik musi zapewniać mechanizmy gwarantujące jednolitą ich obsługę.

Rodzaj wspierania wieloplatformowości

W zależności od przeznaczenia silniki gier wideo wspomagają wieloplatformowość w różny sposób. W związku z tym można je podzielić na typy:

- *Silnik dla wspólnej platformy programowej i wielu platform sprzętowych*

  Najprostszym ich typem są silniki wspierające jedną platformę programową oraz wiele rodzajów sprzętu. Przykładem takiego rozwiązania mogą być silniki służące do tworzenia gier wideo dla jednego z „mobilnych” systemów operacyjnych, takich jak *Android, iOS, Symbian* czy *Windows Mobile*. Systemy te używane są obecnie w ogromnej ilości różnego rodzaju smartfonach, a także w coraz bardziej popularnych tabletach i netbookach. Urządzenia te ze względu na swoją „mobilność” są w głównych założeniach sprzętowych zazwyczaj bardzo do siebie podobne. Różnice pomiędzy nimi nie wpływają w dużym stopniu na ogólną wydajność, nawet jeżeli oparte

są na odmiennych jednostkach centralnych. Istotnym czynnikiem wpływającym na utrzymywanie tych wszystkich urządzeń na jednakowym poziomie wydajności jest wynikające z istoty mobilności - zasilanie bateryjne lub akumulatorowe. Jeżeli producent urządzenia przy zbliżonej pojemności baterii chce utrzymać jego konkurencyjny czas pracy, to ogólna moc obliczeniowa także musi być zbliżona do konkurencji. Różnice niemające wpływu na wydajność, lecz z którymi musi się liczyć twórca oprogramowania, to parametry takie jak rozdzielczość ekranu, rodzaj klawiatury, a także rodzaj wyposażenia, jak sterowanie dotykowe, aparat fotograficzny czy odbiornik GPS. Nie zmienia się natomiast „klasa" sprzętu, co bardzo ułatwia pracę producentom gier i aplikacji oraz niewątpliwie przyczynia się do dostępności bogactwa oprogramowania dla tych systemów.

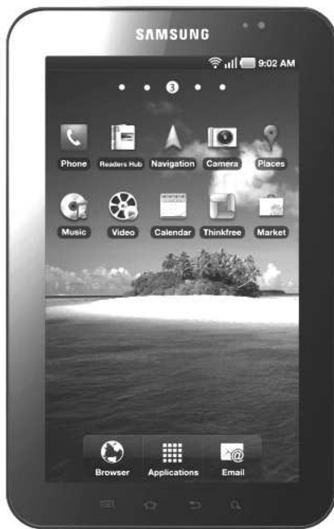
Na rynku zaczyna się jednak pojawiać grupa urządzeń tego typu, które mimo zachowania kompatybilności w zakresie platformy programowej w wyraźny sposób wyprzedzają inne modele pod względem wydajności karty graficznej lub mocy procesora. W tym przypadku będziemy musieli specjalnie obsłużyć mocniejszą wersję urządzenia lub też porzucić zgodność gry ze słabszymi urządzeniami. Przykładem mogą być tutaj tablety *iPad* oraz *iPad2*.

Innym rodzajem wspólnej platformy programowej mogą być silniki zaprojektowane dla języków interpretowanych lub wykorzystujących kod pośredni. Gra wideo oparta na takim języku będzie mogła funkcjonować na każdej platformie, na której będzie dostępny interpreter lub maszyna wirtualna danego języka, które będą właśnie stanowić wspólną platformę programową.

Językami kodu pośredniego wykorzystywanymi w tym celu mogą być na przykład „Flash/ActionScript" lub „Java", które są obecnie niezwykle popularne przy tworzeniu wszelkich gier wideo dostępnych dla użytkownika z poziomu przeglądarek internetowych. W przypadku języków interpretowanych dużą popularność zdobył „Python", który dzięki swojemu wysokiemu poziomowi abstrakcji świetnie nadaje się do szybkiego prototypowania gier wideo lub ich elementów.

- *Silnik dla wielu platform programowych i wspólnej platformy sprzętowej*

  Odrębny typ silnika współpracuje z jednym typem sprzętu, ale z wieloma platformami programowymi. Przykładem może być oprogramowanie przeznaczone dla komputerów osobistych klasy PC, lecz obsługujące więcej niż jeden system operacyjny, np. Windows i Linux. Tworzenie gier jednocześnie na Windowsa i Linuxa nie jest jeszcze zbyt popularne na rynku komercyjnym, lecz pod ten przykład możemy też podciągnąć rozwiązanie częściej spotykane, mianowicie silnik przeznaczony do współpracy z komputerem klasy PC opartym na systemie Windows oraz z komputerem Mac opartym na systemie Mac OS. Obecne serie komputerów Mac oparte są na tej samej rodzinie mikroprocesorów co PC (x86 oraz x86_64), a także większość podzespołów jest ze sobą zgodna. Powoduje to, że PC i Mac stanowią w za-



Samsung Galaxy Tab

Motorola Droid X

Toshiba AC100

Rysunek 2.11. Trzy urządzenia mobilne różnego typu - tablet, netbook i smart fon, pod kontrolą tego samego systemu operacyjnego - Android

sadzie dla programistów wspólną platformę sprzętową, różniącą się jedynie systemem operacyjnym. Te dwie platformy posiadają obecnie największe udziały w światowym rynku komputerów osobistych, a rynkowa walka pomiędzy nimi ciągle trwa. Bezpiecznym posunięciem dla deweloperów gier wideo dla komputerów osobistych jest wydawanie produktów dla obu sys-

temów. Przykładem firmy praktykującej ten zwyczaj jest Blizzard, który już od kilku lat wydaje wszystkie swoje produkcje dla platform Mac/PC. Przykładem takiego typu silnika może być „Irrlicht Engine"[18] - open source'owy silnik gier wideo przeznaczony dla komputerów osobistych korzystających z systemów Windows, Linux i Mac OS.

- *Silnik dla wielu platform programowych i wielu platform sprzętowych*

  Najbardziej zaawansowanym typem silników gier wideo są te, które współpracują z wieloma platformami zarówno sprzętowymi, jak i programowymi. Przykładowo silnik tego typu może wspomagać tworzenie gier dla komputerów osobistych, konsol stacjonarnych i konsol przenośnych. Stworzenie i rozwój tego typu silnika stanowi duże wyzwanie dla projektantów i programistów. Wspierane platformy znacznie różnią się między sobą wydajnością, obsługiwanymi technologiami, a także architekturą samego sprzętu. Duże różnice dotyczą też formatów danych wejściowych w postaci plików danych graficznych czy dźwiękowych, które często są specjalizowane pod konkretny rodzaj platformy, zapewniający szybkie (sprzętowe) ich przetwarzanie. A najlepiej jeśli wszystko to jest przed programistą lub designerem tworzącym grę wideo sprawnie ukryte, aby nie musiał sobie z tych różnic w ogóle zdawać sprawę.

  Trudności, które tworzą się podczas rozwoju takiego silnika, sprawiają, że jeżeli oczekujemy po nim niezawodności, dojrzałości i sprawnej pracy na wszystkich platformach, to w tej dziedzinie na rynku liczą się tylko najwięksi producenci. Wynika to przede wszystkim z ogromu pracy, jaki należy włożyć i to nawet nie w samo jego stworzenie, ale przede wszystkim w utrzymanie i testowanie, przy nieustannie rozwijających się poszczególnych platformach i ich środowiskach programowych oraz wszelkich bibliotekach i technologiach, które silnik obsługuje. Ogrom potrzebnej pracy wymaga dużego i dobrze zorganizowanego projektu open-source'owego lub po prostu pokaźnego budżetu w przypadku projektu komercyjnego.

Tablica 2.1. Popularne, komercyjne wieloplatformowe silniki gier wideo i obsługiwane przez nie systemy

Silnik	**Obsługiwane platformy**
Unreal Engine	Windows, Linux, Mac OS, iOS (iPhone, iPad), Dreamcast, Xbox, Xbox 360, PlayStation 2, PlayStation 3
CryEngine	Windows, Xbox 360, PlayStation 3
Gamebryo[19]	Windows, Xbox, Xbox 360, GameCube, Wii, PlayStation 2, PlayStation 3
Unity[20]	Windows, Mac OS, iOS, Android, Xbox 360, PlayStation 3

[18] Irrlicht Engine - http:/irrlicht.sourceforge.net
[19] Gamebryo - http://www.gamebryo.com
[20] Unity Engine - http:/unity3d.com

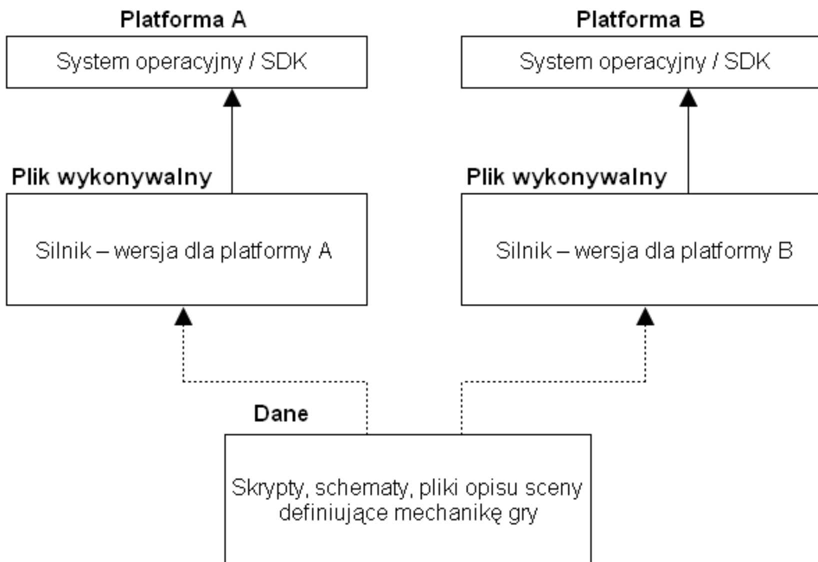
Wspólna implementacja mechaniki gry wideo

Wieloplatformowy silnik edytorowy (sterowany danymi)

W przypadku oparcia gry wideo na silniku edytorowym, efektem pracy programisty lub designera, stanowiącym jej logikę i mechanikę jest zestaw danych, który zależnie od silnika, może być skryptem języka interpretowanego, schematem blokowym przepływu sterowania, sceną z rozmieszczonymi elementami interaktywnymi lub połączeniem wszystkich tych możliwości. Nawet w przypadku skryptu mówimy tutaj o „danych", gdyż jego kod nie stanowi bezpośrednich rozkazów dla procesora i SDK ale będzie interpretowany przez silnik, tak samo jak interpretowane będą dane rozmieszczenia obiektów na scenie czy diagram sterowania. Osiągnięcie dobrej „wieloplatformowości" w tego typu silnikach jest relatywnie proste w stosunku do silników SDK z dwóch powodów.

Po pierwsze, silniki udostępniają użytkownikom obiekty o stosunkowo wysokim poziomie abstrakcji. Różnice w poszczególnych platformach zwiększają się wraz ze zmniejszaniem poziomu abstrakcji, co wiąże się z docieraniem do coraz bardziej specyficznych cech danej platformy programowej lub sprzętowej. Ograniczenie jego minimalnego poziomu odcina większość tych szczegółów, które ukryte będą w implementacji silnika, niedostępnej dla użytkownika.

Drugim ułatwieniem przy implementowaniu silników edytorowych jest to, że wszystkie warunki, jakie przed nim stawiamy, można uprościć do jednego



Rysunek 2.12. Sposób implementacji mechaniki gry wieloplatformowym silniku sterowanym danymi

założenia - na wszystkich platformach dla takiego samego zestawu danych wejściowych chcemy uzyskać takie same lub jak najbardziej zbliżone rezultaty. Sposób realizacji tego założenia jest już sprawą tylko i wyłącznie producenta silnika i może być rozwiązany jakkolwiek, nawet w zupełnie odrębny sposób dla poszczególnych platform. Rzeczywiste implementacje silników łączą oczywiście elementy realizowane wspólnie, jak i oddzielnie, możemy jednak rozróżnić te dwie tendencje. Im więcej elementów wspólnych, tym ta implementacja jest trudniejsza, gdyż sam silnik musi być wtedy pisany według reguł wieloplatformowych. Łatwiejsze będzie natomiast zarządzanie takim silnikiem i jego rozwój, gdyż czynności te wykonywać będziemy na jednym kawałku kodu. Odwrotny skutek ma natomiast zwiększanie liczby elementów implementowanych oddzielnie - sama implementacja będzie prostsza, gdyż w każdej wersji nie musimy brać pod uwagę innych platform, bardziej pracochłonne będzie natomiast zarządzanie, gdyż wszystkie prace musimy wykonać wielokrotnie i z zachowaniem pomiędzy nimi spójności.

Wieloplatformowy silnik w postaci SDK

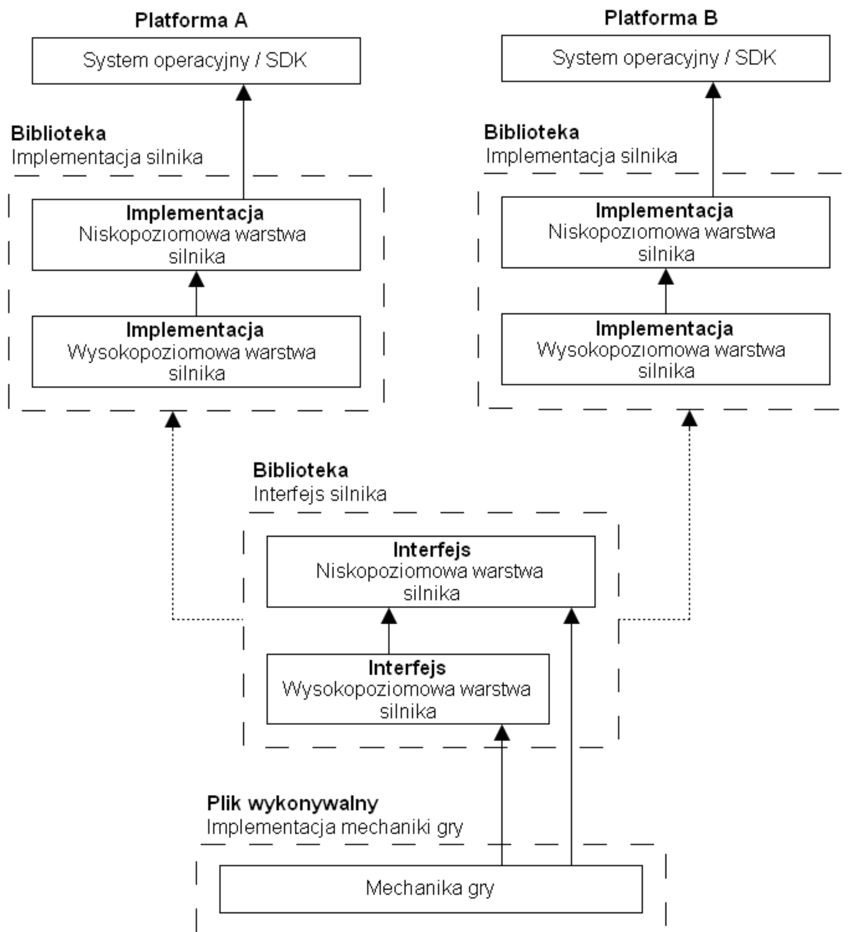
Silnik w postaci SDK stanowi w zasadzie dużą bibliotekę programistyczną zawierająca funkcjonalność wspomagającą tworzenie gier wideo. Przy korzystaniu z takiego typu silnika mechanikę i logikę tworzonej gry piszemy w natywnym języku programowania z wykorzystaniem tej właśnie funkcjonalności. Aby silnik uznać za wieloplatformowy, musi spełnić on warunek, że mechanikę tę tworzymy tylko raz, z wykorzystaniem funkcji, klas i struktur obecnych w silniku, następnie zaś, kodu tego bez zmian (lub prawie bez zmian) możemy użyć do kompilacji gry wideo i będzie on działał w taki sam sposób na różnych platformach. Warunek ten daje nam pewne wskazówki odnośnie tego, jak może w ogólny sposób wyglądać architektura takiego silnika. Zgodnie z wcześniejszymi ustaleniami silnik gier wideo w postaci SDK można podzielić na warstwy odpowiadające różnym poziomom abstrakcji. W podejściu wieloplatformowym możemy ten podział bardziej rozwinąć, dzieląc każdą warstwę na dwie części - podwarstwę interfejsu i podwarstwę implementacji.

Interfejs warstwy stanowi zbiór klas i funkcji, które są widoczne „na zewnątrz" i z których może korzystać programista wykorzystujący silnik. Warstwa ta jest niezmienna, niezależnie of platformy programowej lub sprzętowej, dla której kod jest kompilowany. Zawiera głównie deklaracje klas i funkcji oraz ewentualnie definicje tych, które nie są w żaden sposób zależne od platformy.

Implementacja warstwy gromadzi definicje klas i funkcji zawartych w jej interfejsie oraz inne klasy i funkcje pomocnicze, które jednak nie są widoczne dla programisty użytkującego silnik. Implementacje dla poszczególnych platform mogą być zupełnie odrębne lub przeplecione w jednym wspólnym kodzie, w zależności od przyjętego schematu oraz cech użytego języka programowania.

Nie wszystkie warstwy w silniku SDK muszą posiadać interfejs. Projektując taki silnik, możemy zdecydować, jaki będzie minimalny poziom abstrakcji,

jaki udostępnimy użytkownikom, a w związku z tym, które najniższe warstwy będą ukryte. Po raz kolejny już powtórzmy, że różnice pomiędzy platformami zwiększają się wraz ze zmniejszaniem się poziomu abstrakcji, a skutkuje to tym, że im niższa warstwa, tym trudniej dla niej zaprojektować wspólny, wieloplatformowy interfejs. W związku z tym im więcej warstw „ukryjemy" przed użytkownikami, tym bardziej ułatwiamy sobie zadanie przy tworzeniu silnika. Jednocześnie jednak każda ukryta warstwa zmniejsza pulę możliwości, jaką dajemy użytkownikowi na dokładne dostosowanie gry wideo do własnych potrzeb, gdyż będzie musiał ją budować z „większych klocków".

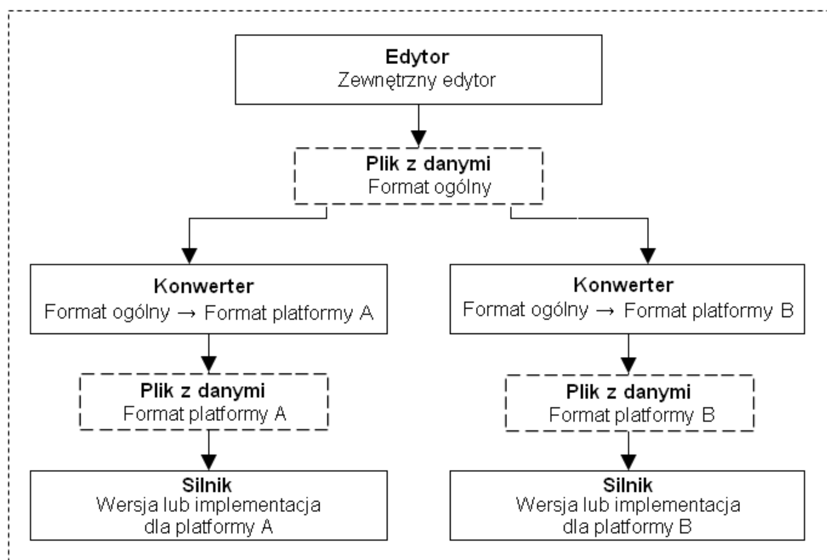


Rysunek 2.13. Sposób implementacji mechaniki gry w wieloplatformowym silniku w postaci SDK

Jednolita obsługa danych w silniku wieloplatformowym

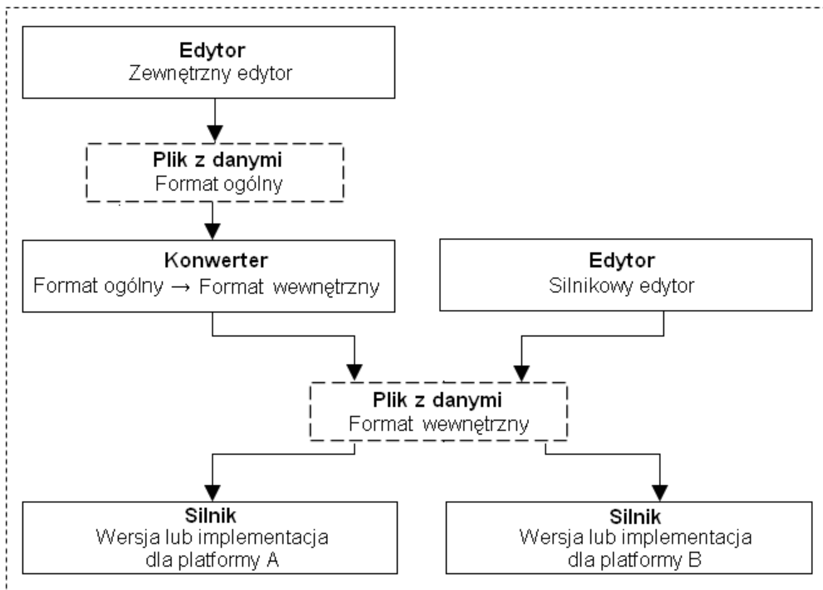
Wieloplatformowy silnik gier wideo musi zapewniać jednolity sposób obsługi danych. Najczęściej jest to realizowane na dwa sposoby.

- *Dostarczanie narzędzi konwertujących na poszczególne platformy*
  Pierwszym z nich jest dostarczanie wraz z silnikiem zestawu narzędzi konwertujących dane do formatu wymaganego przez SDK dla danej platformy. W tym przypadku dane w postaci źródłowej, przygotowane za pomocą narzędzi niewchodzących w skład silnika przygotowujemy raz. Konwertery obsługują najczęściej popularne formaty danych wejściowych, dzięki czemu przy ich tworzeniu można skorzystać z dobrych i sprawdzonych narzędzi takich jak np. „Adobe Photoshop” w przypadku grafiki 2D lub „3ds Max” dla grafiki 3D. Następnie w czasie przygotowywania produktu do uruchomienia na danej platformie dane te są konwertowane do odpowiedniego formatu, co może być procesem wykonywanym ręcznie lub zautomatyzowanym (np. za pomocą skryptu systemowego), w zależności od ilości tych danych i od częstotliwości ich zmian podczas procesu tworzenia gry wideo.



Rysunek 2.14. Obsługa danych w silniku wieloplatformowym przez konwersję do formatu natywnego

- *Własny wspólny format danych*
  Drugim sposobem radzenia sobie z danymi jest stosowanie przez silnik gry wideo swojego własnego, wewnętrznego formatu. W tego typu rozwiązaniu



Rysunek 2.15. Obsługa danych w silniku wieloplatformowym przez konwersję do formatu wewnętrznego

dane w postaci źródłowej przygotowywane są w edytorach zewnętrznych i przekształcane do formatu wewnętrznego przy użyciu dostarczonych konwerterów lub przygotowywane w edytorach dostępnych w silniku i od razu zapisywane w formacie wewnętrznym. Następnie w czasie przygotowywania do uruchomienia na każdej z platform docelowych nie musimy już przygotowywać osobnych wersji danych. Silnik jest przystosowany do obsługi wewnętrznego formatu i korzysta bezpośrednio z niego na każdej platformie. Jeżeli jakieś ewentualne konwersje są jeszcze konieczne, to wykonuje je sam w sposób niewidoczny dla osoby tworzącej grę wideo.

3

Architektura silnika wieloplatformowego

3.1 Wybór języka programowania

Jedną z najważniejszych decyzji, jaką musimy podjąć, decydując się na napisanie własnego silnika gier wideo, jest wybór języka programowania. Wyboru tego powinniśmy dokonać na podstawie kilku kryteriów, z których praktycznie wszystkie można zawrzeć w odpowiedzi na pytanie: „Jakie ma być przeznaczenie tworzonego silnika?”. Gry wideo powstają na tak odmienne platformy i w tak wielu technologiach, że nie istnieje jeden najlepszy, uniwersalny w każdym przypadku język programowania.

3.1.1 Rodzaje języków programowania

Kod programu komputerowego stanowi zestaw instrukcji, które zostaną wykonane podczas jego uruchamiania. Jednak w zależności od rodzaju języka wykonywanie instrukcji może być realizowane w różny sposób.

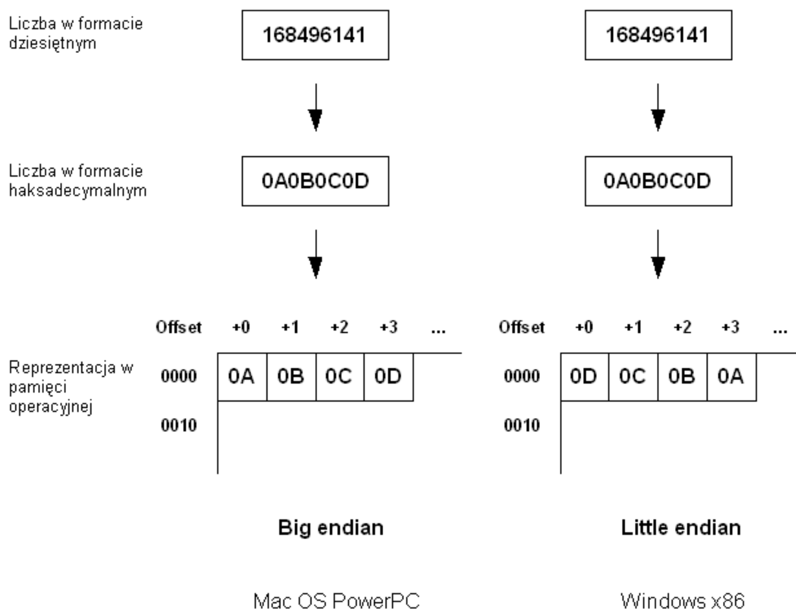
Języki kompilowane do kodu maszynowego (natywne)

Język natywny jest to język, którego kod źródłowy jest konwertowany do kodu maszynowego, czyli postaci binarnej, w której zakodowane są instrukcje bezpośrednio wykonywane przez procesor danego urządzenia. Plik wynikowy tej konwersji jest to plik zawierający kod maszynowy, który jest uruchamiany przez system operacyjny i wykonywany przez procesor. Języki natywne są językami programowania zapewniającymi najszybsze wykonanie programu, gdyż przy ich uruchamianiu nie występuje już żadna konwersja, a tylko przesłanie instrukcji do procesora.

Przy obsłudze wieloplatformowości kod źródłowy programu musi zostać skompilowany osobno dla każdej platformy, do pliku wykonywalnego w formacie dla niej specyficznym. W szczególności oznaczać to może potrzebę generacji

kilku lub kilkunastu wersji pliku wykonywalnego rekompilowanych przy okazji tworzenia każdej nowej wersji programu.

W przypadku kodu źródłowego napisanego w języku niskopoziomowym, takim jak Asembler, nie jest możliwa kompilacja takiego programu na inną architekturę sprzętową - potrzebujemy tu osobnego kodu źródłowego dla każdej platformy. Języki wyższego poziomu, takie jak Pascal, C, C++, są w swojej składni niezależne od architektury sprzętowej i programowej, więc posiadają kompilatory dla różnych platform. W praktyce jednak kompilacja i uzyskanie takiego samego działania nie jest prostą sprawą. Pierwszym problemem są biblioteki, które zapewniają dostęp do SDK systemu operacyjnego, a także różne inne biblioteki obsługujące chociażby system graficzny. Biblioteki te będą się różnić w zależności od platformy, co zmusi nas do modyfikacji kodu źródłowego. Innym problemem dużo trudniejszym do wyeliminowania jest niewłaściwe użycie tych języków. Języki te, pomimo że zaliczane są do języków wysokopoziomowych, zapewniają bezpośredni dostęp do pamięci operacyjnej, co między innymi powoduje, że są tak wydajne. Struktura danych w pamięci jest jednak zależna od architektury procesora lub nawet użytego kompilatora. Programista mógł więc założyć, że dane w pamięci są w pewnym formacie i wykorzystać bezpośredni dostęp do nich w celu optymalizacji, co niestety nie będzie spełnione na innej platformie. Może to powodować błędy, trudne do wykrycia,



Rysunek 3.1. Różna kolejność bajtów liczb całkowitych zapisanych w pamięci dla różnych platform - przykład różnic struktury danych

gdyż program będzie jak najbardziej poprawny pod względem składniowym, skompiluje się i uruchomi, lecz będzie działał nieprawidłowo, co może objawiać się w bardzo niekontrolowany sposób.

Języki kompilowane do kodu pośredniego

Języki kompilowane kodu pośredniego, podobnie jak języki natywne kompilowane są do pewnego rodzaju kodu maszynowego. Ten kod maszynowy nie jest zestawem instrukcji żadnego rzeczywistego procesora, lecz zestawem instrukcji dla maszyny, która jest emulowana na rzeczywistym urządzeniu. To emulowane urządzenie nazywamy maszyną wirtualną. Wirtualna maszyna danego języka jest to program, najczęściej napisany w postaci natywnej, który emuluje urządzenie o ściśle określonej specyfikacji. Wygenerowany kod maszynowy dla wirtualnej maszyny nazywamy bajtkodem (ang. *bytecode*) lub właśnie kodem pośrednim.

Wykonywanie programu w języku kodu pośredniego odbywa się wolniej niż programu natywnego, ze względu na czas potrzebny na konwersję instrukcji w kodzie pośrednim na instrukcję kodu natywnego. Obecnie maszyny wirtualne często optymalizują ten proces kodu pośredniego poprzez kompilację całego fragmentu bajtkodu do kodu natywnego bezpośrednio przed jego wykonaniem (mechanizm JIT). Przykładowymi językami działającymi w tej technologii są Java, C# oraz Flash/ActionScript.

Mechanizm kodu pośredniego znacznie poprawia obsługę wieloplatformowości przez programy napisane w takich językach. Po pierwsze ten sam bajtkod może być użyty do uruchamiania na wielu platformach i niepotrzebna jest wielokrotna kompilacja. Po drugie, ściśle określona specyfikacja maszyny wirtualnej eliminuje różnice wynikające z innej struktury danych w pamięci operacyjnej. Program skompilowany do bajtkodu może w przyszłości uruchamiać się na platformie sprzętowej o zupełnie odmiennej, nieznanej obecnie architekturze, bez konieczności ponownej kompilacji.

Problemy z przenośnością na inne platformy mogą jednak wynikać przez wykorzystanie bibliotek, ekskluzywnych dla danej platformy lub dostępnych tylko na niektóre z nich, np. tylko na systemy operacyjne komputera osobistego PC.

Języki interpretowane

Program w języku interpretowanym - to program, który jest rozprowadzany i uruchamiany jako kod źródłowy w postaci tekstowej. Wykonaniem instrukcji zajmuje się osobny program zwany interpreterem (napisany najczęściej w języku natywnym), który w momencie uruchomienia czyta kolejne instrukcje z kodu źródłowego, interpretuje je i wykonuje.

Z zasady działania tych języków możemy wywnioskować, że są one bardzo wolne w działaniu. Każda najmniejsza nawet instrukcja w kodzie źródłowym,

która przez procesor mogłaby być wykonana w ciągu kilku cykli, musi zostać najpierw odczytana z pliku źródłowego, odnaleziona w zestawie instrukcji języka i dopiero wykonana, co w sumie może zająć tysiące cykli procesora. Dzisiaj wiele interpreterów optymalizuje działanie programu przez wstępną kompilację jego fragmentu do kodu pośredniego lub nawet natywnego (JIT), tuż przed jego wykonaniem.

Języki interpretowane są językami wysokiego poziomu. Oddzielają programistę od szczegółów architektury urządzenia, na którym jest on wykonywany, zwłaszcza szczegółów budowy pamięci operacyjnej poprzez brak bezpośredniego dostępu oraz najczęściej występujący system słabego typowania. Podejście takie z jednej strony uniemożliwia optymalizację pewnych algorytmów, lecz z drugiej strony wymusza pisanie kodu w sposób bardziej wysokopoziomowy, pozostawiając implementacje niskopoziomowych elementów zewnętrznym bibliotekom, które mogą być napisane z wykorzystaniem języków natywnych. Dzięki temu programy w języku skryptowym pisze się relatywnie szybko, gdyż kod programu skupia się bardziej na jego logice niż implementacji drobnych elementów. Zapewnia to także wieloplatformowość. Jeżeli znajdziemy interpreter języka dla danej platformy i wszystkie użyte przez nas biblioteki, to możemy program lub grę bez problemu na niej uruchomić.

Zaletą języków interpretowanych jest także fakt, iż nie wymagają one kompilacji. W przypadku dużych gier proces kompilacji może trwać nawet kilkadziesiąt minut. Tymczasem konieczność czekania nawet kilkudziesięciu sekund na rekompilację podczas wprowadzania i testowania drobnych zmian w kodzie przekłada się bezpośrednio na wydajność i komfort pracy.

Przykładem szeroko wykorzystywanego języka interpretowanego jest Python. Jest to język, którego interpretery możemy odnaleźć na większości możliwych platform, wraz z całym zestawem bibliotek wszelkiego zastosowania. W przypadku tworzenia gier wideo szczególnie warta uwagi jest biblioteka Pygame wspomagająca tworzenie gier wideo i aplikacji multimedialnych.

Mimo technik optymalizacyjnych programy napisane w językach interpretowanych działają wolno. Języki te nie nadają się więc do pisania kluczowych wydajnościowo elementów, lecz doskonale sprawdzają się w prototypowaniu, gdzie szybko chcemy uzyskać efekt i przetestować go w praktyce. Drugim powszechnym zastosowaniem jest użycie tych języków jako językó skryptowych silnika lub dla warstw silnika najwyższych poziomów abstrakcji.

3.1.2 Język programowania a przeznaczenie silnika gier wideo

Rozpatrzmy gry z perspektywy podziału na gry przeglądarkowe, działające w oknie przeglądarki internetowej oraz na gry działające bezpośrednio w systemie operacyjnym danego komputera lub konsoli. Gry przeglądarkowe można podzielić dodatkowo na dwie zupełnie odmienne kategorie, a mianowicie na wykorzystujące tylko standardowe technologie Webowe oraz gry wideo wykorzystujące wtyczki lub inne rozwiązania, zagnieżdżone na stronie internetowej.

Gry przeglądarkowe oparte na bazodanowych technologiach webowych

Technologie webowe są to wszelkie rozwiązania wspomagające tworzenie stron i serwisów WWW. Zaliczyć do nich możemy języki HTML, PHP, JavaScript, technologie ASP, JSP oraz wiele innych. Z wykorzystaniem takich technologii powstają również z powodzeniem gry. Najczęściej są to gry wieloosobowe określane mianem MMO (*Massively Multiplayer Online*) i pozwalają na rozgrywkę setkom lub tysiącom graczy jednocześnie.

Tu pojawia się jednak pytanie, czy tego typu gry możemy zaliczyć do „gier wideo". Mechanika tych gier tworzona jest z wykorzystaniem technologii webowych i bazodanowych, a ich interfejs opiera się na przeglądarkowych formularzach. Brak w nich, tak charakterystycznego dla klasycznych gier wideo, prezentowania przebiegu rozgrywki w czasie rzeczywistym za pomocą urządzenia renderującego. Przebieg gry w czasie rzeczywistym jest realizowany na serwerze, jednakże gracz wykorzystując przeglądarkę internetową, ma możliwość wydawania poleceń poprzez odnośniki lub formularze na stronie, oraz sposobność obserwacji aktualnego stanu rozgrywki jako różnego rodzaju statystyki czy wizualizacje.

Rysunek 3.2. Popularna gra przeglądarkowa OGame

Tworzenie gier tego typu jest procesem zupełnie odmiennym od tworzenia klasycznej gry wideo, przypomina bardziej pracę nad bazodanową aplikacją

webową i wymaga tego typu narzędzi. Z tego powodu trudno jest mówić o silniku gier przeglądarkowych w rozumieniu silnika gry wideo, choć oczywiście istnieją pakiety wspomagające implementację tego rodzaju produktów. Architektura i zasada działania takiego silnika jest zupełnie odmiennym tematem i w związku z tym nie będzie omawiana w tej książce.

Przeglądarkowe gry wideo

Przeglądarkowe gry wideo, w odróżnieniu od powyższego rodzaju, wykorzystują do prezentacji wizualnego stanu metodę renderowania poszczególnych klatek składających się na animację. Urządzeniem renderującym jest w tym przypadku samo okno przeglądarki internetowej lub wtyczka zganieżdżona na stronie, co nie zmienia faktu, że dzięki temu sposobowi działania gry te możemy nazwać już pełnoprawnymi grami wideo.

Jeżeli chcielibyśmy zająć się tworzeniem gry lub silnika tego rodzaju, mamy do wyboru kilka rozwiązań:

- *Języki wbudowane w przeglądarkę*

  Wraz z rozwojem technologii internetowych pojawiają się też metody tworzenia tego typu gier bez korzystania z wtyczek, a jedynie z języków, których obsługa jest wbudowana w przeglądarkę internetową. Językiem takim może być dla przykładu HTML5 w połączeniu z JavaScript. Standard ten znajduje się jeszcze w fazie rozwoju i niewiele przeglądarek, zwłaszcza mobilnych posiada jego pełną obsługę, co nie zmienia faktu, że z jego pomocą jest możliwe stworzenie kompletnej gry wideo. Najlepszym dowodem na to są rozwiązania, które już pojawiają się na rynku, takie jak silnik gier Rocket Engine.
- *Wtyczki ogólnego przeznaczenia*

  Przeglądarki internetowe umożliwiają umieszczanie na stronach zagnieżdżonych elementów za pomocą różnego rodzaju wtyczek. Do najpopularniejszych wtyczek ogólnego przeznaczenia należy technologia Flash wykorzystująca język ActionScript oraz Aplety Javy, wykorzystujące oczywiście język Java. Języki te udostępniają wystarczające możliwości do tworzenia gier wideo lub silnika i obecnie możemy zaobserwować tysiące gier napisanych w tych językach. Do niewątpliwych zalet wtyczek należy ich popularność - większość obecnych komputerów desktopowych posiada te wtyczki zainstalowane oraz niezależność od platformy - jednolitą obsługę na różnych urządzeniach, systemach oraz przeglądarkach gwarantuje producent wtyczki.
- *Specjalizowane wtyczki*

  Oprócz wtyczek ogólnego przeznaczenia istnieją też specjalizowane rozszerzenia dla przeglądarek tworzone przez producentów silników gier wideo. Umożliwiają one uruchomienie gier napisanych za pomocą tych silników ja-

ko zagnieżdżonych elementów strony internetowej. Przykładem mogą być wtyczki udostępnione przez producentów silników Virtools lub Unity. Implementacja gier wideo przy ich użyciu jest ułatwiona dzięki współpracy z wszelkimi narzędziami wchodzącymi w skład silnika, jednak do samej implementacji wykorzystywać będziemy wbudowany w ten silnik język skryptowy. Tak samo jak z wtyczkami ogólnego przeznaczenia, o obsługę wieloplatformową dba jej producent, natomiast gorzej wygląda sytuacja z ich popularnością, ze względu na bardzo specyficzne zastosowanie. Jeśli interesuje nas stworzenie własnego silnika gier w takiej technologii, to będziemy musieli zaimplementować własną wtyczkę do przeglądarki, wykorzystując jeden z natywnych języków programowania, takich jak C++. Jednak w takim przypadku problem obsługi wieloplatformowości leży już po naszej stronie - musimy zapewnić wtyczkę dla każdej platformy, na jakiej chcemy naszą grę wideo uruchomić.

Przeglądarkowe gry wideo zdominowane są przez te zrealizowane w technologii 2D. Wynika to głównie z oczekiwań grupy odbiorców, do jakiej te produkty są adresowane. Wielu spośród nich oczekuje prostej rozrywki podczas przerwy w pracy, a gry oparte na technologiach 2D zazwyczaj są dużo prostsze w opanowaniu i obsłudze. Oczywiście istnieją też gry 3D dla przeglądarek - przykładem może być Minecraft Classic. HTML5 i Java umożliwiają wykorzystanie biblioteki OpenGL, także Flash i ActionsScript udostępnia wsparcie dla sprzętowej akceleracji grafiki 3D.

Niezaprzeczalną zaletą przy tworzeniu przeglądarkowych gier wideo jest ich wieloplatformowość. Jeżeli tylko nie tworzymy własnej wtyczki, a korzystamy z istniejącej technologii, to wieloplatformowość jest już wpisana w jej założenia i uzyskujemy ją niejako „za darmo". Kolejną zaletą jest sposób dystrybucji i uruchamiania, który eliminuje konieczność ściągania i instalacji gry - wystarczy wpisanie odpowiedniego adresu w przeglądarce internetowej. Do jej wad zaliczyć trzeba przede wszystkim ograniczenie w kwestii ilości danych, które może gromadzić gra ze względu na ograniczoną przepustowość łącza internetowego jak również umiarkowaną wydajność działania kodu i ograniczone efekty graficzne.

Gry wideo uruchamiane bezpośrednio w systemie operacyjnym urządzenia

Gry uruchamiane bezpośrednio w systemie operacyjnym danego urządzenia są to samodzielne aplikacje, które nie są zagnieżdżonym elementem innej aplikacji takiej jak przeglądarka internetowa. Można je podzielić na dwie główne grupy:

- *Aplikacje natywne*

  Aplikacje natywne są to aplikacje napisane w języku kompilowanym do natywnego kodu maszynowego danego urządzenia (na przykład dla syste-

Rysunek 3.3. Port gry Quake II w HTML5 uruchomiony w przeglądarce

mu Windows będzie to plik wykonywalny `.EXE` lub biblioteka dynamiczna `.DLL`). Aplikacja natywna umożliwia nam maksymalne wykorzystanie możliwości sprzętowych urządzenia, a także najszybsze wykonywanie napisanego programu. Dlatego jeżeli naszym celem jest tworzenie gier wideo, dla których wydajność sprzętu jest kluczowym parametrem, powinniśmy wybrać lub sami stworzyć silnik gier napisane w technologii natywnej. Do tych celów możemy wykorzystać cały wachlarz języków programowania, które na danej platformie mogą być skompilowane do kodu masznowego, którymi w zależności od potrzebnego poziomu abstrakcji może być Asembler, C++ lub inny obsługiwany na danej platformie język bardzo wysokiego poziomu. Problemem natomiast staje się tutaj wieloplatformowość, gdyż musimy dostarczyć osobne pliki wykonywalne dla każdej platformy. Mimo, że sam wykorzystany język może mieć możliwość kompilacji dla każdej z nich (np. C++), to w praktyce każda platforma korzysta z innych bibliotek i innych funkcji stanowiących chociażby SDK systemu operacyjnego, co wymusza na nas użycie innego lub zmodyfikowanego kodu do kompilacji na każdej z nich.

- *Aplikacje interpretowane lub kodu pośredniego*

  Aplikacje tego typu są napisane w językach, które na danej platformie nie są kompilowane do kodu maszynowego, lecz interpretowane w przypadku języka skryptowego albo wykonywane przez maszynę wirtualną w przy-

padku języka kodu pośredniego. Takimi językami mogą być np. Python, Java lub C#. Silnik napisany w tego typu językach będzie miał oczywiście mniejszą wydajność niż zaimplementowany w sposób natywny, lecz jeżeli naszym celem nie jest tworzenie gier wykorzystujących sprzęt w maksymalny sposób, to języki takie mają swoje zalety. Po pierwsze, są językami wyższego poziomu, co ograniczy możliwości optymalizacji kodu, lecz z drugiej strony zmniejszy potrzebną ilość pracy i przyspieszy proces powstawania silnika. Po drugie, kod silnika oraz kod gry będą działać na wszystkich platformach, na których będzie dostępny interpreter języka lub maszyna wirtualna. Nie musimy się więc martwić tworzeniem wieloplatformowego kodu. Należy jedynie zwrócić uwagę, jeżeli mamy zamiar wykorzystać biblioteki, które nie są zawarte w standardzie danego języka. Niektóre z nich są specyficzne dla jednej konkretnej platformy, a inne mogą obsługiwać tylko niektóre z platform, co musimy mieć na uwadze przy ich wyborze. Wybór języka interpretowanego lub kodu pośredniego może być lepszym wyborem, jeśli nasz silnik ma być przeznaczony do tworzenia mniejszych, prostszych gier, niewymagających maksymalnej wydajności. Implementacja silnika w języku nie-natywnym może być nam też niejako narzucona przez producenta danej platformy programowej, który nie udostępnił dla niej możliwości uruchamiania programów napisanych w sposób natywny. Przykładem może tutaj być Framework XNA, który oficjalnie wspiera jedynie tworzenie aplikacji za pomocą języka C#, lub system operacyjny Android, który do niedawna umożliwiał tworzenie aplikacji jedynie za pomocą języka Java.

3.1.3 Cechy języków programowania

Paradygmat

Paradygmat programowania definiuje, jak za pomocą języka programowania realizowany jest sposób przepływu sterowania w programie komputerowym. W przypadku języków ogólnego przeznaczenia, które można wykorzystać do tworzenia silnika gier komputerowych, możemy je podzielić na spełniające paradygmaty

- *Programowanie obiektowe*

  Programowanie obiektowe jest paradygmatem realizowanym przez większość nowoczesnych wysokopoziomowych języków programowania ogólnego przeznaczenia. Dużą zaletą tego typu programowania jest jego zgodność z naturalnym sposobem myślenia przez umysł ludzki, który definiuje wszystko co widzimy dookoła nas jako obiekty posiadające pewne cechy i właściwości. Mechanizmy obiektowych języków, takie jak dziedziczenie i polimorfizm pozwalają także w naturalny dla nas sposób przedstawić zależności oraz uogólnienia pomiędzy obiektami. Z kolei hermetyzacja i abstrakcja pozwala opisać obiekt za pomocą jego ogólnych właściwości, odgra-

dząjąc nas od nieistotnych szczegółów. Gry wideo często starają się wiernie odwzorować jakiś wycinek rzeczywistości, który widzimy wokół nas, więc języki tego typu doskonale nadają się do modelowania reprezentacji tych obiektów za pomocą kodu programu, co ułatwia projektowanie i implementację gier oraz silników. Przykładami języków obiektowych mogą być języki takie jak C++, C#, Java.

Pierwszy z wymienionych wyżej języków jest językiem obiektowym pozwalającym jednak na korzystanie z wielu mechanizmów nieobiektowych wywodzących się jeszcze z języka C oraz na niskopoziomowy dostęp do pamięci operacyjnej. Ostatnie dwa wymienione języki są to języki w pełni obiektowe. W praktyce, przy tworzeniu wydajnego silnika gier wideo w pewnym sensie „hybrydowe” podejście języka C++ daje nam większe możliwości. Wydajny silnik powinien udostępniać dobrze zaprojektowany, przejrzysty interfejs obiektowy dla jego użytkowników, natomiast wewnątrz, w warstwach ukrytych przed użytkownikiem, może korzystać z niskopoziomowych rozwiązań w celu wydajnych rozstrzygnięć pewnych problemów.

- *Programowanie strukturalne i proceduralne*

  Programowanie strukturalne i proceduralne to paradygmat wywodzący się jeszcze ze starszych języków programowania i obecny chociażby w takich jak C i Pascal. Brak mechanizmów obiektowych spowodował wyparcie tych języków w większości zastosowań ich nowszymi odpowiednikami. Oczywiście stworzenie całego silnika gier wideo za pomocą języków tego typu jest jak najbardziej możliwe, jednak sprawi, że nie byłby on wygodny w projektowaniu i późniejszym użyciu. Ucierpiałaby również współpraca pomiędzy programistami pracującymi nad wspólnym projektem, gdyż mechanizmy abstrakcji i enkapsulacji znacznie ułatwiają rozdzielenie kodu na fragmenty, nad którymi mogą niezależnie pracować programiści.

  Są jednak zastosowania, w których paradygmat ten jest ciągle wykorzystywany. Przykładem będą interfejsy (API) niektórych bibliotek, takich jak chociażby OpenGL. Interfejs OpenGL jest zaprojektowany w postaci proceduralnej i nie zapowiada się, żeby zostały do niego wprowadzone jakieś elementy obiektowe. Po pierwsze, ze względu na kompatybilność wsteczną, a po drugie, ze względu na to, że proceduralność tutaj sprawdza się bardzo dobrze. OpenGL jest interfejsem do urządzenia graficznego, które programista może wyobrazić sobie jako maszynę stanów. Sterowanie maszyną stanów za pomocą wydawania jej rozkazów (wywołania procedur) i przekazywania danych w postaci struktur jest w tym przypadku bardziej intuicyjne oraz wydajniejsze niż rozbicie interfejsu na obiekty.

Wydajność języka

Jeżeli celem silnika gier wideo ma być osiągniecie jak największej wydajności, musimy wybrać odpowiedni do tego język programowania.

Główne cechy języków określające ich wydajność zostały już wymienione wcześniej. Po pierwsze, rodzaj języka. Najszybsze są języki kompilowane do kodu natywnego, a najwolniejsze interpretowane. Po drugie poziom abstrakcji - im niższy, tym pozwala na większą optymalizację i dostosowanie programu do naszych potrzeb, lecz utrudnia implementację.

Inną cechą, na którą warto zwrócić uwagę, jest obecność automatycznego systemu zarządzania pamięcią - tak zwanego *Garbage Collectora*. W nowoczesnych wysokopoziomowych językach programowania, takich jak Java lub C#, system ten dba o dealokację obszarów pamięci zajmowanych przez nieużywane już obiekty. Garbage Collector doskonale sprawdza się przy implementacji różnego rodzaju aplikacji np. bazodanowych, przyspieszając znacznie ich tworzenie i zmniejszając podatność na błędy. Lecz jeśli zależy nam na wydajności, automatyczne odśmiecanie nie zawsze pomaga. Przede wszystkim system ten nie zna intencji programisty i nie zawsze jest w stanie wystarczająco wcześnie odgadnąć, iż dany obiekt nie jest już używany i może zostać zwolniony. W przypadku pisania gier wideo dla komputerów PC, gdzie przeważnie mamy dostępną dosyć sporą ilość pamięci operacyjnej, nie musi być to dużym problemem. Jeżeli natomiast piszemy grę na konsolę, gdzie tej pamięci, nawet w konsolach nowych generacji nie ma zbyt wiele, opóźniona dealokacja może spowodować błąd braku pamięci i zawieszenie urządzenia. Innym problemem jest też nieprzewidywalność tego systemu. Garbage Collector posiada pewne mechanizmy pozwalające sterować jego pracą, lecz nigdy nie mamy nad nim pełnej kontroli. Może to skutkować masowym zwalnianiem wielu obiektów w najmniej odpowiednim momencie, co zaowocuje charakterystycznym zacięciem się gry wideo.

3.1.4 Wybór języka dla celów tej książki

W pracy tej dla przykładów implementacji różnych mechanizmów stosować będziemy język C++. Język ten ze względu na swoje cechy wysoko i niskopoziomowe jest obecnie najczęściej wykorzystywany przy tworzeniu systemów, gdzie wydajność kodu jest kluczową cechą. Ponieważ w przeciwieństwie do języków kodu pośredniego, o wieloplatformowość musimy w C++ zadbać sami, można na jego przykładzie zaprezentować różne podejścia do tworzenia przenośnego kodu źródłowego.

3.2 Organizacja logiczna

3.2.1 Podział funkcjonalny elementów silnika

Zgodnie z przyjętymi założeniami poszczególne elementy silnika będziemy projektować, korzystając z obiektowego paradygmatu programowania, więc projekt architektury silnika będzie się składał ze zbioru klas komunikujących się

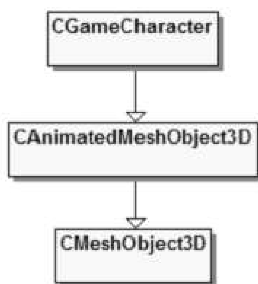
ze sobą i połączonych wzajemnie w celu wykonania określonych zadań. Silnik gier wideo może zawierać setki klas pełniących różnorakie funkcje, dlatego warto uściślić ogólny zarys tej architektury poprzez podział tych klas na podstawowe grupy funkcjonalne.

- *Klasy pomocnicze i narzędziowe*

  Tę grupę stanowią różnego rodzaju klasy niezwiązane bezpośrednio z architekturą silnika gier wideo, lecz przydatne jako elementy do implementacji pozostałych obiektów. Klasy zawarte w tej grupie możemy podzielić na kilka rodzajów. Mogą reprezentować obiekty zależne od platformy, jak choćby klasa opakowująca funkcjonalność wątku systemowego lub obiekty matematyczne, takie jak wektory oraz macierze, które w zależności od architektury sprzętowej mogą korzystać z instrukcji SIMD procesora, przyspieszających obliczenia. Innymi obiektami mogą być ulepszone wersje klas dostępnych w bibliotece standardowej, które w standardowej wersji nie nadają do użycia w silniku ze względów wydajnościowych. Klasami, które często implementowane są na nowo w silnikach gier wideo, są na przykład zamienniki kontenerów takich jak `std::vector` lub klas obsługujących łańcuchy napisowe, jak `std::string`. Pozostałe klasy dostępne w tej grupie to po prostu różne przydatne implementacje algorytmów czy wzorców projektowych, które nie są dostępne w używanych przez nas bibliotekach.

- *Klasy elementów sceny*

  Obserwując ekran wyświetlany przez grę wideo, widzimy wycinek świata, lub raczej scenę, którą gra prezentuje. Scena ta składa się elementów rozmieszczonych przez projektanta w celu uzyskania pożądanego efektu. Przykładowo obserwując ekran trójwymiarowej gry RPG, moglibyśmy zobaczyć teren, po którym chodzimy albo statyczną geometrię użytą do wstawienia drzew, kamieni lub budynków, obiekt światła kierunkowego symulującego słońce, obiekty dźwiękowe, które słyszymy, przechodząc obok nich, oraz obiekty postaci. Wszystko to składa się z elementów, które silnik udostępnił programistom gier lub skrypterom, aby mogli z nich zbudować scenę gry. Do grupy tej należeć mogą różne rodzaje obiektów graficznych, dźwię-

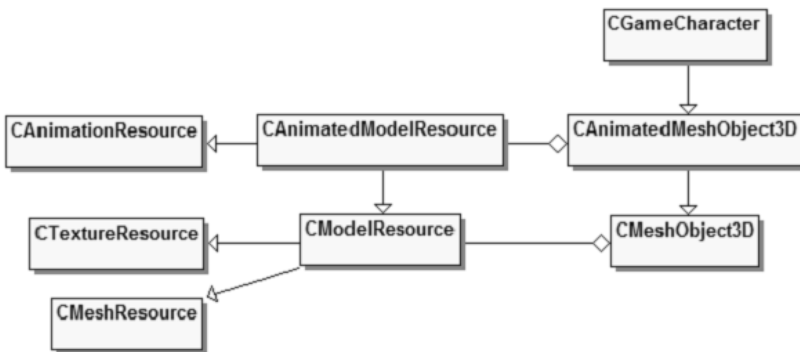


Rysunek 3.4. Hierarchia przykładowych obiektów graficznych

kowych, kolizyjnych, triggerów dla logiki gry i wiele innych. Klasy te mogą także reprezentować różne poziomy abstrakcji silnika, przez co mogą należeć do różnych jego warstw, a także tworzyć hierarchię.

- *Klasy zasobów*

  Zasobem w silniku gier wideo może być dowolna struktura danych tworzonych dynamicznie bądź też wczytywana z pliku. Mogą to być np. tekstury, dane dźwiękowe, dane geometrii 3D. Klasy zasobów są niskopoziomowym elementem silnika gier wideo, które wykorzystywane są przez obiekty sceny jako dane niezbędne do ich funkcjonowania. Klasy zasobów opakowują surowe formaty danych, aby łatwiej można było nimi zarządzać. Dbają one o to, by na przykład nie tworzyć duplikatów tego samego pliku, lub by usuwać dane z pamięci, kiedy nie są już używane. Klasy zasobów mogą także tworzyć hierarchie, stając się coraz bardziej złożone.



Rysunek 3.5. Hierarchia przykładowych obiektów graficznych z zasobami

- *Klasy urządzeń*

  Klasy urządzeń to klasy udostępniające funkcjonalność elementów sprzętowych, z których zbudowana jest konsola lub komputer. Klasy te mogą reprezentować dla przykładu dostępne urządzenia pamięci, urządzenia graficzne i ekrany, urządzenia dźwiękowe lub podłączone kontrolery takie jak gamepad lub mysz. Obiektów klas urządzeń nie tworzy programista, lecz system zarządzający urządzeniami, wykrywając ich podłączanie i odłączanie od konsoli bądź komputera. System ten informuje o ich podłączaniu lub odłączaniu. Większość urządzeń takich jak renderujące lub dźwiękowe jest niskopoziomowymi obiektami wykorzystywanymi wewnątrz silnika. Niektóre z ich, takie jak kontrolery stanowią element wysokopoziomowego interfejsu, z których może korzystać programista gry.

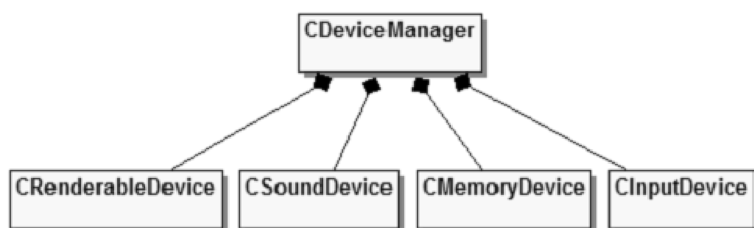
- *Klasy menadżerów*

  Większość obiektów, które istnieją w silniku i stanowią jego właściwą architekturę, możemy przydzielić do jednej z powyższych trzech grup. Prawie

każdy stworzony obiekt w ogólnym pojęciu będzie urządzeniem, zasobem lub elementem sceny. Zarządzaniem tymi obiektami zajmują się klasy menadżerów. W zależności od przeznaczenia menadżera zajmują się one tworzeniem, niszczeniem i obsługą prawidłowego funkcjonowania obiektów.

– *Menadżer urządzeń*

  Menadżer urządzeń zajmuje się tworzeniem i usuwaniem obiektów klas reprezentujących urządzenia, gdy są podłączane lub odłączane. Umożliwia informowanie o pojawieniu się danego typu urządzenia oraz pobranie obiektu reprezentującego urządzenie w celu wykorzystania jego funkcjonalności.



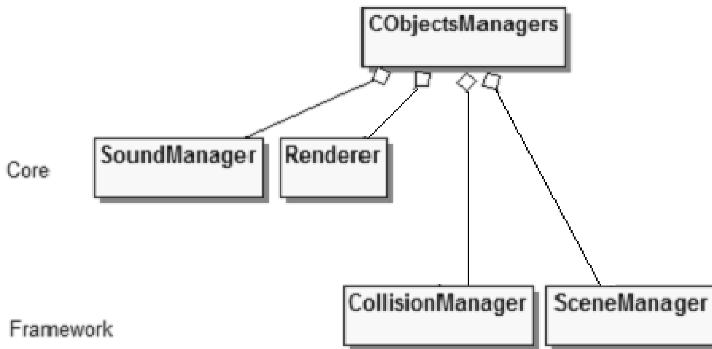
Rysunek 3.6. Menadżer urządzeń

– *Menadżer zasobów*

  Menadżer zasobów zajmuje się tworzeniem obiektów zasobów dla innych obiektów, które tego potrzebują, oraz ich usuwaniem, gdy staną się niepotrzebne. Jego ważnym zadaniem jest dbanie, aby nie dublować zasobów w przypadku, gdy kilka obiektów korzysta na przykład z tej samej tekstury.

– *Menadżery elementów sceny*

  Menadżery elementów sceny to grupa menadżerów zajmujących się obsługą istniejących obiektów, w zależności od ich przeznaczenia. Jednym z takich menadżerów jest renderer, który zarządza wszystkimi obiektami graficznymi oraz kieruje odpowiednie instrukcje do urządzeń renderujących, aby wyświetlić je na ekranie. Analogiczną funkcję pełni menadżer dźwięku. Istnieją także menadżery niepowiązane z urządzeniami, takie jak znajdujący się w wysokopoziomowym interfejsie silnika menadżer kolizji lub fizyki, które zajmują się odpowiednio wykrywaniem kolizji pomiędzy elementami oraz poruszaniem zgodnie z zasadami fizyki. Klas zarządzających tego typu może być w silniku wiele, szczególnie w silniku specjalizowanym, w którym gatunek gry wideo może wymuszać jakieś specjalne właściwości obiektów. Powinna istnieć możliwość ich dynamicznego dodawania i usuwania, bo dzięki temu możliwe jest definiowanie własnych menadżerów przez programistów gry dla potrzeb jakiegoś konkretnego tytułu.



Rysunek 3.7. Menadżery obiektów

3.2.2 Struktura logiczna silnika

Kiedy mamy już zaplanowany funkcjonalny podział klas silnika oraz podział na warstwy niskopoziomową (Core) i wysokopoziomową (Framework), możemy przystąpić do stworzenia pełnej architektury silnika. Oprócz wymienionych poniżej opisów, każda warstwa zawiera także klasy pomocnicze i narzędziowe, umieszczone zależnie od ich poziomów abstrakcji.

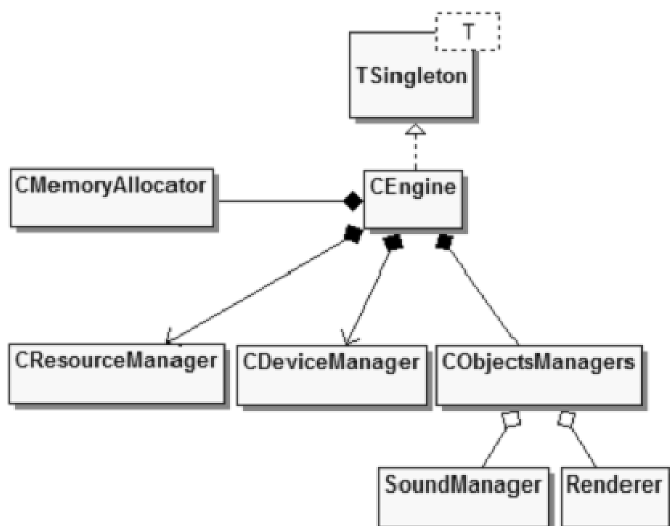
Warstwa niskopoziomowa - Core

Warstwa Core zawiera wszystkie niskopoziomowe elementy silnika niezbędne do jego funkcjonowania oraz silnie związane ze sprzętem. Znajduje się w niej klasa samego silnika będąca implementacją wzorca programowania singleton, który zapewnia, że w aplikacji znajdzie się tylko jedna instancja tej klasy oraz umożliwia dostęp do niej z każdego miejsca w projekcie. W klasie silnika znajdzie się Menadżer Zasobów, Menadżer Urządzeń, Alokator pamięci operacyjnej oraz dynamiczna tablica umożliwiająca wpinanie menadżerów elementów sceny, który możemy nazwać Tablicą Menadżerów Obiektów.

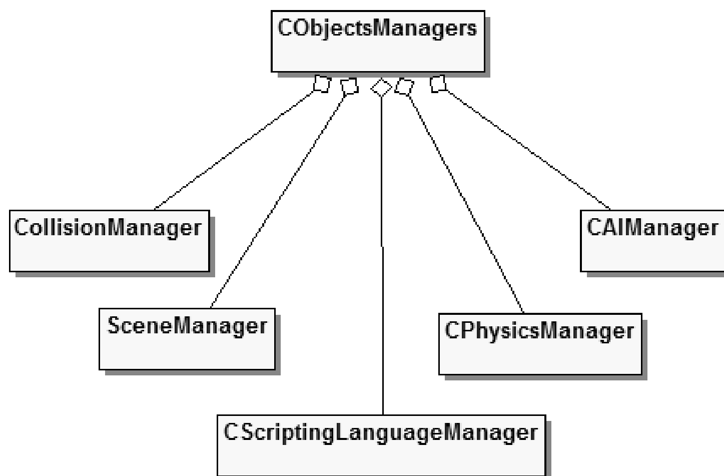
Oprócz tej struktury menadżerów warstwa Core powinna zawierać implementacje klas urządzeń (pamięci, plikowe, renderujące, dźwiękowe, kontrolery) oraz podstawowych zasobów (tekstury, pliki geometrii, pliki dźwiękowe) i obiektów (graficzne i dźwiękowe).

Warstwa wysokopoziomowa - Framework

Warstwa Framework zawiera wszystkie wysokopoziomowe menadżery i obiekty oraz stanowi podstawowy interfejs dla programisty gier korzystającego z silnika. Wszystkie wysokopoziomowe menadżery podłączone są do przygotowanego miejsca w silniku, dzięki czemu możliwa jest ich spójna i jednolita obsługa.



Rysunek 3.8. Diagram menadżerów warstwy Core



Rysunek 3.9. Diagram menadżerów warstwy Framework

System plików

Przy okazji omawiania struktury logicznej warto jeszcze wspomnieć o obsłudze systemu plików w silniku wieloplatformowym. Nasz silnik gier wideo może współpracować z wieloma różnymi typami pamięci masowej. W zależności od komputera czy konsoli może to być np.: dysk twardy, napęd DVD, napęd Blu-Ray, wbudowana pamięć Flash, kartridż, czy nawet dane w chmurze. Powyższy schemat architektury umożliwia bezproblemową obsługę wszystkich tych typów pamięci. W przypadku gdy w urządzeniu, na którym uruchomiony będzie nasz silnik, pojawi się któryś z tych systemów plików, menadżer urządzeń stworzy odpowiednią klasę urządzenia, której będzie można użyć do obsługi w nim plików.

Warstwa Core zapewnia pełny zakres możliwości, należy jedynie wiedzieć, z jakiego typu nośnika chcemy otworzyć plik. Zastanowienia wymaga jedynie warstwa Framework oraz interfejs, jakiego rzeczywiście potrzebuje programista gier z niej korzystający, który tworzy grę wieloplatformową. W najczęstszym przypadku programista otwierający na przykład plik graficzny z teksturą nie będzie chciał się przejmować tym, że na konsoli Nintendo Wii plik ten będzie otwarty z dysku DVD, a na komputerze PC z dysku twardego, na którym ta gra wideo została zainstalowana. Skoro silnik wieloplatformowy ma ukrywać różnice sprzętowe, to najbardziej eleganckim rozwiązaniem byłoby, gdyby taka różnica również była przed programistą ukryta. Podobna sytuacja ma miejsce w przypadku zapisu lub odczytu stanu rozgrywki. Na konsoli Wii programista będzie chciał otworzyć plik usytuowany we wbudowanej pamięci Flash NAND, natomiast na komputerze PC plik znajdzie się na dysku twardym w katalogu z dokumentami lub plikami osobistymi użytkownika.

Można jednak zauważyć pewną regułę. Niezależnie od aktualnie wybranej platformy, większości gier wideo wystarczy dostęp do trzech systemów plików, przy czym z punktu widzenia gry wideo nie jest istotne, w jaki sposób są one fizycznie zrealizowane:

- *System plików tylko do odczytu danych gry*

  Będzie to system plików zawierających wszystkie dane graficzne, dźwiękowe itp. dostarczone razem z grą, niezbędne do jej funkcjonowania - czyli w powyższym przykładzie dysk twardy i katalog z zainstalowaną grą dla komputera PC oraz napęd DVD dla konsoli Wii.
- *System plików odczytu-zapisu z profilami użytkowników*

  To system plików, w którym będzie można zapisywać stany gry, jak również np. ustawienia poszczególnych użytkowników - w powyższym przykładzie katalog z dokumentami dla PC i pamięć Flash NAND dla Wii.
- *System plików tymczasowych odczytu-zapisu*

  Czasami pojawia się potrzeba zapisania pliku tymczasowego, który będzie można usunąć po zamknięciu gry. Przykładem mogą być dane z systemu powtórek w grze wideo czy też pliki buforujące duże pliki audio lub wideo.

Dla komputera PC będzie to jakiś wydzielony katalog, zaś dla konsoli Wii specjalnie do tego przeznaczona przestrzeń w pamięci Flash NAND.

Rozwiązanie problemu systemów plików jest bardzo proste. Wystarczy, żeby warstwa Framework udostępniała wskaźniki do urządzeń plikowych, które zawsze będą dostępne w tym samym miejscu i tak samo się nazywały (np. *MassStorage, UserProfileStorage, TemporaryStorage*), lecz dla poszczególnych platform będą wskazywały na odpowiednie urządzenia plikowe z warstwy Core.

3.2.3 Przepływ sterowania w silniku

Wiemy w jaki sposób będzie zaprojektowana logiczna struktura silnika w postaci diagramu klas, lecz żeby w pełni przedstawić ogólna architekturę silnika, musimy wiedzieć, jak będzie wyglądał przepływ sterowania.

Pętla silnikowa

Główną rolą silnika gier wideo jest renderowanie co pewien czas klatki obrazu przedstawiającej aktualny stan gry, w wyniku czego otrzymujemy zazwyczaj animowany, dynamiczny obraz. Uszczegółowić ten opis możemy informacją, że zanim klatka animacji pojawi się na ekranie, silnik musi przeprowadzić kilka operacji i obliczeń mających na celu przykładowo określenie, co na tej klatce powinno się znaleźć oraz przeprowadzenie samego procesu renderowania tej klatki. Ponieważ cały ten ciąg operacji kończy się wyświetleniem klatki animacji, często nazywany jest klatką silnikową lub ramką silnikową, gdyż w języku angielskim używa się określenia frame. Ramki wyświetlane są jedna po drugiej, dopóki gra wideo nie zostanie wyłączona, więc ramkę silnikową musimy wywoływać w pętli - pętli silnikowej. Najprostszą pętlę silnikową można by przedstawić w ten sposób:

```
//! application main function
void main()
{
    // engine's loop
    while(true)
    {
          EngineFrame();
    }
}
```

Listing 3.2-1 - Kod najprostszej pętli silnikowej

Pętla ta jest naprawdę prosta, brakuje tu na przykład możliwości wyjścia z naszej gry wideo, ale przedstawia ona bazową ideę. Zastanówmy się, co powinna zawierać funkcja `EngineFrame()`.

```
//! engine's frame
void EngineFrame()
{
    // calculate time difference since previous frame
    float fDeltaTime = CalculateTimeDifference();

    // update device manager
    m_DeviceManager.Update(fDeltaTime);

    // update resource manager
    m_ResourceManager.Update(fDeltaTime);

    // update renderer
    m_Renderer.Update(fDeltaTime);

    // update scene meneger
    m_SceneManager.Update(fDeltaTime);

    // render frame
    m_Renderer.Render();
}
```

Listing 3.2-2 - Kod funkcji ramki silnikowej

Jak widać, funkcja ta oblicza czas, jaki minął od poprzedniej ramki, następnie wykonuje aktualizacje wszystkich systemów silnikowych, a na końcu renderuje kolejną klatkę animacji. Jak możemy zauważyć, liczba klatek animacji na sekundę nie jest pozornie tutaj niczym ograniczona. Pętla wykona się tyle razy na sekundę, ile zdąży, a więc *framerate* (parametr oznaczający generowaną liczba klatek na sekundę) będzie zmienny i zależny od skomplikowania aktualnej sceny prezentowanej przez silnik. Tak może być w istocie, lecz czasami stosuje się mechanizmy ograniczające maksymalną liczbą wyświetlanych klatek na sekundę, gdyż zmieniający się skokowo framerate może również powodować poczucie braku płynności. Również samo wywołanie renderingu na urządzeniu może zawierać w sobie mechanizm synchronizujący polegający na oczekiwaniu na informację o zakończeniu ramki z karty graficznej. W takim przypadku liczba obiegów pętli będzie zsynchronizowana z liczbą klatek, jaką odświeża karta graficzna na ekranie, co jest wartością optymalną, jeżeli chodzi o poczucie płynności animacji. Na niektórych platformach synchronizację uzyskuje się w nieco inny sposób. Ramka silnikowa nie jest wykonywana w pętli w funkcji `main()`, lecz SDK danej platformy pozwala na zarejestrowanie funkcji typu *callback* obsługującej zdarzenie wywoływane, gdy karta graficzna odświeża obraz.

Zaprezentowana pętla silnikowa jest bardzo prosta. Najpierw następują obliczenia, później rendering, a następnie wynik jest prezentowany na ekranie. Rozwiązanie to nie jest optymalne, gdyż CPU i GPU muszą dzielić się czasem dostępnym na ramkę. Zaawansowane pętle silnikowe wykorzystują mechanizmy pozwalające na asynchroniczną pracę procesora głównego i procesora graficznego. Dzięki temu każdy z nich ma całą ramkę na wykonanie swoich operacji, co może znacznie zwiększyć wydajność.

Mechanizm scen

Przedstawiona pętla silnikowa demonstruje mechanizm działania silnika oraz rolę renderingu i karty graficznej w tym procesie. Gdzie w takim razie w tym wszystkim jest miejsce na kod gry wideo? Odpowiedzią jest mechanizm scen oraz menadżer scen obecny w warstwie *Framework* silnika.

Jak już wcześniej wspomniano, programista gry lub projektant rozmieszcza obiekty, komponując scenę, którą prezentuje nam gra wideo. Scena ta może być stworzona w edytorze, bądź też napisana w kodzie źródłowym. W każdym przypadku za jej obsługę odpowiedzialna jest silnikowa klasa sceny.

```
//! base interface for scene classes
class IScene
{
      //! scene initialization
      void OnInitialize() = 0;

      //! scene deinitialization
      void OnDeinitialize() = 0;

      //! scene's update function
      void Update(float fDeltaTime) = 0;
};
```

Listing 3.2-3 - Bazowy interfejs klasy sceny

Powyżej przedstawiono najprostszy interfejs klasy sceny, składający się z inicjalizacji, deinicjalizacji oraz metody aktualizującej, która będzie wykonywana podczas każdej ramki silnikowej. Stworzenie własnej sceny polega na podziedziczeniu po tym interfejsie i implementacji tych trzech metod. W inicjalizacji można wczytać scenę skomponowaną w edytorze lub tworzyć jej elementy bezpośrednio w kodzie źródłowym. W deinicjalizacji należy posprzątać, a metoda aktualizująca jest to metoda, która nadaje naszej scenie „życie”. To w niej będziemy implementować całą mechanikę gry wideo, poruszać wszystkimi obiektami i reagować na dane otrzymywane z kontrolerów.

Zarządzaniem scenami zajmuje się menadżer scen. To on wywołuje aktualizację aktualnie działającej sceny. W momencie, gdy zażądamy zmiany obec-

nej sceny na inną, jego zadaniem jest, aby odbyło się to w płynny i przyjazny dla gracza sposób. Menadżer przykrywa aktualną scenę kurtyną, którą może być na przykład plansza z napisem „Loading”, a następnie deaktywuje obecną scenę, aktywuje kolejną i usuwa kurtynę.

Zdarzenia silnikowe

Sposób aktualizowania sceny powoduje, że tworzy się pewnego rodzaju hierarchia wywołań funkcji `Update()`. Funkcja aktualizująca wywoływana jest dla klasy sceny, która wywołuje ją dla swoich elementów składowych, a one dla swoich składowych itd. Jest to dobre podejście, hierarchizacja obiektów pomaga zachować porządek w klasie sceny. Hierarchiczne wywoływanie funkcji aktualizującej pozwala w prosty sposób wpływać na czas, jaki otrzymują wszystkie obiekty, co umożliwia łatwą modyfikację prędkości działania gry lub implementacji pauzy.

Są jednak sytuacje, kiedy funkcja `Update()` nie wystarcza. Ma to na przykład miejsce, gdy musimy zareagować na zdarzenie, które zazwyczaj nie występuje co klatkę. Przykładem takiego zdarzenia może być naciśnięcie przycisku na kontrolerze, podłączenie nowego kontrolera lub na przykład rozłączenie się gracza sieciowego. Odpytywanie przez scenę co klatkę wszystkich obiektów, czy coś się przypadkiem ostatnio nie stało, nie jest najlepszym pomysłem ze względów wydajnościowych.

Dla takich przypadków powinien istnieć mechanizm zdarzeń silnikowych pozwalający scenie bądź dowolnemu innemu obiektowi zarejestrować funkcję typu callback służącą do obsługi wystąpienia takiego zdarzenia za pomocą mechanizmu opartego na wzorcu projektowym lub mechanizmu wykorzystującego na przykład wskaźniki do funkcji.

3.3 Organizacja projektu silnika

Powyżej omówiono logiczną architekturę silnika. Poniżej zostanie omówione fizyczne rozlokowanie wszystkich plików w strukturze projektu oraz sposób ustawienia plików projektu, pozwalający skompilować i zlinkować grę wideo do pliku wykonywalnego.

3.3.1 Struktura podprojektów

Język C++ jest językiem natywnym, więc plikiem wynikowym po zbudowaniu projektu jest plik wykonywalny, umożliwiający uruchomienie gry wideo w systemie operacyjnym danej platformy. Czas kompilacji projektu w tym języku może być jednak dosyć długi. Nie zauważymy tego przy małych projektach składających się z kilku lub kilkunastu plików. Jednak zarówno silnik jak i gra

wideo może osiągnąć poziom kilku tysięcy plików źródłowych, a wówczas czas kompilacji i linkowania może rozciągnąć się nawet do wielu godzin. Są różne metody radzenia sobie z tym problemem. Jednym z nich jest użycie systemów kompilacji wielordzeniowej i rozproszonej na kilka komputerów w sieci. Należy jednak zacząć od kroków na niższym poziomie - podzielenia projektu na podprojekty.

Drugim obok wydajności kompilacji powodem do podzielenia projektu jest idea ponownego wykorzystania kodu. Silnik gier wideo zgodnie z założeniem ma być wspólny dla wszystkich gier, które chcemy napisać z jego wykorzystaniem, więc musi być w jakiś sposób wydzielony, aby mógł być współdzielony pomiędzy projektami. To samo dotyczy wszystkich innych fragmentów kodu, które są na tyle uniwersalne, aby mogły być użyte w innym projekcie.

Podział projektu silnika oraz gry wideo

Najprostszym podziałem, jaki można wybrać, jest podział na podprojekt silnika i podprojekt gry wideo. Jeśli projekty nie są duże, podział ten może okazać się w zupełności wystarczający. Można się jednak dodatkowo zastanowić nad podziałem projektu silnika na podprojekty *Core* i *Framework*, co jeszcze podkreśli rozdzielność ich warstw abstrakcji. Podobnie jest, gdy projekt gry wideo jest duży lub posiada wyraźną logiczną granicę, na przykład pomiędzy implementacją mechaniki gry a implementacją specjalnych funkcjonalności dla poszczególnych poziomów, warto pomyśleć o rozdzieleniu ich na osobne projekty.

Biblioteki statyczne

Biblioteki statyczne są jednym z możliwych sposobów realizacji podziału na podprojekty. Każdy projekt linkowany jest do biblioteki statycznej - czyli pliku z rozszerzeniem `.a` lub `.lib`, z wyjątkiem ostatniego projektu w hierarchii, który jest linkowany do pliku wykonywalnego, włączając podczas linkowania wszystkie wygenerowane wcześniej biblioteki statyczne. W rezultacie otrzymujemy plik wykonywalny, dokładnie taki sam jakby podziału na podprojekty nie było, a każda kompilacja pojedynczego projektu wymusza konieczność linkowania całości. W tej metodzie nie zyskujemy wiele na czasie budowania, lecz uzyskujemy rozdzielność logiczną.

Biblioteki dynamiczne

Wykorzystanie bibliotek dynamicznych różni się tym, że poszczególne podprojekty linkowane są do bibliotek dynamicznych (pod systemem Windows będą to pliki `.DLL`). Ostatni projekt jest linkowany do pliku wykonywalnego, lecz nie musi on linkować statycznie pozostałych bibliotek. Z wykorzystaniem tej

metody każdy podprojekt linkowany jest do „swojego” pliku, więc w przypadku kompilacji jednego podprojektu nie ma konieczności linkowania wszystkich. Zaletą tego rozwiązania jest przyspieszenie ogólnego czasu budowania projektu i separacja logiczna. Wadą natomiast to, że kod źródłowy musi być odpowiednio przygotowany, by mógł być linkowany do bibliotek dynamicznych.

3.3.2 Struktura plików i katalogów

W rozwijającym się projekcie w sposób lawinowy zaczyna przybywać różnego rodzaju plików. Jeżeli chcemy utrzymać w tym porządek, warto zadbać o uporządkowane ich rozmieszczenie już na początku. Przykładowo katalog zawierający projekt mógłby wyglądać w ten sposób:

Rysunek 3.10. Zawartość katalogu z projektem

Pliki źródłowe

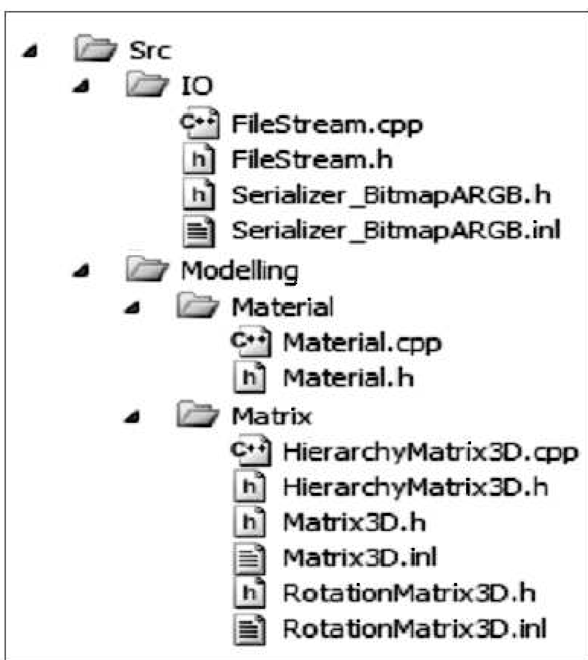
Pliki źródłowe znajdują się w katalogu *Src*. Wraz z rozwojem projektu plików tych będzie bardzo szybko przybywać. Dobrym sposobem na utrzymanie pośród nich porządku jest utrzymywanie hierarchicznej struktury katalogów odpowiadającej podziałowi logicznemu klas silnika oraz trzymanie plików nagłówkowych odpowiadających plikom .cpp obok nich, z tą samą nazwą.

Pliki projektu

Katalog *Prj* zawiera pliki projektu definiujące sposoby budowania wszystkich podprojektów na wszystkie platformy. W zależności od użytego środowiska programistycznego mogą to być pojedyncze pliki lub zestawy plików dla każdej konfiguracji. Przykładowo w przypadku projektu środowiska MS Visual Studio będą to pliki `.prj` oraz `.sln`, zaś w przypadku niektórych środowisk mogą to być standardowe w środowiskach uniksowych pliki typu *makefile*.

Pliki zasobów

Katalog *Resource* zawiera zasoby i pliki danych potrzebne w projekcie. Pliki te mogą być w formatach wymaganych przez wszystkie obsługiwane platformy,



Rysunek 3.11. Przykładowy wycinek hierarchii plików źródłowych

bądź mogą być konwertowane dopiero podczas tworzenia wersji (builda) na daną platformę składającej się z aktualnego pliku wykonywalnego oraz zestawu danych.

Pliki obiektowe

Katalog *Obj* zawiera pliki obiektowe i inne pliki tymczasowe, generowane podczas procesu kompilacji.

Wersje (buildy)

Katalog *Builds* zawiera kompletne wersje naszej gry wideo dla poszczególnych platform składające się z pliku wykonywalnego oraz zestawu danych. Składanie tego zestawu powinno być zautomatyzowane przez skrypt uruchamiany oddzielnie bądź podłączony do systemu kompilacji na daną platformę.

Wersja w zależności od platformy, może mieć postać pliku wykonywalnego i katalogu z danymi obok, bądź postać obrazu binarnego płyty DVD lub kartridża przygotowanego za pomocą narzędzi dostarczonych wraz z SDK danej platformy.

3.4 Organizacja kodu

Język C++ teoretycznie jest językiem przenośnym. W praktyce jednak z uwagi na rozbieżność bibliotek, a także różnice w architekturze sprzętowej o poprawną przenośność kodu trzeba zadbać we własnym zakresie. Poniżej przedstawionych zostanie kilka mechanizmów języka przydatnych przy tworzeniu i zarządzaniu kodu.

3.4.1 Przydatne Mechanizmy C++

Makra preprocesora

Preprocesor prowadzi wstępną obróbkę kodu źródłowego, zanim ten trafi do kompilatora. Dzięki dyrektywom preprocesora można mieć wpływ na generację kodu źródłowego i modyfikować go w zależności od pewnych warunków.

Jedną z najczęściej używanych dyrektyw jest dyrektywa kompilacji warunkowej `#ifdef`, używana zresztą często do implementacji „strażników" plików nagłówkowych. Przy tworzeniu silnika gier wideo dyrektywa ta może mieć jednak inne bardzo przydatne zastosowania. Po pierwsze, jest najprostszym sposobem na uzyskanie przenośnego kodu tam, gdzie występują różnice pomiędzy platformami. Po drugie, umożliwia stworzenie pliku stanowiącego pewnego rodzaju ustawienia dla kompilacji. Mogą one dotyczyć np. włączenia trybu *Debug*, w którym będziemy intensywniej sprawdzać poprawność wszystkich wartości przekazywanych do funkcji i metod lub włączenia w projekcie różnych mechanizmów diagnostycznych.

Drugim przydatnym mechanizmem jest dyrektywa `#define`, pozwalająca definiować wartości, dzięki którym możemy sterować kompilacją warunkową. Umożliwia także definiowanie makr preprocesora, co daje nam możliwość tworzenia w kodzie elastycznych wstawek.

Preprocesor udostępnia też zestaw stałych przydatnych podczas testowania gry wideo w trybie diagnostycznym, takich jak `_LINE_` oraz `_FILE_`, które zwracają numer linii pliku źródłowego oraz nazwę pliku. Dzięki temu w razie wystąpienia błędu można w prosty sposób poinformować użytkownika, w którym miejscu w kodzie źródłowym on wystąpił.

Makra systemowe

Makro `PANIC` umożliwia natychmiastowe zatrzymanie kodu w przypadku, gdy program podczas wykonywania znalazł się w miejscu w kodzie, w którym nie miał prawa się znaleźć. Dobrym miejscem na jego użycie jest na przykład sekcja `default:` instrukcji `switch`, jeżeli zakładamy, że wartość ta może się pojawić tylko w wyniku błędu w kodzie programu. Makro wypisuje na ekranie numer linii i nazwę pliku źródłowego, pomagając szybko odnaleźć miejsce, gdzie błąd wystąpił.

```
// Panic macro
#define PANIC(MessageFormat, ...) \
     printf("Panic in \%s, line: \%d\n", _FILE_, _LINE_), \
     printf(MessageFormat, _VA_ARGS_), \
     abort()

// usage example

enum EDirection
{
     LEFT,
     RIGHT
};

...

switch(eDirection)
{
case LEFT:
     DoSomething();
     break;

case RIGHT:
     DoSomethingElse();
     break;

default:
     PANIC("Direction enumerator value out of range: \%d\n",
     eDirection);
}
```

Listing 3.4-1 - Kod makra preprocesora: Panic

W trybie Debug powinniśmy bardzo dokładnie kontrolować wartości różnych zmiennych, zwłaszcza parametrów przekazywanych do funkcji i metod. Im szybciej przerwiemy wykonywanie programu po wystąpieniu nieprawidłowej wartości, tym łatwiej będzie nam odnaleźć źródło błędu. Makro `ASSERT` daje nam możliwość sprawdzania w trybie Debug poprawności warunku i przerwania programu w przypadku jego niespełnienia. Makro nie zostawia żadnych nadmiarowych sprawdzeń w trybie *Release*.

Funkcje inline

Mechanizm funkcji lub metod *inline* jest to mechanizm, który pozwala na zdefiniowanej funkcji, która będzie rozwijana w miejscu wywołania. Oznacza to, że w miejscu wywołania program nie będzie wykonywał skoku do tej funkcji, ponieważ kompilator „wklei” ciało tej funkcji w miejscu jej wywołania.

```
// Assert macro definition
#ifdef _DEBUG
       #define ASSERT(Condition, MessageFormat, ...) \
          if(!(Condition)) \
             printf("Assert failed in \%s, line: \%d !\n", \
             _FILE_, _LINE_), \
             printf(MessageFormat, _VA_ARGS_), \
             abort()
#else
        #define ASSERT(Condition, MessageFormat, ...)
#endif

...

// usage example

float CalculateSqareRoot(float fValue)
{
      ASSERT(fValue >= 0.0f, "Value must be >= 0, passed \%f !\n",
             fValue);
      return sqrtf(fValue);
}
```

Listing 3.4-2 - Kod makra preprocesora: Assert

Funkcje inline mogą być doskonałym sposobem na optymalizację niektórych fragmentów programu, zwłaszcza jeżeli będą wykorzystane dla krótkich funkcji, gdzie koszt czasowy wykonania skoku do funkcji byłby porównywalny z czasem wykonania jej ciała. Wadą tej funkcji jest rozrastanie się wynikowego pliku wykonywalnego, co może z kolei spowolnić jego wykonanie, zwłaszcza jeżeli mechanizm ten zostanie błędnie użyty dla często wywoływanych, dużych funkcji. Inną wadą może być też utrudniona praca krokowa podczas debugowania kodu, gdyż nie wszystkie debugery potrafią poprawnie wskazywać instrukcje w kodzie źródłowym, jeżeli skompilowane były one jako inline.

Przykładem zastosowania funkcji inline może być zastąpienie w niektórych przypadkach makrodefinicji preprocesora - jeżeli jest to możliwe, ma same zalety. Mechanizm inline w przeciwieństwie do preprocesora jest elementem języka C++, więc jest mniej podatny na błędy oraz zapewnia kontrolę typów. Jednocześnie funkcje inline w porównaniu z makrodefinicjami nie powodują żadnego dodatkowego narzutu w kodzie.

Innym zastosowaniem mechanizmu inline jest optymalizacja krótkich funkcji, które często występują podczas implementacji biblioteki matematycznej. Weźmy dla przykładu definicję operatora sumującego dwa wektory trójwymiarowe.

```
//! operator +=
inline Vector3f & Vector3f::operator+=(const Vector3f & v3fValue)
{
    this->x += v3fValue.x;
    this->y += v3fValue.y;
    this->z += v3fValue.z;
    return *this;
}
```

Listing 3.4-3 - Operator sumowania wektorów napisany jako funkcja inline

Definicja tego operatora zawiera trzy najprostsze operacje, które zostaną wykonane w kilku cyklach procesora. Gdyby operator ten zdefiniowany był jako „prawdziwa" funkcja, to narzut przy wywołaniu tego operatora spowodowany instrukcją skoku, koniecznością zapisania stanu rejestrów i odłożenia parametrów funkcji na stos znacznie by przewyższył czas wykonania samego dodawania. W różnych algorytmach operacje na wektorach takie jak dodawanie mogą być wykonywane bardzo często.

3.4.2 Separacja kodu poszczególnych platform

Projektując wieloplatformowy silnik gier wideo, mamy jedno główne założenie. Chcemy użytkownikowi silnika udostępnić jednolity interfejs (API), niezależny w żaden sposób od platformy, na której będzie ten kod kompilowany. Kod implementacji będzie się jednak różnił na poszczególnych platformach. Należy rozwiązać problem, w jaki sposób oddzielić ten kod tak, aby nie zakłócić interfejsu oraz zachować przejrzystość kodu.

Oddzielna implementacja

Oddzielna implementacja jest radykalnym sposobem rozwiązania tego problemu. Zakłada ona, że implementacje dla poszczególnych platform będą zupełnie odrębnymi projektami, niewspółdzielącymi ze sobą żadnych plików. Identyczność interfejsu dla różnych architektur będzie gwarantowana przez zadeklarowany standard, którego zobowiązujemy się trzymać.

Podejście to jest często stosowane przy implementacji różnego rodzaju interfejsów będących ogłoszonym standardem. Przykładem może być tutaj API OpenGL, które jest tylko zdefiniowanym standardem interfejsu, natomiast implementacja leży już po stronie producentów sprzętu lub oprogramowania, które chcą ten standard spełniać. W związku z tym są one zupełnie odrębne.

Korzystanie z tej metody podczas pracy nad silnikiem gier wideo, który mimo kilku platform jest nadal jednym projektem, byłoby jednak ogromną stratą powodującą niepotrzebny narzut czasowy. Po pierwsze, ogromna część kodu

nie wymaga modyfikacji, a po drugie, dbanie o synchronizację kilku zupełnie oddzielnych grup projektów byłoby bardzo trudne. Jedynym zastosowaniem, gdzie można zastanowić się nad takim rozwiązaniem, jest tworzenie silnika gier wideo w pełni edytorowego (sterowanego danymi).

Kod warunkowy

Kod poszczególnych platform, jak już to zostało wcześniej wspomniane, można odseparować, korzystając z dyrektyw kompilacji warunkowej. Większość platform, na których możemy kompilować język C++, udostępnia definicje preprocesora, dzięki którym możemy jednoznacznie określić platformę. W przypadku, gdyby definicji tych było zbyt mało, można zdefiniować dodatkowe w ustawieniach projektu. Poniżej przykład implementacji wątku systemowego dla platform Windows i Linux odseparowana za pomocą instrukcji warunkowych.

```
//! class implements thread
class CThread
{
public:
       //! starts thread execution  void Start();

       //! thread's task    virtual void Run() = 0;

private:

#ifdef WIN32
       //! thread function
       static DWORD WINAPI EntryPoint(LPVOID pThreadObject);

       //! thread's handle
       HANDLE m_hThreadHandle;

       //! thread Id
       DWORD  m_uiThreadId;
#endif

#ifdef _LINUX
       //! thread function
       static void* EntryPoint(void* pThreadObject)

       //! thread object
       pthread_t m_Thread;
#endif
};

////////////////////////////////////////////////
```

```
//! starts thread execution
void CThread::Start()
{
#ifdef WIN32
     // start thread
     m_hThreadHandle = CreateThread (NULL, 0,  EntryPoint, this,
                       0, &m_uiThreadId);
#endif

#ifdef _LINUX
       pthread_create(&m_Thread, NULL, EntryPoint, (void*)this);
#endif
}

#ifdef WIN32
    //! thread function
    /*static*/ DWORD WINAPI CThread::EntryPoint(LPVOID pThreadObject)
    {
          // get pointer to thread object
          CThread* pObject = (CThread*)pThreadObject;

          // execute thread code
          pObject->Run();

          // exit thread
          return 0;
    }
#endif

#ifdef _LINUX
           //! thread function
           /*static*/ void* CThread::EntryPoint(void* pThreadObject)
     {

           // get pointer to thread object
           CThread* pObject = (CThread*)pThreadObject;

           // execute thread code
           pObject->Run();

           // exit thread
           return 0;
     }
 #endif
```

Listing 3.4-4 - Separacja kodu poszczególnych platform za pomocą kompilacji warunkowej

Zaletami tego rozwiązania są jego szybka implementacja i wykorzystanie wspólnej części kodu. Wadami natomiast brak czytelności, który rośnie z ilością obsługiwanych platform oraz zaśmiecanie interfejsu dodatkowymi atrybutami i funkcjami specyficznymi dla platformy.

Wspólny interfejs - wzorzec fabryki abstrakcyjnej

Separacja kodu za pomocą wzorca fabryki abstrakcyjnej polega na przygotowaniu interfejsu silnika gier wideo jako hierarchii w pełni abstrakcyjnych klas języka C++. Tworzenie poszczególnych obiektów silnika nie jest realizowane przez operator new, lecz poprzez metody obiektu fabryki, które tworzą odpowiedni dla danej platformy obiekt, zwracając jednocześnie wskaźnik na jego interfejs. W tym przypadku implementacje dla poszczególnych platform są zupełnie odrębne, wspólny jest jedynie interfejs, który muszą implementować. Poniżej przykład implementacji klasy wątku w tej technice.

```
//! abstract interface for factory
class IFactory
{
public:
    //! creates thread object
    virtual IThread* CreateThread() = 0;
};

////////////////////////////////////////////////

//! Win32 implementation of factory
class CFactory_Win32
{
public:
    //! creates thread object
    virtual IThread* CreateThread() { return new CThread_Win32; };
};

////////////////////////////////////////////////

//! Linux implementation of factory
class CFactory_Linux
{
public:
     //! creates thread object
     virtual IThread* CreateThread() { return new CThread_Linux; };
};
```

```
////////////////////////////////////////////////

//! interface class for thread task
class IThreadTask
{
    //! thread's task
    virtual void Run() = 0;
};

//! interface class for thread object
class IThread
{
public:
    //! starts thread execution
    virtual void Start(IThreadTask* pTask) = 0;
};

////////////////////////////////////////////////

// Win32 thread implementation
class CThread_Win32 : public IThread
{
public:
    //! starts thread execution
    virtual void Start(IThreadTask* pTask)
    {
       // start thread
       m_hThreadHandle = CreateThread (NULL, 0, EntryPoint, pTask, 0,
                         &m_uiThreadId);
    }

private:
   //! thread function
   static DWORD WINAPI EntryPoint(LPVOID pThreadTask)
   {
       // get pointer to thread object
       IThreadTask* pTask = (IThreadTask*)pThreadTask;

       // execute thread code
       pTask->Run();

       // exit thread
       return 0;

   //! thread's handle
    HANDLE m_hThreadHandle;
   }
```

```
    //! thread Id
    DWORD  m_uiThreadId;
};

////////////////////////////////////////////////

// Linux thread implementation
class CThread_Linux : public IThread
{
public:
    //! starts thread execution
    virtual void Start(IThreadTask* pTask)
    {
        // start thread
        pthread_create(&m_Thread, NULL, EntryPoint, pTask);
    }
private:
    //! thread function
    static void* EntryPoint(void* pThreadTask)
    {
        // get pointer to thread object
        IThreadTask* pTask = (IThreadTask*)pThreadTask;

        // execute thread code
        pTask->Run();

        // exit thread
        return 0;
    }

    //! thread object
    pthread_t m_Thread;
};

////////////////////////////////////////////////

// somewhere in engine

#ifdef WIN32
     IFactory* pFactory = new CFactory_Win32();
#endif

#ifdef LINUX
     IFactory* pFactory = new CFactory_Linux();
#endif
```

Listing 3.4-5 - Separacja kodu poszczególnych platform za pomocą wzorca fabryki abstrakcyjnej

Zaletą tego rozwiązania jest jego względna prostota użycia oraz skuteczność - poszczególne implementacje są zupełnie od siebie odseparowane.

Wadą tego rozwiązania jest konieczność utrzymywania fabryki abstrakcyjnej, która musi być rozbudowywana z każdą nową klasą oraz konieczność przyzwyczajenia się do tworzenia wszystkich obiektów przez fabrykę. Innym problemem jest konieczność tworzenia wszystkich tego typu obiektów jako obiekty dynamiczne i późniejsza ich dealokacja. Nie jest możliwe stworzenie obiektu automatycznego istniejącego tylko w ciele funkcji lub bezpośrednio jako pole innej klasy. Z tego samego powodu niewygodne byłoby użycie tej metody do implementacji przenośnych klas matematycznych.

Warstwa pośrednia - wzorzec mostu

Implementacja silnika wieloplatformowego za pomocą wzorca mostu polega na rozdzieleniu silnika na niezależną sprzętowo warstwę stanowiącą jego użytkowy interfejs (API) oraz zależną od platformy, ukrytą warstwą implementacji. Warstwa implementacji połączona jest z warstwą API przez interfejs definiujący funkcjonalność, jaką musi ona spełnić. Przykład implementacji klasy wątku:

```
//! platform specific thread interface
class IThread_PlatformSpecific
{
public:
    //! constructor()
    IThread_PlatformSpecific(CThread* pOwner): m_pOwner(pOwner) {};

    //! starts thread execution
    virtual void Start() = 0;

protected:
    //! pointer to owner object
    CThread* m_pOwner;
};

//! platform-independent thread
class CThread
{
public:
    //! constructor
    CThread()
    {
    #ifdef WIN32
        m_pThreadImplementation = new CThread_Win32(this);
    #endif
```

```
    #ifdef _LINUX
        m_pThreadImplementation = new CThread_Linux(this);
    #endif
    }

    //! destructor  virtual ~CThread()
    {
        delete m_pThreadImplementation;
    }

    //! starts thread execution
    void Start()
    {
        m_pThreadImplementation->Start();
    }

    //! thread's task
    virtual void Run() = 0;

private:
    //! thread platform-specific implementation
    IThread_PlatformSpecific* m_pThreadImplementation;
};

////////////////////////////////////////////////

// Win32 thread implementation
class CThread_Win32 : public IThread_PlatformSpecific
{
public:
    //! constructor
    CThread_Win32(CThread* pOwner) :
                              IThread_PlatformSpecific(pOwner) {};

    //! starts thread execution
    virtual void Start()
    {
       // start thread
       m_hThreadHandle = CreateThread (NULL, 0, EntryPoint, m_pOwner,
                                                   0, &m_uiThreadId);
    }

private:
    //! thread function
    static DWORD WINAPI EntryPoint(LPVOID pThreadObject)
    {
       // get pointer to thread object
       CThread* pObject = (CThread*)pThreadObject;
```

```
        // execute thread code
        pObject->Run();

        // exit thread
        return 0;
    }

    //! thread's handle
    HANDLE m_hThreadHandle;

    //! thread Id
    DWORD  m_uiThreadId;
};

////////////////////////////////////////////////

// Linux thread implementation
class CThread_Linux : public IThread_PlatformSpecific
{
public:
    //! constructor
    CThread_Linux(CThread* pOwner) :
                         IThread_PlatformSpecific(pOwner) {};

    //! starts thread execution
    virtual void Start()
    {
        // start thread
        pthread_create(&m_Thread, NULL, EntryPoint, m_pOwner);
    }

private:
    //! thread function
    static void* EntryPoint(void* pThreadObject)
    {
         // get pointer to thread object
         CThread* pObject = (CThread*)pThreadObject;

         // execute thread code
         pObject->Run();
```

```
        // exit thread
        return 0;
    }

    //! thread object
    pthread_t m_Thread;
};
```

Listing 3.4-6 - Separacja kodu poszczególnych platform za pomocą wzorca mostu

Wadą wzorca mostu jest jego względna komplikacja, zwiększająca ilość nadmiarowego kodu potrzebnego do implementacji. Zaletą jest jego skuteczność - zupełna separacja i ukrycie przed użytkownikiem faktu, że klasa ta jest zależna od platformy sprzętowej, gdyż sposób jej tworzenia pozostaje niezmienny.

4

Problemy związane z wieloplatformowością

4.1 Wyzwania dla wieloplatformowego silnika

Bardzo ważnym atutem silnika wieloplatformowego jest jego abstrakcyjność. Dobrze zaprojektowany silnik ma całkowicie abstrakcyjny interfejs API, który jest absolutnie niezależny od platform, na których występuje. Podejście takie umożliwia prawie całkowite uwolnienie programistów logiki gry korzystających z silnika od problemów związanych z różnorodnością platform. Z punktu widzenia programisty gry interfejs API silnika wygląda tak samo, niezależnie czy programuje grę na konsole przenośną, czy też na komputer osobisty, a co za tym idzie, jest mu obojętne, na jaką platformę programuje. Przejście programisty z jednej platformy na inną jest bezproblemowe i nie wymaga dodatkowych szkoleń.

W teorii, idea silnika wieloplatoformowego wygląda pięknie, wręcz idealnie. W praktyce okazuje się, że opanowanie projektu wieloplatformowego jest bardzo trudne. Zasadniczym problemem jest zróżnicowanie specyfikacji sprzętu. Producenci konsol do gier próbują uatrakcyjnić swoje produkty poprzez dodawanie nowych, innowacyjnych funkcjonalności, tak aby ich produkt był lepszy i posiadał cechy unikalne, inne niż konkurencja. Różnice te mają ogromny wpływ na ostateczny kształt gry. Dobrze zaprojektowana gra powinna jak najbardziej wykorzystywać ciekawe cechy danej platformy. Gracz, który wybrał pewną konkretną platformę, oczekuje gier wykorzystujących maksymalnie jej charakterystyczne cechy.

Ogromnym problemem jest różna wydajność sprzętu. Tworząc grę, należy starać się wykorzystać jak najlepiej to, co oferuje sprzęt. Dlatego też gry na mocniejszym sprzęcie powinny być bardziej skomplikowane, mieć ładniejszą (a co za tym idzie bardziej złożoną) oprawę graficzną, dźwiękową oraz zawierać odpowiednio więcej treści. Różnice te są tak duże, że nikt nawet nie myśli o tworzeniu jednej gry na wszystkie dostępne platformy. Projektując grę, wybiera się zazwyczaj podobne, jeżeli chodzi o możliwości sprzętowe, platformy. Jednakże nawet pozornie niewielkie różnice w wydajności oraz architekturze

sprzętu mogą powodować ogromne problemy. Jeśli więc gra ma wykorzystywać sprzęt na sto procent, to nawet minimalne różnice między dedykowanymi platformami gry powodują konieczność przygotowania maksymalnie różnych zasobów, takich jak modele, dźwięki czy shadery. Tworzenie zasobów graficznych oraz dźwiękowych o różnym poziomie szczegółowości w praktyce jest bardzo pracochłonne. Dodatkowym problemem może być fakt, że zasoby takie często muszą być przechowywane w danym specyficznym formacie przystosowanym do danej platformy. Bardzo często wiąże się to ze wspomaganiem przez sprzęt pewnych natywnych formatów plików.

Programowanie silnika wieloplatformowego jest bardzo trudne, ponieważ programiści silnika muszą znać API każdej platformy, a ze względu na różną architekturę implementacje mogą być bardzo różne. Producenci sprzętu projektując API swoich bibliotek programistycznych, nastawieni są na maksymalne wykorzystanie sprzętu, przedkładając wydajność nad uniwersalność. Takie podejście jest oczywiste i jak najbardziej poprawne. W końcu optymalne wykorzystanie sprzętu jest celem silnika. Niestety podejście takie powoduje, że biblioteki programistyczne mogą być całkowicie rożne od siebie.

Ze względu na wszystkie wymienione wyżej różnice, konserwacja, testowanie oraz uruchamianie projektu jest bardzo trudne i czasochłonne. Szeregi testów muszą być robione właściwie na każdej platformie osobno. Dodanie lub zmiana funkcjonalności silnika powoduje zmiany we wszystkich implementacjach na wszystkich platformach.

4.2 Różnice sprzętowe

Podczas porównywania platform do gier zwracają uwagę przede wszystkim różnice sprzętowe. Różnice w sterowaniu, wyświetlaniu obrazu czy odgrywaniu dźwięku stają się ewidentne, gdy tylko spojrzy się na rozbieżności: w budowie kontrolerów, liczbie oraz rozdzielczości ekranów czy też budowie systemu audio. Co gorsza, dysproporcje w sprzęcie są znaczne, a niektóre zauważalne dopiero po dokładnym przeanalizowaniu specyfikacji sprzętu. Dedykowane platformy różnią się procesorami, ilością pamięci głównej, graficznej, innymi specyficznymi typami pamięci rozszerzonych oraz pamięciami masowymi.

Projektując silnik na wiele platform, powinno się zacząć od przeanalizowania różnić w sprzęcie, ponieważ dobre rozeznanie w różnicach sprzętowych na początku pozwoli przyjąć lepszą strategię, a podczas projektowania uniknąć wielu błędów architektonicznych. W rozdziale 2 opisano różnice w wersjach gry wideo dla poszczególnych platform. Poniżej (Tab. 4.1-4.2) przedstawiono zestawienie cech wybranych platform gier wideo aktualnej generacji.

Jak widać, pisząc silnik, można spodziewać się znacznych różnic pomiędzy platformami, co może wpłynąć na komplikacje przy projekcie. Z jednej strony należy uważać na architekturę silnika, czyli powinien on być zaprojektowany tak, by różnice w budowie sprzętu nie powodowały żadnych niespójności w in-

Tablica 4.1. Różnice sprzętowe między platformami gier wideo cz. 1

Platforma	Xbox 360	PlayStation 3	Nintendo Wii
Jednostka centralna	Xenon, taktowanie zegara 3.2 GHz, 3 identyczne rdzenie Power PC każdy 64 K pamięci podręcznej L1 i 1 MB pamięci podręcznej L2 dla wszystkich. Każda jednostka posiada jednostkę wektorową VMX	Cell, taktowanie zegara 3.2 GHz, 1 dwu-wątkowy procesor główny wyposażony w pamięć podręczną L1 32 KB, L2 512 KB, jednostkę wektorową VMX i 7 podrzędnych procesorów wektorowych wyposażonych w 256 KB SRAM	Broadway, taktowanie zegara 729 Mhz oraz pamięć podreczna L1 64 KB, L2 256 KB
GPU	Xenos, taktowanie zegara 500 Mhz, 48 potoków zunifikowanych shaderów, korzysta z pamięci systemowej (128 bitowa magistrala, 22.4 GB/s), pamięci podręcznej procesora L2 (128 bitowa magistrala, 24 GB/s) oraz wewnętrznej 10 MB (Z-bufor, alpha blending)	RSX, taktowanie zegara 550 MHz, 24 potoki Pixel Shader, 8 potoków Vertex Shader 256 MB pamięci wewnętrznej	Holywood, 243 Mhz, korzysta z pamięci systemowej oraz wewnętrznej 3 KB (bufor ramek, pamięć tekstur)
Pamięć operacyjna	512 MB (8 chipów po 64 MB), taktowanie zegara 700 Mhz	256 MB, taktowanie zegara 3.2 GHz	88 MB (2 bloki 24 MB szybkiej pamięci wbudowanej oraz 64 MB zewnętrznej)
Pamięć masowa	DVD, opcjonalnie dysk twardy SATA 40/120 GB	Blu-Ray, DVD, CD, dysk twardy	DVD, 512 MB NAND flash, opcjonalnie Xbox Memory Card obsługa kart SD oraz Nintendo GameCube Memory Card
Komunikacja	Ethernet 100 Mbit WiFi	Ethernet 1 Gbps, WiFi, Bluetooth	WiFi
Kontrolery	Klasyczne pady	Klasyczne pady	Kontrolery Wii Remote, zawierające akcelerometry oraz czajniki optyczne, bardzo wiele dodatkowych kontrolerów i rozszerzeń takich jak Wii Board, Wii Motion Plus
Wyjście wideo	1 ekran, obsługiwane wyjścia analogowe i cyfrowe, tryby 720p, 1080i oraz 1080p	1 ekran, obsługiwane wyjścia analogowe i cyfrowe, tryby 480i, 480p, 576i, 576p, 720p, 1080i, 1080p	1 ekran, obsługiwane wyjścia analogowe, tryby 480p 480i, 576i

Tablica 4.2. Różnice sprzętowe między platformami gier wideo cz. 2

Platforma	PlayStation Portable	Nintendo DS	IPhone 4
Jednostka centralna	2 rdzenie oparte na MIPS R4000, 1-333 Mhz z ograniczeniem do 222 Hz, 32 KB pamięci podręcznej, pierwszy związany z grafiką, posiada wbudowaną jednostkę wektorową i układ graficzny, drugi związany z dźwiękiem oraz dekodowanie filmów	ARM 9, 66 MHz i ARM 7, 33 Mhz	Aple A4, taktowanie 1 Ghz oraz pamięć podręczna L1 64 KB, L2 640 KB
GPU	Zintegrowane jest z pierwszym rdzeniem, taktowania zegara 1-166 Mhz z ograniczeniem do 111 Mhz, 4 MB pamięci	Zintegrowane 2 układy do tworzenia grafiki 2D oraz 1 układ do tworzenia grafiki 3D	Zintegrowany oparty o PowerVR
Pamięć operacyjna	64 MB	4 MB	512 MB (2 chipy po 256 MB)
Pamięć masowa	32 MB NAND Flash	EEPROM, FLASH, FRAM do 512 KB	16 GB albo 32 GB flash
Komunikacja	WiFi	WiFi	WiFi, bluetooth
Kontrolery	Wbudowany pad	Wbudowany pad, ekran dotykowy	Ekran dotykowy, żyroskop, akcelerometr
Wyjście wideo	1 ekran LCD, 480×272, 16.77 kolorów	2 ekrany LCD, 256×192 pikseli, 18-bitowa głębia kolorów	1 ekran LCD, 960×640 pikseli

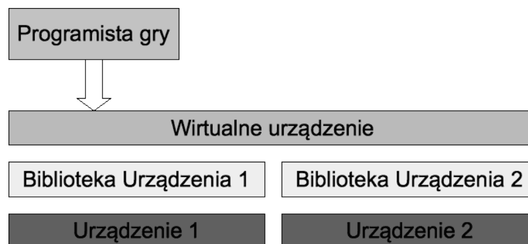
terfejsach API silnika. Z drugiej strony powinien obsługiwać dany sprzęt w jak najlepszy sposób, maksymalnie wydajnie. Żeby zrealizować oba te cele, trzeba się mocno nagimnastykować, gdyż bardzo często stoją one w opozycji. Aby napisać optymalny kod wykorzystujący dany specyficzny sprzęt, najłatwiej jest wykorzystać bezpośrednio związaną z nim bibliotekę. Niestety, biblioteki służące do oprogramowania relatywnie podobnego sprzętu różnych producentów mogą różnić się całkowicie sposobem ich użycia. Czasem różnice są niewielkie i rzeczywiście dotyczą tylko innej nomenklatury - wtedy fragmenty kodu korzystające z takich bibliotek są bardzo podobne. Różnią się tylko nazwami wywoływanych metod oraz ewentualnie sposobem przekazywania argumentów, a struktura algorytmu pozostaje bez zmian. Gorzej jest, gdy zaprojektowane są według całkiem innej wizji. Wówczas korzystanie z bibliotek dotyczących analogicznej tematyki jest całkiem inne. Fragmenty kodu oparte na takich bibliotekach są całkiem niepodobne do siebie. W takiej sytuacji bardzo trudno jest zachować spójność z interfejsem. Projektując interfejs silnika, należy mieć na uwadze takie problemy.

Analizując różnice w budowie współczesnych konsol siódmej generacji, można zauważyć, że niektóre z nich w ogóle nie mogą ze sobą konkurować (różnice w wydajności lub w sterowaniu są ogromne). W takim przypadku trudno sobie wyobrazić grę, która mogłaby zostać zaprojektowana w taki sposób, aby

była równie atrakcyjna na tak różne platformy. Dlatego projektując, na grę wieloplatfomową wybiera się zazwyczaj platformy podobne do siebie.

4.3 Wirtualizacja sprzętu

Wirtualizacja polega na dodaniu pewnej warstwy abstrakcji pośredniej między API danej platformy a warstwą API silnika. Warstwa taka pozwala na dopasowanie tych czasem nie do końca spójnych elementów. W przypadku silnika gry wirtualizacji dokonuje się często względem dostępnego sprzętu. Celem wirtualizacji jest stworzenie wirtualnego, abstrakcyjnego interfejsu urządzenia o pewnej typowej i skończonej funkcjonalności, które przesłoni dla programisty korzystającego z silnika prawdziwe, specyficzne urządzenie. Abstrakcyjne, wirtualne urządzenie powinno posiadać wszystkie typowe cechy urządzenia, które przesłania. Przesłaniać natomiast powinno specyficzne detale, które dla programisty gry nie są tak naprawdę ważne. Podejście takie jest bardzo wygodne, gdyż programista obsługujący wirtualne urządzenie musi znać tylko klasę problemu, nie interesują go natomiast szczegóły. W najprostszym przypadku implementacja wirtualnego urządzenia sprowadza się do oprogramowania analogicznego, rzeczywistego urządzenia. Jest to przypadek optymistyczny, kiedy dana platforma posiada takie dość typowe urządzenia. Implementacja jest wtedy bardzo prosta - kończy się bowiem na wywołaniu odpowiednich funkcji z biblioteki programistycznej danego urządzenia. Najbardziej skrajny przypadek z drugiej strony występuje wtedy, gdy dana platforma w ogóle nie posiada danego rzeczywistego sprzętu, a ze względu na konieczność zachowania spójności istnieje potrzeba stworzenia urządzenia wirtualnego. Wtedy implementacja może emulować rzeczywiste urządzenie programowo lub w oparciu o inny, podobny sprzęt. Sam fakt, że to jest możliwe, świadczy o elastyczności tego podejścia.



Rysunek 4.1. Warstwowy dostęp do urządzeń

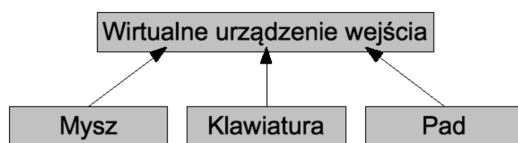
Wirtualizacja przy użyciu współczesnych języków obiektowych jest bardzo prosta. Mechanizm polimorfizmu, który jest podstawą współczesnych języków obiektowych, wręcz idealnie nadaje się do tego zadania. Na wstępie powinno

się zdefiniować interfejs komunikacji z urządzeniem. Interfejs powinien jak najlepiej definiować urządzenie, czyli składać się z szeregu metod wirtualnych, które umożliwiają wygodną komunikacje z urządzeniem. Wirtualne urządzenia pod danymi platformami muszą zaimplementować w odpowiedni sposób istniejący interfejs. Nie ma znaczenia, czy wykorzystają do tego bibliotekę obsługującą rzeczywisty sprzęt, czy implementacja będzie tylko emulacją takiego sprzętu. Ważne jest, że spójność jest zachowana, a programista może używać urządzenia, nie martwiąc się o szczegóły.

4.4 Przykłady wirtualizacji

4.4.1 Wirtualizacja kontrolerów wejścia

Tak jak zostało wcześniej wspomniane, poszczególne platformy znacznie różnią się od siebie pod względem sterowania. Z powodu ogromnych często rozbieżności w budowie kontrolerów (np. ekran dotykowy kontra klasyczny joypad) wydawać się może, że stworzenie wirtualnego kontrolera może być niemożliwe. Wbrew pozorom wcale nie jest to takie trudne. Warto najpierw skupić się na ogólnym celu takiego sprzętu oraz na cechach wspólnych dostępnych na rynku kontrolerów. Ogólnie celem kontrolerów gry jest zdobywanie informacji o stanie przycisków oraz wejść analogowych. Wszystkie kontrolery składają się z pewnej liczby wejść cyfrowych oraz pewnej liczby wejść analogowych. Jedyną różnicą jest liczba wejść. Wirtualny kontroler powinien cechować się ilością oraz stanem wejść cyfrowych i analogowych. Takie informacje są wystarczające dla programisty gry.



Rysunek 4.2. Wirtualizacja urządzeń wejścia

Na ilustracji widoczna jest wirtualizacja dostępnych kontrolerów. Wirtualny kontroler reprezentujący klawiaturę komputerową składa się ze 104 wejść cyfrowych bez żadnego wejścia analogowego. Standardowa mysz składa się z trzech wejść cyfrowych reprezentujących przyciski myszy oraz trzech wejść analogowych reprezentujących odpowiednio wychylenie kursora horyzontalne, wychylenie kursora wertykalne i pozycję kółka. Standardowy pad składa się zazwyczaj z dwóch wejść analogowych na „gałkę” oraz z kilku wejść cyfrowych dla przycisków. Akcelerometr składa się z trzech wejść analogowych dla wychylenia w każdej płaszczyźnie. Standardowy ekran dotykowy składa się z wejścia cyfrowego określającego, czy ekran w danym momencie jest dotknięty i dwóch

wejść cyfrowych reprezentujących pozycje ostatniego dotknięcia. Aby system wejścia był maksymalnie elastyczny, należy dodatkowo uwzględnić różną liczbę kontrolerów wejścia. Nawet w najprostszych platformach system wejścia składa się z kilku kontrolerów.

Interfejs urządzenia wejścia powinien umożliwiać zdefiniowanie urządzenia. Za pomocą tego interfejsu programista powinien umieć się dowiedzieć, z ilu wejść składa się kontroler, ile z nich jest cyfrowych, a ile analogowych, i najważniejsze - dowiedzieć się, jaki jest stan danych przycisków. Poniżej przedstawiona jest propozycja takiego interfejsu w języku C++:

```
//! Virtual controller interface. Defines the contoller build,
enables check input states.
class IController
{
public:
     struct SAnalogState
     {
        float fValue;
        const float fMinValue;
        const float fMaxValue;
        SAnalogState(float fMin, float fMax)
             : fMinValue(fMin),fMaxValue(fMax){}
     };
     virtual ~IController(){}
     //! Returns number of all digital inputs
     virtual unsigned int GetDigitalInputsCount() const = 0;
     //! Returns number of all analogs inputs
     virtual unsigned int GetAnalogInputsCount() const = 0;
     //! Returns state of specified digital input.
     virtual bool GetDigitalInputState(
                          unsigned int iInputIdx) const = 0;
     //! Returns state of specified analog input.
     virtual const SAnalogState&  GetAnalogInputState(
                          unsigned int iInputIdx) const = 0;
     protected:
     IController(){}
};
```

Listing 4.4-1 - Interfejs wirtualnego kontrolera

`IController` jest klasą całkowicie abstrakcyjną, nie posiada żadnych pól oraz posiada wszystkie metody abstrakcyjne, więc odpowiada interfejsowi z innych języków obiektowych. Używając tego interfejsu w łatwy sposób, można scharakteryzować kontroler. Za pomocą metod `GetDigitalInputsCount` oraz `GetAnalogInputsCount` można dowiedzieć się o liczbie wejść cyfrowych oraz analogowych. Aby sprawdzić stan określonego wejścia cyfrowego, należy użyć

metody `GetDigitalInputState`, która zwraca prawdę, gdy przycisk jest wciśnięty, a fałsz w przeciwnym wypadku. Argumentem metody jest indeks wejścia cyfrowego, który może przyjmować wartości od zera do liczby wejść cyfrowych minus jeden. Podobnie w przypadku wejść analogowych - należy tu skorzystać z metody `GetAnalogInputState`, która zwraca referencje do struktury opisującej stan wejścia analogowego. Struktura ta ma 3 pola odpowiadające kolejno wartości minimalnej, maksymalnej oraz wartości stanu wejścia analogowego.

Klasy wirtualnych urządzeń powinny implementować interfejs IController, czyli w przypadku języka C++, publicznie dziedziczyć, przeciążając wirtualne

```
class CMouseController : public IController
{
public:
    enum EDigitalInputs
    {
        BUTTON_LEFT,
        BUTTON_MIDLE,
        BOTTON_RIGHT,
        _DIGITAL_INPUTS_COUNT
    };

    enum EAnalogInputs
    {
        X_AXIS,
        Y_AXIS,
        WHEEL,
        _ANALOG_INPUTS_COUNT
    };

    CMouseController();
    virtual ~CMouseController();
    //! Returns number of all digital inputs
    virtual unsigned int GetDigitalInputsCount() const
                      {return_DIGITAL_INPUTS_COUNT;}
    //! Returns number of all analogs inputs
    virtual unsigned int GetAnalogInputsCount() const
                       {return_ANALOG_INPUTS_COUNT;}
    //! Returns state of specified digital input.
    virtual bool GetDigitalInputState(unsigned int iInputIdx) const;
    //! Returns state of specified analog input.
    virtual const SAnalogState&  GetAnalogInputState(
    unsigned int iInputIdx) const;
};
```

Listing 4.4-2 - Klasa reprezentująca mysz komputerową

metody. Przykładową definicję klasy wirtualnego urządzenia reprezentującego mysz komputerową pokazano na listingu 4.4.1.

Jak widać na powyższym przykładzie, kontroler myszy składa się z trzech wyjść cyfrowych reprezentujących trzy guziki myszy oraz trzech wyjść analogowych reprezentujących wychylenie kursora myszy względem osi wertykalnej, horyzontalnej ekranu i wejść reprezentujących obrót kółka służącego do przewijania ekranu. Definicja metod zwracających stan wejść powinna zostać oparta na bibliotece obsługi myszy na danej platformie.

Korzystanie z tak przygotowanego kontrolera jest bardzo proste. Aby np. sprawdzić, czy użytkownik w danym momencie naciska lewy przycisk myszy, wystarczy wywołać metodę `GetDigitalInputState` z odpowiednim argumentem w następujący sposób:

```
bool bLeftButtonPressed =  pMouseController->
GetDigitalInputState(CMouseController::BUTTON_LEFT);
```

Listing 4.4-3 - Kod sprawdzający naciśnięcie lewego przycisku myszy

Gdzie `pMouseController` jest wskaźnikiem wskazującym na obiekt klasy `CMouseController`.

Oczywiście w analogiczny sposób można zdefiniować inne wymagane klasy kontrolerów dla urządzeń takich jak Pad, Joystick czy Touchpad. Co ciekawe, gdy w danym systemie nie istnieje wymagane urządzenie, można je zaemulować za pomocą innego urządzenia, nawet takiego, które nie jest fizycznie podobne do oryginału. Na przykład, gdyby istniała potrzeba emulacji myszy za pomocą klawiatury, można zdefiniować nową klasę dziedziczącą po klasie `CMouseController`, która w odpowiedni sposób przeciąży metody zwracające stan wejść cyfrowych oraz analogowych w oparciu o stan klawiszy na klawiaturze. Pomimo tego, że klawiatura nie posiada wejść analogowych, to istnieje możliwość ich emulacji. Wystarczy, że klasa rozszerzająca będzie posiadała pola reprezentujące pozycje kursora. Naciśnięcie właściwych klawiszy powinno zwiększać oraz zmniejszać wartości tych pól. Ciekawostką jest, że posiadając w taki sposób zdefiniowane kontrolery, można dokonać jeszcze innych niebanalnych sztuczek. Możliwe jest na przykład zaimplementowanie klasy rozszerzającej dany kontroler, która tylko symuluje jego wykorzystywanie. Modelowym przykładem korzystającym z takiego rozwiązania może być użycie takiego kontrolera przez moduł sztucznej inteligencji.

4.4.2 Wirtualizacja pamięci operacyjnych

Tak jak w wirtualizacji kontrolerów wejścia, tak i w przypadku wirtualizacji pamięci operacyjnej należy stworzyć wirtualne urządzenie pamięci. Urządzenie takie powinno się charakteryzować typowymi cechami pamięci operacyjnej.

Dla zapewnienia większej elastyczności pamięć wirtualna może składać się z bloków. Każdy blok powinien być scharakteryzowany przez rozmiar oraz metody dostępu do takiej pamięci za pomocą alokatorów. Dzięki takiemu podejściu możliwe jest wykorzystywanie odpowiednich bloków zgodnie z ich polecanym zastosowaniem, co ma szczególnie duże znaczenie na konsolach gier, gdzie bardzo często pamięć podzielona jest na obszary służące do konkretnego zastosowania. Pamięć może być też podzielona ze względu na typ danych czy też szybkość, wtedy szybsze bloki służą do przechowywania często używanych zasobów i często wykonanego kodu. Na przykład mogą istnieć bloki do przechowywania tekstur, geometrii lub dźwięków. Praktyczne wykorzystanie alokatorów znajduje się w rozdziale poświęconym zarządzaniu pamięcią.

4.4.3 Wirtualizacja pamięci masowych

Różne platformy mogą posiadać bardzo różne pamięci masowe. Wśród pamięci przeznaczonych do odczytu danych najbardziej popularne są dyski DVD, dyski twarde oraz cartridge. Najbardziej popularne pamięci to dyski twarde i pamięci flash. Na szczęście, pomimo ogromnych różnic w budowie różnych pamięci masowych z punktu widzenia programisty obsługa ich jest bardzo podobna, więc zaprojektowanie uniwersalnego interfejsu nie powinno przysporzyć problemów. Przy dobrze zaprojektowanym systemie wejścia/wyjścia różnice wynikające z budowy urządzeń nie powinny w ogóle mieć znaczenia.

W przypadku wirtualizacji urządzeń pamięci masowych idealnie sprawdza się koncepcja strumieni. Urządzenie wejścia lub wyjścia reprezentowane jest jako strumień, do którego można zapisywać albo odczytywać dane. W strumieniach kolejne operacje zapisu lub odczytu zmieniają stan strumienia. Czyli w przypadku strumienia wejściowego kolejne operacje odczytu w końcu spowodują, że strumień będzie pusty. Koncepcja strumieni jest już bardzo stara, ale do dziś świetnie się sprawdza i właściwie każdy język obiektowy posiada w swojej bibliotece standardowej implementację strumieni. W przypadku komputerów osobistych są to strumienie reprezentujące pliki na dysku twardym, klawiaturę lub konsolę. Koncepcja ta równie dobrze sprawdzi się na innych platformach, wystarczy stworzyć klasy strumieni do operacji wejścia/wyjścia dla typowych dla tej platformy urządzeń pamięci masowych.

W najprostszej implementacji klasa reprezentująca strumień wejściowy powinna posiadać metodę do odczytu danych w formie tablicy znaków oraz metody do sprawdzenia bądź zmiany pozycji w strumieniu. Analogiczna klasa strumienia wyjściowego powinna posiadać metody do zapisu danych z tablicy znaków, odczytu oraz modyfikacji pozycji strumienia. Oczywiście implementacje tych metod są zależne od platformy, dlatego struktura architektoniczna klas strumieni powinna to przewidywać.

Poniżej przedstawiono propozycje interfejsów potrzebnych do implementacji strumieni. Interfejs `IStream` posiada szereg abstrakcyjnych metod związanych z obsługą strumienia. Metody `SetPosition` oraz `GetPosition` służą

```
class IStream
{
public:
   //! virtual destructor
   virtual ~IStream(){}
   //! Sets the stream position.
   virtual void SetPosition(unsigned int iPosition) = 0;
   //! Returns the stream position.
   virtual unsigned int GetPosition() = 0;
   //! Returns true when End Of File is reached.
   virtual bool IsEof() = 0;
   //!  Flushes the stream.
   virtual void Flush() = 0;
};

class IInputStream : public IStream
{
public:
   virtual ~IInputStream(){}
   //! Reads from stream specified number of characters to
   //! specified buffer. Returns the total ammount of successful
   //! bytes read.
   virtual unsigned int Read(void *pBuffer, unsigned int iBytes) = 0;
};

class IOutputStream : public IStream
{
   virtual ~IOutputStream(){}
   //! Write to stream specified number of characters from
   //! specified buffer. Returns the total amount of successful
   //! bytes write.
   virtual unsigned int Write(void *pBuffer,
           unsigned int iBytes) = 0;
};
```

Listing 4.4-4 - Interfejsy strumieni

do modyfikacji pozycji zapisu lub odczytu w strumieniu, zaś metoda `IsEof` do informowania o osiągnięciu końcowej pozycji strumienia. W przypadku strumieni plikowych oznacza to koniec pliku. Metoda `Flush` synchronizuje strumień, co ma ogromne znaczenie, gdy implementacja strumienia wykorzystuje buforowanie. Po wywoływaniu tej metody w przypadku strumienia plikowego stan pliku powinien się zsynchronizować ze strumieniem.

Przedstawione interfejsy, pomimo że są bardzo proste, umożliwiają wygodną obsługę strumieni. Strumienie wejściowe pod konkretną platformę powinny implementować interfejs `IInputStream`, wyjściowe interfejs `IOutputStream`.

Dla platformy obsługującej DVD Rom można zadeklarować klasę strumienia w następujący sposób:

```
class CDvdInputStream : public IInputStream
{
//...
}
```

Listing 4.4-5 - Klasa strumienia obsługi DVD Rom

Na platformie obsługującej odczyt z cartridge można stworzyć implementacje strumienia tak:

```
class CCardInputStream : public IInputStream
{
//...
}
```

Listing 4.4-6 - Klasa strumienia obsługi cartridge

Jak widać, obsługa strumieni jest całkowicie niezależna od platformy. Odczyt z DVD czy z cartridge przebiega w identyczny sposób, wystarczy się odwoływać do odpowiedniego strumienia poprzez referencję lub wskaźnik na interfejs `IInputStream`.

Zaproponowane tu interfejsy są bardzo proste, żeby przedstawiony przykład był możliwie przejrzysty. Warto jednak wyposażyć też strumienie w inne użyteczne metody. Bardzo często strumienie posiadają metody do zapisywania lub odczytywania konkretnych typów - przeważnie są to typy proste języka i inne często używane typy, takie jak łańcuchy znakowe czy typy matematyczne (wektory, macierze). Podczas implementacji warto rozważyć taką funkcjonalność, ponieważ wykorzystywanie tylko zaproponowanych metod `Read` oraz `Write` może być niewygodne. Jednak powinny one zostać w interfejsie, gdyż nie jest możliwe stworzenie metod dedykowanych dla wszystkich możliwych typów. Metody `Read` i `Write` w takiej postaci są bardzo funkcjonalne, ponieważ umożliwiają operacje na dowolnych typach, choć zdecydowanie łatwiej używa się metod dla konkretnych typów. W przypadku języka C++ bardzo dobrym pomysłem jest przeciążenie operatorów strumieniowych (» oraz «), których nazwa nawet sugeruje, że idealnie nadają się do tego zadania.

W przypadku, gdy strumienie służą do operacji na dużych danych, warto rozważyć dodanie operacji asynchronicznych. Funkcjonalność taka jest zazwyczaj bardzo potrzebna w przypadku dużych gier. Ilość danych koniecznych do wczytania, aby stworzyć świat gry, jest zazwyczaj tak duża, że czas oczekiwania mógłby znudzić gracza, lub co gorsza, spowodować, że gracz mylnie uzna, iż gra się zawiesiła. Aby temu zapobiec, w najprostszym przypadku można wyś-

wietlić prostą animację lub pasek postępu podczas wczytywania, a w bardziej złożonym można doczytywać odpowiednie fragmenty sceny podczas trwania gry. Aby jednak było to możliwe, potrzebne są asynchroniczne strumienie. Wówczas zaproponowany interfejs powinien posiadać możliwość zarejestrowania wywołania zwrotnego (ang. *callback*) np. funktora lub obserwatora, służących do informowania o stanie odczytu.

4.4.4 Wirtualizacja urządzeń graficznych

Obsługa układów graficznych jest dziś dość mocno ustandaryzowana, co nie oznacza niestety, że wszystkie dostępne biblioteki graficzne są takie same. Na skutek ogromnej popularności dwóch bibliotek graficznych *OpenGL* i *DirectX* standaryzacji uległ sposób renderowania obrazów dwu i trójwymiarowych. Biblioteki te w gruncie rzeczy są do siebie podobne, pomimo dość sporych różnic w API. Sama koncepcja renderowania obrazów jest taka sama. Programista ma do dyspozycji metody oraz funkcje:

- do renderowania obiektów graficznych zbudowanych z wielościanów,
- do dokonywania transformacji (translacji, rotacji, skalowania) renderowanych obiektów,
- do obsługi oświetlenia oraz do manipulacji właściwościami materiałów obiektów,
- służące teksturowaniu obiektów,
- do obsługi shaderów,
- do obsługi buforów kolorów, głębokości itd.

Obie biblioteki mają ogromne rzesze fanów debatujących nad tym, która z nich jest lepsza. Silnik wieloplatformowy w idealnym przypadku powinien obsługiwać co najmniej obie te biblioteki. Jeżeli ma to być tylko jedna z nich, zazwyczaj lepszym wyborem będzie *OpenGL* ze względu na to, że jest wspierany na większej liczbie platform.

```
class IGraphicsDevice
{
public:
    virtual ~IGraphicsDevice(){}
    //! Returns image size
    virtual const Dimension2i& GetDimension() const = 0;
    //! Returns pixel format descriptor
    virtual const IPixelFrmt&  GetPixelFormat() const = 0;
    //! Returns render context
    virtual IRenderContext& GetRenderContext() const = 0;
};
```

Listing 4.4-7 - Interfejs urządzenia graficznego

Tak jak w poprzednich przykładach, dobrym wyjściem jest stworzenie wirtualnego urządzenia graficznego (Listing 4.4.7). Urządzenie takie cechuje się przed wszystkim kontekstem, czyli aktualnym stanem. Kontekst to stan poszczególnych podukładów układu graficznego, którego zmiany mają wpływ na powstający w urządzeniu obraz. Oprócz tego urządzenie graficzne jest scharakteryzowane przez wymiary oraz format piksela renderowanego obrazu.

Interfejs urządzenia graficznego dostarcza informacje o wymiarach i o formacie renderowanego obrazu. Dodatkowo też zwraca referencje do kontekstu, za pomocą którego możliwe jest renderowanie obrazu.

Za pomocą interfejsu `IRenderContext` możliwe powinno być wyrenderowanie każdego obrazu. Jego definicja powinna zawierać metody, umożliwiające wykorzystanie całego potencjału dostępnych bibliotek graficznych. W idealnym przypadku za pomocą `IRenderContext` powinno się oferować dokładnie to, co oferuje biblioteka *OpenGL* lub *DirectX*. Zresztą implementacje tego interfejsu powinny być oparte na tych bibliotekach.

Poniższa definicja interfejsu kontekstu jest najprostsza z możliwych. Daje możliwość wyrenderowania obiektu składająceco się z trójkątów położonych w konkretnym miejscu sceny.

```
class IRenderContext
{
public:
    virtual ~IRenderContext(){}
    //! Sets the projection matrix.
    virtual void SetProjectionMatrix(const Matrix4x4f& mat) = 0;
    //! Sets model view matrix.
    virtual void SetModelViewMatrix(const Matrix4x4f& mat) = 0;
    //! Draws primitive from specified triangles
    virtual void DrawTriangles(
            unsigned short *aIndices,
            Vertex *aVertices,
            unsigned int iTrianglesCount) = 0;
    //! Swaps buffers (finish rendering)
    virtual void SwapBuffers() = 0;
};
```

Listing 4.4-8 - Interfejs kontekstu

Za pomocą metody `SetProjectionMatrix` możliwe jest ustawienie macierzy rzutowania. Metoda `SetModelViewMatrix` służy do ustawienia macierzy modelowania. Za pomocą metody `DrawTrinagles` można wyrysować określoną liczbę trójkątów, których wierzchołki podane są w tablicy wierzchołków, pierwszym argumentem jest tablica indeksów do wierzchołków. Metoda `SwapBuffers` przerzuca z kolei tylni bufor do bufora wynikowego, czyli generuje obraz.

Po tak zadeklarowanym interfejsie mogą dziedziczyć implementacje oparte na bibliotekach graficznych. Gdy istnieje taka konieczność, można też stworzyć całkowicie niezależną od sprzętu implementację, czyli renderer software'owy.

```
class DXRenderContext : public IRenderContext
{
//...
};

class OGLRenderContext : public IRenderContext
{
//...
};

class SoftRenderContext : public IRenderContext
{
//...
};
```

Listing 4.4-9 - Klasy renderera software'owego

Kontekst renderujący należy do urządzenia graficznego i jest z nim bezpośrednio związany. Jest on także przez takie urządzenie tworzony. Implementacje wirtualnych urządzeń powinny inicjalizować odpowiednią bibliotekę oraz stworzyć odpowiedni kontekst renderujący.

4.5 Wykorzystanie specyficznych jednostek w procesorach

Bardzo ważnym zagadnieniem jest wykorzystanie specjalistycznych układów jednostki centralnej. Jeżeli sprzęt wyposażony jest w dodatkowe układy i narzędzia te rozwiązują jakieś zagadnienia, trzeba to wykorzystać, gdyż rozwiązywanie ich bez udziału tych układów jest marnotrawstwem w dwojaki sposób. Po pierwsze, dedykowane układy do specyficznych zadań są zawsze szybsze niż jednostka centralna wykorzystana w takim celu. Po drugie, gdy nie wykorzystujemy dedykowanego układu, jednostka główna jest bardziej obciążona.

Bardzo wiele współczesnych platform wyposażonych jest w układy do obliczeń macierzowych. Jest to spowodowane tym, że w przypadku gier trójwymiarowych ogromna większość obliczeń dotyczy operacji na macierzach i wektorach, które określają względne położenie obiektów trójwymiarowych. Z drugiej strony na rynku ciągle istnieją platformy, na których wszelkie operacje macierzowe należy wykonać osobiście.

Stworzenie szybkiej i uniwersalnej międzyplatformowej biblioteki matematycznej nie jest trywialne pomimo tego, że większość platform udostępnia API do takich operacji. Z jednej strony użycie operacji na macierzach, wektorach czy kwaternionach musi być bardzo łatwe i intuicyjne, z drugiej zaś, biblioteka taka musi być maksymalnie wydajna i w idealnym przypadku nie powinna generować żadnego narzutu. Oba te cele jest dość trudno pogodzić. Rozpatrzmy problem na przykładzie prostej klasy wektora. Wektor składa się z 3 pól: x, y, z oraz metod do standardowych operacji na wektorach, takich jak: dodawanie, odejmowanie, mnożenie skalarne oraz wektorowe. Ze względu na wygodę przy deklaracji klas matematycznych warto stosować przeciążone operatory matematyczne.

```
class Vector3f
{
public:
   union
   {
      struct
      {
         float x;
         float y;
         float z;
      };
      float elements[VECTOR3F_ALIGMENT];
   };

public:
   inline Vector3f(float x,float y,float z);
   inline Vector3f(const Vector3f& v3Other);
   inline Vector3f operator+(const Vector3f& v3Other);
   inline Vector3f operator-(const Vector3f& v3Other);
   inline float operator*(const Vector3f& v3Other);
   inline Vector3f operator^(const Vector3f& v3Other);
   inline Vector3f& operator=(const Vector3f& v3Other);
};
```

Listing 4.5-1 - Klasa wektora

Powyższa deklaracja wektora jest bardzo prosta i wygodna w użyciu. Na przykład dodawanie wektorów jest intuicyjne:

```
   Vector3f a(10.0f,10.0f);
   Vector3f b(-15.0f,10.0f);
   Vector3f c = a + b;
```

Listing 4.5-2 - Dodawanie wektorów

Tak zadeklarowany wektor ma funkcjonalny interfejs, więc jego użycie nie powinno sprawiać problemów. Niestety, przy takiej deklaracji dość trudno jest uniezależnić implementację od platformy. Implementacja operacji dodawania, odejmowania czy mnożenia może być inna w zależności od platformy. Na platformy nieposiadające jednostki macierzowej metoda służąca do dodawania może wyglądać następująco:

```
Vector3f Vector3f::operator+(const Vector3f& v3Other)
{
   return Vector3f(x+v3Other.x,y+v3Other.y,z+v3Other.z);
}
```

Listing 4.5-3 - Operator dodawania wektorów bez wykorzystania jednostki macierzowej

W przypadku, gdy sprzęt wyposażony jest w jednostkę do obliczeń wektorowych, kod metody dodającej może wyglądać następująco (przykładowy kod wykorzystuje bibliotekę *Accelerate Framework* firmy Apple, która w zależności od dostępnego procesora wykorzystuje rozkazy AltiVec lub SSE):

```
Vector3f Vector3f::operator+(const Vector3f& v3Other)
{
    Vector3f v3Result;
    vDSP_vadd( elements, 1, v3Other.elements, 1,
    v3Result.elements, 1, VECTOR3F_ALIGMENT );
    return v3Result;
}
```

Listing 4.5-4 - Operator dodawania wektorów z wykorzystaniem jednostki macierzowej

Jak widać na powyższym przykładzie, definicje metod wektora są rożne w zależności od platformy. Obsługa operacji na wektorach przy użyciu polimorfizmu jest niemożliwa przynajmniej z dwóch powodów. Po pierwsze, typowe operacje takie jak dodawanie, odejmowanie czy mnożenie skalarne jako wynik powinny stworzyć oraz zwrócić nowy obiekt wektora. Nie może to być referencja lub wskaźnik do interfejsu bazowego. Jest to związane z problem własności obiektu. Gdyby operacje zwracały tylko referencje do obiektu, musiałyby przechowywać wynik. Rozwiązaniem tego problemu są metody które wynik zwracają za pomocą referencji przekazanej jako argument. Jednak są one nieintuicyjne i nie nadają się do stosowania jako przeciążone operatory arytmetyczne. Po drugie, operacje matematyczne są dość krótkie, ale złożone obliczeniowo, powinny być więc maksymalnie zoptymalizowane. Jedną z me-

tod optymalizacji jest napisanie ich w postaci funkcji inline. W przypadku interfejsu oraz metod wirtualnych wywołanie inline działa tylko w sytuacji, gdy metoda wywołana jest jawnie z konkretnego poziomu dziedziczenia (za pomocą operatora zakresu). Dlatego wywoływanie metod wirtualnych z wykorzystaniem polimorfizmu jest wolniejsze. Projektując klasy do obsługi matematyki, działające pod wieloma platformami, należy skorzystać z kompilacji warunkowej. Nie jest to może tak eleganckie rozwiązanie jak zastosowanie polimorfizmu, ale w tym przypadku konieczne.

Należy zauważyć, że w przypadku wektora nawet definicja klasy może być różna w sytuacji rożnych platform. W jednostkach wektorowych VMX, stosowanych w konsolach do gier, wektor danych powinien mieć długość 128 bitów, co odpowiada czterem 32 bitowym liczbom zmiennoprzecinkowym (`float`). Dlatego w deklaracji klasy zastosowano unię, która w razie potrzeby rozszerzy rozmiar wektora za pomocą makra `VECTOR3F_ALIGMENT`.

Załóżmy, że silnik będzie domyślnie liczył operacje matematyczne w sposób standardowy za pomocą jednostki głównej. Jeżeli zostanie skompilowany z zdefiniowanym makrem `DIRECT_VMX_ENABLE`, będzie korzystał z jednostki wektorowej VMX bezpośrednio za pomocą rozkazów Altivec. Dla zdefiniowanego makra `ACCELERATE_ENABLE` skorzysta z biblioteki *Accelerate*, która w przypadku procesora PowerPC wyposażonego w jednostkę VMX skorzysta z rozkazów Altivec, a w przypadku procesora Intel skorzysta z rozkazów SSE.

Ze względu na to, że metody powinny być inline i że deklaracja klasy jest różna, w zależności od wersji, sekcje warunkowe muszą już znajdować się w pliku nagłówkowym. Ważne jest, aby kod warunkowy nie zdominował pliku nagłówkowego. Spowodowałoby to pogorszenie czytelności tego pliku, a pliki nagłówkowe są bardzo pomocne programistom.

```
#ifndef_VECTOR3F_H
#define_VECTOR3F_H

#ifdef DIRECT_VMX_ENABLE
    #define VECTOR3F_ALIGMENT 4
#elif defined(ACCELERATE_ENABLE)
    #define VECTOR3F_ALIGMENT 4
#else
    #define VECTOR3F_ALIGMENT 3
#endif

class Vector3f
{
//..
};
```

```
#ifdef VMX_ENABLE
    #include "Vector3fVMX.inl"
#elif defined(ACCELERATE_ENABLE)
    #include "Vector3fAccelerate.inl"
#else
    #include "Vector3f.inl"
#endif

#endif
```

Listing 4.5-4 - Użycie kompilacji warunkowej dla klasy wektora

Jak widać powyżej, makro `VECTOR3F_ALIGMENT` jest różne w zależności od wersji. Dla wersji domyślnej wektor składa się z trzech liczb zmiennoprzecinkowych, czyli zajmuje 12 bajtów, dla wersji obsługującej jednostkę wektorową wektor zajmuje 16 bajtów. Na dole pliku nagłówkowego znajduje się sekcja warunkowa odpowiedzialna za dołączenie odpowiedniego pliku zawierającego definicje funkcji inline dla specyficznej platformy. Dla wersji obsługującej VMX dołączony jest plik z definicjami funkcji korzystającymi z biblioteki do obsługi jednostki wektorowej, domyślnie dołączony jest również plik z definicjami, w których obliczenia wykonywalne są normalnie, korzystając z jednostki głównej.

W analogiczny sposób można oczywiście zaprojektować klasy macierzy, kwaternionów i innych, w których można wykorzystać obliczenia wektorowe.

4.6 Różnice formatów danych

Różnice w budowie sprzętu oraz w bibliotekach programistycznych mogą za sobą pociągać różnice w formacie danych. Niektóre platformy mogą posiadać własne natywne formaty plików tekstur, dźwięków, shaderów, modeli trójwymiarowych oraz innych zasobów. Spowodowane to może być (choć nie musi) chęcią optymalnego wykorzystania sprzętu. Formaty takich plików są wtedy specjalnie zaprojektowane tak, że dane zawarte w plikach mogą być bezpośrednio wykorzystane przez sprzęt. Różnice w formatach danych mogą być też spowodowane wydajnością sprzętu. Na mocniejszych platformach można sobie pozwolić na przykład na tekstury o większej głębi kolorów niż na słabszej, która teoretycznie również obsługuje taki format.

Różnice w formatach danych mogą powodować problemy w projekcie na poziomie gry, ponieważ pomimo tego, że mechanizmy obsługi różnych formatów plików zawarte są w warstwie silnika, to samym wczytywaniem zasobów zajmują się programiści logiki gry. To oni wczytują i zwalniają modele, tekstury oraz inne potrzebne im zasoby, dlatego oni są odpowiedzialni za wczytanie zasobu w konkretnym formacie na daną platformę.

Dobrą praktyką, która pozwala zachować porządek w strukturze zasobów jest przechowywanie zasobów na różne platformy w różnych repozytoriach. Ścieżki do katalogów głównych z danymi na różne platformy są wtedy inne. Wszystkie ścieżki do zasobów powinny być podawane względem takiego katalogu głównego danej platformy. Umożliwia to posiadanie zasobów o takich samych nazwach, a o innej zawartości w zależności od platformy.

Załóżmy, że w danej grze korzystamy z tekstury zawartej w pliku „`wood.tga`", odwzorowującej drewnianą powierzchnię. Na mocniejszej platformie wyposażonej w lepsze wyjście wideo i o wyższej rozdzielczości oraz głębi koloru, tekstura ma wymiar 1024x1024 tekseli oraz głębię 32 bity. W przypadku słabszej platformy skorzystanie z takiej samej tekstury mogłoby być złą decyzją, gdyż z jednej strony wyświetlacz mógłby mieć za małą rozdzielczość oraz głębię koloru, by w pełni wykorzystać rozdzielczość tekstury, a z drugiej tekstura taka zajęłaby za dużo miejsca w pamięci. Lepszą więc decyzją w takiej sytuacji mogłoby być skorzystanie na przykład z „odchudzonej" tekstury o wymiarach 512x512 pikseli oraz głębi 24 bity.

Gdy posiadamy dwa osobne katalogi główne dla każdej platformy, takie że struktura ich wewnętrznych katalogów jest identyczna, programista nie musi rozróżniać tych tekstur w kodzie. Tekstura dla pierwszej platformy może znajdować się w katalogu „`/data/platform1/tex/wood.tga`", dla drugiej platformy analogicznym katalogiem może być „`/data/platform2/tex/wood.tga`", czyli katalog główny dla pierwszej tekstury to „`/data/platform1`", dla drugiej „`/data/platform2`". Po wcześniejszej właściwej konfiguracji katalogów głównych dla konkretnych platform wczytanie tekstury dla programisty gry wygląda identycznie:

```
CTexture *pWoodTexture = CTexture::Load("tex/wood.tga");
```

Listing 4.6-1 - Wczytywanie tekstur TGA

Problemem jednak dalej mogą być natywne formaty zasobów. Nie dość, że pliki natywnych formatów mogą mieć inne rozszerzenia, to jeszcze sposób tworzenia takich zasobów może wymagać innej implementacji.

Załóżmy, że pewna platforma obsługuje natywne formaty tekstur w formacie NTGA ze wsparciem sprzętowej dekompresji i obsługuje zwykłe standardowe TGA, ale bez sprzętowego wsparcia dekompresji. Użycie tekstur w formacie TGA na tej platformie jest mało optymalne, ponieważ tekstury zajmują więcej pamięci. Z drugiej strony, funkcjonalność tworzenia tekstur z plików TGA może być przydatna, szczególnie we wczesnej fazie projektu, kiedy optymalność nie jest bardzo ważna, a korzystanie z TGA jest wygodniejsze (na przykład, gdy projekt powstaje równolegle na wiele platform i gdy już istnieją grafiki w formacie TGA). Dlatego dla tej platformy powinny istnieć funkcje zarówno tworzące tekstury z formatu TGA, jak i NTGA.

Żeby móc skorzystać z tekstury NTGA, należy ją stworzyć w następujący sposób:

```
CTexture *pWoodTexture = CTexture::LoadNative("tex/wood.ntga");
```

Listing 4.6-2 - Wczytywanie tekstur NTGA

Pojawia się tu jednak problem spójności kodu. Dla platformy wspierającej NTGA należy wczytywać pliki w tym formacie, zaś dla platform niewspierających należy wczytywać zwykłe TGA. Skutecznym rozwiązaniem tego problemu może być znów kompilacja warunkowa oraz stworzone za pomocą niego makra do tworzenia zasobów.

```
#ifdef Platform1
    #define LoadTexture(name) \
    CTexture::LoadNative(#name".ntga")
#elif
    defined(Platform2)
    #define LoadTexture(name) \
    CTexture::Load(#name".tga")
#endif
```

Listing 4.6-3 - Makra tworzenia zasobu tekstury

Wtedy tworzenie zasobu jest całkowicie niezależne od platformy:

```
CTexture *pWoodTexture = LoadTexture(tex/wood);
```

Listing 4.5-3 - Wczytywanie tekstur w jednolity sposób

4.7 Różnice w implementacji gry

Stworzenie nawet bardzo dobrze zaprojektowanego silnika nie gwarantuje, że na poziomie gry nie wystąpią żadne problemy związane z różnicami między platformami. Różnice w budowie i liczbie kontrolerów czy liczba ekranów danej platformy muszą mieć wpływ na implementację samej gry. Projektując architekturę gry, warto od razu mieć na uwadze takie różnice, by w elegancki sposób rozwiązać wszystkie następujące z tego powodu problemy. Rozwiązaniem ostatecznym zawsze pozostaje kompilacja warunkowa, lecz można starać się rozwiązać te problemy w bardziej wyrafinowany sposób.

Podczas projektowania na przykład gry typu FPS przeznaczonej na różne platformy napotkamy na pewno problem sterowania. Na komputerach osobistych w tego typu grach najlepiej sprawdza się sterowanie za pomocą klawiatury i myszy. Na konsolach zaś, z braku lepszej alternatywy, wykorzystuje się pada.

Załóżmy, że posiadamy klasę gracza:

```
class CPlayer
{
//..
public:
//..
    void IncPostion(const Vector3f& v3Pos);
    void Rotate(const Quaternion& qRot);
};
```

Listing 4.7-1 - Klasa gracza

Wspomniana klasa posiada między innymi metody do poruszania gracza. Metoda `IncPosition` zmienia położenia gracza o podany wektor, a `Rotate` odpowiedzialna jest za obrót gracza (czyli w grze FPS za jego rozglądanie się). W przypadku gry na PC metoda `IncPostion` powinna zostać wywoływana po naciśnięciu klawiszy odpowiedzialnych za poruszanie (np. W,S,A,D), a metoda `Rotate` na wychylenie myszki. Na konsoli analogiczne interakcje powinny nastąpić na wychylenie „gałek" na padzie.

Oczywiście można wykorzystać kompilację warunkową i w zależności od definicji makra definiującego daną platformę, wykonywać bezpośrednio kod odpowiedzialny za translację stanu kontrolera na stan gracza. Lepszym rozwiązaniem jest jednak dodanie kolejnej warstwy abstrakcji, która zawiera kontroler gracza.

```
class IPlayerController
{
public:
    //! Updates player state.
    virtual void Update(CPlayer *pPlayer,float fDt);
};
```

Listing 4.7-1 - Interfejs kontrolera gracza

Metoda Update aktualizuje pozycje gracza na podstawie stanu urządzeń wejściowych. Na tym poziomie abstrakcji nie ma znaczenia, jakie to są urządzenia. Wywołując metodę Update, mamy pewność, że stan gracza się odpowiednio zmieni, a nie interesuje nas platforma. Dopiero klasy implementujące

kontroler gracza dla poszczególnych platform muszą znać takie szczegóły, jak rodzaj występujących kontrolerów.

```
class CPlayerControllerPC : public IPlayerController
{
public:
    //! Constructs and initialize controller using specified
    //! keyboard and mouse.
    CPlayerControllerPC(CKeyboard *pKB,CMouse *pMouse);

    //! Updates player state.
    virtual void Update(CPlayer *pPlayer,float fDt);
};

class CPlayerControllerXBox : public IPlayerController
{
public:
    //! Constructs and initialize controller using specified
    //! Xbox pad
    CPlayerControllerXBox(CPadXBox *pPad);

    //! Updates player state.
    virtual void Update(CPlayer *pPlayer,float fDt) = 0;
};
```

Listing 4.7-1 - Klasy implementujące kontroler gracza

Klasy kontrolerów na określonych platformach w konstruktorach otrzymują wskaźniki do specyficznych kontrolerów tych platform, dzięki temu są w stanie w metodzie Update dokonać translacji stanu odpowiednich kontrolerów na pozycje gracza.

5

Zarządzanie pamięcią operacyjną

5.1 Pamięć operacyjna

Efektywne zarządzanie pamięcią operacyjną jest niezmiernie ważnym aspektem silnika. Na zarządzanie tą pamięcią składa się przydzielanie grze pamięci na obiekty, struktury oraz inne dane z dostępnej pamięci ograniczonej przez system. Zarządzanie pamięcią polega również na zwalnianiu pamięci do systemu, gdy już nie jest ona wymagana przez grę. Systemy zarządzania pamięcią mogą być bardzo różne w zależności od typu aplikacji oraz budowy systemu na której jest uruchamiana, niemniej można powiedzieć, że wykonują dwa powiązane ze sobą zadania:

- Na niższym poziomie: zarządzają bezpośrednio dostępem do pamięci w celu rezerwowania oraz zwalniania obszarów pamięci na dane, czyli umożliwiają wszystkie dynamiczne alokacje oraz dealokacje pamięci. Za tę część odpowiedzialny jest tak zwany alokator pamięci.
- Na wyższym poziomie: zarządzają strategią tworzenia, zwalniania i współdzielenia zasobów wykorzystujących pamięć. Ogromnie ważnym aspektem jest optymalne zwalnianie niepotrzebnych już obszarów pamięci, aby było możliwe ponowne ich wykorzystanie. Zarządzanie na tym poziomie sprowadza się do kontrolowania, gdy dane są jeszcze potrzebne lub kiedy można zwolnić nieużywany blok pamięci do ponownego użycia. Pomimo, że brzmi to banalnie, jest problemem bardzo poważnym.

Słabe zarządzanie pamięcią prowadzi do wielu problemów takich jak:

- Przedwczesne zwolnienie pamięci - program zwalnia pamięć, a następnie się do niej odwołuje, zazwyczaj kończy się natychmiastowym zawieszeniem programu.
- Wyciek pamięci - program nie zwalnia wcześniej zaalokowanej pamięci, działa wolniej, a w ostateczności kończy się zawieszeniem, gdy cała pamięć zostaje zajęta, a konieczne są kolejne alokacje.

- Fragmentacja pamięci - słaby alokator rezerwuje oraz zwalnia bloki pamięci w taki sposób, że po jakimś czasie brakuje dostatecznie dużego wolnego bloku do alokacji, pomimo tego, że sumarycznie tej wolnej pamięci jest wystarczająco dużo.
- Niedopasowanie do projektu - zarządzanie pamięcią powinno być dopasowane odpowiednio do specyficznego projektu, platformy itd. Złe dobranie pewnych założeń (np. wielkość domyślnego bloku pamięci, czas życia zasobu) może powodować nieoptymalne wykorzystanie pamięci lub powolne działanie programu.

5.2 Alokacja oraz dealokacja pamięci

Alokacja pamięci to zarezerwowanie ciągłego obszaru pamięci przez proces do swojego użytku. Dealokacja jest działaniem odwrotnym - powoduje zwolnienie pamięci, czyli przywrócenie obszaru pamięci do puli pamięci w systemie. W przypadku C++ alokacje można podzielić na automatyczne, dynamiczne oraz statyczne.

- Do **alokacji automatycznych** zazwyczaj służy stos lub część rejestrów procesora. Alokacje automatyczne wykonywane są w trakcie wykonywania programu podczas deklaracji zmiennych. Dealokacje wykonywane są automatycznie.
- Do **alokacji dynamicznych** oraz dealokacji dynamicznych na stercie służą zazwyczaj funkcje biblioteki standardowej danej platformy lub operatory new oraz delete. Alokacje oraz dealokacje wykonane są w czasie trwania programu poprzez jawne ich wywołanie. Niezwolnienie wcześniej zarezerwowanej pamięci powoduje tak zwany wyciek pamięci.
- **Alokacja statyczna** wykonywana jest w czasie kompilacji w specjalnym bloku pamięci statycznej i jest dostępna od początku do końca działania programu. Rozmiar jej musi być znany na etapie kompilacji programu.

W dalszej części tekstu skupimy się na alokacjach automatycznych i dynamicznych.

5.2.1 Sterta vs Stos

W trakcie trwania programu język C++ umożliwia alokowanie pamięci w dwojaki sposób: za pomocą stosu (alokacje automatyczne) oraz sterty (alokacje dynamiczne).

Stos

Stos (ang. *stack*) to liniowa struktura danych LIFO (ang. *last in, first out*). Jej główną cechą jest kolejność dodawania oraz odejmowania danych. Ostatnie

dane dodawane na stos muszą zostać z niego ściągnięte jako pierwsze. Każdy wątek posiada zarezerwowany ciągły obszar pamięci na stos. Zasadę działania stosu najłatwiej jest zilustrować na przykładzie stosu książek poukładanych jedna na drugiej. Dostęp do książek możliwy jest w kolejności odwrotnej do kolejności położenia. Książka położona na stosie jako ostatnia jest dostępna jako pierwsza. Aby dostać się do kolejnej książki, należy najpierw zdjąć tę ze szczytu stosu.

Stos zazwyczaj służy do przechowywania wielu danych, takich jak:

- adres powrotu,
- parametry przekazane do funkcji czy metody,
- wskaźnik aktualnej instancji (wskaźnik this),
- zmienne lokalne (automatyczne).

Zmienne automatyczne odkładane na stosie mają bardzo wiele zalet. Najważniejszą ich cechą jest automatyczna alokacja i dealokacja, gdy przebieg programu wchodzi i wychodzi z zakresu, w którym są zadeklarowane. Dzięki temu używanie ich jest bardzo proste - nie powodują wycieków pamięci. Alokacja zmiennych automatycznych jest bardzo szybka, zdecydowanie szybsza niż alokacja na stercie, ponieważ nie rezerwuje ona rzeczywiście nowej pamięci, a wykorzystuje pamięć zarezerwowaną przez system operacyjny na stos. Również dzięki relatywnemu położeniu zmiennych względem stosu, dostęp do zmiennych lokalnych jest szybszy. Żeby było jeszcze szybciej, nowoczesne kompilatory część zmiennych automatycznych umieszczają w rejestrach procesora, do których dostęp jest natychmiastowy.

Z drugiej strony zmienne automatyczne mają wiele ograniczeń. Alokacja zmiennych automatycznych następuje podczas ich deklaracji, dlatego wielkości zmiennych muszą być znane na etapie kompilacji programu. Kolejną ich wadą jest ograniczony rozmiar stosu, w związku z czym, nie nadają się one do alokacji dużych bloków pamięci. Alokacja dużych zmiennych może bowiem spowodować przepełnienie stosu (ang. *stack overflow*). Również największa zaleta zmiennych lokalnych, czyli automatyczna dealokacja związana jest z ogromnym ograniczeniem. Czas życia zmiennej ograniczony jest do czasu trwania zakresu. Gdy przebieg programu wyjdzie poza zakres, następuje automatyczna dealokacja, w związku czym bardzo utrudnione jest elastyczne zarządzanie pamięcią.

Sterta

Sterta (ang. *heap*) to obszar pamięci dostępnej w systemie dla alokacji dynamicznych. Sterta jest nazwą zwyczajową, która wzięła się od tego, że implementacja przydziału pamięci dla takich alokacji bardzo często oparta jest na drzewiastej strukturze - stercie, często zwanej także kopcem. Z punktu

widzenia programisty rzadko potrzebna jest wiedza, jaki wykorzystano algorytm alokacji dynamicznych. Ważne natomiast jest to, że zmienne tworzone na starcie są dynamicznie, czyli dopiero w czasie trwania programu musi być znana ich wielkość. Pamięć zarezerwowana jest aż do jawnego jej zwolnienia i jest całkowicie niezależna od aktualnego zakresu programu. Sterta jest wielokrotnie większa od stosu i doskonale nadaje się do alokacji dużych bloków pamięci. Jej wielkość ograniczona jest przez system operacyjny. Elastyczność alokacji na stercie okupiona jest niestety wadami. Niezwolnienie wcześniej zarezerwowanej pamięci powoduje wyciek. Alokacje są wolne i defragmentują pamięć, dlatego ważne jest optymalne zarządzanie alokacjami oraz dealokacjami, ponieważ po jakimś czasie działania programu pamięć może być tak zdefragmentowana, że uniemożliwi to wykonanie kolejnych alokacji. Używanie zmiennych dynamicznych jest znacznie trudniejsze od używania zmiennych automatycznych.

Podsumowując, języki programowania udostępniają alokacje automatyczne oraz dynamiczne, ponieważ oba sposoby alokacji są potrzebne. Alokacje dynamiczne są niezbędne, pomimo że jest możliwe napisanie prostego programu bez ich używania. Implementacja bardziej złożonego programu bez możliwości alokacji na stercie jest właściwie niemożliwa. Elastyczność, z jaką związane są alokacje dynamiczne powoduje, że ich używanie jest znacznie bardziej skomplikowane, niż zmiennych automatycznych. Alokowanie małych danych w ten sposób jest nieoptymalne, gdyż jest wolne i powoduje defragmentację pamięci. W przypadku zmiennych o niewielkich rozmiarach służących do przechowywania tymczasowych danych zdecydowanie lepiej jest użyć alokacji automatycznych. Rozwiązanie takie jest szybsze, nie powoduje defragmentacji pamięci jest też zdecydowanie łatwiejsze i mniej podatne na błędy, co ma ogromne znaczenie podczas testowania oraz uruchamiania programu.

5.2.2 Alokator małych obiektów

Częste alokacje i dealokacje małych obszarów pamięci powodują bardzo szybką defragmentację pamięci. Ponadto alokacja bloku pamięci trwa mniej więcej tyle samo czasu niezależnie od jego wielkości, dlatego też rezerwacja wielu małych bloków pamięci jest znacznie dłuższa niż jednego o rozmiarze sumarycznym wszystkich małych bloków. Z uwagi na powyższe, warto rozpatrzyć specjalną strategię alokacji małych bloków pamięci.

Do tego celu idealnie sprawdzają się tak zwane pule pamięci (ang. *Memory Pool*). Idea ich działania opiera się na wstępnej alokacji dużych fragmentów pamięci, z kolei fragmenty te dzielone są na bloki o zadanym rozmiarze. Każda alokacja pamięci w puli rezerwuje blok o tym samym rozmiarze. Dzięki temu nie występuje w ogóle zjawisko defragmentacji. Po każdym zwolnieniu takiego bloku jest on ponownie gotowy do alokacji. Implementacje pul pamięci mogą być bardzo różne. W najprostszym przypadku może to być jedna pula, która np. służy do alokacji obiektów mniejszych niż 64 bajty, a wszystkie rezerwacje

traktuje tak, jakby miały one rozmiar 64 bajtów. Bardziej złożona implementacja może alokować pule o różnych rozmiarach bloków przeznaczone dla wielu małych alokacji. Gdy nie ma miejsca we wszystkich pulach, alokowana jest kolejna. Implementacji puli pamięci może być bardzo wiele, a co najważniejsze powinny być dopasowane do konkretnego projektu. Każdy program ma specyficzne dla siebie zapotrzebowanie na pamięć.

Korzystanie z pul pamięci ma bardzo wiele zalet:

- Unikamy defragmentacji pamięci;
- Może znacząco skrócić czas alokacji programu.

Korzystanie z pul ma również wady:

- Pule pamięci nie są uniwersalne i powinny zostać dostosowane do konkretnego zastosowania;
- Alokowane są nadmiarowe obszary pamięci.

5.2.3 Pamięci o różnym czasie dostępu

Niektóre platformy mają pamięć rozbitą na kilka banków. Często jest tak, że czas dostępu do poszczególnych banków jest różny. Alokując pamięć, warto się zastanowić, który blok wykorzystać do danego celu. Dane, które są odczytywane lub modyfikowane bardzo często powinny znaleźć się w bloku o krótkim czasie dostępu, z kolei te rzadziej używane mogą być w blokach wolniejszych.

5.2.4 Implementacja alokatorów

Alokator pamięci odpowiedzialny jest za rezerwowanie nowych bloków pamięci oraz zwalnianie tych wskazanych przez użytkownika. Jego interfejs może wyglądać następująco:

```
class IMemoryAllocator
{
public:
    virtual void *Alloc(size_t size) = 0;
    virtual void Free(void* pPtr) = 0;
};
```

Listing 5.2-1 - Interfejs alokatora pamięci

Wszystkie klasy alokatorów powinny dziedziczyć po powyższym interfejsie. Metoda `Alloc` powinna zwrócić wskaźnik do zarezerwowanego obszaru pamięci o rozmiarze podanym jako argument. Metoda `Free` powinna zwolnić wcześniej zarezerwowany blok pamięci spod adresu podanego jako argument.

By możliwy był dostęp do alokatora z każdego miejsca programu, może on korzystać z wzorca projektowego singletona. Singleton jest wzorcem, którego zadaniem jest ograniczenie możliwości tworzenia wielu instancji danej klasy. Ponadto zapewnia on globalny dostęp do stworzonego obiektu. Implementacja singletona opiera się na klasie posiadającej statyczne pole będące wskaźnikiem na instancję klasy, której liczba obiektów ma zostać ograniczona. Klasa ta posiada ponadto statyczną metodę `GetInstance`, która w pierwszej kolejności sprawdza czy instancja klasy już istnieje, ewentualnie tworzy obiekt, a ostatecznie zwraca wskaźnik do obiektu. Aby klasa singletona była wygodna w użyciu i uniwersalna, a więc niezależna od typu klasy, warto oprzeć jej implementację na szablonie.

```
template<class T> class TSingleton
{
public:
    static T *&GetInstance()
    {
       if (m_pInstance == NULL)
             m_pInstance = new T();
       return (*m_pInstance);
    }

private:
    static T *m_pInstance;
};
```

Listing 5.2-2 - Wzorzec klasy singletonu

Dzięki wykorzystaniu wzorca projektowego singletona istnieje tylko jedna instancja danego alokatora, do której można się bez problemu dostać. Innym dobrym rozwiązaniem byłoby stworzenie menadżera pamięci, który przechowywałby wszystkie dostępne w systemie alokatory.

Prosty alokator korzystający ze standardowych funkcji `malloc` oraz `free` wygląda następująco:

```
class CStandardAllocator :
    public IMemoryAllocator,
    public TSingleton<CStandardAllocator>
{
public:
    virtual void *Alloc(size_t size){return malloc(size);}
    virtual void Free(void* pPtr){free(pPtr);}
};
```

Listing 5.2-3 - Klasa prostego alokatora

W analogiczny sposób można stworzyć alokatory korzystające z puli pamięci. W tym przypadku implementacja alokacji oraz dealokacji nie będzie jednak tak trywialna.

```
class CMemPoolAllocator :
    public IMemoryAllocator,
    public TSingleton<CMemPoolAllocator>
{

public:
    virtual void *Alloc(size_t size);
    virtual void Free(void* pPtr);

};
```

Listing 5.2-4 - Klasa alokatora korzystającego z puli pamięci

Aby możliwe było używanie własnych alokatorów podczas tworzenia obiektów, należy przeciążyć operatory `new` i `delete`. Alokatory służą tylko do rezerwowania bloków pamięci. Operator `new` podczas tworzenia obiektu odpowiedzialny jest dodatkowo za wywołanie odpowiedniego konstruktora, a podczas zwalniania bloku pamięci po obiekcie, za wywołanie destruktora. Operatory `new` oraz `delete` należy przeciążyć w taki sposób, aby wywoływały specyficzne alokatory podczas tworzenia nowych obiektów, dzięki czemu użycie alokatorów jest wtedy automatyczne.

Przeciążyć operatory można w dwojaki sposób - dla konkretnych klas lub globalnie.

Przeciążanie operatorów new oraz delete dla konkretnych klas

```
class A { public:
    A(){};
    virtual ~A(){};
    static void* operator new (size_t size)
    {
        return CMemPoolAllocator::GetInstance().Alloc(size);
    }

    static void* operator new [](size_t size)
    {
        return CMemPoolAllocator::GetInstance().Alloc(size);
    }
```

```
    static void operator delete (void *pPtr)
    {
        return CMemPoolAllocator::GetInstance().Free(pPtr);
    }

    static void operator delete[] (void *pPtr)
    {
        return CMemPoolAllocator::GetInstance().Free(pPtr);
    }
};
```

Listing 5.2-4 - Przeciążenie operatorów new oraz delete dla konkretnych klas

Jak widać w powyższym przykładzie klasa A ma przeciążone operatory `new` oraz `delete`, które korzystają z alokatora dla małych obiektów. Utworzenie obiektu klasy A za pomocą operatora `new` wywoła wpierw przeciążony operator `new`, który dokona alokacji pamięci z pomocą `CMemPoolAllocator`. Następnie zostanie wywołany konstruktor. Po usunięciu obiektu klasy A za pomocą operatora `delete` zostanie wywołany destruktor, a następnie przeciążony operator `delete`, który zwolni pamięć za pomocą alokatora `CmemPoolAllocator`. Dodatkowo zostały przeciążone operatory tablicowe uruchamiane podczas tworzenia oraz usuwania tablic obiektów.

Operatory `new` oraz `delete` mogą być definiowane ponownie dla klas dziedziczących po klasach, które już je sobie zdefiniowały. Wtedy nowo zdefiniowane operatory przesłaniają operatory odziedziczone.

```
class B : public A
{
public:
    B(){};
    virtual ~B(){};
    static void* operator new (size_t size)
    {
        return CStandardAllocator::GetInstance().Alloc(size);
    }
    static void* operator new [](size_t size)
    {
        return CStandardAllocator::GetInstance().Alloc(size);
    }
    static void operator delete (void *pPtr)
    {
        return CStandardAllocator::GetInstance().Free(pPtr);
    }
```

```
    static void operator delete[] (void *pPtr)
    {
        return CStandardAllocator::GetInstance().Free(pPtr);
    }
};

class C : public A
{
public:
    C(){};
    virtual ~C(){};
};
```

Listing 5.2-5 - Przeciążenie operatorów new oraz delete dla klas dziedziczących

Stworzenie obiektu klasy A lub klasy C przydzieli pamięć za pomocą alokatora `CMemPoolAllocator`, natomiast stworzenie obiektu klasy B skorzysta z alokatora standardowego `CStandardAllocator`.

```
A* pA = new B();
delete pA;
```

Listing 5.2-6 - Tworzenie i usuwanie obiektu

W przykładzie powyżej, wywołany zostanie najpierw operator `new`, który przydzieli pamięć za pomocą `CStandardAllocator`, po czym uruchomi konstruktory najpierw klasy A, potem klasy B. Dzięki użyciu destruktora wirtualnego najpierw uruchomi się destruktor klasy B, a następnie klasy A, by ostatecznie zwolnić pamięć zajmowaną przez obiekt za pomocą metody `Free` z klasy `CStandardAllocator`.

Przeciążanie operatorów new oraz delete globalnie

```
void* operator new (size_t size)
{
    return CStandardAllocator::GetInstance().Alloc(size);
}
void  operator delete (void *pPtr)
{
    return CStandardAllocator::GetInstance().Free(pPtr);
}
```

```
void* operator new[] (size_t size)
{
    return CStandardAllocator::GetInstance().Alloc(size);
}
void  operator delete[] (void *pPtr)
{
    return CStandardAllocator::GetInstance().Free(pPtr);
}
```

Listing 5.2-7 - Globalnie przeciążone operatory new i delete

Globalnie przeciążone operatory `new` oraz `delete` pozwalają używać zdefiniowanego alokatora dla wszystkich obiektów, których klasy nie przeciążają ich lokalnie. Bardzo ważne jest, aby deklaracje przeciążonych operatorów były znane przed ich użyciem. To oznacza, że plik nagłówkowy, w którym zadeklarowane są operatory, musi być dołączony do pliku, w którym wykonywane są alokacje i dealokacje.

Warto wiedzieć, że operator new posiada bardzo ciekawą cechę - może być przeciążany w różny sposób. Standardowo - operator `new` przyjmuje jeden argument typu `size_t` określający rozmiar alokacji podany w bajtach. W zaawansowany sposób - operator `new` można przeciążyć z dodatkowymi argumentami. Pierwszy argument musi być taki jak w wersji standardowej, zaś dodatkowe argumenty mogą wystąpić w dowolnej liczbie i dowolnym typie. Cechę tę można wykorzystywać na wiele sposobów, tworząc różne definicje `new`. Operator `new` z rozbudowaną listą argumentów w literaturze bardzo często nazywany jest „placement new", ponieważ najczęściej wykorzystuje się go do tworzenia obiektów w pewnym określonym miejscu pamięci. Najprostsza implementacja wygląda następująco:

```
void* operator new (size_t size,void* ptr)
{
    return ptr;
}
```

Listing 5.2-8 - Dwuargumentowe przeciążenie operatora new

Działanie powyższego operatora może wydawać się bezsensowne, lecz jak się wkrótce okaże, ma ono swoje zastosowanie. W przypadku, gdy mamy już zarezerwowaną pamięć i chcemy ją koniecznie wykorzystać na konkretne obiekty, operatory tego typu nadają się idealnie. Na przykład, jeśli chcemy mieć pewność, że kolejno tworzone i zwalniane obiekty będą się znajdować dokładnie w tym samym miejscu pamięci. Użycie takiego opratora `new` pokazuje poniższy przykład.

```
class CBaseWorker
{
//...
public:
    void* operator new (size_t size,void* ptr)
    {
        return ptr;
    }
    void operator delete (void* ptr){}
    virtual ~CBaseWorker(){};
    virtual void DoHeavyJob() = 0;
};

class CWorker1 : public CBaseWorker
{
    float m_afBigArray[1024 * 1024];
public:
    CWorker1(float fStart){};
    virtual void DoHeavyJob() {};
};

class CWorker2 : public CBaseWorker
{
    int m_aiHugeArray[1024 * 1024 * 1024];
public:
    virtual void DoHeavyJob() {};
};
```

Listing 5.2-9 - Klasy wykorzystujące nietypowy operator new

Klasy `CWorker1` oraz `CWorker2` rozszerzają klasę `CBaseWorker`, implementując wirtualną metodę `DoHeavyJob`.

Przyjrzyjmy się poniższemu przykładowi:

```
size_t size = (sizeof(A) > sizeof(B) ? sizeof(A) : sizeof(B));
CBaseWorker *pBaseWorker =
    (CBaseWorker*) CFastMemoryAlocator::GetInstance().Alloc(size);

A* pA = new (CBaseWorker)CWorker1();
pBaseWorker-> DoHeavyJob();

delete pA;
B* pB = new (CBaseWorker)CWorker2();
pBaseWorker->
DoHeavyJob();
```

Listing 5.2-10 - Wykorzystanie nietypowego operatora new

W związku z faktem, że operator `new` zdefiniowany w tym przykładzie nie rezerwuje w ogóle pamięci, to na początku wykonywana jest alokacja ręczna za pomocą wybranego alokatora. Rozmiar alokowanego bloku musi być przynajmniej równy rozmiarowi większego obiektu.

Po utworzeniu obiektu klasy `CWorker1` przez operator `new` możliwe jest wywołanie metody na wskaźniku `pBaseWorker`, gdyż obiekt ten został utworzony dokładnie w miejscu wskazywanym przez ten wskaźnik. Wywołana zostanie metoda zdefiniowana w klasie `CWorker1`.

W momencie usunięcia obiektu klasy `CWorker1` w jego miejsce tworzony jest obiekt klasy `CWorker2`. Wywołanie metody `DoHeavyJob()` na wskaźniku `pBaseWorker` jest ciągle poprawne, ponieważ adres obiektu pB jest taki sam jak adres `pA`. Oczywiście w tym przypadku zostanie wywołana metoda z klasy `CWorker2`.

5.2.5 Alokatory a wielowątkowość

Jeżeli implementowane alokatory mają pracować w środowisku wielowątkowym metody Alloc oraz Free, to muszą być atomowe. A to oznacza, że funkcje te muszą zawsze wykonać się w całości, nie będąc przerwane przez inny wątek. W tym celu najlepiej jest zastosować Mutex. Mutex powinien zostać zablokowany już na samym początku metod Alloc oraz Free, a zwolniony dopiero na ich końcu.

5.2.6 Podsumowanie alokatorów

Podsumowując, stosowanie własnych alokatorów ma bardzo wiele zalet:

- Może bardzo mocno zoptymalizować działanie programu. Standardowe metody do alokacji pamięci (`malloc` oraz `free`) są bardzo powolne, więc bardzo ważne jest, by wywoływać je najrzadziej, jak tylko to możliwe. Szczególnie ważne jest to w przypadku małych obiektów, które często muszą być tworzone oraz zwalniane. Stosowanie pul pamięci pozwala w dużej mierze rozwiązać ten problem.
- Alokatory dają ogromne możliwości w zarządzaniu przydziałem pamięci. Jeżeli system wyposażony jest w bloki pamięci o różnym czasie dostępu, za pomocą własnych alokatorów, można przydzielać szybszą pamięć dla obiektów, które wykonują krytyczne operacje. Poza tym można za ich pomocą podzielić pamięć na logiczne bloki przeznaczone na obiekty różnego typu. Na przykład wszystkie obiekty tekstur mogą być alokowane w konkretnym miejscu w pamięci.
- Alokatory mogą również posłużyć jako narzędzie diagnostyczne, służące do zbierania statystyk wykorzystania pamięci przez program.

5.3 Menadżer zasobów

5.3.1 Idea menadżera zasobów

Bardzo ważnym elementem silnika jest system zarządzania zasobami. Zasobami w silnikach gier są dane gry, takie jak obiekty graficzne, dźwięki czy też poziomy. Cechą charakterystyczną zasobów jest ich sumarycznie duży rozmiar. Ze względu na rozmiary swoich zasobów zajmują dużo miejsca w pamięci, a ich przetwarzanie jest czasochłonne. Głównym celem menadżera zasobów jest wydajne zarządzanie zasobami oraz umożliwianie efektywnego współdzielenia ich przez różne obiekty.

Przykładami zasobów, które idealnie nadają się do współdzielenia przez różne obiekty gry, są:

- *Geometrie modeli trójwymiarowych* - bardzo dużo obiektów graficznych w grach może być opartych na wspólnej geometrii. Mogą to być modele postaci przeciwników w grach typu FPS lub modele samochodów w grach wyścigowych.
- *Animacje modeli trójwymiarowych* - tak jak w przypadku geometrii, opis animacji modeli może być współdzielony przez wiele obiektów, które mają się poruszać w identyczny sposób.
- *Tekstury* - ze względu na duże rozmiary w pamięci, zysk przy współdzieleniu tekstur jest ogromny. Czasem wystarczy nawet niewielka ilość tekstur, żeby pokryć w sensowny sposób elementy sceny - szczególnie, gdy tych samych tekstur używamy w połączeniu z różnymi shaderami.
- *Shadery* - współdzielenie shaderów z bazy typowych, często używanych shaderów jest wygodne, a gdy używa się ich z różnymi parametrami, ich powtarzalność nie jest widoczna,
- *Dźwięki* - współdzielenie dźwięków, szczególnie krótkich i generycznych, pozwala uzyskać niezły efekt, którego powtarzalność może być prawie niezauważalna dla gracza, a umożliwia oszczędzenie pamięci.
- *Efekty cząsteczkowe* - skomplikowane efekty cząsteczkowe mogą zajmować dość sporo pamięci, a bez problemu mogą być wykorzystane w wielu miejscach przez różne obiekty.
- *Czcionki* - współdzielenie czcionek przez obiekty tekstowe jest bardzo dobrym pomysłem. Duże czcionki, zawierające znaki specjalne dla wszystkich języków obsługiwanych w grze mogą zajmować bardzo dużo pamięci.

Pierwszym etapem życia zasobu jest jego stworzenie, czyli wczytanie z dysku lub z serwera, bądź stworzenie za pomocą programu. Zasób jednego typu może mieć bardzo wiele różnych źródeł, z których może zostać stworzony. Warto to uwzględnić, projektując menadżer zasobów. Dobrym przykładem tego jest tekstura. Tekstury najczęściej tworzone są z plików w różnych formatach graficz-

nych, przechowywanych na dysku lub serwerze. Tekstury mogą być tworzone również funkcyjnie. Sam proces tworzenia zasobu tekstury może trwać długo, dlatego powtarzanie procesu tworzenia tego samego zasobu powinno być ograniczone do minimum. Z drugiej strony, może być niemożliwe stworzenie wszystkich zasobów na początku programu i trzymanie ich w pamięci do jego końca ze względu na ograniczoną ilość pamięci.

Zadaniem menadżera zasobów jest ich utworzenie na żądanie i udostępnianie każdemu obiektowi, który o nie poprosi. Zasób po stworzeniu jest cache'owany, czyli przechowywany w pamięci tak długo, jak jest to możliwe. Zasób może być usunięty na żądanie lub wtedy, gdy od dłuższego czasu nie jest potrzebny, a brakuje miejsca na inne zasoby. Menadżer powinien zwalniać mniej potrzebne zasoby gdy zaczyna brakować pamięci na tworzenie nowych, potrzebnych w danej chwili przez obiekty gry.

5.3.2 Uchwyty, klucze

Najprostszym rozwiązaniem wydaje się być umożliwienie dostępu do zasobu poprzez referencje cz też wskaźnik do niego. Niestety, nie jest to bezpieczne, gdyż istnieje ryzyko, że staną się one nieważne. Taka sytuacja może wystąpić np. wówczas, gdy zasób został już usunięty, a potem ponownie stworzony. Nie jest możliwa wewnętrzna reorganizacja pamięci menadżera zasobów, bowiem przeniesienie danych powoduje bałagan w referencjach lub wskaźnikach. Rozwiązaniem tego problemu jest dodanie warstwy abstrakcji w dostępie do zasobu, by nie używać referencji lub wskaźników, zaś cały ciężar zarządzania referencjami przenieść na menadżer zasobów.

Cel ten można zrealizować na wiele sposobów. Bardzo popularne jest odwoływanie się do zasobów za pomocą klucza tekstowego - czyli inaczej mówiąc za pomocą jego nazwy. Nazwa zasobu przechowywana jest jako łańcuch znaków zawierający ścieżkę do pliku, z którego zasób jest stworzony.

Aby dostać wskaźnik do zasobu, należy wywołać odpowiednią metodę z menadżera zasobów:

```
CResource* CresourceManager::GetResource(const char *pszResName);
```

Listing 5.3-1 - Pobieranie wskaźnika do zasobu

Nazwa musi się zgadzać oczywiście z tą podaną podczas tworzenia zasobu. Jeśli zasób nie istnieje, ponieważ nie został stworzony lub został już zwolniony, bardzo łatwo można ten błąd wychwycić i w odpowiedni sposób zareagować, na przykład wczytując zasób z pliku.

Wadą kluczy tekstowych jest ich rozmiar. Za każdym razem, gdy dokonywana jest translacja z nazwy na zasób, konieczne jest przeanalizowanie całego

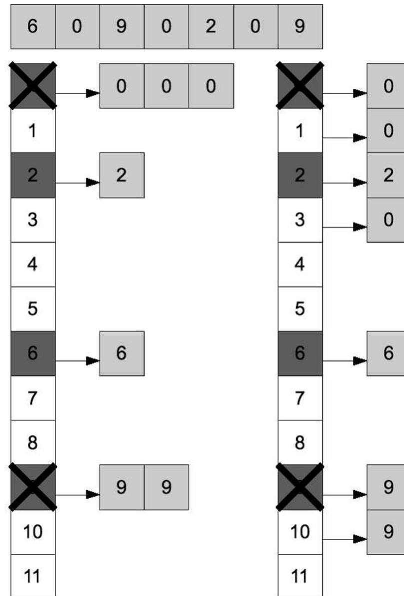
ciągu znaków. Dodatkowo kopiowanie łańcuchów znaków jest uciążliwe zarówno jeżeli chodzi o pamięć, jak i o czas wykonania.

Tablice haszujące

Do szybkiej translacji łańcucha znakowego na wskaźnik bardzo często stosuje się tablice haszujące. Tablica haszująca jest to struktura danych służąca do przechowywania danych w taki sposób, aby możliwy był do nich szybki dostęp przy użyciu kluczy. Tablice haszujące bardzo często opiera się na zwykłych tablicach indeksowanych liczbami. Każdą daną w tablicy identyfikuje klucz, który może być na przykład ciągiem znaków. Kluczem może być również każdy inny obiekt pod warunkiem, że na jego podstawie możliwe jest wyznaczenie indeksu. Do wyznaczania indeksu w tablicy służy funkcja mieszająca. Oczywiście jest ona specyficzna dla danego typu klucza. Ważne jest, aby dla takiego samego klucza zawsze wygenerowała dokładnie taki sam indeks. Funkcja mieszająca powinna być prosta, aby czas liczenia indeksu nie zdominował czasu wyszukiwania obiektu, jednak powinna dla różnych kluczy generować maksymalnie różne indeksy. Oczywiście nie zawsze jest możliwe, aby dla każdego kolejnego klucza wygenerować unikalny indeks z zakresu od zera do rozmiaru tablicy. Gdy funkcja mieszająca wygeneruje indeks już zajęty przez inny klucz, występuje kolizja. Stosunek liczby kolizji do liczby wszystkich wygenerowanych indeksów informuje o skuteczności funkcji haszującej. Aby stworzyć doskonałą funkcję haszującą, czyli taką, która nigdy nie spowoduje kolizji, konieczna jest znajomość wszystkich możliwych kluczy. Oczywiście stopień komplikacji takiej funkcji jest odwrotnie proporcjonalny do nadmiarowości rozmiaru tablicy. Przeważnie czas wykonania funkcji mieszającej jest ważniejszy, niż jej wysoka skuteczność. Kolizje można rozwiązać na wiele sposobów. Najczęściej stosowane to:

- *Tablica list* - sposób ten polega na przechowywaniu elementów nie bezpośrednio w tablicy, ale w listach (lub wektorach), które są zawarte w tablicy. Gdy występuje kolizja, lista pod danym indeksem wydłuża się.
- *Przyrost indeksu* - gdy następuje kolizja, element wstawia się w pod innym indeksem, wyznaczonym za pomocą funkcji przyrostu indeksu. Parametrem funkcji jest indeks wygenerowany przez funkcję mieszającą. Często stosuje się funkcję liniową lub kwadratową.

Na poniższej ilustracji zaprezentowane są dwie wyżej omówione implementacje tablicy haszującej. Po lewej stronie widoczna jest tablica list, natomiast po prawej stronie znajduje się tablica, która radzi sobie z kolizjami za pomocą przyrostu indeksu o jeden. Ilustracja prezentuje jak wyglądają obie tablice po wstawieniu kolejno siedmiu elementów, których wartości indeksów wyliczone przez funkcje haszującą wynoszą odpowiednio 6, 0, 9, 0, 2, 9. W przypadku tych elementów pojawiają się trzy elementy, które mają wartość 0 i dwa, które mają wartość dziewięć. Indeksy, dla których występuje kolizja, zaznaczono



Rysunek 5.1. Rozwiązywanie kolizji w tablicy mieszającej

krzyżykiem. Dla elementów o indeksie 0 kolizja wystąpiła dwa razy, a dla elementów o indeksie równym 9 jeden raz. Listy wydłużają się na skutek kolizji. Dlatego w tablicy pod indeksem 0 znajduje się trójelementowa lista, a pod indeksem 9 lista o dwóch elementach. Dla indeksów 2 i 6 listy zawierają tylko po jednym elemencie. Pozostałe listy są puste. Druga implementacja reaguje na kolizje przez znalezienie pierwszego wolnego miejsca w tablicy dla kolidującego elementu. Pierwsza kolizja występuje dla elementu o indeksie zero. Kolejne miejsce w tablicy jest wolne więc element zostaje umieszczony pod indeksem o wartości 1. W przypadku następnej kolizji dla elementu o indeksie równym 0 algorytm poszukuje miejsca w tablicy dłużej, ponieważ miejsca pod indeksami 1 i 2 są już zajęte przez inne elementy. Pierwsze, wolne miejsce znajduje się pod indeksem numer 3.

Obydwa rozwiązania mają swoje wady i zalety. Pierwsze rozwiązanie może być wolniejsze podczas dodawania obiektów, gdyż w przypadku konfliktu konieczna może być realokacja pamięci. Będzie ono za to na ogół szybsze podczas usuwania oraz znajdowania obiektów, gdyż na skutek kolizji nie jest potrzebne wywoływanie funkcji przyrostu. Za drugim rozwiązaniem może przemawiać prostota implementacji, jednak wadą jest to, że tablica musi lepiej mieć dobrany rozmiar początkowy.

Dobór rozmiaru tablicy haszującej jest tematem wielu opracowań. Istnieje rozmiar idealny - taki, że dla każdego możliwego klucza istnieje pole w tablicy, jednak zajmuje on zbyt wiele miejsca w pamięci. Zakładając, że klucz tablicy

będzie 4 bajtowym ciągiem znaków (czyli bardzo krótki - zazwyczaj stosuje się klucze o wiele dłuższe), tablica musiałaby mieć rozmiar 4 gigabajtów:

$$n = \sum_{i=1}^{4} 256^i = 4311810304$$

Przyjęło się, że rozmiar tablicy powinien być większy od maksymalnej liczby elementów o jakieś 25-30% oraz powinien być liczbą pierwszą. Kolejnym problemem jest dobór funkcji mieszającej. Optymalna funkcja mieszająca dla każdego kolejnego klucza powinna wygenerować inny indeks - wtedy nigdy nie wystąpiłyby konflikty. W praktyce funkcję zazwyczaj wybiera się empirycznie dla konkretnych danych, aby konflikty występowały możliwie rzadko.

Implementacja nieskomplikowanej tablicy haszującej jest relatywnie prosta. Podstawą jest interfejs klucza.

```
class IHashTableKey
{
public:
    virtual ~IHashTableKey(){};
    //! Returns key hash code.
    virtual unsigned int GetHashCode() = 0;
    //! Checks if other key is equal.
    virtual bool IsEqual(const IHashTableKey\& other) = 0;
};
```

Listing 5.3-2 - Interfejs klucza tablicy haszującej

Interfejs dla kluczy składa się z dwóch metod. Pierwsza z nich, `GetHashCode`, zwraca wartość policzoną poprzez funkcję mieszającą. Druga, `IsEqual`, służy do porównywania dwóch kluczy. Zwraca `true`, gdy klucze są takie same. Metoda ta jest bardzo ważna. Wykorzystywana jest do sprawdzania, czy tablica zawiera już obiekt o takim kluczu. Jest ona wywoływana podczas wstawiania, wydobywania obiektu z tablicy oraz podczas jego usuwania.

Klasa klucza zbudowanego z łańcucha znaków wygląda następująco:

```
class CStringKey : public IHashTableKey
{
private:
    CString m_strKey;
    unsigned int m_iHashCode;
    void CalculateHashCode(unsigned int iHashTableSize);
```

```
public:
    //! Constructs and initialize key.
    CStringKey(const char *pszKey,unsigned int iHashTableSize)
        : m_strKey(pszKey)
    {
     CalculateHashCode(iHashTableSize);
    }
    //! Returns key hash code.
    virtual unsigned int GetHashCode(){return m_iHashCode;}
    //! Checks if other key is equal.
    virtual bool IsEqual(const IHashTableKey& other)
    {
    returnm_strKey.IsEqual(((const CStringKey &)other).m_strKey);
    }
};
```

Listing 5.3-3 - Klasa klucza zbudowanego z łańcucha znaków

Łańcuch znaków w obiekcie klasy `CString` zapisywany jest w konstruktorze. Liczony jest też indeks za pomocą prywatnej funkcji mieszającej `CalculateHashCode`. Metoda `GetHashCode` zwraca wcześniej policzony indeks. Metoda `IsEqual` zwraca prawdę, gdy klucze zawierają taki sam łańcuch znakowy.

Przykładowa funkcja haszująca może wyglądać następująco:

```
CStringKey::CalculateHashCode(iHashTableSize)
{
    unsigned int iResult;
    unsigned int iLen = m_strKey.GetLenth();
    for (int i = 0; i < iLen; i++)
    {
        char cLetter =  strKey[i];
        iResult += (unsigned int)cLetter * Pow(31,(iLen-i));
    }
    m_iHashCode = iResult % iHashTableSize;
}
```

Listing 5.3-4 - Metoda implementująca funkcję haszującą

Uchwyty

Inny sposobem odizolowania się od wskaźników lub referencji są uchwyty. Uchwyt jest to obiekt przechowujący informacje potrzebne do wydobycia zasobu z menadżera. Obiekt ten pomimo tego, że jest bardzo mały, jest wystarczający do jednoznacznego zidentyfikowania zasobu. Jego zaletą jest rozmiar

- dzięki czemu łatwo można go kopiować oraz w bardzo szybki sposób identyfikować powiązany z nim zasób.

```
class CHandle
{
private:
    int m_iIndex;
    int m_iMagicNumber;

public:
    CHandle(const CHandle& other)
        : m_iIndex(other.m_iIndex),
          m_iMagicNumber(other.m_iMagicNumber)
    {}

    CHandle(int iIndex,int iMagicNumber)
        : m_iIndex(iIndex),
          m_iMagicNumber(iMagicNumber)
    {}

    bool operator ==(const CHandle& other)
    {
        return (m_Index == other.m_iIndex &&
            m_iMagicNumber == other.m_iMagicNumber);
    }

    CHandle& operator =(const CHandle& other)
    {
        m_iIndex = other.m_iIndex;
        m_iMagicNumber = other.m_iMagicNumber;
        return *this;
    }

};
```

Listing 5.3-5 - Klasa uchwytu

Uchwyt składa się z dwóch pól:

- Indeksu zasobu, który w jednoznaczny sposób pozwala menadżerowi zasobów określić lokalizacje zasobu w pamięci. Wartość indeksu określa indeks w tablicy lub na liście zasobów znajdującej się w menadżerze zasobów.
- Magicznej liczby, która pozwala w jednoznaczny sposób określić, czy uchwyt jest aktualny i czy dotyczy istniejącego w pamięci zasobu. Jest ona generowana przez menadżer zasobów podczas tworzenia nowego zasobu. Jej wartość może być wyznaczana właściwie w dowolny sposób, ważne jest,

aby była możliwie unikalna. Bardzo prostym, a jednocześnie skutecznym sposobem, jest przypisywanie magicznej liczbie wartości losowej. Istnieje znikome prawdopodobieństwo wylosowania dwukrotnie 32 bitowej liczby w czasie trwania programu.

Oprócz tego uchwyt składa się z konstruktora kopiującego oraz przeciążonego operatora przypisania, dzięki którym uchwyt w łatwy sposób się kopiuje. Za pomocą przeciążonego operatora równości można szybko porównać dwa uchwyty.

Zasoby tworzy się za pomocą metody:

```
CHandle& CresourceManager::CreateReasource(const char *pszResName);
```

Listing 5.3-6 - Tworzenie zasobów

Podczas tworzenia nowego zasobu tworzony jest nowy uchwyt, który przechowuje pierwszy wolny indeks w tablicy wskaźników na stworzenie zasobu i wylosowaną magiczną liczbę. Uchwyt zostaje zwrócony przez funkcję tworzącą zasób. Menadżer po utworzeniu zasobu zapamiętuje pod wybranym wcześniej indeksem w tablicy wskaźników wskaźnik do niego, a w tablicy magicznych liczb przypisaną mu magiczną liczbę.

Podczas pobierania zasobu z menadżera za pomocą metody:

```
CResource* CresourceManager::GetResource(const CHadle& handle);
```

Listing 5.3-7 - Pobieranie zasobu

Menadżer w pierwszej kolejności sprawdza, czy istnieje zasób pod indeksem przechowywanym w uchwycie, a następnie weryfikuje magiczne liczby. Jeżeli magiczna liczba z uchwytu rożni się od tej z menadżera, oznacza to, że uchwyt nie jest już ważny. Sytuacja taka może wystąpić wtedy, kiedy zasób związany z podanym uchwytem został już zwolniony, a w jego miejsce został utworzony nowy zasób.

Podsumowanie uchwytów i kluczy

Wydobywanie zasobów za pomocą uchwytów jest bardzo szybkie, gdyż sprowadza się do weryfikacji magicznej liczby oraz ewentualnego zwrócenia zasobu. Klucze są wolniejsze, ponieważ za każdym razem podany łańcuch znakowy musi zostać przeanalizowany w celu obsłużenia tablicy haszującej. Za to ich użycie wydaje się być bardziej intuicyjne, a kod bardziej czytelny.

Obu sposobów można używać przemiennie - starając się jednak w krytycznych momentach wykorzystywać uchwyty, ponieważ są szybsze. Dobrą praktyką jest stosowanie kluczy w funkcjach inicjalizacyjnych, aby pobrać uchwyt, który następnie wykorzystywany jest w funkcjach wykonywanych cyklicznie. Dzięki takiemu podejściu kod jest łatwy do interpretacji przez programistę i do tego wystarczająco szybki.

Zarówno klucze, jak i uchwyty mają następującą przewagę nad bezpośrednim dostępem przez wskaźnik czy referencje:

- Łatwo można sprawdzić ich poprawność.
- Dodatkowy poziom abstrakcji pozwala na reorganizacje danych w pamięci.
- Ułatwiają zapis oraz odczyt z dysku.

5.3.3 Zliczanie referencji

Aby możliwe było odpowiednie zarządzanie zwalnianiem zasobów z pamięci, menadżer zasobów musi wiedzieć, ile obiektów korzysta z danego zasobu. Dlatego każdy użytkownik zasobu powinien zwiększyć licznik referencji do tego zasobu przed użyciem, a gdy już jest pewien, że nie będzie z niego korzystać, zmniejszyć licznik. Menadżer zasobów, gdy potrzebuje zwolnić z pamięci nieużywane zasoby, sprawdza, który licznik referencji jest równy zero, co oznacza, że zasób jest nieużywany i istnieje możliwość jego usunięcia. Zasada działania jest bardzo prosta - w związku z tym implementacja też nie powinna być skomplikowana.

Najprościej jest udostępnić metody inkrementujące i dekrementujące licznik referencji danego zasobu w klasie menadżera zasobów. Metody takie mogłyby mieć następujące deklaracje:

```
void CresourceManager::IncrementReferenceCounter(const CHadle &
                                                          handle);
void CresourceManager::DecrementReferenceCounter(const CHadle &
                                                          handle);
```

Listing 5.3-8 - Metody obsługujące licznik referencji

Jest to oczywiście rozwiązanie poprawne, powinno też całkiem dobrze działać, lecz przy jednym założeniu: że każdy użytkownik zasobu będzie pamiętał o wywołaniu powyższych metod przed i po użyciu zasobu. Pomimo, że spełnienie tego założenia nie jest bardzo trudne, to wymaga od programistów korzystających z menadżera zasobów pewnej sumienności. Błąd spowodowany złym użyciem menadżera może być bardzo poważny w skutkach. Gdy licznik referencji ma wartość wyższą niż powinien - zasób nigdy nie zostanie zwolniony z pamięci, co ostatecznie może doprowadzić do braku pamięci. W przeciwnym

wypadku, gdy licznik referencji ma wartość niższą, zasób może zostać usunięty z pamięci, pomimo że będzie jeszcze wykorzystywany przez inne obiekty.

W związku z powyższym dobrze jest zastanowić się nad zautomatyzowaniem aktualizowania wartości licznika referencji. Do tego celu idealnie nadają się sprytne wskaźniki (ang. *smart pointers*). Sprytne wskaźniki to abstrakcyjne typy symulujące zwykły wskaźnik posiadający dodatkową funkcjonalność. W tym przypadku inteligentny wskaźnik zajmuje się zliczaniem referencji.

Aby zrealizować idee zliczania referencji, potrzebne są dwie klasy. Pierwsza z nich jest klasą bazową dla wszystkich zliczanych obiektów (w tym przypadku jest to klasa bazowa dla klasy zasobów) oraz dla klasy sprytnego wskaźnika.

```
class  CbaseObject
{
private:
    int m_iCounter;

public:
    void AddReference(){m_iCounter++;}
    void ReleaseRefernce(){m_iCounter--;}
    int GetRefernceCouter(){return m_iCounter;}
};
```

Listing 5.3-9 - Klasa bazowa licznika referencji

Klasa `BaseObject` zawiera licznik referencji oraz metody do jego zwiększania oraz zmniejszania.

```
template<class T>
class TreferencePointer
{
private:
    T* m_pObject;

public:
    TreferencePointer() :  m_pObject(NULL){}
    TreferencePointer(T *pObject) :  m_pObject(pObject)
    {
        m_pObject->AddReference();
    }
    TreferencePointer(const ReferencePointer<T>& pointer) :
    m_pObject(pointer.m_pObject)
    {
        m_pObject->AddReference();
    }
```

```
        ~TreferencePointer()
        {
            m_pObject->ReleaseReference();
        }
        operator =(T *pObject)
        {
            if (m_pObject) m_pObject->ReleaseReference();
            m_pObject = pObject;
            if (m_pObject) m_pObject->AddReference();
        }
        operator =(const TreferencePointer<T>& pointer)
        {
            if (m_pObject) m_pObject->ReleaseReference();
            m_pObject = pointer.m_pObject;
            if (m_pObject) m_pObject->AddReference();
        }
        T& operator *() const
        {
            return *m_pObject;
        }
        T* operator ->() const
        {
            return m_pObject;
        }
        operator T*() const
        {
            return m_pObject;
        }
    };
```

Listing 5.3-10 - Wzorzec klasy sprytnego wskaźnika

Klasa sprytnego wskaźnika odpowiedzialna jest za zwiększanie oraz zmniejszanie licznika referencji. Licznik jest zwiększany w konstruktorach z parametrami oraz w operatorach przypisania, a zmniejszany w destruktorze i w operatorze przypisania, gdy przypisany jest nowy wskaźnik. Dzięki przeciążeniu operatorów wyłuskania oraz rzutowania na typ `T`, obiekt sprytnego wskaźnika zachowuje się jak zwykły wskaźnik i tak też się go używa.

```
class CResource : public CBaseObject {
    //..
public:
    void ExampleMethod();
};
```

```
void main() {
    //..
    ReferencePointer<CResource> spResPointer1 =
            CresourceManager::GetResource(handle);
    spResPointer1->ExampleMethod();
    ReferencePointer<CResource> spResPointer2 =  NULL;
    spResPointer2 =  spResPointer1
    spResPointer2->ExampleMethod();

    //..
}
```

Listing 5.3-11 - Wykorzystanie sprytnych wskaźników

Na powyższym przykładzie widać, że sprytnego wskaźnika używa się tak, jak zwykłego. Wywołanie metody `ExampleMethod` jest identyczne jak przy użyciu zwykłego wskaźnika. To samo dotyczy też kopiowania wskaźnika. W powyższym przykładzie licznik referencji został powiększony dwukrotnie. Najpierw na skutek uruchomienia drugiego konstruktora klasy `ReferencePointer`, a następnie podczas wywołania operatora przypisania `spResPointer1` do `spResPointer2`. Gdy zakres widoczności zmiennych się kończy, wywołane są destruktory obu wskaźników, w których zmniejszony zostaje licznik referencji. Po wyjściu z zakresu licznik referencji zasobu wynosi tyle samo co przed wejściem.

5.3.4 Buforowanie i cache'owanie zasobów

Ze względu na długi czas tworzenia zasobu oraz jego duży rozmiar w pamięci, zasób powinien być tworzony oraz zwalniany z pamięci najrzadziej jak to możliwe. Nie można dopuścić do sytuacji, że zasób jest wczytywany, następnie zwalniany, a za moment znowu wczytywany i ponownie zwalniany. W takiej sytuacji program mógłby działać bardzo powoli. Jednym z powodów jest długi czas tworzenia zasobu, który składa się zazwyczaj na operacje alokacji dużych bloków pamięci oraz odczytów z pamięci masowej. Drugim powodem jest fakt, iż częste zwalnianie zasobu może powodować defragmentację pamięci operacyjnej, której skutkiem jest wolniejsze działanie programu. Bardzo ważne również jest kontrolowanie, czy konkretny zasób jest już w pamięci. Wielokrotne tworzenie oraz przechowywanie w pamięci tego samego zasobu jest oczywiście bez sensu.

Aby uniknąć powyższych błędów, menadżer zasobów w momencie tworzenia zasobu powinien sprawdzać, czy istnieje już zasób utworzony z podanego źródła. Jeśli tak, to nie powinien tworzyć go na nowo, lecz zwrócić uchwyt na wcześniej utworzony zasób. W przeciwnym wypadku powinien stworzyć zasób oraz zapamiętać jego źródło. Źródłem zasobu zazwyczaj jest ścieżka do pliku

zasobu zapisanego w pamięci masowej. Ścieżka w postaci łańcucha znakowego jednoznacznie określa zasób i idealnie nadaje się do użycia jako klucz do tablicy haszującej. Dzięki temu, gdy użytkownik ponownie chce stworzyć zasób z tego samego źródła, menadżer jest w stanie bardzo szybko sprawdzić za pomocą tablicy haszującej, czy zasób już istnieje. Czas translacji ścieżki do wskaźnika przechowującego adres miejsca zasobu w pamięci jest relatywnie krótki i zdecydowanie krótszy niż ponowne wczytywanie zasobu. Do sprawdzenia, czy zasób nadal jest potrzebny, służy zliczanie referencji. Wiedza o liczbie użytkowników danego zasobu umożliwia menadżerowi podjąć decyzję o ewentualnym zwolnieniu zasobu. Gdy licznik referencji ma wartość równą zero, to znaczy, że zasób może zostać zwolniony z pamięci. Jednak nie zawsze warto jest taki zasób od razu zwalniać. Gdy na przykład wiemy, że dane zasoby są związane z główną postacią bohatera gry, to nawet jeżeli chwilowo żaden obiekt z nich nie korzysta (ponieważ na przykład bohater gry nie jest w danym momencie widoczny na ekranie), to raczej pewne jest, że zasób za chwilę znów będzie potrzebny. Dlatego warto zasobom nadać priorytety ważności. Zasoby o największym priorytecie nie powinny być zwalnianie nigdy automatycznie. Zasoby o średnim priorytecie powinny być oznaczone jako gotowe do zwolnienia po upływie jakiegoś czasu od ostatniego użycia. Na końcu zasoby o niskim priorytecie są gotowe do zwolnienia zaraz, gdy ostatni użytkownik zasobu przestanie go używać.

5.3.5 Współpraca menadżera zasobów z alokatorem

Bardzo ważnym aspektem pracy menadżera zasobów jest współpraca z alokatorem pamięci. Menadżer zasobów na podstawie licznika referencji zasobów wie, czy zasób jest używany i czy można go zwolnić. Oczywiście, gdy zasób nie jest używany, a także jego priorytet na to pozwala, można go od razu zwolnić z pamięci. Jednak nie zawsze jest to opłacalne. Jeżeli w danym momencie program posiada jeszcze spory zapas wolnej pamięci, zwolnienie zasobu nie wiąże się z żadną korzyścią. A może wystąpić sytuacja, że zasób za moment znowu będzie potrzebny. Dlatego też zasoby powinny być zwalnianie tylko wtedy, gdy istnieje pewność, że nigdy nie będą potrzebne oraz gdy są gotowe do zwolnienia i brakuje pamięci. Żeby menadżer pamięci wiedział, ile jest dostępnej pamięci, musi współpracować z alokatorem pamięci. Współpraca menadżera zasobów z konkretnym, przeznaczonym do tego alokatorem pamięci może mieć wiele innych zalet:

- Alokator może mieć ograniczenie do ilości maksymalnego zużycia pamięci przez menadżer zasobów.
- Alokator specjalnie zaimplementowany dla menadżera konkretnych zasobów może być bardziej optymalny od uniwersalnego. Przykładowo, menadżer tekstur, bardzo często będzie korzystać z alokacji o takim samym rozmiarze. Do tego większość tekstur w danej grze prawdopodobnie będzie w jednym formacie. Dlatego alokator takiego menadżera może spodziewać

się większości alokacji o konkretnym rozmiarze, dzięki czemu lepiej może zarządzać alokacjami. Załóżmy, że 60% tekstur ma wymiary 1024 na 1024 oraz jeden teksel zajmuje cztery bajty (na trzy składowe koloru oraz przezroczystość), co oznacza, że 60% alokacji będzie miało rozmiar 4 MB. Taka wiedza pozwala bardzo zoptymalizować alokator. Na przykład może on działać na zasadzie puli bloków o rozmiarze 4MB. W takim przypadku zwolnienie i utworzenie nowej tekstury o takich wymiarach w ogóle nie potrzebuje zwalniania oraz rezerwowania nowych bloków, a wystarczy ponownie je wykorzystać.

5.4 Wielowątkowość a zasoby plikowe

W przypadku gier, które zawierają rozległe otwarte światy, elementy sceny muszą być doczytywane w czasie trwania rozgrywki. Wiele z tych elementów graficznych może być powiązanych z jeszcze niewczytywanymi zasobami. Aby animacja gry w czasie doczytywania zasobów była płynna, tworzenie zasobów powinno być asynchroniczne. To oznacza, że gra powinna sobie radzić z sytuacją, w której nie wszystkie zasoby są jeszcze wczytane. Bardzo często rozwiązuje się ten problem poprzez niewyświetlanie obiektów, których zasoby nie są jeszcze wczytane. Oczywiście to rozwiązanie może dotyczyć mniej ważnych w danym momencie dla rozgrywki obiektów. W przypadku obiektów, które mają krytyczne znaczenie, trzeba poczekać na załadowanie ich wszystkich zasobów do pamięci. Takie obiekty najlepiej jest stworzyć na początku sceny i oznaczyć ich zasoby wysokim priorytetem, uniemożliwiającym usunięcie ich z pamięci. Ze względu na to, że liczba obiektów potrzebnych od samego początku sceny zazwyczaj jest dość duża, czas trwania ich tworzenia może być długi (kilka, kilkanaście lub kilkadziesiąt sekund). Aby przez ten czas gracz nie doszedł do wniosku, że gra się zawiesiła, zazwyczaj wyświetla się w tym czasie specjalnie przygotowane animowane ekrany informujące o postępie wczytywania się sceny (tzw. „loading screeny”).

6

Narzędzia diagnostyczne

6.1 Błędy

Błąd programistyczny, w żargonie bardzo często nazywany bugiem (ang. *bug*), jest przyczyną niepoprawnego, niespodziewanego działania programu, wynikającą z błędu człowieka powstałego podczas tworzenia oprogramowania.

Współczesne gry komputerowe uważa się za bardzo skomplikowane oprogramowanie, z uwagi na ogromną liczbę różnych zagadnień, z którymi związany jest taki projekt. Do zagadnień tych należą między innymi tworzenie w wydajny sposób grafiki trójwymiarowej, obsługa kolizji i symulacja fizyki, sztuczna inteligencja czy też zarządzenie gigantycznymi ilościami danych. Tak wielka różnorodność wymaga konieczności zaangażowania całej rzeszy specjalistów. Duży stopień skomplikowania tematów, nad którymi pracują programiści, praca pod presją terminów oraz naturalne trudności w komunikacji dużych grup ludzi, muszą powodować pomyłki. Błędy popełnia każdy programista, nawet pracując w pojedynkę, a im większy projekt i więcej osób w niego zaangażowanych, tym większe prawdopodobieństwo wystąpienia trudnych do znalezienia błędów programistycznych.

6.1.1 Przyczyny błędów

Na liczbę występujących błędów w programie wpływ ma wiele czynników:

- *Złożoność problemu* - im program jest bardziej rozbudowany pod względem ilości kodu oraz zawiera więcej skomplikowanych algorytmów, tym więcej jest miejsc, gdzie mogą wystąpić błędy.
- *Umiejętność stosowania zasad języka programowania przez programistów* - biegli w programowaniu, bardziej doświadczeni i lepiej znający projekt programiści popełniają mniej błędów. Oczywiście w dużych zespołach muszą się znaleźć ludzie o różnych poziomach umiejętności.

- *Czytelność kodu* - im większą dbałość przyłoży się do czytelności kodu, tym mniejsze prawdopodobieństwo wystąpienia błędu. Stosowanie czytelnych identyfikatorów oraz komentarzy objaśniających działanie poszczególnych partii kodu, pozwala również łatwiej znajdować i usuwać powstałe błędy. Niechlujny kod wydaje się zawikłany, co może być przyczyną błędów innych programistów, którzy mają problemy z jego interpretacją.
- *Ponowne wykorzystanie kodu* - kod, który był już używany i przetestowany we wcześniejszych projektach, jest pewniejszy niż nowo powstały.
- *Komunikacja i stosunki między programistami* - zgrany zespół lepiej działa, ludzie lepiej się komunikują, dzięki czemu rzadziej dochodzi do nieporozumień, które bardzo często są przyczyną błędów.
- *Optymalizacje* - krytyczne partie kodu wymagają często optymalizacji, które zazwyczaj zmniejszają czytelność kodu. Problem ten często dotyczy np. wstawek asemblerowych, czy innych optymalizacji w algorytmach, które można by uznać za nieeleganckie z punktu widzenia stylu programowania. Oczywiście optymalizacje takie utrudniają interpretacje kodu przez innych programistów.

Gry komputerowe nie są trywialne w produkcji. Pracują przy nich duże zespoły, piszące kod zazwyczaj w języku C++, który jest uważany za język trudny. Dodatkowo kod musi być pisany tak, by po kompilacji program gry działał maksymalnie wydajnie, co nie sprzyja tworzeniu przejrzystego i prostego kodu. Wszystkie te czynniki powodują, że gry są szczególnie narażone na występowanie błędów programistycznych.

6.1.2 Typy błędów

W programach mogą wystąpić następujące błędy:

- *Błędy składniowe* - nie pozwalają na kompilację. Łatwe do usunięcia. Nie są istotnym problemem w skali projektu.
- *Błędy arytmetyczne* - to błędy spowodowane dzieleniem przez zero, przekroczeniem zakresów liczb lub zaokrągleniami. Błędy te mogą czasem być bardzo trudne do wykrycia. Same w sobie nie powodują zawieszania programu, dlatego ich skutki mogą być zauważalne z bardzo dużym opóźnieniem. Błędy takie lubią się propagować, co oznacza, że wystąpienie jednego takiego błędu może powodować cały szereg innych błędów.
- *Błędy pamięciowe* - błędy typu odwołanie się do niezainicjalizowanej pamięci czy też odwołanie się do nieprzydzielonego lub wcześniej zwolnionego bloku pamięci. Są relatywnie łatwe do wykrycia, ponieważ wskutek ich wystąpienia program się bardzo szybko zawiesza.
- *Błędy logiczne* - są to błędy typu niekończące się pętle, rekurencyjne wywoływanie się funkcji bez warunku stopu, wykonanie się pętli za dużo lub za

mało razy. Błędy takie są trudne do wykrycia. Zazwyczaj powodują wtórne błędy innego typu, dzięki którym można zidentyfikować błąd logiczny jako podstawową przyczynę powstania błędów wtórnych.

- *Błędy związane z wieloma wątkami* - to błędy takie jak zakleszczenie się wątków, hazard danych, zawłaszczenie sobie zasobu przez wątek. Zazwyczaj prowadzą do zawieszenia się programu. Ze względu na niesekwencyjne wykonywanie się części programu błędy takie są bardzo trudne do namierzenia.

6.1.3 Wyszukiwanie błędów

Debugowanie jest to czynność polegająca na wyszukiwaniu błędów w programie. Program debuguje się zazwyczaj w specjalnie skompilowanej, uruchomieniowej wersji programu (ang. *debug build*), dzięki której możliwe jest użycie narzędzi ułatwiających debugowanie, które omówione zostaną w dalszej części rozdziału. Wersja uruchomieniowa różni się od finalnej (ang. *release build*) przede wszystkich tym, że zawiera w sobie dodatkowe dane, tak zwane symbole debugowe, które są niezbędne do prawidłowego działania debuggera. Ponadto nie korzysta zazwyczaj z żadnych optymalizacji kompilacji oraz linkera. Zoptymalizowany kod jest trudniejszy do analizy („znikają" z kodu często wykorzystywane zmienne lokalne, które trafiają do rejestrów procesora, zoptymalizowane konstrukcje mogą się znacząco różnić od oryginalnych), poza tym kompilacja oraz linkowanie z włączonymi optymalizacjami trwa zdecydowanie dłużej. W przypadku częstej rekompilacji kodu podczas testowania i uruchamiania, mogłoby to być bardzo uciążliwe. Wersja finalna jest zoptymalizowana oraz wolna od jakichkolwiek nadmiarowych danych, czego skutkiem jest mniejszy rozmiar plików wykonywalnych i bibliotek oraz zdecydowanie szybsze działanie (w specyficznych przypadkach przyspieszenie będzie kilkunastokrotne!) - niestety, jest bardzo trudna do analizy.

Czasami analiza wersji uruchomieniowej jest w dalszym ciągu zbyt skomplikowana. W takich przypadkach do procesu debugowania może być wymagana specjalna wersja programu, która jest bardzo mocno okrojona względem oryginału. Pozwala to wyeliminować nadmiarowe dane, które mogą utrudniać wyszukanie konkretnych błędów. Wersja ta może być również tak skonstruowana, aby dany błąd występował najczęściej jak to jest możliwe.

Debugowanie krytycznych błędów zazwyczaj jest bardzo trudnym i pracochłonnym procesem, dlatego bardzo ważna jest profilaktyka przeprowadzana między innymi za pomocą narzędzi diagnostycznych. Program powinien być uruchamiany i testowany od najwcześniejszej fazy, co uniemożliwi nagromadzenia się dużej liczby błędów. W przeciwnym przypadku może dojść do sytuacji, w której nikt nie ma kontroli nad programem, w którym jest tyle błędów, że trudno jest zacząć sensowne testy. W trakcie testowania każde nieprzewidziane działanie programu powinno być sygnałem alarmowym oraz powinno zostać odnotowane. Aby proces uruchamiania i testowania był jak

najbardziej efektywny, warto zaopatrzyć się w szereg narzędzi diagnostycznych. Jakość oraz liczba takich narzędzi znacząco może wpływać na czas diagnostyki programu.

6.2 Debugger

Debugger jest niezastąpionym narzędziem podczas szukania błędów w programie. Jest podstawowym narzędziem programistycznym i powinien być wykorzystywany zawsze, gdy zachodzi potrzeba analizy działania programu. Narzędzie to pozwala na szereg działań umożliwiających badanie przebiegu programu i analizę działania fragmentów kodu. Pozwala obserwować, w jaki sposób program wykonuje się krok po kroku i jaki to ma wpływ na środowisko, w jakim jest uruchomiony.

Najważniejszymi cechami debuggera są:

- *Pułapka* (ang. *breakpoint*) - programista może wskazać miejsce w kodzie programu, w którym program powinien się zatrzymać. Dzięki temu istnieje możliwość skorzystania z innych narzędzi debuggera, umożliwiających identyfikację błędu w danym momencie działania programu. Stawiając pułapkę, bardzo często mamy możliwość wprowadzenia dodatkowych warunków, które muszą zostać spełnione, aby program się zatrzymał.
- *Praca krokowa* - umożliwia wykonywanie programu instrukcja po instrukcji, dzięki czemu w wygodny sposób można obserwować przebieg programu. Zazwyczaj istnieje także możliwość wykonywania kroków na różnym poziomie szczegółowości - bardziej pobieżnie, pomijając całe wywołania funkcji lub bardzo szczegółowo, rozkaz po rozkazie procesora.
- *Podgląd stosu wywołań* - narzędzie to pozwala podglądać aktualną zawartość stosu wywołań, co z kolei w bardzo wygodny sposób umożliwia obserwację przebiegu programu oraz szybkie przemieszczanie się po poziomach szczegółowości testów.
- *Podgląd zawartości zmiennych lokalnych* - z jego pomocą możliwe jest podglądanie zawartości zmiennych lokalnych z aktualnego zakresu programu. Wiele debuggerów umożliwia też podgląd zmiennych w formie rozwijanego drzewka, dzięki czemu możliwy jest dostęp do innych zmiennych przez lokalne wskaźniki oraz wskaźnik `this`.
- *Podgląd zdefiniowanych obszarów pamięci* - umożliwia podglądanie dowolnych obszarów pamięci programu. Najczęściej jednak wykorzystywany do sprawdzania poprawności dużych danych używanych w programie (zawartość wczytanych plików, wartości łańcuchów znakowych, tablic itd.).
- *Podgląd rejestrów procesora* - umożliwia sprawdzenie aktualnego stanu procesora - funkcjonalność wykorzystywana raczej tylko w bardzo zaawansowanych przypadkach, niemniej może być konieczna w przypadkach, gdy na

skutek optymalizacji kodu przez kompilator często używane zmienne lokalne umieszczane są rejestrach procesora.

- *Modyfikacja zawartości zmiennych lokalnych, zdefiniowanych obszarów pamięci, rejestrów procesora* - bardziej zaawansowane debuggery umożliwiają zmianę zawartości obszarów pamięci oraz stanu procesora w trakcie działania programu. Jest to potężne narzędzie, pozwalające zmienić całkowicie przebieg działania programu. Pozwala bardzo przyspieszyć testowanie, ponieważ możemy przykładowo zmienić zawartość błędnych danych na prawidłową przed zawieszeniem się programu i testować dzięki temu jego dalszy przebieg.
- *Skok do funkcji* - zaawansowane debuggery umożliwiają w dowolnym momencie wykonać dowolną funkcje programu (zmodyfikować stos wywołań lub rejestry instrukcji procesora). Dzięki temu podczas testów w dowolnym momencie możliwa jest zmiana przebiegu działania programu.
- *Obserwacje stanu wątków* - pozwala on sprawdzać aktualny stan programu w wielu wątkach. Umożliwia wykrycie błędów synchronizacji wątków.
- *Edytuj i kontynuuj (ang. edit and continue)* - zaawansowane debuggery dają możliwość rekompilacji po edycji programu „w locie" i wznowienia go dokładnie w miejscu przerwania.

Debuggery są dostępne zazwyczaj jako elementy środowisk programistycznych. Najczęściej stosowane debuggery to:

- **Visual Studio Debugger** - bardzo dobry i popularny debugger, głównie z uwagi na integrację z popularnym środowiskiem programistycznym Visual Studio. Posiada wszystkie cechy nowoczesnego debuggera. Visual Studio głównie wykorzystywane jest do tworzenia aplikacji na komputery i urządzenia przenośne z systemem operacyjnym Microsoftu, a także na konsole Xbox 360.
- **WinDbg** - to alternatywny debugger Microsoftu. Jest mniej popularny od swojego brata Visual Studio Debugger, głównie ze względu na trudność użycia. Uważa się go za zdecydowanie mocniejszy oraz bardziej funkcjonalny od Visual Studio Debugger.
- **GNU Debugger** - tekstowy debugger dostępny na licencji GNU. Nie posiada graficznego interfejsu użytkownika. Większość środowisk programistycznych udostępnia integrację z tym debuggerem. Oprócz tego jest możliwość uruchomienia GNU Debuggera za pomocą programu Visual GNU Debugger, który udostępnia dla niego interfejs użytkownika. GNU Debugger jest bardzo popularny, ponieważ służy do analizy programów dla bardzo wielu różnych architektur oraz może być uruchamiany pod większością systemów operacyjnych, między innymi jest wykorzystywany przez środowisko XCode dla systemów MacOS, a co za tym idzie, jest wykorzystywany podczas tworzenia aplikacji dla urządzeń przenośnych działających na systemie iOS.

- **CodeWarrior Debugger** - bardzo dobry element popularnego środowiska programistycznego Metrowerks Code Warrior, szczególnie często wykorzystywanego do tworzenia aplikacji na urządzenia przenośne z systemem PalmOS, Symbian oraz konsole do gier Playstation 2, Nintedo GameCube, Nintendo DS oraz Nintendo Wii.

6.3 Wykorzystanie kompilacji warunkowej do tworzenia wersji testowych

W trakie tworzenia wersji debugowej programu bardzo często korzysta się z kompilacji warunkowej. Za pomocą dyrektyw kompilacji warunkowej można tworzyć różne wersje programu.

```
#ifdef DEBUG
    #define CONST_VAL 3.14f
    #define MACRO_TEST(param)
#else

    void FunctionTest(float a)
    {
    }
    #define CONST_VAL 1.0f
    #define MACRO_TEST(param) FunctionTest(param)
#endif
```

Listing 6.3-1 - Kompilacja warunkowa dla trybu debugowania

Gdy w kodzie poprzedzającym powyższy kod znajdzie się definicja stałej `DEBUG` albo też program zostanie skompilowany ze specjalną opcją (w przypadku GCC jest to `-DDEBUG`), wówczas w kodzie wynikowym wszystkie wystąpienia `CONST_VAL` zostaną zastąpione wartością `3.14f`, natomiast każde wywołanie `MACRO_TEST` zostanie zastąpione pustą instrukcją. W przeciwnym wypadku `CONST_VAL` zastąpione zostanie wartością `1.0f`, a `MACRO_TEST` wywołaniem funkcji `FunctionTest`.

Rozważmy poniższy przykład:

```
//!
float fTestVariable =  CONST_VAL;
MACRO_TEST(CONST_VAL + 100.0f);
```

Listing 6.3-2 - Użycie testowego makra

Dla programu z definiowaną stałą DEBUG powyższy kod zostanie zamieniony na:

```
//!
float fTestVariable =  3.14f; ;
```

Listing 6.3-2 - Rezultat użycia makra ze zdefiniowaną stałą `DEBUG`

Jeśli natomiast taka definicja nie wystąpi, otrzymamy następujący efekt:

```
//!
float fTestVariable =  1.0f;
FunctionTest(101.0f);
```

Listing 6.3-3 - Rezultat użycia makra bez zdefiniowanej stałej `DEBUG`

6.4 Asercje

Asercje służą do sprawdzania poprawności danych w kodzie. Asercja jest warunkiem sprawdzającym poprawność pewnych założeń. Gdy wynik testu jest sprzeczny z założeniami, to program się kończy. Asercje powinny być stosowane najczęściej jak to jest możliwe - najlepiej gdyby sprawdzane za ich pomocą były wszystkie argumenty funkcji oraz metod. Ich częste stosowanie zapobiega propagacji błędu i umożliwia lokalizację błędu dużo szybciej. Dodatkowo sprytna implementacja asercji za pomocą makra jest uruchamiana tylko w wersji włączeniowej, a więc nie ma w ogóle wpływu na szybkość wykonywania się programu w wersji finalnej.

```
#ifdef DEBUG
    void EngineAssert(bool bCondition)
    {
    if (!bCondition)
       Halt();
    }
    #define ASSERT(condition) EngineAssert(condition)
#else
    #define ASSERT(condition)
#endif
```

```
void Function(CData* pData)

{
    ASSERT(pData);
}
```

Listing 6.4-1 - Wykorzystanie makra `ASSERT`

Gdy w powyższym przykładzie przekazany wskaźnik na klasę `CData` będzie równy `NULL`, asercja zatrzyma program. Aby tak zdefiniowane asercje działały prawidłowo, należy skompilować program ze zdefiniowaną stałą `DEBUG`. Funkcja `Halt` użyta w przykładzie powinna zatrzymać program. Jest to funkcja, której definicja jest zależna od platformy.

Należy pamiętać, że asercja jest makrem, które w finalnej wersji programu w ogóle nie występuje. Dlatego bardzo ważne jest, aby nie wstawiać do makra żadnych wyrażeń mających wpływ na działanie programu.

```
ASSERT(i == 10);
ASSERT(strlen(pszStrnig));
```

Listing 6.4-2 - Prawidłowe wykorzystanie makra `ASSERT`

Powyższe użycia są poprawne, bowiem wyrażenia nie zmieniają stanu programu - odczytują tylko stan zmiennych.

```
ASSERT(i++ == 10);
ASSERT(OpenFile("file.dat"));
```

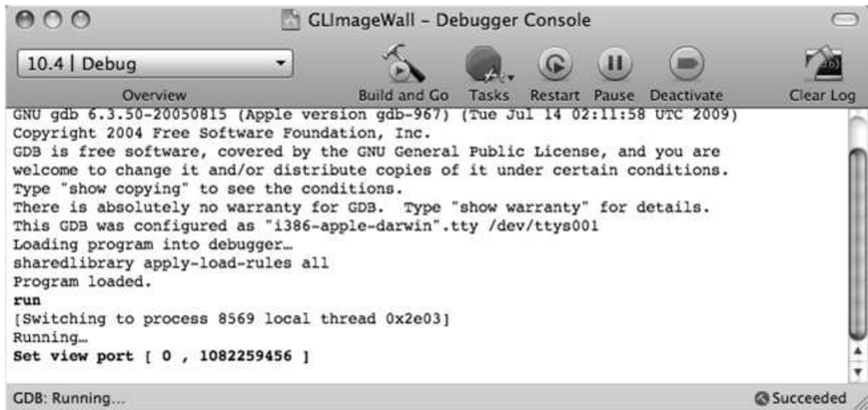
Listing 6.4-2 - Nieprawidłowe wykorzystanie makra `ASSERT`

Takie wykorzystanie asercji jest błędne, ponieważ zarówno w pierwszej linii jak i podczas działania programu stan zmiennej w wersji finalnej będzie różny niż w wersji debugowej. W drugiej linii plik „file.data" w ogóle nie zostanie otworzony w wersji finalnej.

Korzystanie z asercji jest niezmiernie proste i skuteczne. Asercje powinny być stosowane często, gdyż umożliwi to bardzo szybkie wychwycenie błędów relatywnie blisko jego przyczyny. Asercje sprawdzane są tylko w wersji debugowej programu, dlatego ich stosowanie nie ma żadnego wpływu na szybkość działania programu w wersji finalnej.

6.5 Konsola diagnostyczna

Konsola diagnostyczna jest bardzo prostym, a przy tym bardzo przydatnym narzędziem. Składa się zazwyczaj z dodatkowego okienka z konsolą tekstową, które uruchamia się w wersji debugowej programu. Strumień konsoli połączony jest z testowanym programem za pomocą makra, które umożliwiają drukowanie na niej dowolnych tekstów. Konsola diagnostyczna zazwyczaj jest dostarczana razem ze środowiskiem programistycznym na daną platformę, ale jeżeli jej braknie, bardzo łatwo można ją zaimplementować. Za pomocą konsoli można śledzić między innymi poprawność danych.



Rysunek 6.1. Konsola debugowa środowiska XCode

```
void ExampleClass::Method1(int a,int b,float c)
{
    DEBUG_PRINT("ExampleClass::Method1 - a %d, b %d, c %f \n",a,b,c);
    //..
}
```

Listing 6.5-1 - Wykorzystanie konsoli diagnostycznej

W powyższym przykładzie zaraz po wywołaniu metody `Method1` na obiekcie klasy `ExampleClass` na konsoli diagnostycznej zostaną wypisane wartości przekazanych argumentów.

Oczywiście konsola może być również wykorzystana do śledzenia przebiegu programu oraz do wypisywania innych ważnych informacji. Warto jednak korzystać z niej z umiarem, ponieważ gdy jest ona nadużywana, na konsoli pojawia się natłok informacji, przez co staje się ona nieczytelna. Ponadto zbyt

częste wywoływanie operacji I/O (do których należy wypisywanie informacji na konsoli) może znacznie spowolnić wykonywanie programu.

Identyfikator `DEBUG_PRINT` tak jak w przypadku asercji powinien być zdefiniowany jako makro, aby w wersji finalnej program nie „tracił" czasu na wypisywanie bezużytecznych z punktu widzenia użytkownika informacji.

6.6 Śledzenie wycieków pamięci

Błędy spowodowane złym zarządzaniem pamięcią występują często i w dodatku są trudne do wykrycia. W większości przypadków prowadzą do zawieszenia się programu. Najczęściej występującym błędem związanym z zarządzaniem pamięcią jest tak zwany wyciek pamięci, polegający na niezwolnieniu wcześniej zarezerwowanego bloku pamięci. Szczególnie problematyczne są wycieki cykliczne. Nawet jeżeli dotyczą relatywnie małych rezerwacji pamięci, to prędzej czy później spowodują, że cała pamięć zostanie wykorzystana. Wyszukiwanie wycieków pamięci bez specjalnie do tego zaimplementowanych narzędzi jest bardzo trudne, ponieważ skutki takiego błędu objawiają się zwyczaj bardzo późno - podczas alokacji pamięci, gdy cała pamięć jest już wyczerpana. Program wtedy zawiesza się w zupełnie niezwiązanym z wyciekiem miejscu. Znaleźć wyciek w pamięci można w dwojaki sposób. Jednym sposobem jest analiza statyczna programu - sprowadza się do przeanalizowaniu całego kodu programu przy szczególnym skupieniu na miejscach alokacji oraz dealokacji pamięci. Jak łatwo sobie wyobrazić, sposób ten jest jedynie dopuszczalny w przypadku niewielkich projektów. Zdecydowanie lepszym rozwiązaniem jest implementacja mechanizmu umożliwiającego śledzenie wszystkich alokacji oraz dealokacji podczas trwania programu (ang. *leak tracer*). Rejestrując w automatyczny sposób wszystkie alokacje, a także odpowiadające im dealokacje będzie można zlokalizować miejsca powstawania wycieku pamięci.

Do śledzenia wycieków pamięci najlepiej jest wykorzystać wspomnianą już wcześniej cechę operatora `new`, tak zwany „placement new". Operator `new` może przyjmować dowolną liczbę argumentów dowolnego typu po pierwszym argumencie typu `size_t`. W związku z tym możliwa jest taka definicja operatora `new`:

```
void* operator new (size_t size,const char *pszFileName,
                                                    int iFileLine);
void* operator new[] (size_t size,const char *pszFileName,
                                                    int iFileLine);
```

Listing 6.6-1 - Definicje operatora `new`

W definicji tej pozostałymi argumentami są nazwa pliku oraz numer linii pliku, w którym dokonana została alokacja. Znając takie dane, w bardzo łatwy

sposób można zlokalizować wszystkie alokacje w programie. Dla każdej alokacji należy zapamiętać zwrócony wskaźnik, nazwę pliku i numer linii. Podczas dealokacji danego wskaźnika należy usunąć odpowiedni wpis. W chwili poprzedzającej zamknięcie programu w tablicy nie powinien znaleźć się żaden wpis. Jeśli w tablicy znajdują się wpisy, oznacza to wycieki - na szczęście dość łatwe do usunięcia, ponieważ znamy plik, z którego pochodzą alokacje oraz odpowiadające im numery linii w kodzie.

W takiej postaci definicji operatora `new` podczas każdej alokacji należy podawać nazwę pliku oraz linie. Należy do tego celu użyć predefiniowanych zmiennych preprocesora `_FILE_` oraz `_LINE_`, zwracających odpowiednio nazwę pliku źródłowego i numer linii w aktualnym pliku źródłowym.

Alokacja klasy `CObject` w takiej sytuacji wyglądałaby następująco:

```
CObject *pObject = new(_FILE_,_LINE_) CObject();
```

Listing 6.6-2 - Alokacja obiektu klasy `CObject`

Nie jest to rozwiązanie idealne. Problematyczne jest już samo wywołanie - powtarzanie skomplikowanego wywołania funkcji zaciemnia kod. Poza tym w wersji finalnej nie powinno się śledzić wycieków, bo to zajmuje niepotrzebnie pamięć oraz czas procesora.

Rozwiązaniem tego problemu jest skorzystanie z parametrów domyślnych oraz funkcji typu inline.

```
#ifdef LEK_TRACE inline void* operator new (size_t size,const char
*pszFileName =
                                        _FILE_, int iFileLine = _LINE_)
{
    //! leak trace
    //! alloc mem
    //!...
}

inline void* operator new[] (size_t size,const char *pszFileName =
                                        _FILE_, int iFileLine = _LINE_)
{
    //! leak trace
    //! alloc mem
    //!...
} #else
```

```
inline void* operator new (size_t size)
{
    //! alloc mem
    //!...
}

inline void* operator new[] (size_t size)
{
    //! alloc mem
    //!...
}
#endif
```

Listing 6.6-3 - Zmodyfikowane definicje operatora `new`

Po powyższych modyfikacjach wywołanie operatora `new` bez dodatkowych parametrów:

```
CObject *pObject = new CObject();
```

Listing 6.6-3 - Alokacja obiektu klasy Cobject przy użyciu zmodyfikowanego operatora `new`

dla zdefiniowanej stałej `LEAK_TRACE` spowoduje wywołanie wersji `new` z włączonym śledzeniem wycieków. W przeciwnym wypadku użyte zostanie standardowe `new`.

6.7 Profiler

Profiler to narzędzie służące do analizy kodu pod względem szybkości jego wykonania. Za pomocą profilera dokonuje się analizy dynamicznej - czyli w trakcie wykonywania się programu. Jest to bardzo ważne narzędzie, szczególnie w programowaniu gier lub innych aplikacji, które wykonują się w czasie rzeczywistym, a szybkość ich działania wpływa bezpośrednio na jakość. Za pomocą profilera programista może zorientować się, ile czasu wykonują się poszczególne elementy programu. Większość profilerów mierzy częstotliwość oraz czas trwania poszczególnych funkcji, co umożliwia programiście zorientować się, które części programu mają znaczący wpływ na czas wykonywania kodu. Na podstawie tych danych można zdecydować się, które fragmenty kodu wymagają optymalizacji. W kodzie zazwyczaj znajdują się tak zwane wąskie gardła, czyli fragmenty kodu, których wykonanie jest zdecydowanie dłuższe niż pozostałego kodu. W przypadku gier wąskie gardło stanowią zazwyczaj rendering,

wykrywanie kolizji oraz operacje macierzowe. Empirycznie zostało dowiedzione, że w optymalizacji kodu można stosować zasadę Pareto, która w przypadku programowania mówi, że 10% objętości kodu jest odpowiedzialne za 90% czasu wykonywania. I rzeczywiście, fragmenty kodu odpowiedzialne za rendering czy wykrywanie kolizji są bardzo krótkie w stosunku do całego projektu - a to one mają zazwyczaj największy wpływ na szybkość działania. Podczas optymalizacji należy się skupiać rzeczywiście na tych fragmentach kodu, które mają znaczący wpływ na rezultat optymalizacji. Przesadne optymalizacje mogą nieść za sobą szereg problemów:

- Kod może przestać być multiplatformowy.
- Kod może stracić na czytelności.
- Optymalizacje mogą za sobą pociągnąć komplikacje algorytmów, co zwiększa możliwość wystąpienia błędów i utrudnia zarządzanie kodem.
- Optymalizacje pod względem czasu działania programu prawie zawsze mają skutek uboczny, polegający na większym zapotrzebowaniu algorytmu na pamięć operacyjną.

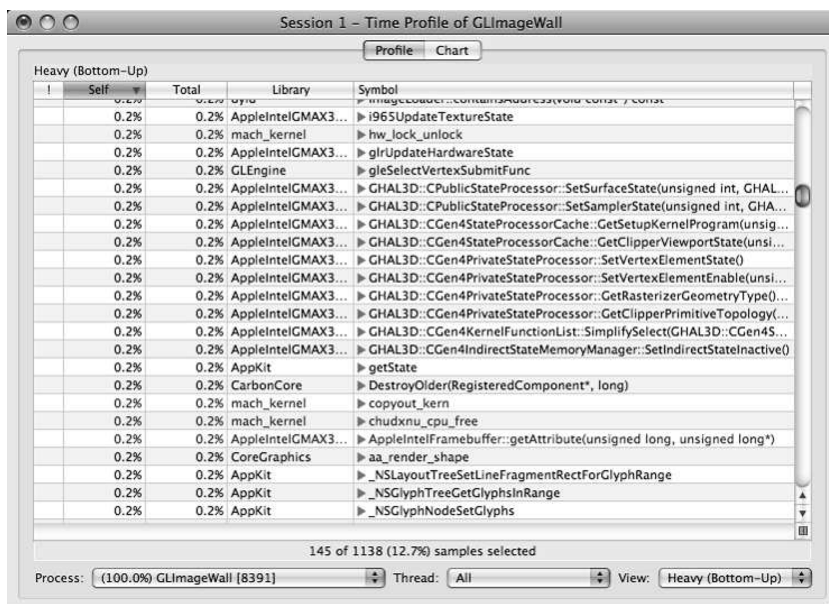
Oczywiście nie są to powody, dla których powinno się zaniechać optymalizacji, lecz warto je mieć na uwadze i w związku z tym optymalizować tylko te fragmenty, które mają znaczący wpływ na prędkość działania kodu.

Profilery można podzielić na dwie grupy:

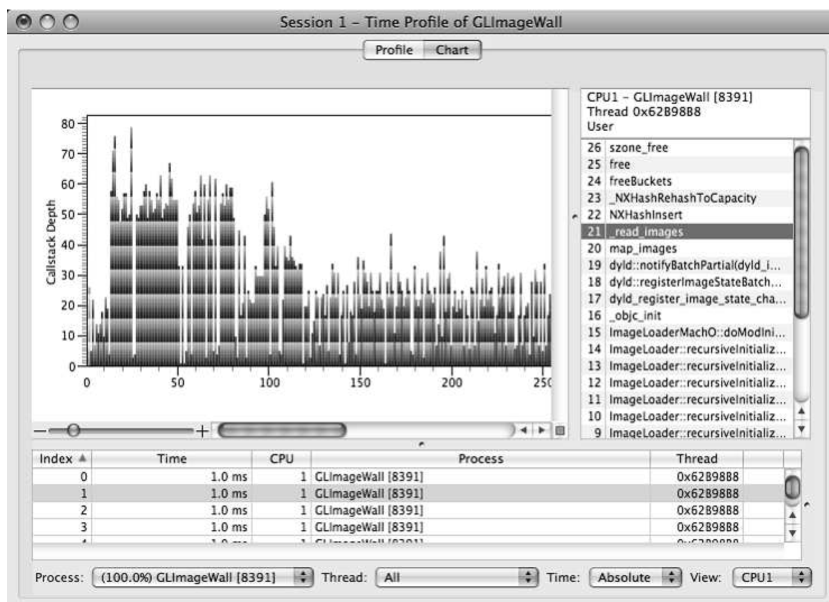
- Profilery, które ze względu na sposób działania wymagają wkompilowania specjalnych instrukcji służących do zbierania danych. Wadami tego typu narzędzi są konieczność modyfikowania kodu programu oraz fakt, że wersja do profilowania jest wolniejsza niż wersja finalna programu. Zaletami są prostota rozwiązania oraz dokładne dane profilujące. Dodatkowe instrukcje mogą zostać wprowadzane na wiele sposobów: manualnie przez programistę, dodane do źródeł przez specjalne narzędzie, przez kompilator na etapie kompilacji, dodane do plików binarnych przez specjalne narzędzie, dodane do pliku wynikowego przed uruchomieniem oraz przez modyfikację kodu wykonywanego „w locie”.
- Profilery, które zbierają próbki danych periodycznie, a wynik opiera się na podstawie statystyki. Próbki danych zbierane są w stałych odstępach czasu w oparciu o przerwania systemowe. Dokładność wyników zależy od częstotliwości próbkowania, która ma wpływ oczywiście na prędkość działania programu.

Poniżej wymieniono najczęściej stosowane profilery:

- **Intel VTune Amplifier** - profiler opracowany przez firmę Intel. Służy do analizy kodu przeznaczonego na procesory tej firmy oraz kompatybilne z nimi (pełna funkcjonalność występuje tylko dla procesorów Intel). Przeznaczony jest dla systemów operacyjnych z rodziny Microsoft Windows



Rysunek 6.2. Profiler Shark - dane w postaci tabeli



Rysunek 6.3. Profiler Shark - dane w postaci wykresu

oraz Linux. Aby korzystać z profilera wystarczy jedynie program skompilować za pomocą dowolnego kompilatora z odpowiednimi danymi debugowymi (symbolami). Program posiada bardzo wiele narzędzi umożliwiających wygodną analizę zebranych danych, które przedstawione są wizualnie za pomocą grafów, wykresów itd.

- **AMD Codeanalyst** - profiler firmy AMD służący do analiz kodu uruchamianego na procesorach tej firmy. Pracuje w systemie Windows oraz Linux. Profiler ten, tak jak poprzedni nie wymaga żadnych zmian w kodzie ani pliku binarnym - analizowany program musi jedynie zawierać wkompilowane symbole. Profiler może działać jako samodzielna aplikacja lub zostać zintegrowany ze środowiskiem Microsoft Visual Studio. Wyniki analizy prezentowane w postaci graficznej. AMD Codeanalyst jest darmowy.
- **GNU profiler gprof** - to darmowy profiler przeznaczny do programów skompilowanych za pomocą kompilatora GCC. Wymaga specjalnie skompilowanego programu (`gcc` z opcją `-pg`). Dane prezentowane są w postaci testowej. Dostępny we wszystkich systemach operacyjnych, na których występuje gcc. W przypadku systemu Linux można skorzystać z darmowego narzędzia `Kprof`, które wyświetla dane w bardziej przejrzystym interfejsie okienkowym.
- **Shark** - profiler przeznaczony do analizy programów napisanych dla systemu Mac OS X. Jest on zintegrowany ze środowiskiem programistycznym XCode. Dane z analizy prezentowane są w formie graficznej. Środowisko XCode wyposażone jest ponadto w dodatkowy profiler - `OpenGL Profiler`, który umożliwia profilowanie kodu na procesorze graficznym. Narzędzie to oprócz analizy pod względem czasu pozwala podglądać bufory urządzenia graficznego w dowolnym momencie renderingu.
- **Sleepy, Very Sleepy** - darmowe, otwarte profilery pod system Windows. Do analizy kodu wymagają standardowych informacji debugowych. Dane prezentowane w interfejsie graficznym.

6.8 Monitor zasobów

Bardzo ważne dla optymalnego działania programu jest dobre zarządzanie dostępną pamięcią. Szczególnie ważne jest to dla dużych bloków pamięci. Czas rezerwacji oraz zwalniania dużych bloków ma duże znaczenie w systemach czasu rzeczywistego, do których zaliczają się gry. Optymalizacja tego zagadnienia sprowadza się do zminimalizowania liczby rezerwacji oraz zwolnień dużych bloków pamięci. W przypadku gier największe alokacje dotyczą zasobów takich jak tekstury, dźwięki, efekty cząsteczkowe itd. Dlatego właśnie menadżer zasobów jest elementem silnika, który wymaga obserwacji. Optymalne zarządzanie parametrami odpowiedzialnymi za zwalnianie nieużywanych zasobów może mieć ogromny wpływ na wydajność gry.

Aby zarządzanie zasobami było wygodne oraz intuicyjne, warto wyposażyć się w narzędzie do obserwacji stanu menadżera zasobów oraz poszczególnych zasobów. Monitor taki w najprostszej postaci może mieć interfejs tekstowy i na żądanie drukować na konsoli diagnostycznej aktualny stan menadżera oraz zasobów. Oczywiście dużo wygodniejsze w obsłudze będzie narzędzie, które będzie posiadało przejrzysty interfejs graficzny. Obie implementacje są możliwe w oparciu o zaproponowane poniżej interfejsy. Jednak ze względu na prostotę, zaprezentowany zostanie monitor tekstowy. Aby możliwe było śledzenie stanu, należy wyposażyć menadżer zasobów w mechanizm, za pomocą którego można poznać jego stan. Innymi słowy, musimy wyposażyć menadżer w szereg publicznych metod, za pomocą których można dowiedzieć się, ile zajmują poszczególne zasoby, czy są jeszcze używane, jak też kiedy były ostatni raz używane, poznać ich priorytet itd.

Najprostszy monitor zasobów mógłby korzystać tylko z tych metod. Byłoby to rozwiązanie niezbyt eleganckie, bowiem gdyby monitor miał pracować ciągle (znać aktualny stan menadżera cały czas), musiałby co klatkę wywoływać wszystkie metody menadżera, mimo, iż większość z nich zwróciłaby tę samą wartość, co w poprzedniej klatce.

Lepszym rozwiązaniem jest wyposażenie w mechanizm wywołania zwrotnego (ang. *callback*), w którym menadżer zasobów będzie mógł informować zainteresowane tym obiekty o zmianie swojego stanu. Zazwyczaj będzie to miało miejsce w sytuacji, gdy zostanie dodany lub usunięty z pamięci zasób. Dzięki temu zredukujemy znacznie liczbę metod wywoływanych w celu monitorowania menadżera zasobów. W takim celu można użyć:

- Zdarzeń (ang. *events*) - menadżer zasobów wysyła zdarzenia o stworzeniu oraz zwolnieniu zasobu. Monitor nasłuchuje odpowiednich zdarzeń.
- Obserwatora (ang. *listeners, observer*) - menadżer zasobów udostępnia możliwość zarejestrowania oraz odrejestrowania obserwatorów. W momencie tworzenia lub zwolnienia zasobu powiadamia o tym wszystkich obserwatorów. Monitor zasobów jest obserwatorem.
- Funktorów (ang. *functors*) - menadżer zasobów udostępnia możliwość zarejestrowania funktora na stworzenie zasobu oraz na jego zwolnienie. Menadżer wywołuje w odpowiednich momentach stosowne funktory. Monitor rejestruje funktory na dodanie oraz zwolnienie zasobu.

Wybór implementacji wywołań zwrotnych może być różny w zależności od preferencji. Najlepiej w całym silniku stosować jedną wybraną implementację, aby zachować spójność. Poniżej (List. 6.8.1) przedstawiono przykładowy kod dla wariantu obserwatora.

Interfejs obserwatora posiada dwie wirtualne metody, które należy przeciążyć. Metoda `OnResourceCreate` jest wywoływana przez menadżer zasobów zaraz po utworzeniu zasobu. Metoda `OnResourceDelete` jest wywoływana przez menadżer chwilę przed usunięciem zasobu z pamięci.

```
//! Resource manager observer
class IResourceManagerListener
{
public:

   virtual ~IResourceManagerListener(){}
   //! Method called after resource create.
   virtual void OnResourceCreate(const CHandle& resourceHandle) = 0;
   //! Method called before resource delete.
   virtual void OnResourceDelete(const CHandle& resourceHandle) = 0;
};
```

Listing 6.8-1 - Interfejs obserwatora zasobów

```
class CResourceManager
{
//..

public:
//..
    //! Adds new listener to manager.
    void AddListener(IResourceManagerListener *pListener);
    //! Removes specific listener from manager.
    void RemoveListener(IResourceManagerListener *pListener);

private:
//..
    void NotifyOnResourceCreate(const CHandle & handle);
    void NotifyOnResourceDelete(const CHandle & handle);

    TVector<IResourceManagerListener*> m_vListeners;
};
```

Listing 6.8-2 - Klasa menadżera zasobów

Menadżer zasobów powinien mieć publiczne metody służące do dodawania i usuwania obserwatorów oraz metody, które powiadomią obserwatorów o zmianie stanu.

Metoda `AddListener` dodaje podany jako argument wskaźnik do wektora zawierającego wszystkie wskaźniki obserwatorów. Metoda `RemoveListener` ma odwrotne zadanie - usuwa podany jako argument wskaźnik z wektora.

Metody `NotifyOnResourceCreate` oraz `NotifyOnResourceDelete` wywołują odpowiednio metody `OnResourceCreate` oraz `OnResourceDelete` na

wszystkich wskaźnikach zawartych w wektorze. Obie powinny być wywoływane przez menadżer w odpowiednich momentach - zaraz po utworzeniu i zaraz przed usunięciem zasobów.

Klasa monitora zasobów powinna implementować interfejs `IResourceManagerListener`, celem zbierania informacji o aktualnym stanie zasobów w systemie.

```
class CTxtResourceManager: public IResourceManagerListener
{
public:
    //! Method called after resource create.
    virtual void OnResourceCreate(const CHandle & resourceHandle);
    //! Method called before resource delete.
    virtual void OnResourceDelete(const CHandle & resourceHandle);
    //! Prints raport on the console.
    void PrintRaport();
};
```

Listing 6.8-3 - Klasa monitora zasobów

Metoda `PrintRaport` służy do wydrukowania na konsoli informacji, jakie zasoby są aktualnie w systemie oraz bardziej szczegółowych informacje na ich temat takich jak, ile zajmują miejsca, ile obiektów z nich korzysta i kiedy ostatnio były używane. Dodatkowo może wypisać, ile cały menadżer zajmuje sumarycznie miejsca w pamięci.

7

Podsumowanie

Na zakończenie niniejszej książki przedstawione zostaną istniejące obecnie na rynku silniki komercyjne oraz silniki opensource'owe. Zachęcamy czytelnika, aby zapoznał się przynajmniej z częścią z nich. Umożliwi to przekonanie się o ich zaletach oraz wadach i wraz z niniejszą książką będzie stanowić dobrą podstawę do opracowania założeń co do własnego pomysłu na nowy silnik gier wideo lub napisania modyfikacji do istniejących silników.

7.1 Współczesne silniki komercyjne

`CryENGINE 3`

Silnik stworzony przez firmę Crytek, znany ze swych ogromnych możliwości i wspaniałej jakości grafiki (sławna gra *Crysis*). Ma duże wymagania sprzętowe. Obecną wersją silnika jest `CryENGINE 3`, dla którego zapowiedziany jest m.in. Sniper: Ghost Warrior 2 tworzony przez polskie studio City Interactive.

Obsługiwane platformy: Windows, Xbox 360, PlayStation 3.

Język skryptowy: Lua.

Licencja: Oprogramowanie zamknięte.

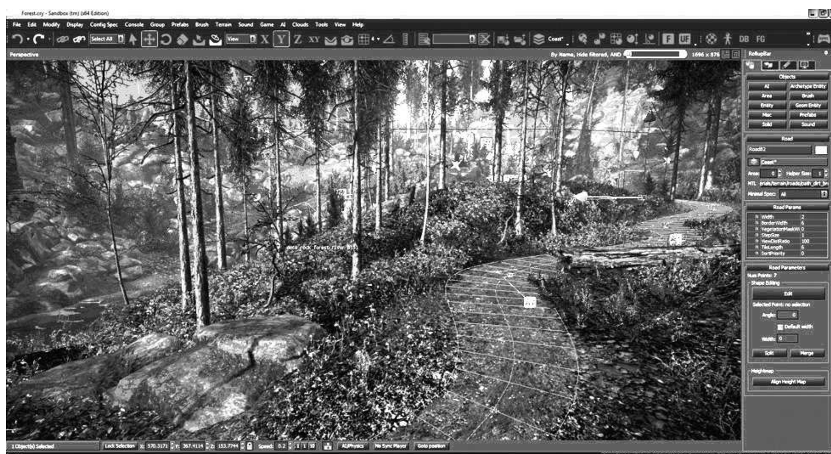
Strona WWW: `http://mycryengine.com/`

Wybrane tytuły: Crysis 2
Sniper: Ghost Warrior 2 (City Interactive).

`Chrome Engine 4 i 5`

Jest to silnik opracowany przez polską firmę Techland na potrzeby ich produkcji. Na jego podstawie stworzonych zostało kilka głośnych polskich gier oferujących przyzwoitą jakość grafiki.

Obsługiwane platformy: Windows, Xbox 360, PlayStation 3.



Rysunek 7.1. Edytor „Sandbox” silnika CryENGINE 3

Licencja: Oprogramowanie zamknięte.
Strona WWW: `http://gdh.techland.pl/`

Wybrane tytuły: Call of Juarez: Bound in Blood
Sniper: Ghost Warrior Dead Island.

Rysunek 7.2. „Dead Island” oparta o silnik Chrome Engine 5

`Gamebryo`

Jeden z częściej wykorzystywanych silników wieloplatformowych. Oferuje dużą liczbę obsługiwanych platform, w tym wszystkie konsole obecnej generacji.

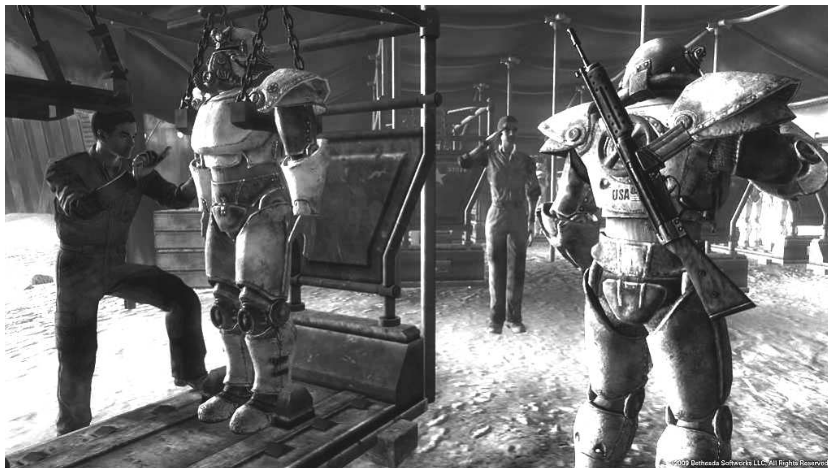
Obsługiwane platformy: Windows, Xbox, Xbox 360, GameCube, Wii, PlayStation 2, PlayStation 3.

Język skryptowy: Lua.

Licencja: Oprogramowanie zamknięte.

Strona WWW: `http://www.gamebryo.com/`

Wybrane tytuły: The Elder Scrolls IV: Oblivion Sid Meier's Civilization IV Fallout 3.



Rysunek 7.3. „Fallout 3" wykorzystujący silnik Gamebryo

`RAGE`

Silnik opracowany przez firmę Rockstar na potrzeby ich własnych produkcji. Cechą charakterystyczną tego silnika jest zdolność do generowania ogromnych, żywych światów, które możemy podziwiać chociażby w grach z serii GTA.

Obsługiwane platformy: PC, PS3, Wii, Xbox 360.

Licencja: Oprogramowanie zamknięte.

Wybrane tytuły: Grand Theft Auto IV Red Dead Redemption.

`Unreal Engine 3`

Jeden z najpopularniejszych silników do tworzenia gier wieloplatformowych. Z jego wykorzystaniem powstały dziesiątki gier wideo. Jest to silnik w pełni sterowany danymi (edytorowy).

Obsługiwane platformy: Windows, Linux, Mac OS, iOS (iPhone, iPad), Dreamcast, Xbox, Xbox 360, PlayStation 2, PlayStation 3.

Rysunek 7.4. „Red Dead Redemption" i silnik RAGE

Język skryptowy: UnrealScript.
Licencja: Oprogramowanie zamknięte.
Strona WWW: `http://www.unrealengine.com/`

Wybrane tytuły: Unreal Tournament 3 Gears of War (1, 2, 3) Mass Effect (1, 2, 3) BioShock (1,2) Batman: Arkham Asylum Mortal Kombat.

`Unreal Development Kit (UDK)`

Dla potrzeb edukacyjnych oraz mniejszych deweloperów opracowana została wersja silnika Unreal Engine pod nazwą UDK. Umożliwia on darmowe zapoznanie się z nim, a licencję wykupić musimy dopiero w przypadku zdecydowania się na wydanie gry.

Obsługiwane platformy: Windows, iOS (iPhone, iPad).
Język skryptowy: UnrealScript.
Licencja: Oprogramowanie zamknięte (darmowe dla zastosowań edukacyjnych).
Strona WWW: `http://www.udk.com/`

`Unity`

Wieloplatformowy silnik gier wideo, którego cechą charakterystyczną jest możliwość tworzenia produkcji uruchamianych z poziomu przeglądarki internetowej. Drugą ważną informacją jest to, że licencja silnika zezwala na darmowe jego użycie dla podstawowych zastosowań i platform.

Obsługiwane platformy: Windows, Mac OS, iOS, Android, Xbox 360.

Język skryptowy: UnityScript, C#, Boo.

Licencja: Oprogramowanie zamknięte (darmowe dla podstawowych zastosowań).

Strona WWW: `http://unity3d.com/`

Rysunek 7.5. Gra „Tiger Woods PGA Tour Online" uruchamiana z poziomu przeglądarki internetowej oparta na silniku Unity3D

7.2 Współczesne silniki opensource'owe

`Irrlicht`

Silnik graficzny, opracowywany oraz rozprowadzany na zasadzie open-source. Dzięki idei otwartego oprogramowania umożliwia zapoznanie się z silnikiem gier wideo „od środka".

Obsługiwane platformy: Windows, Linux, Mac OS.

Język skryptowy: Lua.

Licencja: Zlib (darmowe, open-source).

Strona WWW: `http://irrlicht.sourceforge.net/`

`Ogre3d`

Podobnie jak Irrlicht, jest to silnik graficzny, opracowywany i rozprowadzany na zasadzie open-source.

Obsługiwane platformy: Windows, Linux, Mac OS.

Licencja: MIT (darmowe, open-source).
Strona WWW: `http://www.ogre3d.org/`

`Panda3D`

Jest to kompletny silnik gier wideo, pierwotnie rozwijany oraz wykorzystywany przez firmę Disney. Z biegiem czasu licencja tego silnika została „uwolniona" i jest on obecnie darmowy i rozprowadzany na zasadzie open-source.

Obsługiwane platformy: Windows, Linux, Mac OS, FreeBSD.
Język skryptowy: Python Licencja: BSD (darmowe, open-source).
Strona WWW: `http://www.panda3d.org/`

Wybrane tytuły: Disney's Pirates of the Caribbean Online Disney Pinball.

Rysunek 7.6. „Disney's Pirates of the Caribbean Online" wykorzystująca silnik Panda3D

Literatura

1. Game Engine Architecture, Jason Gregory, wyd. Taylor & Francis Ltd., 2009.
2. Game Coding Complete, Mike McShaffry, wyd. Charles River Media, 2009.
3. Perełki programowania gier. Vademecum profesjonalisty. Tom 1, Mark DeLoura, wyd. Helion, 2002.
4. Perełki programowania gier. Vademecum profesjonalisty. Tom 2, Mark DeLoura, wyd. Helion, 2002.
5. Perełki programowania gier. Vademecum profesjonalisty. Tom 2, Dante Treglia, wyd. Helion, 2003.
6. Triki najlepszych programistów gier 3D. Vademecum profesjonalisty, Adre LaMothe, wyd. Helion, 2004.
7. Język C++, Stroustrup Bjarne, wyd. Wydawnictwa Naukowo-Techniczne WNT, 2004.
8. C++ Kruczki i fortele w programowaniu, Stephen C. Dewhurst, wyd. Helion, 2003.
9. C++. 50 efektywnych sposobów na udoskonalenie Twoich programów - Scott Meyers, wyd. Helion, 2003.
10. Code Complete- Steve McConnel, wyd. Microsoft Press, 2004.

ポーランド日本情報工科大学

POLSKO-JAPOŃSKA
WYŻSZA SZKOŁA
TECHNIK KOMPUTEROWYCH

WARSZAWA
tel.: 22 58 44 500, fax: 22 58 44 501
e-mail: inform@pjwstk.edu.pl
www.pjwstk.edu.pl
Skype: pjwstk_info
facebook: http://www.facebook.com/pjwstk

Wydział Informatyki
Kierunek: informatyka
Studia I, II i III stopnia, studia podyplomowe

Wydział Sztuki Nowych Mediów
Kierunek: architektura wnętrz
Studia I stopnia
Kierunek: grafika
Studia I i II stopnia

Wydział Zarządzania Informacją:
Kierunek: zarządzanie
Studia I stopnia

Wydział Kultury Japonii
Kierunek: kulturoznawstwo
Studia I i II stopnia

Akademickie Liceum Ogólnokształcące przy PJWSTK
www.liceum.pjwstk.edu.pl

Niepubliczne Liceum Plastyczne przy PJWSTK
www.liceumplastyczne.pjwstk.edu.pl

Akademickie Centrum Szkoleniowe
www.acs.pjwstk.edu.pl

WYDZIAŁY ZAMIEJSCOWE:

GDAŃSK
e-mail: gdansk@pjwstk.edu.pl
tel.: 58 683 59 75
fax: 0-58 682 10 67
http://gdansk.pjwstk.edu.pl

Kierunek: informatyka
Studia I stopnia,
studia podyplomowe
Kierunek: grafika
Studia I stopnia

BYTOM
41-902 Bytom, Aleja Legionów 2
tel.: 32 387 16 60, fax: 32 389 01 31
e-mail: bytom@pjwstk.edu.pl
http://bytom.pjwstk.edu.pl

Kierunek: informatyka
Studia I stopnia,
studia podyplomowe
Kierunek: grafika
Studia I stopnia

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Publikacja współfinansowana ze środków
Unii Europejskiej w ramach
Europejskiego Funduszu Społecznego

Egzemplarz bezpłatny