POLSKO-JAPOŃSKA
WYŻSZA SZKOŁA
TECHNIK KOMPUTEROWYCH

Łukasz Hacura, Wojciech Borczyk

Projektowanie i programowanie gier video

WYDAWNICTWO
PJWSTK

Notka biograficzna

Łukasz Hacura (mgr inż.), absolwent kierunku Informatyka na Politechnice Śląskiej, zajmuje się zawodowo tworzeniem gier video, głównie programowaniem i zarządzaniem zasobami ludzkimi. Aktualnie pracuje na stanowisku Lead Programmer w największej polskiej firmie tworzącej gry video – City Interactive, w dziale w Katowickim, gdzie jest odpowiedzialny za kierowanie zespołem programistów oraz tworzenie kodu gry. Wykłada przedmiot „Programowanie gier video" na PJWSTK. Łukasz Hacura posiada doświadczenie w tworzeniu gier na takie platformy jak PlayStation3, Xbox360, PC, Nintendo Wii, Nintendo DS oraz iPhone.

Wojciech Borczyk (mgr inż.), absolwent kierunku Informatyka na Politechnice Śląskiej. Pisze 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.

Streszczenie

Książka omawia najważniejsze zagadnienia projektowania i tworzenia kodu gier video. Układ materiału książki kolejno wprowadza w coraz bardziej szczegółowe elementy, na jakie składa się programowanie gier. Nacisk w omawianiu zagadnień położony jest na wyróżniki tworzenia gier video od tworzenia pozostałych rodzajów oprogramowania. Książka jest przeznaczona głównie dla studentów kierunków informatycznych oraz programistów, którzy nie mieli, lub mieli niewielką, styczność z tworzeniem gier video.

Seria: Podręczniki akademickie

Edytor serii: Leonard Bolc
Tom serii: 50

POLSKO-JAPOŃSKA
WYŻSZA SZKOŁA
TECHNIK KOMPUTEROWYCH

Łukasz Hacura, Wojciech Borczyk

Projektowanie i programowanie gier video

WYDAWNICTWO
PJWSTK

Edytor
Leonard Bolc

Kierownik projektu
Prof. dr hab. inż. Konrad Wojciechowski

Korekta
Anna Bittner

Redaktor techniczny
Ada Jedlińska

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 978-83-63103-02-6
nakład: 150 egz.

Wersja elektroniczna
ISBN 978-83-63103-56-9

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt „ Nowoczesna kadra dla e-gospodarki – program rozwoju Wydziału Zamiejscowego Informatyki w Bytomiu Polsko- Japońskiej Wyższej Szkoły Technik Komputerowych współfinansowany za 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:
Hacura, Ł. & Borczyk, W., 2011. Projektowanie i programowanie gier video. Warszawa: Wydawnictwo PJWSTK.

Spis treści

1

Wstęp

Programowanie gier wideo jest diametralnie różne od jakiegokolwiek innego rodzaju programowania. Wyróżnikiem jest tutaj konieczność innego nastawienia programisty pracującego przy tworzeniu gry. Standardowy programista myśli bardzo funkcjonalnie i metodycznie, i chociaż są to ogromne zalety również przy tworzeniu gier, to jednak współczesnej branży gier wideo bliżej jest do przemysłu filmowego w kategoriach podejścia i oczekiwań, niż do rozwiązań sektora informatycznego. Choć może to brzmieć banalnie, tworzenie gier wideo jest pewnego rodzaju sztuką, która wymaga nie tylko umiejętności technicznych, ale również wyczucia artystycznego i ciągłego zadawania sobie pytania „Czy to się graczowi spodoba?". Nie sposób porównać gracza do np. użytkownika bazy danych. Zadaniem programisty tworzącego bazę danych nie jest wyzwolenie żadnych emocji w użytkowniku, ma co najwyżej unikać wyzwalania frustracji podczas używania jego oprogramowania. W przypadku gier zależy nam często na wywołaniu takich uczuć, jak radość, wzniosłość czy też strach. Konieczność wyzwalania tych emocji poprzez gry wideo jest największym wyróżnikiem tego gatunku spośród wszystkich typów oprogramowania. Podsumowując, tworzenie gier wideo wymaga od programisty unikalnego dla tej branży podejścia, również od strony projektowania i programowania, o których traktuje ta książka.

1.1 O czym i dla kogo jest ta książka

Prezentowana książka przeznaczona jest przede wszystkim dla programistów gier wideo - zarówno przyszłych, którym pomoże poznać podstawy tego rzemiosła, jak również obecnym, którzy będą mogli uporządkować zdobytą dotychczas wiedzę i przy okazji nauczyć się kilu nowych rzeczy. Na rynku księgarskim, zwłaszcza w Stanach Zjednoczonych, istnieje wiele pozycji poświęconych opisom technologii, przy użyciu których tworzone są gry wideo. W tytułach tych książek pojawiają się takie wyrażenia jak: *DirectX* czy *Unreal Engine*.

Jednak bardzo niewiele z tych książek tak naprawdę traktuje o faktycznym projektowaniu i programowaniu gier wideo, a jedynie o technologiach i komponentach używanych w tym celu. Do stworzenia gry wideo z punktu widzenia programistycznego potrzeba znacznie więcej niż znajomości języka programowania i biblioteki graficznej, tak samo jak znajomość C# i SQL nie wystarczy do zaprojektowania i zaimplementowania dowolnego systemu biznesowego opartego na bazie danych.

Książka ta nie ma na celu uczyć ani języka programowania, ani technologii, jakich się używa przy programowaniu gier wideo. Głównym zadaniem tej książki jest pokazanie, co tak naprawdę oznacza dla programisty tworzenie gry wideo, z jakimi wyzwaniami i zadaniami się zetknie przy tego typu projektach. Jest to idealna pozycja dla programisty, który nie miał styczności zawodowej z branżą gier wideo i chciałby się dowiedzieć, na czym zagadnienie tworzenia gier polega od strony programowania.

Założeniem tej książki jest, że czytelnik posiada przynajmniej podstawową umiejętność programowania w języku C++. Większość opisanych tu przykładów oraz technik programowania oparta jest właśnie na tym języku, gdyż jest on podstawowym językiem używanym w branży gier wideo. Oczywiście programiści znający dowolny inny język obiektowy typu Java czy C# nie powinni mieć większych problemów ze zrozumieniem opisywanych tu zagadnień. Podstawowa znajomość zasad działania komputera, rozróżnianie funkcji poszczególnych jego elementów takich jak procesor, karta graficzna czy pamięć RAM na pewno pomoże w zrozumieniu zawartości tej książki.

1.2 Przewodnik po rozdziałach

Treść książki „Projektowanie i programowanie gier wideo" została podzielona na sześć rozdziałów, poniżej przedstawiono krótki opis zawartych w nich treści.

Rozdział 1. Wstęp - wprowadza w tematykę książki oraz pokrótce omawia jej zawartość.

Rozdział 2. Programowanie obiektowe - w tym rozdziale opisane są najlepsze praktyki programowania obiektowego, pod kątem tworzenia gier wideo. Nacisk kładziony jest tutaj bardziej na zamysł programowania obiektowego niż jego techniczne wykorzystanie.

Rozdział 3. Programowanie gier - rozdział ten opisuje cechy charakterystyczne dla programowania gier wideo, w odróżnieniu od tradycyjnego programowania, przykładowo aplikacji biznesowych. Opisane są w nim również współczesne platformy, na które tworzone są gry wideo oraz wyjaśniona została standardowa struktura zespołu projektowego tworzącego gry.

Rozdział 4. Architektura silnika technologicznego - rozdział zawiera opisy gatunków gier w zestawieniu z wymaganiami, jakie poszczególne gatunki sta-

wiają silnikom technologicznym, zestawienie dostępnych współcześnie gotowych silników jak również strukturę uniwersalnego silnika technologicznego wraz z opisem każdej warstwy.

Rozdział 5. Architektura silnika gry - w rozdziale opisana została struktura silnika gry, największy nacisk został położony na architekturę zarządzającą wszystkimi obiektami znajdującymi się w grze.

Rozdział 6. Narzędzia - rozdział zestawia oprogramowanie używane przy produkcji gier wideo. Są to narzędzia nie tylko wspomagające samo tworzenie gier, ale również odpowiedzialne za pomoc w organizacji pracy.

W poszczególnych rozdziałach znajdują się ramki z definicjami, ciekawostkami i dobrymi praktykami związanymi z branżą gier wideo, które wyglądają następująco:

Definicja

W takich ramkach zawarto definicję różnych pojęć stosowanych zarówno w programowaniu, jak i ogólnie w branży gier wideo.

Ciekawostka

W takich ramkach zawarte są różnego rodzaju ciekawostki z branży gier wideo.

Dobra praktyka

W takich ramkach zawarto informację na temat dobrych praktyk zalecanych przez autora.

2

Programowanie obiektowe

2.1 Dlaczego obiektowo?

Wielu programistów przekonanych jest, że programuje obiektowo, ponieważ używa obiektowego języka programowania, takiego jak C++, Java czy C#. Nic bardziej mylnego. Można swobodnie programować strukturalnie w tych językach, gdyż programowanie obiektowe to sposób myślenia, a nie dodatkowe możliwości, jakie daje składnia języka. Żeby zrozumieć zalety programowania obiektowego, warto się zastanowić, skąd tak naprawdę wziął się ten kierunek rozwoju języków programowania. Kiedyś języki programowania takie jak Basic czy C miały na celu uproszczenie komunikacji pomiędzy maszyną a człowiekiem. Języki asemblerowe, czyli najbardziej bezpośrednia forma komunikacji z procesorem, były trudne w opanowaniu, a kod wytworzony przy ich użyciu - choć działał po skompilowaniu bardzo szybko - był nieprzejrzysty i trudny w zrozumieniu. Stworzenie jakiegokolwiek większego programu w języku asemblerowym stanowiło nie lada wyczyn i generowało mnóstwo problemów w trakcie jego testowania oraz podczas wprowadzania w nim jakichkolwiek zmian. Wysokopoziomowe języki programowania miały usprawnić i ułatwić proces tworzenia oprogamowania. Udostępniały szerszy w stosunku do asemblera zestaw słów kluczowych, które w połączeniu z takimi mechanizmami jak procedury i funkcje pozwalały na enkapsulację problemów i tworzenie przejrzystego, zrozumiałego kodu. Powstał paradygmat tak zwanego programowania proceduralnego. Jednak wszystkie te zabiegi miały na celu tylko uproszczenie komunikacji z maszyną. Nadal wymuszały sposób myślenia w kategoriach wykonywania kodu krok po kroku, jak kodu maszynowego. Programowanie obiektowe ma na celu zastosowanie ludzkiego sposobu myślenia i przeniesienie go w drugiej kolejności na język maszyny. Faktycznie, nawet najbardziej obiektowy kod, zawile stosujący polimorfizm, mający wiele warunków etc., koniec końców zostaje przetworzony na strukturalny ciąg instrukcji wykonywanych jedna po drugiej, gdyż tak stworzone są komputery: przyjmują ciąg instrukcji, które wykonują.

! Definicja 2.1. Polimorfizm

Mechanizm w obiektowych językach programowania umożliwiający wywołanie różnych fragmentów kodu przy użyciu wywołania tej samej funkcji lub odwołania się do różnych typów zmiennych za pomocą jednego typu.

Przykład polimorficznego wywołania funkcji:

```
class CVehicle {
    virtual void Drive(){cout << "I don't know how";}
};

class CCar: public CVehicle {
    void Drive(){cout << "I'm rotating my wheels";}
};

int main() {
    CVehicle * pVehicle = new CCar();

    pVehicle->Drive();
}
```

Polimorficzne wywołanie funkcji wirtualnej Drive powoduje wypisanie na ekranie tekstu „*I'm rotating my wheels*", mimo iż pochodzi ze wskaźnika na klasę bazową *CVehicle*. Gdyby funkcja ta nie była wirtualna, wywoływałaby się metoda klasy bazowej i na ekranie zostałoby wypisane „*I don't know how*".

Człowiek postrzega świat w sposób obiektowy. Obiekt posiada w sobie zarówno cechy, jak i zachowania. Na przykładzie samochodu, człowiek postrzega go jako jedną całość, obiekt, który posiada takie funkcje jak jeżdżenie i takie atrybuty jak prędkość. Bardzo ważne jest tutaj połączenie tych dwóch rzeczy: cech i zachowań, czyli w programowaniu obiektowym pól oraz metod klas. W kodzie proceduralnym poszczególne zachowania i cechy są oderwane od siebie, a ich logiczna spójność w działaniu jest całkowicie umowna i występuje tylko w umyśle programisty. Przykładowo programista wie, że funkcja przeliczająca prędkość samochodu powinna być używana tylko w połączeniu z danymi samochodu, jednak niekoniecznie musi to wynikać bezpośrednio z wyglądu kodu źródłowego. Kod obiektowy jest bardziej naturalny dla człowieka, przez co projektowanie aplikacji w takiej metodologii jest dla programisty znacznie bardziej intuicyjne. Poniżej znajduje się kod źródłowy pod postacią obiektową i proceduralną, wykonujący dokładnie ten sam algorytm, przeliczania odległości, jaką przebył abstrakcyjny samochód zależnie od ustawionej mu prędkości:

Kod obiektowy:

```
// the class representing a~car
class CCar
{
private:
    // car current speed
    float m_fSpeed;

    // car acceleration parameter
    float m_fAcceleration;

    // car current mileage
    float m_fMileage;

public:
    // constructor
    CCar()
    {
        m_fSpeed = 0.0f;
        m_fAcceleration = 1.0f;
        m_fMileage = 0.0f;
    }

    // returns the car mileage
    float GetMileage(){return m_fMileage;}

    // the car update function
    void Update()
    {
        m_fMileage += m_fSpeed;
    }

    // accelerates the car
    void Accelerate()
    {
        m_fSpeed += m_fAcceleration;
    }
};

int main()
{
    // we create the car
    CCar car;

    // we give a~car some starting speed
    car.Accelerate();

    // we update the data
    while(1)
    {
      car.Update();
      cout << "\nCar␣current␣mileage:␣";
      cout << car.GetMileage();
    }
    return 0;
}
```

Kod proceduralny:

```
// car data structure
struct SCar
{
    // car current speed
    float fSpeed;

    // car acceleration parameter
    float fAcceleration;
```

```
    // car current mileage
    float fMileage;
};

// function that updates the car
void CarUpdate(SCar * pCar)
{
    pCar->fMileage += pCar->fSpeed;
}

void CarAccelerate(SCar * pCar)
{
    pCar->fSpeed += pCar->fAcceleration;
}

int main()
{
    // we create the car data structure
    SCar car;
    car.fMileage = 0.0f;
    car.fSpeed = 0.0f;
    car.fAcceleration = 1.0f;

    // we give a~car some starting speed
    CarAccelerate(&car);

    // we update the data
    while(1)
    {
      CarUpdate(&car);
      cout << "\nCar␣current␣mileage:␣";
      cout << car.fMileage;
    }

    return 0;
}
```

Jakkolwiek kody te wykonują dokładnie to samo zadanie, w pewnych subtelnych różnicach widać większą elegancję kodu obiektowego. Przede wszystkim nie ma twardego rozdziału między danymi a funkcjami samochodu. W kodzie obiektowym samochód traktowany jest jako jedna całość, w proceduralnym rozbity jest na funkcje i struktury danych. Kod obiektowy jest również naturalnie enksapsulowany, czyli na zewnątrz udostępnianie są tylko te dane i te funkcje, które programista uznaje za konieczne. W tym przypadku udostępnione jest pole `m_fMileage`, ale tylko do odczytu poprzez funkcje `GetMileage`.

Enkapsulacja kodu

Enkapsulacja, czy też inaczej hermetyzacja, jest jednym z głównych założeń programowania obiektowego. Zakłada udostępnienie na zewnątrz obiektu tylko tych metod i pól, z których muszą korzystać inne klasy, pozostałe powinny się znajdować w sekcji chronionej lub prywatnej. Najlepszą praktyką jest tzw. pełna enkapsulacja, czyli całkowity brak udostępniania pól klas publicznie. Jeśli jakaś klasa potrzebuje dostępu do pola innej klasy, powinno być to zrealizowane odpowiednio przez funkcje `Set` lub `Get`.

Możliwość zastosowania ludzkiego, bardziej intuicyjnego sposobu myślenia to jedna z największych zalet programowania obiektowego, ale nie jedyna. O ile programowanie proceduralne poczyniło w tym kierunku już pierwsze kroki, programowanie obiektowe pozwala na wprowadzenie swoistej taśmowej specjalizacji do programowania. W projekcie zorientowanym obiektowo każdy programista może zająć się osobnym kawałkiem kodu, nie wchodząc w drogę innemu programiście. Wracając do przykładu samochodu, jeden programista może zająć się klasą odpowiedzialną za modelowanie silnika, a inny za klasę modelującą skrzynię biegów. Tak długo, jak dobrze zaprojektowane zostaną interfejsy komunikacji pomiędzy tymi dwiema klasami, prace nad nimi mogą przebiegać niezależnie i równolegle. Pozwala to również w łatwy sposób zastosować metodę „dziel i rządź" - jest to koncepcja w programowaniu stwierdzająca, iż każdy problem może zostać podzielony na zestaw mniejszych problemów, których rozwiązywanie po kolei jest prostsze niż rozwiązanie całego problemu naraz.

Definicja 2.2. Programowanie proceduralne

Model programowania zalecający dzielenie kodu na procedury, które nie powinny korzystać ze zmiennych globalnych, lecz pobierać i przekazywać wszystkie dane, jako parametry.

Programowanie obiektowe należy rozumieć bardziej w kategoriach koncepcji niż składni i możliwości języka. Nic nie stoi na przeszkodzie, żeby programować w sposób obiektowy w języku całkowicie do tego nieprzystosowanym, jak np. C. Jednak należy pamiętać, że obok koncepcji, w obiektowych językach programowania stworzono dużo mechanizmów mających ułatwić programiście zastosowanie obiektowych założeń. Najbardziej charakterystycznym elementem w językach obiektowych jest możliwość tworzenia klas obiektów, dziedziczenie oraz mechanizm polimorfizmu. Umiejętności efektywnego wykorzystania tych mechanizmów mogą zaoszczędzić programiście wiele problemów projektowych.

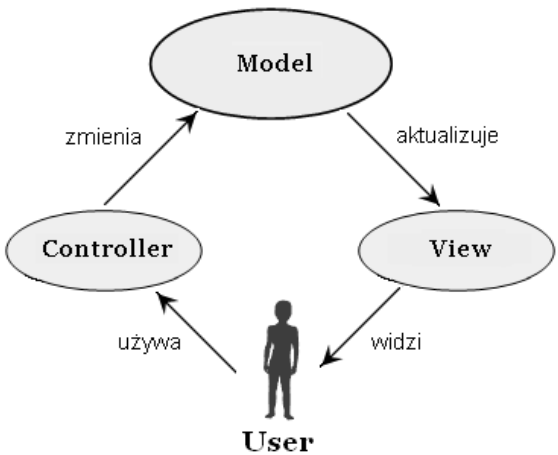
2.2 Warstwowość programowania

Do projektowania aplikacji podchodzi się zazwyczaj na jeden z dwóch sposobów: *top - down* (pol. góra - dół) lub *bottom - up* (pol. dół - góra). Pierwsze podejście oznacza rozpoczęcie prac od zagadnień ogólnych, aż do stopniowego ich uszczegóławiania, podczas gdy druga metoda stosuje koncepcję odwrotną: pracę rozpoczyna się od najniższych poziomem abstrakcji problemów i stopniowo przechodzi do najwyższych. Fundamentalna zasada inżynierii oprogramowania mówi, że każdy problem projektowy może zostać uproszczony po-

przez wprowadzenie dodatkowego, pośredniego poziomu pojęciowego. Jest to obiektowa parafraza ogólnej zasady programowania: „dziel i rządź".

Jednym z najbardziej podstawowych zastosowań powyższej zasady projektowania jest architektoniczny wzorzec projektowy MVC (z ang. *Model View Controller*, pol. Model Widok Kontroler). Wzorzec ten opisuje model trójwarstwowy aplikacji:

- Warstwa modelu - opisuje zależności pomiędzy danymi i logikę programu. Wszelkie algorytmy i działania, jakie może wywołać użytkownik przy pomocy kontrolera, są zaprogramowane w tej warstwie. Przykładowo w grach w tej warstwie znajdują się algorytmy sztucznej inteligencji czy też zapisane dane użytkownika takie jak stan gry.
- Warstwa widoku - zajmuje się wyświetlaniem danych dla użytkownika, zazwyczaj poprzez pewną część GUI (z ang. *Graphical User Interface*, pol. Graficzny Interfejs Użytkownika). Dane do wyświetlania są pobierane z warstwy modelu. W grach komputerowych jest to najczęściej warstwa odpowiedzialna za wyświetlenie wszelkiego rodzaju grafiki bezpośrednio na ekranie.
- Warstwa kontrolera - jest warstwą komunikacji pomiędzy użytkownikiem a modelem. Użytkownik może wywołać pewien określony przez kontroler zestaw operacji na modelu, po ich wykonaniu widzi efekty na warstwie widoku, którą model zaktualizuje. Przykładowo kod obsługujący urządzenia wejścia, np. myszkę czy też pada powinien się znajdować w warstwie kontrolera.



Rysunek 2.1. Diagram procesów MVC

Chociaż MVC jest już dosyć starą koncepcją, gdyż pochodzi z roku 1978, to nadal jej idea jest aktualna i dynamicznie rozwijana. W grach wideo często wa-

rstwa modelu jest rozbijana na kolejne dwie: algorytmiczną i danych, w takim układzie kontroler wywołuje metody z warstwy algorytmów, te z kolei operują na danych. W aplikacjach biznesowych istnieją już wzorce, które mówią nawet o siedmiu warstwach aplikacji. Jakkolwiek optymalna liczba warstw w programie jest mocno uzależniona od jego rodzaju oraz rozmiaru, to już sam fakt rozdzielenia pewnych poziomów abstrakcji jest kluczowy dla stworzenia zrozumiałej programistycznie aplikacji bez zbędnych zależności pomiędzy jej modułami.

2.3 Projektowanie

2.3.1 Wstęp

Faza projektowania w oprogramowaniu jest bardzo często traktowana pobieżnie, czasem wręcz pomijana. Zazwyczaj wynika to pośrednio z faktu niewielkiego doświadczenia programisty w programowaniu obiektowym - programista nie wie do końca, jak napisać daną aplikację. Nie ma jej obrazu w głowie. Dlatego chce jak najszybciej rozpocząć programowanie, rozpoznać problemy i zwalczać je kolejno wraz z ich występowaniem, pomijając kwestię projektowania „z góry". O ile jest to na pewno świetny sposób na zdobycie doświadczenia w rozwiązywaniu problemów, jakie mogą się pojawić przy oprogramowywaniu danego typu aplikacji, tak niekoniecznie jest to najbardziej efektywne podejście do jej stworzenia. Natychmiastowe rozpoczęcie implementacji jest też często pożądane przez osoby zarządzające programistą, gdyż istnieje przeświadczenie, że im szybciej nastąpi implementacja, tym szybciej będzie można oglądać efekt.

Niezależnie od przyczyn, efekt zazwyczaj jest taki sam, tzn. programowanie z pominięciem fazy projektowania prowadzi do bałaganu w kodzie i znacznego zwiększenia iteracji poprawek, zanim osiągnie się założony efekt. Często pominięcie fazy projektowania prowadzi do konieczności zarzucenia kodu dotychczas stworzonego i rozpoczęcia pracy od początku, co jest nieporównywalnie bardziej kosztowne czasowo niż zaprojektowanie aplikacji przed rozpoczęciem implementacji. Oczywiście można również przesadzić w drugą stronę: projektowanie kodu źródłowego na wypadek każdej możliwej potrzeby i przypadku użycia jest marnowaniem czasu, ponieważ wszystkich i tak nie uda się przewidzieć, a wiele z nich w ogóle nie wystąpi lub też nie będzie przez użytkownika wykorzystywane. Niezależnie jednak od sytuacji zawsze warto się najpierw nad problemem zastanowić, a dopiero później przystąpić do działania. Ta ogólna życiowa zasada ma, jak widać, również swoje zastosowanie w programowaniu.

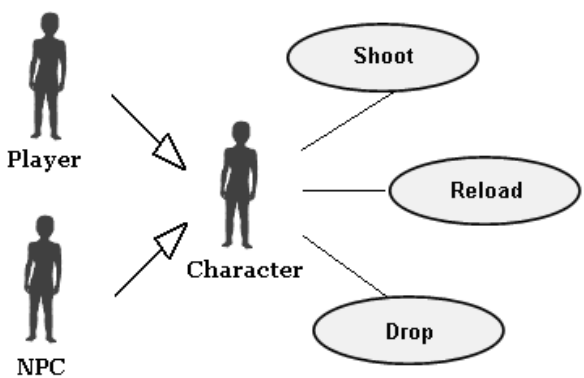
Do procesu projektowania powstały metodologie narzędzia, które wspomagają jego przeprowadzenie. Poniżej zostaną przedstawione najważniejsze spośród nich.

2.3.2 Język UML

Ujednolicony język modelowania, czyli UML (od ang. „*Unified Modeling Language*") udostępnia zestaw narzędzi w postaci diagramów, służących do zaprojektowania dowolnego systemu informatycznego. UML ma za zadanie ujednolicić projektowanie obiektowe w prosty i zrozumiały dla wszystkich sposób, jednocześnie na tyle dokładny, by uniknąć nieporozumień. UML jest językiem formalnym, czyli jednoznacznym, jak języki programowania. Z gotowego projektu UML można bezpośrednio wygenerować kod źródłowy, pozwalając na znaczne skrócenie czasu fazy projektowania i przygotowania projektu do implementacji.

UML udostępnia kilkanaście rodzajów diagramów. Poniżej podano krótkie omówienie tych najczęściej wykorzystywanych:

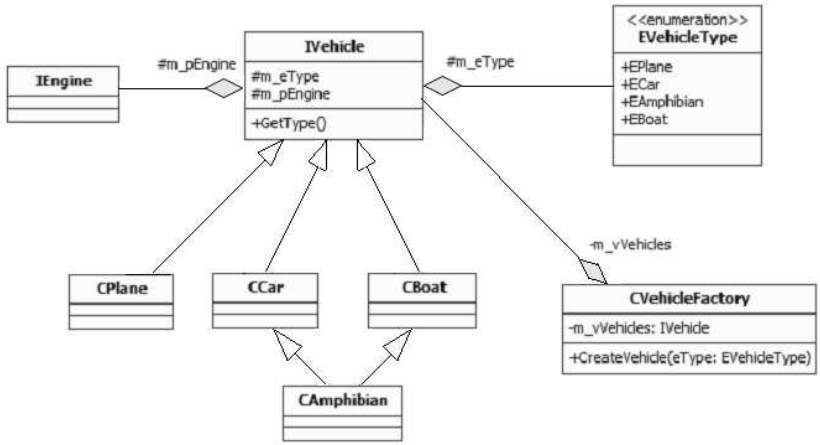
- *Diagram przypadków użycia* - odpowiada na pytanie, w jaki sposób będzie używana dana aplikacja i na jakie składowe czynności dzielić się będzie korzystanie z niej. Diagram ten ma najwyższy poziom abstrakcji, często używany jest bezpośrednio przy kontakcie z klientem, np. producentem gry jako forma sprecyzowania wymagań użytkowych aplikacji. Na diagramie wyróżnia się dwa zasadnicze rodzaje bloczków: aktora i przypadku użycia. Aktor jest to osoba wykonująca konkretną czynność na aplikacji, będącą właśnie przypadkiem użycia. Poniżej przykład takiego diagramu opisujący przypadki użycia broni przez postać w grze (rys. 2.2)



Rysunek 2.2. Diagram przypadków użycia

- *Diagram klas* - jest to podstawowy element każdego programu projektowanego obiektowo, pokazujący zależności pomiędzy klasami, takie jak dziedziczenie, zawieranie czy komunikację. Przykład diagramu klas przedstawiono na rysunu 2.3.

  Diagram klas, jak widać, ma dość wysoki poziom szczegółowości. Na schemacie tym, oprócz relacji typu generalizacja, kompozycja oraz agregacja,

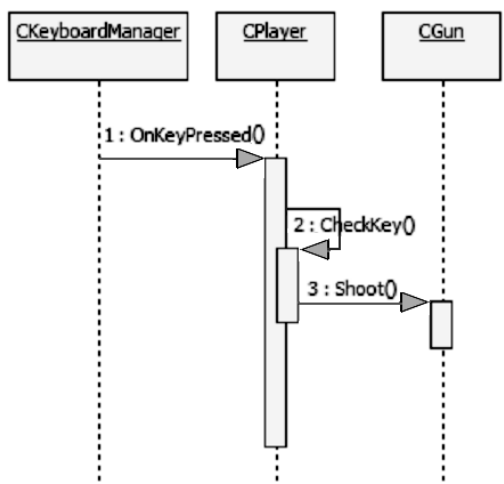


Rysunek 2.3. Diagram Klas

przedstawione są również pola i metody poszczególnych klas oraz ich zakres widoczności.

Jeśli programista ma za zadanie napisanie aplikacji, która będzie składała się chociażby z kilku klas, warto najpierw stworzyć diagram klas takiego projektu. Na etapie tworzenia diagramu często udaje się wychwycić luki już w pierwszym pomyśle implementacji, zmusza to również programistę do przemyślenia projektu o kilka kroków naprzód.

- *Diagram sekwencji* - jest świetnym uzupełnieniem diagramu klas dla uzyskania pełnego obrazu architektury aplikacji. Stworzenie samego diagramu klas często nie definiuje sposobów komunikacji i konkretnych sekwencji ich



Rysunek 2.4. Diagram sekwencji

użycia. Diagram sekwencji pokazuje po kolei, jak wygląda wywołanie konkretnego zdarzenia w systemie, np. logowania, których klas dotyczy i w jaki sposób następuje kolejność wykonywania poszczególnych metod. Tworzenie diagramów sekwencji jest zazwyczaj bardziej czasochłonne niż stworzenie diagramu klas, gdyż dla pojedynczego diagramu klas można wygenerować kilka sekwencji, zależnie od liczby przypadków użycia danej klasy. Przykład diagramu sekwencji pokazano na rysunku 2.4.

Jak widać w diagramie sekwencji po kolei przedstawia się wywołania odpowiednich funkcji i sposób komunikacji pomiędzy klasami. Wypisanie diagramów sekwencji dla wielu przypadków użycia danego zestawu klas, zgodnie z tym jak będą one używane w aplikacji, pozwala na szybkie wychwycenie błędów w pierwotnych założeniach.

Oprócz powyżej wymienionych diagramów istnieje wiele innych wspomagających prace przy projektowaniu aplikacji, takich jak np. diagramy czynności czy wdrożenia. Każdy większy projekt programistyczny powinien zostać zaprojektowany w UML, gdyż pozwoli to na proste zarządzanie zmianami, uprości rozdział zadań, ocenę skomplikowania projektu i generowanie dokumentacji. Przykładowo przy rozrośnięciu się danej klasy i potrzeby rozdzielenia jej na dwie, diagram klas znacznie uprości proces rozdziału i w łatwy sposób pokaże, jakie elementy będą musiały pozostać wspólne, by zachować dotychczasową funkcjonalność. Stopień użycia języka UML powinien być wprost proporcjonalny do rozmiaru projektu. Dla bardzo małych projektów wystarczyć może sam diagram klas. Dla trochę większych na pewno będą przydatne diagramy przypadków użycia i sekwencji, podczas gdy diagramy wdrożeniowe zastosowane powinny zostać już w przypadku dużych aplikacji biznesowych. Wyróżnia się trzy zasadnicze sposoby wykorzystania języka UML:

- *Szkic* - wykorzystywany zazwyczaj przez pojedynczego programistę, nie zespół. Szkic używany jest przed przystąpieniem do implementacji większego modułu kodu obiektowego. Po stworzeniu szkicu, który najczęściej składa się po prostu z diagramu klas, przystępuje się do implementacji, natomiast do szkicu nigdy się już nie wraca.
- *Plan* - służy do zaprojektowania większego modułu kodu obiektowego, często będącego częścią jeszcze większej aplikacji. Plan jest obowiązkowy, gdy w projekcie uczestniczy więcej niż jeden programista. Często utrzymuje się kompatybilność planu UML z implementacją w czasie trwania projektu, by cały zespół miał ciągły, prosty wgląd w architekturę aplikacji.
- *Implementacja* - sposób wykorzystywany tylko przy bardzo dużych aplikacjach, gdzie rozmiar projektu sprawia, że ani jeden programista nie jest w stanie całkowicie objąć działania aplikacji. UML, jako język formalny, przy użyciu wielu narzędzi służy w tym momencie bezpośrednio do wygenerowaniu kodu źródłowego. Wymaga to pełnego wykorzystania wszystkich mechanizmów UML, schodząc na najniższy poziom abstrakcji. Ogromny narzut pracy, jaki jest potrzebny, by aplikacja była cały czas utrzy-

mana w stu procentach zgodnie z UML, czyli żadna modyfikacja kodu źródłowego nie może nie mieć swojego odzwierciedlenia w diagramach UML, sprawia, że takie wykorzystanie zaczyna się opłacać przy zespołach programistów składających się co najmniej z kilkudziesięciu osób, co rzadko ma miejsce w procesie tworzenia gier.

Niezależnie jednak od rozmiaru, sam fakt użycia języka UML w dowolnym projekcie informatycznym to ruch w dobrą stronę. Każdy programista powinien znać dobrze to narzędzie wspomagające.

2.3.3 Wzorce projektowe

Nie jest celem tej książki przybliżenie konkretnego zestawu wzorców projektowych - na ten temat można by napisać kilka osobnych tomów - jednak nie sposób nie wspomnieć o samym zagadnieniu, pisząc o projektowaniu kodu. Wzorce projektowe są to pewne gotowe schematy zależności i diagramów klas zaprojektowane do rozwiązywania konkretnego problemu.

Definicja 2.3. Wzorce projektowe

Wzorce projektowe (nie mylić z wzorcami klas!) są to ogólne rozwiązania do często powtarzających się problemów projektowania oprogramowania. Wzorzec projektowy nie jest gotowym rozwiązaniem, nie może być jednoznacznie przepisany na kod źródłowy. Pojedynczy wzorzec projektowy może być wykorzystany w wielu różnych sytuacjach. Obiektowe wzorce projektowe zazwyczaj pokazują zależności i interakcje pomiędzy klasami, bez podania bezpośredniej implementacji klas. Dzięki takiemu podejściu wzorce projektowe są łatwe do modyfikacji i możliwe do zastosowania w praktycznie dowolnym języku programowania.

Jednym z prostszych wzorców projektowych, świetnie nadającym się do przedstawienia ogólnej idei, jest „*Singleton*". Wzorzec ten ma na celu stworzenie klasy, której jest tylko jedna instancja w aplikacji.

Poniżej prosta implementacja tego wzorca projektowego:

```
// a~design pattern singleton class
class CSingleton
{
private:
    // instance of the class
    static CSingleton * m_pInstance;

protected:
    // constructor
    CSingleton(){}
```

```
    // destructor
    ~CSingleton()
    {
        m_pInstance = NULL;
    }

public:
    //! returns the instance of the class
    static CSingleton * GetInstance()
    {
        if (m_pInstance == NULL)
        {
            //if the instance does not exist yet, create one
            m_pInstance = new CSingleton();
        }

        return m_pInstance;
    }

    // removes current instance
    void DeleteInstance()
    {
        delete m_pInstance;
    }
};

//! initialization of the singleton instance
CSingleton * CSingleton::m_pInstance = NULL;
```

Jak widać, dostęp do tej klasy jest realizowany za pomocą statycznej metody `CSingleton::GetInstance()`. Wywołanie tej metody po raz pierwszy stworzy pierwszą i jedyną instancję obiektu `CSingleton`. Każde następne wywołanie będzie zwracało wskaźnik na ten sam obiekt. Ten wzorzec obiektowy często jest wykorzystywany w miejscach, gdzie chcemy być pewni, że nigdy nie powstanie więcej niż jedna instancja danej klasy, gdyż mogłoby to spowodować błędne działanie programu. Przykładowo klasa zajmująca się jakimś zasobem, np. kartą muzyczną, nie powinna mieć kilku instancji, gdyż mogłoby to doprowadzić do konfliktów i próby na przykład wielokrotnej inicjacji karty muzycznej, która została zainicjowana przy pierwszym stworzeniu obiektu tej klasy. Nie musi koniecznie chodzić o zasób, może to być unikalny zbiór danych, który powinien być uporządkowany i wyjątkowy, ale dostęp do niego będzie potrzebny w bardzo wielu miejscach aplikacji, np. dane zapisanego stanu gry.

Należy pamiętać, że implementacje wzorców projektowych mogą się różnić od siebie i to bardzo. Wzorzec jest tylko ogólną ideą rozwiązania pewnego problemu. Sam sposób implementacji leży już w gestii programisty.

Wzorce projektowe są bardzo potężnym narzędziem, które może oszczędzić programiście wielu trudów z próbami wynalezienia rozwiązania dla problemów, które już dawno znalazły swoje optymalne rozwiązanie. Jednak nie można traktować ich jako metody na poradzenie sobie z każdym problemem projektowym w programowaniu obiektowym. Należy pamiętać, że wzorzec projektowy to jest konkretna aplikacja ogólnych zasad programowania obiektowego na potrzeby konkretnego problemu, nie należy starać się używać na siłę wzorców projektowych, tak by cały kod źródłowy wpisywał się w jakieś istniejące ramy. Każdy projekt jest inny i może się rządzić własnymi prawami,

wzorce projektowe należy traktować jako zestaw dobrych pomysłów, z których można czerpać i modyfikować na własne potrzeby w konkretnej aplikacji.

2.4 Style programowania

2.4.1 Formatowanie kodu

Sposób sformatowania kodu źródłowego jest wizytówką programisty. Prawidłowe formatowanie, które bezpośrednio przekłada się na większą przejrzystość kodu źródłowego, jest bardzo ważnym elementem programowania. Zwiększona przejrzystość kodu oznacza skrócony czas potrzeby na wdrożenie się w dany fragment na nowo, czy to przez autora danego kodu źródłowego po jakimś czasie, np. w celu naprawienia błędu, czy przy wdrażaniu innego programisty. Cytując Brucea Eckela, autora książki „Thinking in C++": „Elegancja zawsze się opłaca". Niestety zazwyczaj potrzeba trochę doświadczenia w programowaniu, żeby w to uwierzyć.

Dla każdego języka programowania istnieje kilka zasad związanych z formatowaniem kodu, niektóre z nich są wspólne, jak np. stosowanie wcięć, zależnie od zmiany zakresu. Znacznie poprawia to przejrzystość kodu i jest w zasadzie podstawowym wymogiem prawidłowego formatowania kodu. Przykład nieprawidłowo sformatowanego kodu:

```
int t[10][10];

for (int i~= 0; i~< 10; i++)
{
for (int j = 0; j < 10; j++){
t[i][j]
= 0;}
}
```

Jak widać, niezależnie od zagłębienia się zakresu ważności zmiennych w powyższym przykładzie każda linijka zaczyna się tym samym odstępem od początku akapitu. Każdy poziom zakresu powinien być odsunięty o jedną tabulację. Należy stosować konsekwencję - tu klamra otwierająca po instrukcji for raz znajduje się w tej samej linii, raz w nowej. Nie powinno być wielokrotnych odstępów pomiędzy liniami kodu, a wszystkie pojedyncze ciągi instrukcji powinny się znajdować w jednej linii, jeśli nie są za długie. Brakuje wreszcie komentarza objaśniającego działanie pętli. Przykład tego samego kodu, z uwzględnieniem poprawnego stylu i formatowania:

```
int aTab[10][10];

//set all matrix values to zero
for (int i~= 0; i~< 10; i++)
```

```
{
    for (int j = 0; j < 10; j++)
    {
        aTab[i][j] = 0;
    }
}
```

W języku potocznym często mówi się o prawidłowym „falowaniu" kodu w odniesieniu do wcięć. Pomiędzy językami występują drobne różnice w formatowaniu kodu, przykładowo w Java nawias otwierający `{` jest zostawiany zaraz po nazwie funkcji czy klasy, podczas gdy w C++ zazwyczaj przenosi się go do następnej linii. Nie ma tak naprawdę wielkiej różnicy w tego typu przypadkach, tak długo jak w całym obrębie kodu dany sposób stosowany jest konsekwentnie.

Pisząc o prawidłowym formatowaniu kodu, nie sposób nie wspomnieć tu o stosowaniu notacji. Poprzez notacje w językach programowania rozumie się pewien zbiór nazewnictwa takich elementów jak zmienne, nazwy klas, funkcje etc. Celem takiego zabiegu jest zwiększenie przejrzystości, zwłaszcza w przypadku kodu, z którego korzysta kilku programistów jednocześnie.

Jedną z najpopularniejszych stosowanych notacji jest tzw. notacja węgierska (ang. *Hungarian notation*) - choć nie tak popularna jak kiedyś, zwłaszcza z powodu dynamicznego rozwoju takich języków jak C# czy Java, gdzie jej zastosowanie jest niewielkie, nadal jest świetnym przykładem uporządkowanego stylu kodowania. Autorem notacji węgierskiej jest Charles Simonyi, który pracował w firmie Microsoft, gdzie jego notacja była szeroko stosowana w czasach świetności C/C++ w aplikacjach biznesowych. Microsoft odstąpił od promowania notacji węgierskiej w momencie wzrostu popularności języka C#.

CamelCase
Sposób zapisu poprzez łączenie kolejnych słów bez spacji, gdzie każde słowo pisane jest z wielkiej litery. Nazwa wzięła się od skojarzenia dużych liter w tak zapisanym wyrażeniu do garbów wielbłąda (ang. *Camel* - wielbłąd).

Notacja węgierska w nazwach zmiennych stosuje system prefiksów oznaczających typ oraz tzw. `CamelCase`: wszystkie człony nazw, bez żadnego rozdziału, zaczynają się z wielkiej litery (tak jak w wyrażeniu `CamelCase`). `CamelCase` stosuje się również w nazwach funkcji i klas. Przykładowo w notacji węgierskiej, deklaracja zmiennej typu *integer*, która odpowiadałaby za zliczanie samochodów, mogłaby wyglądać tak:

```
int iCarCounter;
```

Kolejnym elementem notacji węgierskiej jest dodanie przed prefiksem litery z podbiciem oznaczającej zakres zmiennej. Litera „m" oznacza pole klasy, litera „g" zmienną globalną, zaś litera „s" statyczne pole klasy. Przykład licznika samochodów będącego zmienną globalną:

```
int g_iCarCounter;
```

Notacja węgierska bardzo dobrze sprawuje się w języku C++ w technologiach opartych głównie na alokacji dynamicznej pamięci przy pomocy operatora *new*, gdzie przykładowo zmienna będąca wskaźnikiem na obiekt klasy `CarManager` wyglądałaby tak:

```
CarManager * pCarManager;
```

Jak widać wskaźnik w notacji węgierskiej oznacza się prefixem „p" (od ang. *„pointer"*).

W przypadku lokalnej alokacji pamięci pojawia się problem z gimnastyką słowną: jaki wymyślić prefiks dla typu klasy `CarManager`? Może „cm"? W takim razie, w jaki sposób odróżnić prefiks klasy `CarManager` od prefiksu klasy `CameraManager`? Coś, co miało pomagać w przejrzystości kodu, nagle go zaciemnia, bo wszystkie zmienne mają jako prefiks niezrozumiałe ciągi liter, wyglądające często bardzo podobnie. Ten właśnie problem sprawia, że w technologiach, gdzie głównie używana jest alokacja lokalna, lub w językach typu C#, gdzie w zasadzie prawie wszystko jest polem klasy i klasą jednocześnie, stosowanie notacji węgierskiej mija się z celem. Co nie oznacza, że można sobie pozwolić w tym momencie na dowolność. Styl programowania i notacje zawsze należy dostosować do języka i technologii, jednak zawsze powinny one być spójne. Przykładowo często stosowaną notacją w języku C# jest połączenie `CamelCase` z rozwiniętym prefiksem z notacji węgierskiej do pełnej nazwy klasy. Przykład zmiennej będącej przyciskiem służącym do zaktualizowania bazy danych w języku C#:

```
buttonDataBaseUpdate = new Button();
```

Jak widać, zakres został pominięty, jednak nazwa klasy została wpisana w nazwę zmiennej tylko małą literą. Może się to na pierwszy rzut oka wydawać nadmiarowe i niepotrzebne, jednakże po dłuższej pracy z takim zapisem jest bardzo pomocne. Okazuje się, że programista znacznie częściej pamięta typ zmiennej, której chce użyć, niż jej konkretną nazwę. We współczesnych środowiskach kodowania, gdzie mamy pomoc w postaci automatycznego uzupełnienia składni, przykładowo IntelliSense w środowisku *Visual Studio*, wystarczy wpisać słówko button, by wyświetliły nam się wszystkie zmienne zaczynające się od tej nazwy w danym zakresie formatki czy też klasy. Podczas pracy przy większych projektach jest to bardzo pomocne.

Ogólnie rzecz biorąc, nie ma notacji idealnej. Pracując na różnych technologiach, pisząc w różnych językach programowania, nie sposób posiadać jedną,

zawsze spójną notację. Ale nie zmienia to faktu, że konsekwentna notacja, w obrębie projektu lub technologii, bardzo pomaga. Ułatwia prace pojedynczemu programiście, który powracając do fragmentów kodu sprzed pewnego okresu czasu, łatwiej się w takim kodzie odnajduje, jednak przede wszystkim ułatwia prace całym zespołom programistycznym. Jeśli kilku programistów pracuje wspólnie nad jednym projektem, a w dzisiejszych czasach nie sposób napisać większości aplikacji samemu, taki sam styl programowania i notacji bardzo ułatwia pracę. Jeśli ktoś woli używać przykładowo podbić, zamiast systemu `CamelCase`, i nazywać swoje zmienne tak:

```
int variable_name;
```

to nie ma w tym nic złego, tak długo jak w swoim stylu i notacji jest konsekwentny i cały zespół do takowej notacji się stosuje.

Jaką notację stosować?

Zdecydowaną większość gier wideo pisze się w języku C++, dlatego notacja węgierska nadal pozostaje w wielu przypadkach aktualna. Mimo kontrowersji co do jej użyteczności, w pewnym zakresie zawsze będzie pomagała utrzymać przejrzystość kodu i porządek. Jedyną modyfikacją, jaką trzeba do niej wprowadzić, żeby używanie jej nie przysporzyło zbytnich kłopotów, to niewymyślanie prefixów do lokalnie tworzonych obiektów klas.

Wszystkie powyższe przykłady nazw klas, zmiennych oraz funkcji podano w języku angielskim. Nie jest to przypadek. W dzisiejszych czasach coraz częściej pracuje się w zespołach wielokulturowych, co wymaga ujednolicenia warstwy językowej, przynajmniej na poziomie kodu źródłowego, jeśli nie dokumentacji. Najczęściej stosowanym na świecie językiem w programowaniu jest oczywiście języki angielski, słowa kluczowe języka programowania praktycznie zawsze są w języku angielskim. Dobrą praktyką jest stosować język angielski zarówno we wszelkich nazwach, jak i w komentarzach kodu źródłowego.

Jednym z bardziej kontrowersyjnych tematów dotyczących stylu programowania jest komentowanie kodu. Z jednej strony wielu programistów, zwłaszcza młodych stażem w profesjonalnym programowaniu, komentarzy nie stosuje w ogóle lub w znikomej formie. Z drugiej strony mamy opinie akademicką inżynierii programowania na temat prawidłowej objętości komentarzy w stosunku jeden do dwóch do instrukcji kodu, co oznacza, że w przypadku optymalnym ponad jedna trzecia tekstu kodu źródłowego powinna być komentarzem. Kiedy programista po raz pierwszy słyszy prawidłowe proporcje instrukcji kodu do komentarzy, zazwyczaj reaguje powątpiewaniem. Stwierdza, że dążenie do tych proporcji na siłę jest sztuką dla sztuki i porzuca pomysł całkowicie. Jak się okazuje, wcale nie jest dobrym pomysłem starać się utrzymać stale tę optymalna proporcję. Kod należy komentować w sposób bardzo zrozumiały,

jednak zwarty, nie opisowy. Większość kodu źródłowego w postaci instrukcji powinna być tzw. „samokomentująca się". Oznacza to, że nazwy zmiennych i funkcji i sposób zapisu algorytmu powinien być na tyle jasny, by wymagał jak najmniej komentarzy. Przykład kodu źródłowego, który nie wymaga komentarza:

```
CCarFactory * pCarFactory = new pCarFactory();

CCar * pCar;

pCar = pCarFactory->CreateCar();
```

Same nazwy klas i funkcji oraz prostota wykonywanych operacji, czyli wywołanie jednego konstruktora i jednej funkcji, sprawiają, że komentarz do takiego kodu jest zbędny. Dodanie komentarza do tego typu kodu spowoduje niepotrzebną redundancję informacji, jak w przykładzie poniżej:

```
// creating a new car factory
CCarFactory * pCarFactory = new pCarFactory();

// declaration of a pointer to the car object
CCar * pCar;

// creating a car and assigning it to the pCar
pCar = pCarFactory->CreateCar();
```

Jak widać, komentarze te nie wniosły nic do przejrzystości kodu. Mówi się w takich przypadkach o zbędnym komentowaniu oczywistości. Natomiast kod poniżej bez żadnego komentarza jest bardzo mało czytelny:

```
for(i = 0; i~< m_vDrawableObjects.size() - 1; i++)
{
  t = i;
  for (j = i~+ 1; j < m_vDrawableObjects.size(); j ++)
  {
      if (m_vDrawableObjects[j].m_pObject->GetPriority()
          < m_vDrawableObjects[t].m_pObject->GetPriority())
          t = j;
  }
  swap(m_vDrawableObjects[t],m_vDrawableObjects[i]);
}
```

Chociaż składniowo kod jest bardzo prosty, a stosowane nazewnictwo przejrzyste, to na pierwszy rzut oka trudno stwierdzić, co tak naprawdę wykonuje dany fragment kodu. Wystarczyłoby jedno zdanie komentarza przed algorytmem, mówiące o tym, że kod odpowiada za sortowanie obiektów według priorytetów rysowania ich na ekranie, a przejrzystość kodu znacznie by wzrosła:

```
/ sorts the objects by their draw priority
```

Pytanie więc, skąd w takim razie bierze się tak wielki stosunek czystego kodu do komentarzy, skoro często wystarczy jedno zdanie, by wyjaśnić kilkanaście linijek kodu, w połączeniu ze stosowaniem odpowiedniego nazewnictwa, czyli kodu „samokomentującego się"? Odpowiedź jest prosta: najwięcej komentarzy wymagają metody i pola. Na przykładzie języka C++, plik nagłówkowy klasy powinien mieć stosunek komentarzy do kodu co najmniej jeden do jednego, czyli każde pole i metoda klasy powinny być skomentowane. Często te proporcje w plikach nagłówkowych są przesunięte nawet bardziej w kierunku komentarzy, gdyż jeśli dana metoda klasy przyjmuje argumenty, każdy z nich powinien być opisany w osobnej linijce komentarza.

Takie podejście do komentowania plików nagłówkowych jest zgodne z zasadą enkapsulacji kodu. Często programisty korzystającego z danej klasy nie interesuje, jak ona działa w środku i jeśli klasa jest dobrze napisana, zaś jej metody dobrze opisane poprzez prawidłowe nazwy oraz wyczerpujące komentarze, prawdopodobnie nigdy nie będzie musiał zajrzeć do pliku źródłowego. Komentarze konkretnej implementacji algorytmu najczęściej służą programiście, który sam ten kod stworzył. W momencie pisania kodu często wydaje się, że tok rozumowania jest prosty i zrozumiały, jednakże powrót do implementacji po kilku dniach lub tygodniach może być bardzo rozczarowujący pod względem przejrzystości kodu, jeśli nie jest on prawidłowo skomentowany. Tak więc okazuje się, że idealny stosunek kodu źródłowego do komentarzy nie jest odgórnie narzuconą i wymyśloną zasadą, a jedynie wywodzi się ze statystyki dobrze skomentowanych i przejrzystych kodów źródłowych. Dlatego nie należy na siłę dążyć do odpowiednich proporcji, które w wielu przypadkach wydają się abstrakcyjne, lecz starać się zwięźle i jednocześnie przejrzyście komentować własny kod, wtedy z czasem proporcje wyrównają się same.

Abstrahując od wszelkich notacji, zasad formatowania, komentarzy i dobrych praktyk z tym związanych, jest jedna podstawowa zasada w programowaniu, do której niestosowanie się grozi katastrofą w każdym większym projekcie. Nazwa zmiennej, klasy lub też funkcji **MUSI** jasno sugerować, do czego służy. W przeciwnym wypadku prędzej czy później, w kodzie nastanie chaos nie do opanowania - jeśli tylko mamy do czynienia z jakimkolwiek większym programem, niż szybki algorytm do napisania w funkcji main na potrzeby pojedynczego laboratorium na studiach.

Podsumowując, nie można poznać dobrego programisty po samym formatowaniu kodu i stosowanej notacji, ale można poznać słabego po niedostatkach w tym względzie. Jeśli kod ma odpowiednie wcięcia, konsekwentnie i sensownie nazwane zmienne i funkcje, to nie znaczy, że sam kod jest optymalny, a algorytm prawidłowy. Jednakże brak tych wszystkich elementów od razu świadczy o tym, że dany programista nie należy do najlepszych. Poniżej lista podsumowująca dobre praktyki w stylu programowania:

- Nazywać zmienne, klasy i funkcje tak, by jasno sugerowały, do czego służą;
- Prawidłowo i konsekwentnie formatować kod źródłowy („falowanie" kodu);

- Konsekwentnie stosować notację (dowolną);
- Często stosować komentarze;
- Używać języka angielskiego.

2.4.2 Programowanie monolitowe

Programowanie monolitowe polega na tworzeniu kodu źródłowego jako zwartej masy instrukcji kolejno wykonujących algorytm, bez podziału na moduły, jak też bez wydzielania funkcjonalności. Charakteryzuje się również częstym przeklejaniem kodu z drobnymi modyfikacjami. Styl programowania monolitowego bezpośrednio wynika z programowania strukturalnego. W takim stylu funkcjonalności do kodu dodawane są na bieżąco, w miarę powstałej potrzeby. Kod monolitowy zazwyczaj jest pochodną małego fragmentu algorytmu, który rozrósł się do pełnej aplikacji. Prawie każdy programista na początku swej kariery programuje monolitowo, czy to ucząc się samemu w domu podczas pierwszych prób uruchomienia prostego „hello word", czy też na studiach, mając za zadanie zaimplementowanie w krótkim czasie zajęć konkretnego algorytmu. Wraz z rozwojem aplikacji i dodawaniem funkcjonalności taki styl programowania oraz czas i wysiłek potrzebne na rozwój aplikacji stają się nieproporcjonalnie duże w stosunku do osiąganych efektów. Kolejnym problemem jest przekazanie pracy innemu programiście. Kod monolitowy po kilku iteracjach zmian i poprawek jest zazwyczaj bardzo mało czytelny, nawet jeżeli autor stosuje się do wszystkich zasad prawidłowego formatowania kodu. Trudno jest dostrzec zależności i ogólną zasadę działania poszczególnych algorytmów i komunikację pomiędzy komponentami, które zazwyczaj są bardzo mocno zazębione. Z powodu takiego zazębienia i ogólnie zwartej, monolitowej konstrukcji kodu, bardzo trudne, czasem wręcz niemożliwe, jest prowadzenie pracy równoległej na takim kodzie. Programowanie monolitowe można porównać do budowy domu. Po wylaniu fundamentów, zbudowaniu ścian i położeniu dachu bardzo trudne może być np. przesunięcie murowanego budynku o dwadzieścia metrów w prawo. W programowaniu monolitowym relatywnie proste życzenia klienta mogą przerodzić się w zadania o podobnym skomplikowaniu. Niestety, zjawisko programowania monolitowego również często występuje w profesjonalnych firmach tworzących oprogramowanie. Przyczyny zazwyczaj są dwie, małe doświadczenie programistów i ogromna presja ze strony kierownictwa czy też klienta na bardzo szybkie wyniki prac. Chociaż programowanie monolitowe faktycznie dosyć szybko pozwala pokazać pierwsze efekty, to na dłuższą metę duża aplikacja napisana w tym stylu jest bardzo trudna w utrzymaniu i rozwoju.

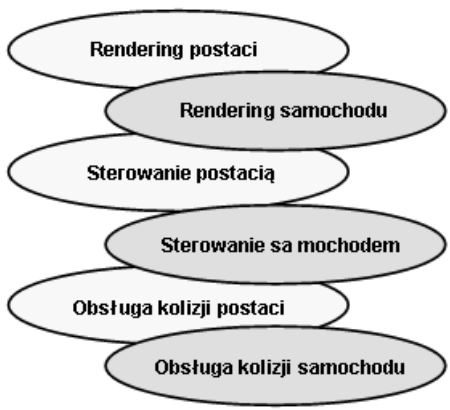
Żeby lepiej zrozumieć ideę programowania monolitowego, można sobie wyobrazić następujący przykład:

Niech założeniem danej aplikacji będzie stworzenie postaci, która będzie się renderować na ekranie, będzie można nią sterować oraz będzie ona obsługiwała kolizje z terenem.

Proces tworzenia tych funkcjonalności prezentowałby się liniowo, jedna po drugiej, jak pokazano na rysunku 2.5.

Rysunek 2.5. Kolejne elementy monolitowe procesu tworzenia postaci

Jeżeli po napisaniu takiego programu, programista otrzymałby zadanie dodania przy identycznych założeniach obiektu samochodu, wpinałby się w kolejnych miejscach, dodając jego obsługę. Jak pokazuje rys. 2.6, ilość kodu prawdopodobnie by się w takim przypadku podwoiła.



Rysunek 2.6. Dodanie obiektu samochodu do kodu monolitowego

Zalety programowania monolitowego:

- Szybkie pierwsze rezultaty dla małych aplikacji

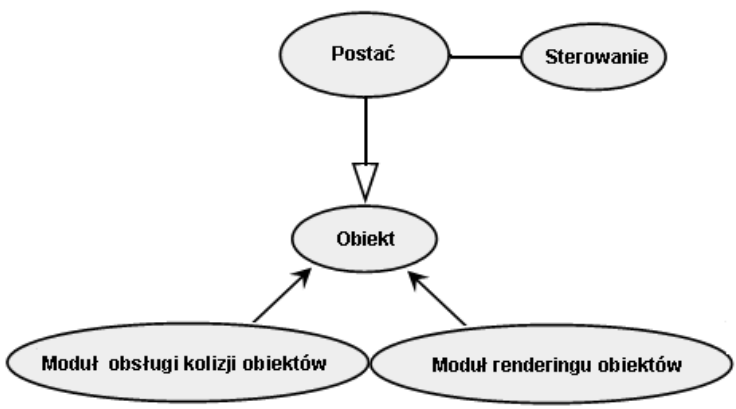
Wady programowania monolitowego:

- Bardzo utrudnione wprowadzenie zmian
- Nieproporcjonalny czas rozwoju do efektów
- Bardzo utrudniona możliwość równoległej pracy
- Bardzo mała przejrzystość kodu

2.4.3 Programowanie modułowe

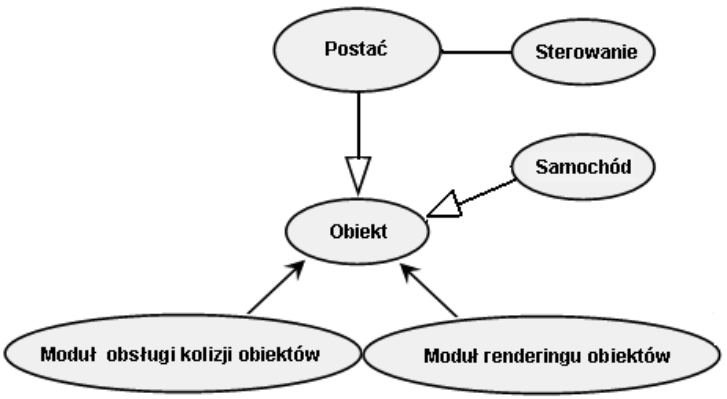
Programowanie modułowe jest naturalną pochodną języka obiektowego, połączonego z odpowiednim zaprojektowaniem kodu. Jak sama nazwa wskazuje, polega na tworzeniu niezależnych od siebie modułów, które potrafią się pomiędzy sobą komunikować. Zazwyczaj każdy moduł jest klasą, na którą mogą się składać inne klasy. Programowanie modułowe pozwala w dosyć łatwy sposób prowadzić równoległe prace programistyczne, zmiany jednego modułu bardzo rzadko oznaczają konieczność wprowadzania zmian w innych modułach. Jednak przy większym rozmiarze aplikacji czasem trudno jest dobrze zaprojektować architekturę kodu źródłowego. Posługując się przykładem z programowania monolitowego, łatwo byłoby przesunąć murowany dom, gdyby możliwość potrzeby takiego zabiegu była znana przed wylaniem fundamentów, można by wtedy np. pomyśleć o konstrukcji na szynach lub kołach. Klient, w przypadku branży gier - producent, będzie chciał zawsze wprowadzać zmiany do produktu (dlaczego tak się dzieje, wyjaśnione jest w rozdziale 3). Pytanie jednak, czy projekt architektury aplikacji te zmiany wytrzyma? Jeśli nie, to zaczyna się przebudowa architektury oraz przeróbka znaczącej części kodu, a wszystko odbywa się zazwyczaj pod sporą presją czasu, bez możliwości porządnego przeprojektowania całej aplikacji. Taki rozwój sytuacji prowadzi prostą drogą z powrotem do stylu monolitowego kodowania. Niestety, przewidzenie wszystkich możliwości rozwoju aplikacji jest bardzo trudne i wymaga sporego doświadczenia programisty w projektowaniu podobnego typu aplikacji.

Kolejnym problemem jest termin realizacji. Programista bardzo często musi działać pod presją czasu, porządne zaprojektowanie aplikacji modułowej wymaga go sporo, zaś efektów pracy projektowania długo nie widać. Rodzi to kolejny problem rozminięcia się wizji programisty i klienta. Jeśli klient widzi efekty pracy relatywnie późno, to nie ma absolutnie żadnej gwarancji, że to, co zrozumiał programista jako swoje zadanie, będzie tym, czego klient oczekuje.



Rysunek 2.7. Modułowy model przykładu aplikacji tworzącej postać

Posługując się przykładem postaci i samochodu z opisu programowania monolitowego, pierwszą fazę pokazuje rysunek 2.7. Widać już z samego zarysu, że stworzenie postaci zajmie prawdopodobnie więcej czasu niż w przypadku kodu monolitowego, gdyż wymagać będzie wyabstrahowania uniwersalnego mechanizmu renderingu oraz detekcji kolizji. Jednak już rysunek 2.8 pokazuje, że dodanie obiektu samochodu do tak zaprojektowanego kodu jest znacznie łatwiejsze niż w przypadku kodu monolitowego.



Rysunek 2.8. Dodanie obiektu samochodu do modelu modułowego aplikacji

Zalety programowania modułowego:

- Proste wprowadzanie zmian i dodawanie funkcjonalności
- Ułatwiona praca równoległa
- Duża przejrzystość kodu

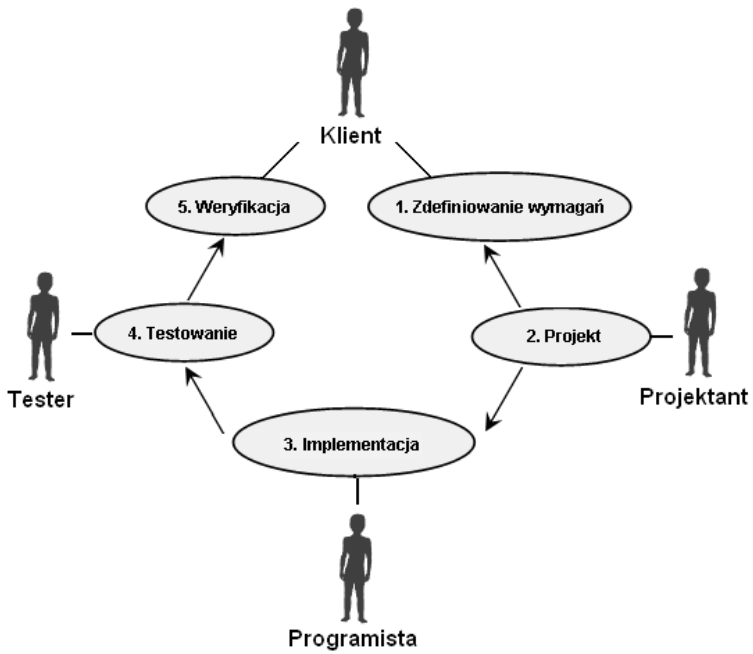
Wady programowania modułowego:

- Wymaga dużego doświadczenia w zaprojektowaniu
- Błędy w fazie projektowania mogą łatwo przekształcić projekt modułowy w monolitowy
- Relatywnie późne rezultaty prac
- Możliwość rozminięcia się oczekiwań z efektami w związku z późnym przedstawieniem rezultatów

2.4.4 Programowanie ekstremalne

Programowanie ekstremalne jest jednym z nurtów wywodzących się z tak zwanego programowania zwinnego (z ang. *AGILE development*). Temat zwinności w programowaniu jest opisany w rozdziale „Programowanie w systemie agile",

w tym miejscu większa uwaga zostanie poświęcona samemu zagadnieniu programowania ekstremalnego. Najważniejszą rzeczą w programowaniu ekstremalnym jest założenie, prawdziwe dla każdego projektu programistycznego, że klient będzie często wprowadzał zmiany w specyfikacji aplikacji - w przypadku gier producent czy projektant gry mogą sobie zażyczyć np. zupełnie innego modelu zniszczeń pod sam koniec projektu. Powody takich zmian są różne, najczęściej wynikają z tego, że dopiero po fizycznym kontakcie z aplikacją wychodzą braki w zaprojektowanej funkcjonalności. Dlatego też, w takim sposobie programowania ważne jest jak najszybsze stworzenie prototypu funkcjonalności. Przez prototyp nie jest rozumiany kod napisany szybko i niechlujnie, by uzyskać jak najwcześniej efekt, ale funkcjonalność okrojona ze wszelkich zbędnych na tak wczesnym etapie elementów dodatkowych. Jeśli przykładowo prototyp ma przedstawiać sterowanie postacią w grze, wiele elementów takich jak grafika terenu czy postaci może być uproszczona, przedstawiająca jedynie szczegółowość potrzebną do przetestowania samego sterowania. Pozwala to na ewentualne wprowadzenie modyfikacji już na bardzo wczesnym etapie, kiedy z programistycznego punktu widzenia wprowadzenie tych zmian jest nadal proste i nie narusza w żaden sposób architektury, która na tym etapie zazwyczaj nie jest jeszcze do końca określona.



Rysunek 2.9. Pojedyncza iteracja w programowaniu ekstremalnym

Najważniejszym mechanizmem działania w programowaniu ekstremalnym są iteracje, w cyklach iteracji mierzone jest życie projektu. Każda iteracja ma

za zadanie implementacje pewnego zestawu żądanych funkcjonalności i poprawek do istniejącego już kodu, po zakończeniu pracy programistycznej oprogramowanie poddawane jest intensywnym testom i pokazywane klientowi. Po wynikach testów i informacji zwrotnej uzyskanej od klienta przygotowuje się zestaw funkcjonalności i poprawek do kolejnej iteracji.

Takie jednak podejście może wygenerować nieskończoną ilość iteracji i projekt może nigdy nie osiągnąć końca. Dlatego spośród całego zbioru zadań na daną iterację wybierany jest relatywnie mały zestaw poprawek krytycznych oraz odpowiednie ułożenie według priorytetów pozostałych zadań. Następnie dla każdej iteracji określane są sztywne ramy czasowe - muszą one być na tyle rozsądne, by bez większego wysiłku zmieściły się w nich wszystkie zadania krytyczne. Pozostałe zadania, które nie zostaną zaimplementowane, przenosi się z ewentualną zmianą priorytetu do następnej iteracji lub w ogóle usuwa z projektu. Takie podejście wyznacza jakość produktu definiowaną poprzez czas, jaki mu poświęcono oraz umiejętności zespołu. O ile w programowaniu ekstremalnym nie można stwierdzić dokładnie, jaki zestaw funkcjonalności będzie miał finalny produkt, to w prawidłowo poprowadzonym w ten sposób projekcie pewne są dwie rzeczy: produkt będzie gotowy zawsze na czas i będzie posiadał co najmniej minimalną wymaganą przez klienta funkcjonalność. Brzmi to na pewno zachęcająco, jednak programowanie ekstremalne niesie ze sobą spore ryzyko, jeśli nie jest umiejętnie stosowane. Ciągłe zmiany w kodzie mogą doprowadzić do ogromnych problemów architektonicznych i ogólnego bałaganu od strony programistycznej, dlatego tak ważne są intensywne testy stabilności przy każdej iteracji i dobre przemyślenie architektury kodu, tak by była otwarta na modyfikację.

Największą zaletą programowania ekstremalnego jest częsty i bezpośredni kontakt z klientem. Częste iteracje wymuszają natomiast stały przepływ informacji wewnątrz zespołu i pomiędzy zespołem a klientem. Pozwala to na stworzenie aplikacji w znacznym stopniu spełniającej wszelkie wymagania i nie zostawia wiele pola dla pomyłek i błędnych interpretacji życzeń klienta przez programistę.

Posługując się kolejny raz przykładem aplikacji z postacią i samochodem, proces jej tworzenia w przypadku programowania ekstremalnego podzielony byłby na etapy. Sama architektura od razu przewidywałaby modułowość, jednak zależnie od priorytetów najpierw zrealizowany i przetestowany mógłby być rendering. Następnie mógłby zostać dodany samochód, a przykładowo sterowanie i kolizje na końcu. Na każdym z tych etapów aktualny stan aplikacji można by ocenić i aplikacja posiadałaby gotową konkretną funkcjonalność.

Zalety programowania ekstremalnego:

- Jasna wspólna wizja projektu wspólna dla całego zespołu
- Relatywnie najszybsze możliwe rezultaty
- Proste wprowadzanie zmian
- Stabilny kod

Wady programowania ekstremalnego:

- Z założenia nie wszystkie zaprojektowane funkcjonalności trafią do finalnego produktu
- Ryzyko przerodzenia się w monolit przy nieumiejętnym stosowaniu

2.4.5 Programowanie napędzane testami

Programowanie napędzane testami (ang. „*Test Driven Development*" - TDD) wywodzi się bezpośrednio z programowania ekstremalnego. Skoro w programowaniu ekstremalnym tak ważne są testy po każdej iteracji danego modułu oprogramowania, dlaczego nie stworzyć mechanizmów automatycznie wykonujących takie testy? Inżynieria programowania mówi o tzw. obwodach sterujących, czyli programach mających na celu zasilanie danymi aplikacji stworzonej przez programistę w celu przetestowania możliwie największego zbioru danych testowych, dla jakich znany jest prawidłowy wynik. Napisanie takiego programu tylko do celów testowych wiąże się oczywiście z pewnym narzutem czasowym, jednak zależnie od testowanego algorytmu, możliwość zautomatyzowania generowania danych testowych może być nieoceniona. TDD idzie o krok dalej, polega mianowicie na napisaniu programu testującego dany fragment kodu zanim przystąpi się do jego implementacji. Taki program nazywany jest testem jednostkowym, gdyż powinien testować relatywnie mały fragment kodu źródłowego, na poziomie przykładowo jednej metody czy też małej klasy. Testy jednostkowe tworzone przed samym napisaniem programu dla wielu programistów są czymś dziwnym. Jak się okazuje, jest to jednak bardzo efektywna forma programowania.

Stworzenie testu zanim napisze się algorytm wymusza automatycznie zastanowienie się, jak dany fragment kodu będzie używany, jaki dana klasa będzie miała interfejs oraz jakie metody będzie udostępniała. Ułatwia to znacznie projektowanie architektury kodu. Należy pamiętać, by przy stosowaniu tej metodologii implementować tylko te elementy funkcjonalności, które są aktualnie potrzebne, bez wychodzenia naprzeciw ewentualnym przyszłym potrzebom. Zasada ta bierze się stąd, iż TDD zajmuje więcej czasu niż tradycyjne programowanie, tak więc implementacja nadmiarowa może okazać się większą niż zazwyczaj stratą czasu, jeśli dany fragment kodu nie będzie nigdy wykorzystywany.

Schemat programowania przy użyciu testów jednostkowych wygląda następująco:

- Stworzenie testu jednostkowego
- Stworzenie pustej architektury kodu testowanego
- Uruchomienie testów jednostkowych i sprawdzenie, czy wszystkie wyniki są błędne
- Implementacja kodu

- Uruchomienie testów jednostkowych i sprawdzenie, czy wszystkie wyniki są poprawne, ewentualne naprawienie wynikłych błędów

Poniżej znajduje się prosty przykład funkcji podnoszącej argument do potęgi drugiej oraz testu jednostkowego napisanego do niej w języku C#:

```
// returns the square of a~given float
   static float Square(float fBase)

   {
     return fBase * fBase;
   }


/// <summary>
/// A test for Square
/// </summary>
     [TestMethod()]
     [DeploymentItem("UnitCaseTest.exe")]
     public void SquareTest()

   {
     float fBase = 10F; // TODO: Initialize to an appropriate value
     float expected = 100F; // TODO: Initialize to an appropriate value
     float actual;
     actual = Program_Accessor.Square(fBase);
     Assert.AreEqual(expected, actual);
   }
```

Jak widać, test ten uruchamia funkcję Square dla przykładowego argumentu 10 i sprawdza, czy wynik równa się 100.

Największa zaleta TDD ujawnia się przy wprowadzeniu modyfikacji do kodu. Wprowadzone zmiany do kodu źródłowego mogą być natychmiast przetestowane przez poprzednie testy jednostkowe. Jeżeli w dowolnym teście zostanie zgłoszony błąd, zazwyczaj oznacza to, że wprowadzona zmiana wywołała błędne działanie aplikacji.

Programowanie napędzane testami nie jest jednak bez wad. Przede wszystkim dla niewprawnego w takim sposobie pisania aplikacji programisty jest to metoda bardzo mało intuicyjna i uciążliwa. Trzeba również sporego doświadczenia oraz wyobraźni by napisać dobry zbiór testów jednostkowych, co np. dla aplikacji odpowiedzialnych za obliczenia matematyczne jest relatywnie proste, jednak już dla zagadnień związanych z wyświetlaniem efektów graficznych na ekranie nie jest takie łatwe. Przy pierwszych próbach użycia TDD narzut czasowy zazwyczaj jest bardzo duży, wiąże się to również z koniecznością użycia osobnych narzędzi, chociaż wiele środowisk programistycznych obecnie udostępnia wsparcie dla pisania testów jednostkowych. Zazwyczaj rozwiązane jest to na poziomie testowania pojedynczej klasy.

Niezależnie jednak od skomplikowania przy wdrażaniu tej metody programowania, w środowisku programistycznym jest wiele głosów mówiących, że jest to przyszłość profesjonalnego programowania w każdej dziedzinie, czy aplikacji biznesowych, czy też gier wideo.

Zalety TDD:

- Bardzo wysoka stabilność kodu
- Natychmiastowe wychwycenie błędów przy wprowadzeniu zmian do kodu
- Uproszczone zadanie projektowania architektury kodu

Wady TDD:

- Wymaga dużego doświadczenia i odpowiedniego przeszkolenia programisty
- Narzut czasowy tworzenia testów jednostkowych może być nieproporcjonalny do korzyści dla niektórych przypadków
- Wymaga korzystania z dodatkowych narzędzi

3

Programowanie gier

3.1 Dlaczego ten typ aplikacji jest szczególny

Wyłączając oprogramowywanie symulacji fizycznych działających w czasie rzeczywistym oraz zaawansowanych technologii wojskowych takich jak samoloty sterowane przez sztuczną inteligencję, programowanie gier wideo jest jednym z najbardziej skomplikowanych programistycznie zagadnień na świecie. Mało który typ aplikacji wymaga skupienia tak wielu różnych rozwiązań w jednym miejscu: grafika, dźwięk, sieć, urządzenia wejścia i zaawansowana algorytmika. Dodatkowo gra wideo najczęściej operuje w czasie rzeczywistym, co znacznie zwiększa jej skomplikowanie z punktu widzenia wydajności. Wszystkie te elementy muszą sprawnie ze sobą współpracować w aplikacji liczącej zazwyczaj kilkaset tysięcy linii kodu źródłowego.

Definicja 3.1. IP

Intelectual Property (pol. Własność intelektualna) - pojęcie szersze, jednak w środowisku twórców gier wideo w ten sposób określa się zestaw elementów wymyślonych na potrzeby konkretnej gry (postacie, świat, styl graficzny), często wykorzystywany wielokrotnie, przykładowo przy seriach gier (np. *Final Fantasy*) lub grach rozgrywanych w jednym świecie (np. Gwiezdnych Wojen).

Programista może natknąć się na krańcowo różne problemy zależnie od gatunku tworzonej gry. Gra traktująca o wyścigach samochodowych będzie wymagała znajomości innych algorytmów niż strategia czasu rzeczywistego. Wiele firm decyduje się na tworzenie kolejnych części istniejących tytułów zamiast tworzenia nowych zupełnie innych gier nie tylko ze względu na wartość marketingową ich IP, ale też ze względu na zaplecze technologiczne, jakie powstało podczas tworzenia poprzedniej części wraz z ogromnym bagażem doświadczeń.

Programista, który chce poświęcić się programowaniu gier musi zdawać sobie sprawę, że bardzo często będzie wymagana od niego wiedza z zakresu wielu różnych problemów programistycznych. Mało tego, często będzie wymagana wiedza dalece odbiegająca od informatycznej, potrzebna do prawidłowego zaprojektowania algorytmów. Przykładowo może to być wiedza z zakresu fizyki prowadzenia samolotu czy wiedza o wpływie wiatru na trajektorię kuli snajperskiej.

Kolejny problem charakterystyczny dla gier wideo to utrzymanie aplikacji, czyli wprowadzanie poprawek i zmian po faktycznym sprzedaniu produktu. Często nie ma możliwości wprowadzania zmian do gry wideo po jej wydaniu. Owszem, większość platform przewiduje systemy wprowadzania łatek do projektu, ale takie przypadki z przyczyn kosztowych zazwyczaj są ograniczane do niezbędnego minimum i rzadko kiedy zmiany wprowadzane przez łaty w jakikolwiek sposób modyfikują samą mechanikę gry, raczej tylko jej stabilność.

Większość aplikacji robionych na zamówienie ma konkretnego klienta, który jest jego odbiorcą. Taki klient staje się zazwyczaj częścią zespołu, na każdym etapie tworzenia aplikacji ma wgląd w jej aktualny stan, i może zgłaszać swoje uwagi, tak aby projekt końcowy jak najlepiej odzwierciedlał jego oczekiwania. W przypadku gier wideo, czyli produktów, które trafiają na półki sklepowe albo są do ściągnięcia poprzez dystrybucję elektroniczną, bezpośredni kontakt z klientem końcowym - czyli graczem - jest utrudniony. Często zdarzały się przypadki, że gra była tworzona pieczołowicie, dopracowana w każdym aspekcie, jednak jej wydanie nie trafiło w dobry moment rynkowy i sprzedaż okazała się fatalna. Aby lepiej wyczuć oczekiwania graczy i poznać ich opinie, korzysta się z testów fokusowych. Jednak w zespole tworzącym grę komputerową osobą odpowiedzialną za całokształt gry wideo jest producent. Jest to osoba, która w oczach zespołu jest właśnie klientem. Dlatego producent, jak każdy klient, zgłasza uwagi do produktu, tak by jak najlepiej odzwierciedlał jego wizję.

Zmiany będą zawsze

Jest to prawda uniwersalna, niezależnie od tego, czy tworzona aplikacja jest grą wideo, czy jakimkolwiek innym projektem. Programista musi być przygotowany na dosłownie KAŻDĄ zmianę w projekcie. Wymaga to odpowiedniego zaprojektowania architektury aplikacji. Zmiany będą wprowadzane przez klienta zawsze z prostego powodu: klient nie wie do końca, czego chce. Dopiero, kiedy zobaczy jakiś fragment funkcjonalności, to jest w stanie z większą dozą pewności stwierdzić, czy dane rozwiązanie mu się podoba, czy też nie. Dlatego tak ważne jest szybkie prototypowanie i dlatego programowanie zwinne jest zazwyczaj dużo skuteczniejsze od kaskadowego.

Powyższe zestawienie elementów wyróżniających tworzenie gier wideo od innych rodzajów aplikacji z pewnością nie jest kompletne, ale pozwala zrozumieć, jak poważnym przedsięwzięciem jest tworzenie gier wideo. Jednak elementem, który szczególnie je wyróżnia, jest pasja osób zaangażowanych w ten proces. Na pewno istnieją na świecie osoby, które są zafascynowane tworzeniem baz danych na potrzeby wielkich korporacji finansowych, jednak wielu programistów traktuje tego typu prace jedynie jako sposób zarabiania na życie. Programowanie gier wideo wiąże się z procesem twórczym, który jeśli tylko połączony jest z pasją do gier, daje w efekcie programiście niesamowitą satysfakcję i sprawia, że mimo niewątpliwego ciężaru obowiązków, jaki tego typu praca niesie, często jest po prostu przyjemnością.

3.2 Platformy i języki programowania

Pojęcie gry wideo jest bardzo szerokie, obejmuje gry pisane na każdą istniejącą platformę, od telefonów komórkowych, po duże konsole i komputery osobiste. Poniżej znajduje się zestawienie aktualnie najpopularniejszych istniejących na rynku platform wraz z krótkim opisem oraz językiem programowania używanym do tworzenia na daną platformę gier.

Definicja 3.3. SDK

Software Development Kit (pol. Zestaw do tworzenia oprogramowania) jest to zestaw narzędzi udostępnionych programiście w celu tworzenia oprogramowania pod konkretną platformę lub system operacyjny. Składa się zazwyczaj z skompilowanych bibliotek z plikami nagłówkowymi i dokumentacji z przykładami. Zależnie od rodzaju licencji może mieć również udostępnione pliki źródłowe.

Platformy stacjonarne:

- *Playstation 3* - konsola siódmej generacji firmy Sony, następczyni Playstation 2. Pod względem parametrów aktualnie najpotężniejsza konsola na rynku, dodatkowo posiada wbudowany czytnik płyt Blu-ray. Gry programuje się na SDK dostarczanym przez Sony, w oparciu o język C++.

  Parametry techniczne:

Procesor	3.2GHz Cell z 7 pojedynczymi elementami procesorowymi SPE
Koprocesor graficzny	550 Mhz RSX, obsługa HD (1080p)
Pamięć	256MB * pamięci XDR DRAM i 256MB pamięci GDDR3 RAM
Nośnik danych	Blu-ray

- *Xbox 360* - konsola siódmej generacji firmy Microsoft, następca Xboxa. Gry programuje się na SDK dostarczanym przez Microsoft, opartym na DirectX 9 w języku C++. Rozwiązanie to pozwala tworzyć gry praktycznie jednocześnie na PC i Xboxa 360, z drobnymi modyfikacjami.

  Parametry techniczne:

Procesor	3.2 GHz PowerPC Xenon z trzema rdzeniami
Koprocesor graficzny	500 Mhz ATI Xenos, obsługa HD (1080p)
Pamięć	512MB GDDR3 RAM
Nośnik danych	DVD

- *Wii* - konsola „prawie" siódmej generacji firmy Nintendo, posiada parametry bardzo zbliżone do poprzedniej konsoli tej firmy - GameCube'a, dlatego znacznie odbiega parametrami od konsol konkurencji. Jednak akcelerometr i sensor optyczny wbudowane w główny kontroler zwany Wii Remote oraz uplasowanie produktu na rynku masowym (tzw. casual gaming) sprawiło, że konsola swą popularnością znacznie prześcignęła konkurencję. Gry tworzy się w oparciu o SDK zwanym Revolution, w języku C++.

  Parametry techniczne:

Procesor	729 MHz PowerPC „Broadway" IBM
Koprocesor graficzny	243 Mhz ATI „Hollywood", obsługa 576i i 480p
Pamięć	64 MB GDDR3, 24MB 1T-SRAM i 3MB eDRAM
Nośnik danych	Wii Disc

Definicja 3.4. Casual game

Casual game (pol. gra rekreacyjna) jest to gra przeznaczona dla masowego odbiorcy. Cechuje się prostotą rozgrywki, bardzo krótkim czasem nauki oraz możliwym krótkim czasem pojedynczej rozgrywki.

- *PC* - komputer osobisty. Gry na tę platformę powstają od początku jej istnienia. Choć rynek gier wideo w ostatnich latach został w znacznym stopniu zdominowany przez konsole, głównie z powodu ogromnego piractwa na rynku PC, nadal jest to ważna platforma, zwłaszcza z punktu widzenia programistycznego, gdyż jest to zazwyczaj pierwsza platforma, z jaką programista ma kontakt. Gry na PC pisze się przy użyciu wielu języków i wielu różnych SDK, jednak najpopularniejszy zestaw w profesjonalnym programowaniu gier komputerowych to język C++ oraz biblioteka DirectX. Mowa tu oczywiście o większych produkcjach. Gry internetowe pisane przykładowo pod przeglądarki mogą być tworzone w różny sposób jako aplety JAVA, we FLASHu itp.

Platformy przenośne:

> **Generacje konsol**
> Ogólnie przyjęte zostało dzielenie dużych konsol, czyli nieprzenośnych, na „generacje". Związane jest to z tym, że wypuszczeniu nowej konsoli na rynek zazwyczaj towarzyszyło kilka konsol konkurencji w miarę zbliżonych technologicznie. Każda taka fala konsol nazywana była generacją. Aktualna generacja konsol zaczęła swój żywot w 2005 roku, wraz z wypuszczeniem na rynek przez Microsoft konsoli Xbox 360, wkrótce, w 2006 roku, Sony rozpoczęło sprzedaż Playstation 3, a Nintendo - konsoli Wii. Te trzy konsole zaliczane są do siódmej generacji. Pierwszą generację konsol zapoczątkowało pojawienie się na rynku konsoli Odyssey firmy Magnavox w roku 1972.

- *DS* - Konsola przenośna firmy Nintendo, podobnie jak Wii, odbiega od konkurencji pod względem parametrów technicznych. W zamian za to oferuje podwójny ekran, jeden z nich dotykowy, oraz w nowych wersjach (DSi i DSi XL) dwie kamery. Na DS programuje się w języku C++ z użyciem SDK o nazwie NITRO (w nowych wersjach TWL).

  Parametry techniczne (DSi):

Procesor	134 MHz ARM9 i 33 MHz ARM7
Pamięć	16MB RAM i 656kB VRAM
Nośnik danych	Nintendo DS/DSi Game Card

- *PSP* - konsola przenośna firmy Sony, na którą programuje się przy użyciu SDK o nazwie PSPSDK w języku C++. Wszystkie wersje PSP poza PSP Go, używają jako nośnika danych specjalnej płyty, stworzonej przez Sony na potrzeby tej konsoli, o nazwie UMD (od ang. *Universal Media Disc*). Aktualnie na rynku są trzy takie wersje (PSP-1000, PSP-2000 i najnowsza PSP-3000).

  Parametry techniczne (PSP-3000):

Procesor	333 MHz MIPS R4000
Pamięć	64MB RAM i 4MB VRAM
Nośnik danych	UMD

- *iPhone* - choć początkowo traktowany przez środowisko developerskie bardziej jako smartfon niż platforma do grania, to liczba gier, jaka na niego powstała, szybko sprawiła, że stał się pełnoprawną platformą do gier przenośnych. Posiada ekran dotykowy i akcelerometr. W założeniu SDK przystosowane jest do pisania na niego aplikacji w języku Objective C, jednak nic

nie stoi na przeszkodzie, żeby używać do tego C++. Wyszło kilka kolejnych wersji tego urządzenia (iPhone 3G, iPhone 3GS, iPhone 4G) różniących się między sobą nieznacznie wyglądem, jednak znacznie parametrami. Oprócz telefonów iPhone istnieje cała gama urządzeń działających pod kontrolą tego samego systemu operacyjnego iOS i będących kompatybilnymi platrformami do gier. Każda wersja iPhone ma swój odpowiednik w postaci iPod'a Touch, który jest praktycznie identycznym urządzeniem, tylko bez funkcjonalności telefonu. Oprócz tego Apple produkuje urządzenie o nazwie iPad, które jest urządzeniem typu tablet pracującym na tym samym systemie operacyjnym.

Parametry techniczne (iPhone 4G):

Procesor	1 GHzARM Cortex-A8
Koprocesor graficzny	200 MHz PowerVR SGX 535
Pamięć	512MB eDRAM

- Inne - na rynku istnieje wiele innych platform, na które pisze się gry, zdecydowana większość z nich to różnego rodzaju telefony. SDK do pisania na nie aplikacji jest udostępnianie przez producenta i może być przystosowane do różnych języków, zależnie od systemu operacyjnego urządzenia. Najpopularniejszym językiem w tym wypadku jest JAVA (np. platforma Android), w drugiej kolejności C++.

Jak widać, bez znajomości języka C++ bardzo trudno będzie się programiście odnaleźć w programowaniu gier wideo, gdyż jest to natywny język dla większości platform.

Każda platforma posiada swoje własne SDK i rządzi się swoimi prawami. Choć sposób rozwiązania problemów w różnych SDK może być inny, to cel jest wszędzie taki sam: wyświetlenie grafiki, odegranie dźwięku, przyjęcie komunikatu z kontrolera wejścia. Mimo iż te zagadnienia mogą być bardzo skomplikowane, to lata doświadczeń środowiska twórców gier i sprzętu sprawiły, że bardzo wiele rozwiązań zostało ustandaryzowanych i są one wspólne między platformami. Przykładowo SDK pod platformy Nintendo zarówno Wii, jak i DS mają bardzo podobną architekturę do biblioteki OpenGL. Dlatego jeśli ktoś miał doświadczenie z tą biblioteką komputerową, bez większych problemów poradzi sobie z opanowaniem bibliotek Nintendo.

3.3 Programowanie gier a techniki zarządzania

Początki komercyjnych produkcji gier wideo, czyli pierwsze lata siedemdziesiąte ubiegłego wieku, były relatywnie proste z punktu widzenia zarządzania. Najważniejszy w tamtych czasach był dobry pomysł oraz umiejętności. Plat-

formy sprzętowe miały bardzo ograniczone możliwości, zaś gracze mieli niewielkie oczekiwania względem gier. Dzięki temu jeden programista z jednym grafikiem i jednym muzykiem mogli napisać naprawdę konkurencyjną na rynku grę. Co więcej, często w zespołach pewne role były obsadzane przez tę samą osobę. Przykładowo grafik był również programistą, a designer muzykiem itd. W takim zespole nie potrzeba wiele, by kontrolować stan projektu, a komunikacja jest ułatwiona. Jednak w dzisiejszych czasach mamy do czynienia z dużo bardziej skomplikowanym sprzętem oraz dużo większymi oczekiwaniami graczy. Zespoły tworzące gry składają się z kilkudziesięciu osób i choć programiści należą do raczej mało licznej grupy, to nie da się ukryć faktu, że dla sprawnego przebiegu tworzenia takiej gry osoby odpowiedzialne za projekt muszą stosować pewne techniki zarządzania.

! Definicja 3.5. Projekt a proces

Definicja projektu wg PMI (ang. *Project Managment Institute*) jest następująca: „Projekt to tymczasowe przedsięwzięcie podejmowane w celu wytworzenia unikalnego wyrobu lub dostarczenia unikalnej usługi. Przez tymczasowość należy rozumieć więc, iż każdy projekt ma swój początek i koniec". Proces natomiast jest zbiorem działań, które rozwiązują dany problem lub prowadzą do osiągnięcia konkretnego efektu. Charakterystyczny dla procesu jest zawsze ten sam zbiór czynności i ten sam efekt. Każde oprogramowanie, zwłaszcza gra wideo, jest projektem, gdyż posiada pewne unikalne elementy. Im więcej elementów unikatowych w projekcie, tym większe ryzyko przy wycenie czasowej i budżetowej takiego projektu. Między innymi, dlatego tak wiele twórców gier trzyma się pewnych utartych schematów.

Zarządzanie projektem polega głównie na zorganizowaniu pracy specjalistów z różnych dziedzin, by wspólnie użyli swoich umiejętności do stworzenia jakiegoś unikalnego produktu. Słowo „unikalnego" jest tutaj kluczowe: bowiem gdyby każda gra wideo tworzona przez daną firmę wyglądała identycznie, to już nie byłaby projektem, a procesem, jak np. zbijanie drzwi na taśmie w stolarni. Poniżej znajduje się zestawienie specjalistów, którzy zazwyczaj pracują przy tworzeniu gry wideo:

- **Programiści** - projektują i implementują kod samej gry oraz ewentualnie narzędzia do jej tworzenia. Choć fakt, jak ważna jest ich rola, nie budzi żadnych wątpliwości, nie jest to zazwyczaj liczna grupa. W największych zespołach tworzących grę wideo programiści rzadko stanowią więcej niż dziesięć procent. Dzieje się tak, ponieważ programiści są bardzo mocno wzajemnie uzależnieni od wyników swojej pracy i tym samym, im więcej jest ich w zespole, tym większe są kłopoty komunikacyjne i wzajemnej zależności. Dalszy rozrost zespołu programistycznego powodowałby, iż problemy ko-

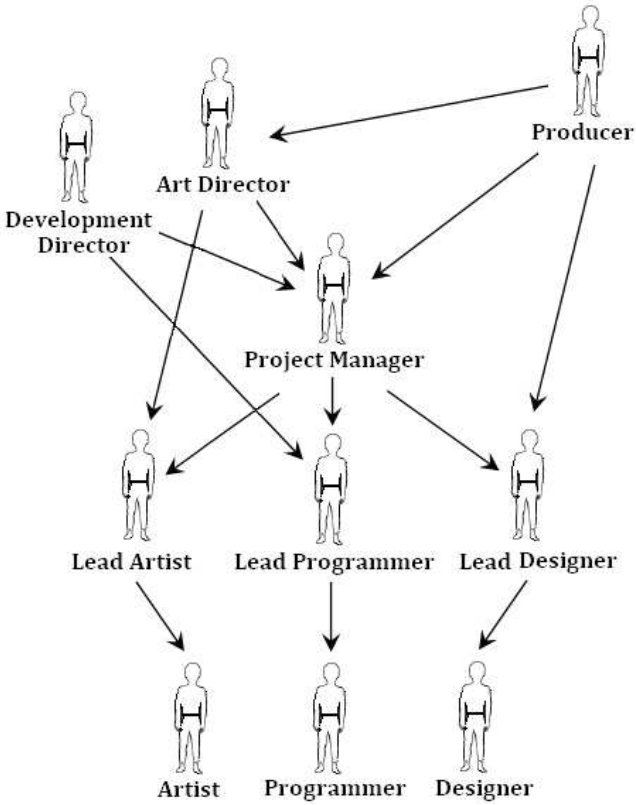
munikacyjne oraz rosnące zależności wprowadziłyby narzut czasowy, który zniwelowałby oszczędność uzyskaną z zatrudnienia większej liczby osób do tego samego celu. Dlatego w dużych zespołach istnieją czasem mocne specjalizacje, przykładowo: programista fizyki lub sztucznej inteligencji. Najbardziej podstawowy podział jest następujący:

- *Engine programmer* - programista silnika gry. Jego praca często jest w mocnym oderwaniu od samej gry. Dostarcza programiście gry pewną potrzebną funkcjonalność, zazwyczaj na tyle ogólną, że obejmuje cały gatunek tworzonej gry wideo. Są to często programiści o mocnym zacięciu niskopoziomowym, gdyż często pracują bezpośrednio na sprzęcie, a najczęściej na SDK platformy.
- *Game programmer* - programista odpowiedzialny za implementację zawartości gry. Mniej zajmujący się technicznymi aspektami projektu, a bardziej funkcjonalnością gry.
- *Tool programmer* - odpowiedzialny za implementację wszelkiego rodzaju narzędzi usprawniających tworzenie gry wideo, od konwerterów grafiki po edytory gry.
- *Lead programmer* - zazwyczaj programista z największym doświadczeniem w zespole. Jego rola jest nie tyle kierownicza, co polega na pilnowaniu poziomu architektury i kodu w projekcie. Jest również odpowiedzialny za jasne przedstawienie technicznych możliwości programistów designerom, grafikom i wszystkim innym zainteresowanym.
- *Development director* - odpowiedzialny za wszystkie zespoły programistyczne w danej firmie. Wyznacza pewne technologiczne trendy, jak decyzje o stworzeniu wewnątrzfirmowej technologii lub zakupieniu konkretnego rozwiązania z zewnątrz, np. silnika fizycznego. Często kieruje się opiniami programistów, jednak konfrontując je już z czysto managerskimi zagadnieniami, takimi jak koszta, jakie może ponieść firma, ryzyko itp.

- **Artyści** - odpowiedzialni za stworzenie całości grafiki i dźwięku do gry. Ich rola w branży gier wideo jest ogromna, ponieważ już sama doskonała grafika i udźwiękowienie może przesądzić o sukcesie gry, nawet przy relatywnie niskiej grywalności. Zdecydowanie najbardziej liczna grupa uczestnicząca przy tworzeniu gry wideo. Artyści mogą się dzielić na wiele różnych typów:
  - *Concept artist* - tworzy szkice do gry, przeważnie we wczesnej fazie jej powstawania, na potrzeby stworzenia wizji, jak będzie wyglądała gra.
  - *3D modeler* - tworzy geometrię do gry, zazwyczaj dzielącą się na dwa rodzaje: statyczną (np. budynki, teren) oraz dynamiczną (np. postacie, pojazdy).
  - *Texture/2D artist* - w grach 3D odpowiada za tworzenie tekstur oraz elementów 2D np. UI. W grach 2D tworzy praktycznie całą zawartość graficzną.
  - *Lightning artist* - odpowiada za oświetlenie w grach 3D.

- *Animator* - jedna z najbardziej wymagających funkcji artystycznych. Odpowiada za animację modeli 3D. W przypadku nieużywania technologii motion capture jest to zadanie wysoce skomplikowane i trudne artystycznie przy wszelkich modelach postaci.
- *Motion capture actor* - osoba, z której pobierane są dane na temat jej ruchu na potrzeby stworzenia animacji postaci. Zależnie od rodzaju gry video, werbuje się w tym celu różne osoby, począwszy od osób z zespołu developerskiego, poprzez profesjonalnych kaskaderów, na mistrzach sztuki walk kończąc.
- *Sound artist* - osoba odpowiedzialna za stworzenie dźwięków do gry.
- *Voice actor* - podkłada głos pod postacie w grze.
- *Composer* - tworzy muzykę specjalnie dla konkretnej gry. Stosowany głównie w wysokobudżetowych produkcjach, w innych przypadkach kupuje się najczęściej licencję już istniejącej muzyki.
- *Lead artist* - wyłaniany w zespołach tworzących konkretny rodzaj zawartości gry, typu modele 3D lub dźwięki. Odpowiada w tym momencie za spójność oraz jakość produktów wytwarzanych przez jego zespół.
- *Art director* - odpowiada za spójność wrażeń artystycznych w grze, zależnie od rozmiaru firmy może czuwać nad więcej niż jednym projektem. Do wrażeń artystycznych zalicza się najczęściej jakość grafiki, jednak nie można nie doceniać roli, jaką pełni udźwiękowienie w tworzeniu klimatu gry.

- **Designerzy** - projektują zachowania gry, następnie ich wizja jest przelewana przez artystów i programistów na zewnętrzny efekt pewnego zestawu mechanizmów interaktywnych dla gracza. Designerzy pracują na różnych poziomach abstrakcji, a standardowy podział wśród nich jest następujący:
  - *Desinger* - najogólniejsza możliwa forma pracy nad projektem gry. Odpowiada za zaprojektowanie działania wszelkich mechanizmów w grze oraz przedstawienie go odpowiednim osobom, aby mogły te mechanizmy stworzyć.
  - *Writer* - designer, którego głównym zadaniem jest tworzenie fabuły do gry, dialogów oraz wszelkiego rodzaju dłuższych tekstów, np. fragmentów książek występujących w grze.
  - *Scripter* - znany też jako „level designer", zajmuje się projektowaniem gry na bardzo technicznym poziomie, np. rozłożeniem przeciwników na planszy w edytorze, oskryptowaniem scenariusza w danym miejscu gry w jakimś języku skryptowym (np. Lua), modyfikowaniem parametrów w celu zbalansowania rozgrywki na danym poziomie itp.
  - *Lead designer* - osoba odpowiedzialna w całości za ogólny projekt gry, ma zazwyczaj największy wpływ na wygląd gry obok producenta. Odpowiada za spójność wizji projektu. Często jest autorem pomysłu na samą grę.



Rysunek 3.1. Standardowy „łańcuch dowodzenia" w zespole gier wideo - część osób odpowiada bezpośrednio przed więcej niż jedną osobą

Definicja 3.6. Lua

Lua jest w pełni funkcjonalnym językiem programowania, jednak została zaprojektowana głównie jako język skryptowy. Bardzo szeroko używana w branży gier wideo do przechowywania logiki gry i wszelkiego rodzaju parametrów. Więcej informacji o językach skryptowych w rozdziale 6.1.1.3.

- *Producer* - główne ogniwo łączące zespół tworzący grę z osobami, które na stworzenie tej gry dają pieniądze, czyli przykładowo zarządem firmy. Jego głównym zadaniem jest opiniowanie postępów w grze i jej ogólnego odbioru, tak żeby gra spełniała narzucone jej wymagania, m.in.: gatunek, grupa docelowa, ogólna jakość. Patrzy na grę z trochę szerszej perspektywy niż pozostali designerzy, uwzględniając np. jej konkurencyjność rynkową.

- **Inni**
  - Wymienione powyżej osoby to ludzie mający bezpośredni wkład w zawartość projektu, jednak za nimi stoi cała rzesza ludzi obecnych w większości firm, bez których wykonanie gry wideo nie byłoby możliwe, od kierowników projektów, zarządzających całym zespołem i pilnujących terminów, poprzez tłumaczy, wsparcie IT, marketing, pomoc biurową, księgowość, dystrybucję itd. Przy dużych produkcjach może pracować nawet kilkuset ludzi, na kilkudziesięciu różnych stanowiskach.

Należy pamiętać, że powyższe zestawienie to tylko pewien ogólny schemat. Zależnie od firmy, obowiązki na danym stanowisku mogą się różnić, czasem nawet znacznie. Przykładowo w niektórych firmach obowiązki producenta opisane powyżej posiada główny designer, producent jest managerem, a managera nie ma w ogóle. Z uawgi na to, w wielu firmach w CV mile widziany jest opis obowiązków obok nazwy stanowiska, by nie było nieporozumień związanych z doświadczeniem kandydata na pracownika.

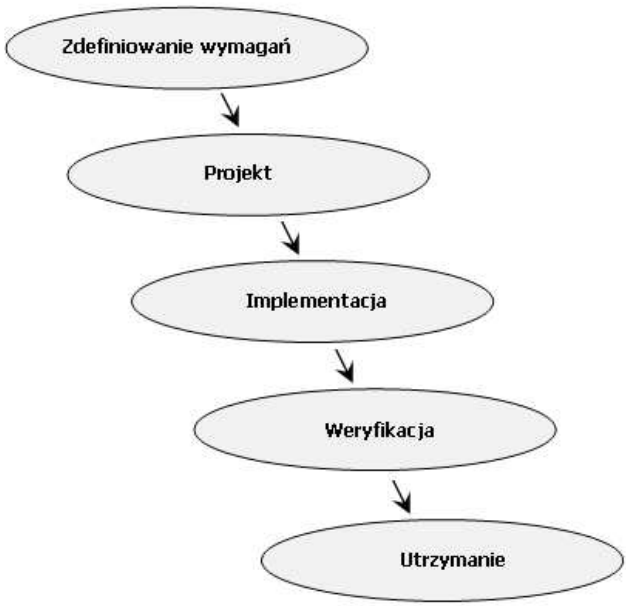
3.3.1 Wpływ rodzaju zarządzania na development

Na podstawie przedstawionej struktury osobowej firmy produkującej gry wideo można wysnuć wniosek, że zarządzanie tego typu projektem to nie lada wyzwanie. Mało który programista jednak zdaje sobie sprawę, że sposób zarządzania firmą, w której pracuje, będzie miał wpływ na jego sposób programowania.

3.3.2 Programowanie w systemie waterfall

Programowanie w modelu kaskadowym (ang. *waterfall*) polega na sekwencyjnym wykonywaniu kolejnych elementów składających się na tworzenie aplikacji (rys. 3.2). Jest to najbardziej intuicyjna kolejność programowania i zazwyczaj w taki sposób każdy programista początkowo podchodzi do swojej pracy. Wydaje się oczywiste, że należałoby zacząć od jasno zdefiniowanych wymagań, następnie zaprojektować na tej podstawie aplikację, zaimplementować ją, przekazać klientowi do zatwierdzenia i ewentualne zadbać o jej utrzymanie. Rzeczywistość jednak jest zazwyczaj znacznie bardziej skomplikowana, a model kaskadowy nie przewiduje np. zarządzania zmianami, co czyni z niego bardzo nieelastyczny model programowania. Kiedy klient widzi aplikację czy też nawet fragment funkcjonalności na etapie weryfikacji, zazwyczaj jest już zdecydowanie za późno na proste wprowadzanie zmian.

Zasady modelu kaskadowego mówią wtedy o ponownym zdefiniowaniu wymagań, wprowadzeniu zmian do warstwy projektowej, ponownej implementacji i weryfikacji. Taki scenariusz zazwyczaj jest jednak nierealny, gdyż nie ma po prostu na to czasu ani budżetu. Im później w projekcie nastąpią zmiany, tym większy będzie narzut związany z ich wprowadzeniem. W branży gier vi-



Rysunek 3.2. Fazy modelu kaskadowego

deo model kaskadowy jednak dobrze się sprawdza przy niewielkich powtarzalnych projektach, np. jeśli dana firma zajmuje się tworzeniem gier tylko z jednego gatunku, stworzyła już ich kilka, ma gotową technologię i wprowadza tylko do niej usprawnienia. Programowanie w tym modelu projektu całkowicie nowego, z całą technologią tworzoną od podstaw lub nieznaną zespołowi programistycznemu, może się skończyć w pewnym momencie koniecznością przepisywania dużych fragmentów gry, które powstały w początkowej fazie projektu.

3.4 Programowanie w systemie agile

Programowanie zwinne, to zbiór metodologii opartych na iteracyjnym i przyrostowym developmencie. Sposoby jego wdrożenia, takie jak np. programowanie ekstremalne czy TDD były opisane już w rozdziale 2. Od strony organizacji pracy i zarządzania istnieją też różne metody wprowadzania zwinności do projektów. Jednak by efektywnie zastosować wszystkie te narzędzia, należy najpierw odpowiedzieć sobie na pytanie, w czym tak naprawdę tkwi największy problem przy tworzeniu gier wideo i dlaczego Agile pomaga w jego rozwiązaniu. Problemem tym jest komunikacja. Absolutnie w każdej firmie, w każdym zespole, w każdej branży i przy każdym projekcie jest ułomna. Dzieje się tak dlatego, że brak perfekcyjnej komunikacji to problem na poziomie ludzkim. Klient, przykładowo producent, nawet jeśli wie dokładnie,

czego oczekuje, nie jest w stanie tego dobrze przekazać wykonawcy. Będzie szukał porównań, przykładów, wspomagał się szkicami, słowem, doświadczeniami z poprzednich projektów, jednak nigdy dokładnie nie przekaże drugiemu człowiekowi, czego chce. Tak naprawdę najczęściej sam tego do końca dokładnie nie wie - posiada pewną wizję, ale jest ona niekompletna i zazwyczaj nie do końca przemyślana pod każdym kątem. Jest to naturalne i zazwyczaj nie jesteśmy w stanie tego zmienić. Dlatego tak ważne jest prototypowanie, aby móc na każdym etapie projektu pokazać pewne funkcjonalności i od razu skonfrontować i zweryfikować je z oczekiwaniami klienta. W modelowo prowadzonym projekcie według metodologii Agile, oprogramowanie powinno być funkcjonalne i testowane od pierwszego dnia pracy nad nim.

W branży gier wideo bardzo często w czasie życia projektu trzeba pokazać aktualny stan prac. Może przykładowo pojawić się natychmiastowa potrzeba pokazania jakiejś wersji kontrahentowi w sieci sklepów lub wydawcy albo konieczne będzie nakręcenie zwiastuna i potrzebne będą materiały z gry do jego nagrania. Programista nie może pozwolić, by w takim momencie projekt nie kompilował się i na przykład usprawiedliwiać ten fakt trwającymi pracami nad ważnym modułem, które powodują niemożliwość kompilacji projektu przez najbliższe dwa tygodnie.

Skracanie linii komunikacji

Najczęściej programista otrzymuje polecenia bezpośrednio od jednej osoby, zazwyczaj od kierownika zespołu. Jednak wykonuje zadania pośrednio zlecane przez wiele innych osób: designera, producenta, managera, grafika. Warto dowiedzieć się, od kogo dane zadanie pochodzi i jeśli nie jest to wbrew obyczajom panującym w firmie i istnieje taka fizyczna możliwość, skonsultować się z daną osobą, w celu ujednolicenia wizji efektu danego zadania. Przykładowo, na pomysł poprawy jakiegoś efektu graficznego wpadnie grafik, przekaże to swojemu kierownikowi, ten, zależnie od rozmiaru zmiany, skonsultuje się ze swoim przełożonym lub od razu wystawi zlecenie kierownikowi programistów, który z kolei przekaże zadanie do implementacji konkretnemu programiście. Jeśli programista ma jakieś pytania lub wątpliwości, jak powinien wyglądać dany efekt, najlepiej, by bezpośrednio skontaktował się z grafikiem, który wpadł na ten pomysł. Nie w każdej firmie jest to technicznie możliwe, ale samą ideę skracania linii komunikacji powinno się wdrażać w możliwie szerokim zakresie.

Ciągłe prototypowanie i utrzymanie wersji uruchamiającej się gry ma jednak również inne, ważniejsze zastosowanie, o którym wspomniano wcześniej. Dzięki niemu można w prosty sposób poznać oraz ocenić, w którym kierunku zmierza gra. Wszyscy zainteresowani, np. producent czy designer mogą na bieżąco kontrolować, czy gra rozwija się w żądanym kierunku i na bieżąco zgłaszać uwagi i modyfikować projekt, co w zasadniczy sposób odróżnia ten model

od kaskadowego. Utrzymanie kodu w stanie użytkowym przez cały czas życia projektu to technika zwinna zwana „*Continous integration*" (pol. „ciągła integracja"). Jej bezpośrednia implementacja to zazwyczaj zautomatyzowany system budowania gry z najnowszego kodu. Takie wersje tworzone są najczęściej codziennie i przechowywane w jednym, ogólnodostępnym miejscu, np. na serwerze firmowym.

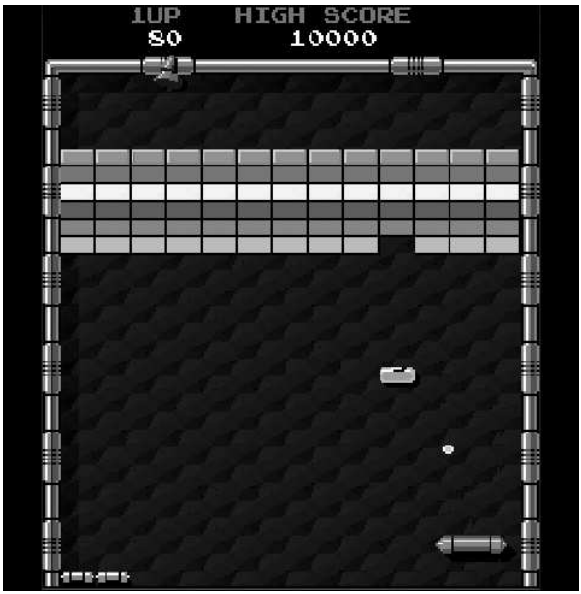
Abstrahując od wszelkich technik oraz procesów, z jakich może się składać w danej firmie programowanie zwinne, najważniejszy jest jeden przekaz tej metodologii: zmiany w projekcie będą zawsze i trzeba się do tego przystosować, najlepiej będąc zwinnym.

3.5 Warstwy w programowaniu gier

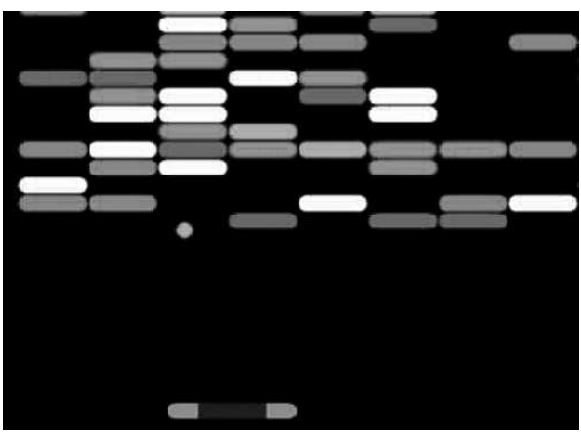
3.5.1 Wstęp

Pisząc grę po raz pierwszy, wielu programistów próbuje stworzyć kolejno elementy gry, dopasowując następny element do poprzedniego. Żeby dobrze zrozumieć ten proces, a przede wszystkim zdać sobie sprawę, co w nim jest nieprawidłowego, najlepiej będzie prześledzić przykład tworzenia małej gry. Zakładając, że zadaniem programisty byłoby stworzenie gry typu *Arkanoid* na platformę PC, jaki jest zwyczajowy schemat postępowania? Najczęściej programista najpierw chce stworzyć kod testowy, którego efekt zobaczy na ekranie. Takim „Hello Word" gier jest na ogół wyświetlenie w żądanym miejscu dowolnej grafiki. Na tym etapie programista musi się zdecydować na technologię, w jakiej chce stworzyć swoją grę. Jako że jest to prosta gra 2D, najlepszym sposobem będzie wykorzystanie jakiejś gotowej biblioteki udostępniającej pełną potrzebną programiście funkcjonalność. Może się np. zdecydować na bibliotekę Allegro. Studiując dokumentację lub, co bardziej prawdopodobne, przerabiając jakiś przykład znaleziony w Internecie, programista znajdzie wszystkie potrzebne linijki kodu do inicjalizacji biblioteki (na potrzeby tego przykładu pominięte zostały problemy związane z przygotowaniem środowi-

Arkanoid
Gra stworzona przez firmę Taitio w 1986 r., w oparciu o serie gier firmy Atari z lat 70 „Breakout". Gra polega na odbijaniu znajdującą się na dole planszy paletką piłki. Celem jest zbijanie znajdujących się w górnej części planszy klocków. Paletką można poruszać tylko w lewo lub w prawo. Choć sama idea została zapoczątkowana przez serię „Breakout", to nazwa „Arkanoid" jest szerzej rozpoznawalna jako typ gier arcade'owych opartych na mechanice oryginału.



Rysunek 3.3. Arkanoid



Rysunek 3.4. Breakout

ska, czyli podpięcie biblioteki Allegro do projektu w dowolnym GUI programistycznym z kompilatorem) oraz dowie się, jak wygląda kod odpowiedzialny za wyświetlenie grafiki na ekranie w pożądanym miejscu. Następną myślą programisty bywa najczęściej „Mam paletkę, jak ją teraz poruszyć?"

Ponownie za pomocą dokumentacji lub przykładu z Internetu, programista poznaje tajniki obsługi urządzeń wejścia w bibliotece Allegro - na potrzeby

gry Arkanoid wystarczy klawiatura. Po rozpracowaniu technologii przechodzi do już relatywnie prostego zadania poruszania paletką. Kod w tym momencie może wyglądać mniej więcej tak:

```
#include <allegro.h>

int main(void)
{
    // initializing allegro routines
    allegro_init();
    install_keyboard();
    set_color_depth(32);
    set_gfx_mode(GFX_AUTODETECT_WINDOWED, 800, 600, 0, 0);

    // we create the paddle graphics
    BITMAP * bmpPaddle;
    bmpPaddle = load_bitmap("pad.bmp", NULL);

    // we create the screen buffer
    BITMAP * bmpScreenBuffer;
    bmpScreenBuffer = create_bitmap(800, 600);
    clear_bitmap(bmpScreenBuffer);

    // we define the starting position of the paddle
    float fPositionX = 400;
    float fPositionY = 550;

    // main loop of the game
    while(!key[KEY_ESC])
    {
        // steering the paddle
        if (key[KEY_LEFT])
        {
            fPositionX = fPositionX-1.5f;
        }
        else
        if(key[KEY_RIGHT])
        {
            fPositionX = fPositionX+1.5f;
        }

        // drawing the paddle into the screen buffer
        draw_sprite(bmpScreenBuffer, bmpPaddle, fPositionX, fPositionY);

        // the double buffering mechanism
        draw_sprite(screen, bmpScreenBuffer, 0, 0);
        clear_bitmap(bmpScreenBuffer);
    }

    return 0;
}
END_OF_MAIN()
```

Skoro programista potrafi już wyświetlić grafikę, poruszyć ją oraz odczytać sygnały z klawiatury, nie ma już żadnych technologicznych przeszkód, żeby przejść do implementacji właściwej mechaniki i logiki rozgrywki. Następnym krokiem może być np. stworzenie piłeczki odbijającej się od ścian i od paletki oraz klocków do zbijania. Wyłączając dodatkowe elementy, takie jak obsługa zakończenia gry, naliczanie punktów, efekty cząsteczkowe, dźwięk itd., można powiedzieć, że główny rdzeń gry zostałby skończony.

Allegro
Darmowa biblioteka typu open source służąca do tworzenia gier wideo. Początkowo przeznaczona na platformę Atari ST, stąd też nazwa będąca skrótem od „Atari Low Level Game Routines". W późniejszym okresie przepisana na komputery osobiste. Obsługuje głównie grafikę 2D, a w ograniczonym zakresie również 3D (tylko programowo, bez wsparcia sprzętowego). Z założenia jest przenośna między systemami operacyjnymi PC (DOS, Windows, BeOS, Mac OS X, systemy oparte na Unixksie). Stworzona w języku C, obsługiwana poprzez C lub C++.

Na powyższym przykładzie kod z punktu widzenia formatowania wygląda poprawnie i czytelnie. Algorytmicznie zaś trudno w tak mało skomplikowanym projekcie popełnić jakiś karygodny błąd. Jednak w tak krótkim fragmencie kodu naruszone zostały praktycznie wszystkie prawidła programowania gier wideo. Z przyczyn objętościowych, by nie zaciemnić obrazu, nie został zamieszczony tutaj pełny kod gry. Należy sobie jednak zdawać sprawę, że choć te kilkadziesiąt linijek nie wygląda na razie groźnie, to wraz z rozrostem aplikacji, bez rozbicia jej na warstwy i bez odpowiedniego systemu przepływu komunikatów, implementacja stanie się bardzo uciążliwa. Przykładowo każda kolejno dodana grafika do takiego kodu będzie musiała mieć ręcznie obsłużone buforowanie. Pozycja grafiki z kolei jest całkowicie logicznie oderwana od niej samej, są to po prostu współrzędne odrysowania na ekranie. W kolejnych podrozdziałach tematu warstw programowania gier pokazane zostanie prawidłowe rozdzielenie powyższego kodu i omówione zostaną podstawowe popełnione w nim błędy.

Więcej klas często znaczy lepiej

Jeśli istnieje jakakolwiek wątpliwość, czy dany problem rozwiązać, tworząc jedną klasę lub większą ich ilość albo czy rozbić daną klasę na mniejsze, prawidłowa odpowiedź jest zawsze taka sama: lepiej stworzyć więcej klas niż mniej. W większości platform rozmiar kodu nie jest już tak ważny jak kiedyś. Gdyby się okazało, że dana klasa została rozbita niepotrzebnie, znacznie łatwiej jest połączyć ją powrotem w jedną, niż jedną dużą klasę w połowie projektu podzielić sensownie na dwie. Są dwa podstawowe kryteria oceny, czy dana klasa powinna zostać podzielona na więcej klas. Pierwszym kryterium jest jej rozmiar: jeśli klasa jest zbyt duża, zawiera przykładowo ponad tysiąc linii, należy ją podzielić. Drugie kryterium to określenie, do czego dana klasa służy. Jeśli zajmuje się więcej niż jedną funkcjonalnością i wyjaśnienie, do czego służy dana klasa wymaga więcej niż jednego zdania komentarza, taką klasę również należy podzielić.

3.5.2 SDK Platformy

Każda platforma, na którą pisze się gry, posiada swoje SDK, zazwyczaj dostarczone przez producenta danej platformy. SDK umożliwia działanie na wszystkich podzespołach platformy: grafice, dźwięku, urządzeniach wejścia, procesorach i pamięci. Duża część programistów pracujących w branży gier wideo nigdy nie ujrzy na oczy linijki wywołania funkcji SDK, pod warunkiem, że zajmować się będą implementacją samej gry, nie technologii. Zazwyczaj istnieje jeden podstawowy podział: na programistów silnikowych i programistów gry. Programista silnikowy głównie pracuje z SDK oraz wystawia potrzebną funkcjonalność programiście gry. W samym kodzie gry nigdy nie powinno się jawnie wywoływać metod SDK. Po pierwsze, wprowadza to bałagan, ponieważ następuje rozsynchronizowanie pomiędzy warstwą silnika a samym kodem gry. Dla przykładu obsłużenie z poziomu SDK w kodzie gry klawiatury w danym miejscu może spowodować nieprawidłowe działanie pauzy zaimplementowanej na poziomie silnika. W silniku zostałyby wtedy zatrzymane komunikaty z wejścia, natomiast we fragmencie gry pomijającej silnik tak by nie było. Po drugie, używanie SDK wymaga od programisty gry jego dobrej znajomości, co przecież należy do programistów silnika. Po trzecie i najważniejsze, uniemożliwia tworzenie wieloplatformowego kodu. Wywołania SDK powinny się odbywać tylko na poziomie silnika technologicznego.

3.5.3 Silnik technologiczny

W najprostszym możliwym spojrzeniu silnik technologiczny w oparciu o SDK platformy, na którą jest pisany, ma za zadanie wystawić spójny interfejs dostępu do konkretnych funkcjonalności, najlepiej niezależny od platformy. Choć większość fragmentów w różnych silnikach technologicznych jest wspólna, zazwyczaj jest on implementowany z myślą o tworzeniu gier z konkretnego gatunku. Na przykładzie fragmentu kodu z ruszającą się paletką (rozdz. 3.4.1) wydzielona zostanie funkcjonalność, jaką miałby mieć silnik technologiczny takiego miniprojektu.

Pierwszą rzeczą byłoby stworzenie warstwy silnika technologicznego. Można to zrobić w oparciu o wzorzec singleton, tak żeby zawsze była tylko jedna instancja silnika, lecz dostęp do niej byłby możliwy z każdego miejsca w kodzie:

```
// an engine class, created as a~singleton
class CEngine
{

private:
  // instance of the engine
  static CEngine * m_pInstance;

  // the graphic manager
  CGraphicManager * m_pGraphicManager;
```

```
protected:
  // constructor
  CEngine()
  {
     m_pGraphicManager = new CGraphicManager();
  }

  // destructor
  CEngine()
  {
     m_pInstance = NULL;
     delete m_pGraphicManager;
  }

public:
  // ! returns the instance of the engine
  static CEngine * GetInstance()
  {
     if (m_pInstance == NULL)
  {

  // if the instance does not exist yet, create one
  m_pInstance = new CEngine();
  }
     return m_pInstance;
  }

  // removes current instance
  void DeleteInstance()
  {
     delete m_pInstance;
  }

  // initializes the allegro
  void Init()
  {

  // initializing allegro routines
      allegro_init();
      install_keyboard();
  }

  // returns the graphic manager
  CGraphicManager * GetGraphicManager(){return m_pGraphicManager;}

  // step of the engine, should be run in the main game loop
  void Step()
  {
      m_pGraphicManager->Step();
  }
};
```

Jak widać, została tutaj użyta klasa *CGraphicManager*, zajmuje się ona w tym prostym przykładzie przechowywaniem wszystkich grafik i wyrysowywaniem ich, co klatkę na ekran, jej kod wygląda następująco:

```
// handles the graphic drawing of sprites
class CGraphicManager
{
friend class CEngine;

private:
   // list of all registered sprites
   std::list<CSprite*> m_lSprites;
```

```
  // the main screen buffer
  BITMAP * m_pScreenBuffer;

  // function called every frame by the engine
  void Step()
  {
     std::list<CSprite*>::iterator sprites = m_lSprites.begin();
     while (sprites != m_lSprites.end())
  {

  // drawing all the sprites into the screen buffer
  draw_sprite(m_pScreenBuffer, (*sprites)->GetBitmap(),
  (*sprites)->GetPosition().x, (*sprites)->GetPosition().y); sprites++;
  }

  // the double buffering mechanism
  draw_sprite(screen, m_pScreenBuffer, 0, 0);
  clear_bitmap(m_pScreenBuffer);
  }

  // constructor
  CGraphicManager(){}

  // destructor
  CGraphicManager(){}

public:
  // adds the sprite to the graphic manager
  void AddSprite(CSprite * pSprite)
  {
     m_lSprites.push_back(pSprite);
  }

  // removes the sprite from the graphic manager
  void RemoveSprite(CSprite * pSprite)
  {
     m_lSprites.remove(pSprite);
  }

     void InitGraphics(bool bWindowed, int iResX, int iResY, int
          iColorDepth)
   {
     set_color_depth(iColorDepth);
     if (bWindowed)
         set_gfx_mode(GFX_AUTODETECT_WINDOWED, iResX, iResY, 0, 0);
     else
         set_gfx_mode(GFX_AUTODETECT_FULLSCREEN, iResX, iResY, 0, 0);

     // we create the screen buffer
     m_pScreenBuffer = create_bitmap(iResX, iResY);
     clear_bitmap(m_pScreenBuffer);
   }
};
```

Do kompletu implementacji tego maksymalnie uproszczonego renderera 2D opartego o bibliotekę Allegro brakuje już tylko klasy *CSprite*:

```
// class representing single sprite
class CSprite
{
friend class CGraphicManager;

private:
    // the allegro bitmap
    BITMAP * m_pSprite;
```

```
    // the position of the sprite
    Vector3D m_v3Position;

    // returns the allegro bitmap
    BITMAP * GetBitmap(){return m_pSprite;}

public:
    // constructor
    CSprite(const char * pchResourcePath)
    {
       m_pSprite = load_bitmap(pchResourcePath, NULL);
       CEngine::GetInstance()->GetGraphicManager()->AddSprite(this);
    }

    // destructor
    CSprite()
    {
       CEngine::GetInstance()->GetGraphicManager()->RemoveSprite(this);
       destroy_bitmap(m_pSprite);
    }

    // returns the position of the sprite
    Vector3D GetPosition(){return m_v3Position;}

    // sets the position of the sprite
    void SetPosition(Vector3D v3Position){m_v3Position = v3Position;}
};
```

Takie podejście pozwoliło całkowicie oderwać wyrysowywanie na ekran i buforowania grafik od warstwy samej gry. Na takiej bazie można już swobodnie rozwijać ten minisilnik. Przykładowo wykorzystując fakt, że pozycja 2D zapisana jest w klasie *Vector3D*, to nieużywanej współrzędnej „z" można użyć w klasie *CGraphicManager* do sortowania kolejności wyrysowywania grafik. Do klasy *CEngine* można dopisać obsługę stałego lub zmiennego *framerate'u*, gdyż w tej chwili po prostu wykonuje tyle klatek na sekundę, ile tylko potrafi, ale bez mierzenia czasu i ewentualnego opóźnienia w odrysowywaniu ekranu. Wszystkie tego typu mechanizmy i wiele innych można teraz swobodnie dokładać, kiedy poszczególne elementy są zamknięte w osobnych klasach opartych na pewnej architekturze. Skróciło to kod samej funkcjonalnej aplikacji do takiego rozmiaru:

```
CEngine * CEngine::m_pInstance = NULL;
int main(void)
{
    // initializing the engine
    CEngine::GetInstance()->Init();

    // initializing the renderer
    CEngine::GetInstance()->GetGraphicManager()->InitGraphics(true, 800,
        600, 32);

    // we create the paddle graphics
    CSprite * pPaddle = new CSprite(,,pad.bmp,,);

    // we define the starting position of the paddle
    pPaddle->SetPosition(Vector3D(400, 550, 0));

    // main loop of the game
    while(!key[KEY_ESC])
```

```
    {
        CEngine::GetInstance()->Step();

        // steering the paddle
        if (key[KEY_LEFT])
        {
            pPaddle->SetPosition(pPaddle->GetPosition() - Vector3D(1.5f
                ,0,0));
        }
        else
        if(key[KEY_RIGHT])
        {
            pPaddle->SetPosition(pPaddle->GetPosition() + Vector3D(1.5f
                ,0,0));
        }
    }

    return 0;
}
END_OF_MAIN()
```

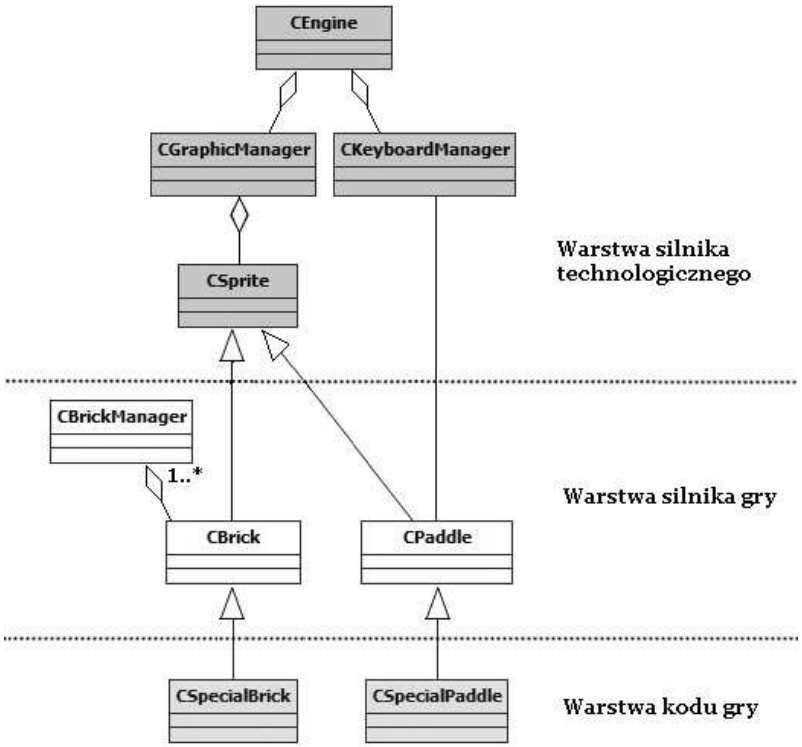
W powyższym kodzie, który wygląda już zdecydowanie lepiej niż oryginalny przykład, nadal znajduje się kilka elementów wymagających dopracowania. Przykładowo urządzenie wejścia, w tym przypadku klawiatura, obsłużone jest z poziomu SDK w kodzie gry. Odpowiednia funkcjonalność do tego celu powinna zostać udostępniona przez silnik. Jeśli spojrzy się na liczbę linii pierwszego przykładu i obecną, można dojść do błędnego wniosku, że przecież te dwa kawałki kodu robią dokładnie to samo, a pierwsza wersja powstała szybciej i jest krótsza. Jest to w pewnym sensie prawda, ale próba wprowadzenia przykładowo wyświetlenia piłeczki w drugim przypadku, to w tej chwili jedna linijka kodu. W pierwszym przypadku, z każdą dodaną grafiką musiałaby ona zostać ręcznie uwzględniania w mechanizmie buforowania, co jest absolutną stratą czasu i zaciemnia kod. Już taki mały kawałek kodu potwierdza zasadę, że elegancja się opłaca.

Więcej na temat architektury silnika technologicznego opisane jest w rozdziale czwartym.

3.5.4 Silnik gry

W odróżnieniu od silnika technologicznego, silnik gry jest tworzony nie tylko pod konkretny gatunek, ale wiele jego mechanizmów tworzone jest na potrzeby konkretnej gry. Choć w branży gier wideo często silnik technologiczny nazywany jest również silnikiem gry, a reszta po prostu kodem gry, takie podejście od strony programistycznej może spowodować zbytnie zazębienie logiki gry z jej funkcjonalnością. Poniżej pokazany jest diagram klas pojedynczego poziomu *Arkanoida*, z rozdzieleniem na warstwę silnika technologicznego, silnika gry oraz kodu gry.

Rysunek 3.5 pokazuje, że kod gry w takiej architekturze sprowadzałby się praktycznie tylko do implementacji jakiś specjalnych zachowań dla klocków



Rysunek 3.5. Diagram klas pojedynczego poziomu gry Arkanoid

i paletki, reszta kodu rozdzielona byłaby pomiędzy poszczególne warstwy silników. Takie podejście sprawia, że bardzo łatwo jest przenieść podstawową mechanikę *Arkanoida* do następnego projektu opartego na tym samym pomyśle, co w branży gier wideo zdarza się bardzo często.

Więcej informacji na temat architektury silnika gry zawiera rozdział 5.

3.5.5 Narzędzia do tworzenia gry

Pojęcie „narzędzie do tworzenia gry" nasuwa na myśl w pierwszej kolejności edytor, czyli narzędzie, które jest ogniwem łączącym silnik gry z wizją designerów. Na przykładzie Arkanoida takie narzędzie umożliwiałoby edycję parametrów paletki oraz piłeczki, takich jak na przykład prędkość, oraz przede wszystkim umożliwiałoby rozłożenie klocków w dowolny sposób na poszczególnych poziomach gry. Jednak narzędzia to nie tylko edytor. Istnieje cała gama narzędzi wspomagających development, od konwerterów grafiki po systemy wersjonowania kodu. Temat narzędzi rozwinięto w rozdziale 6.

3.5.6 Inne rozwiązania

Podział warstw przedstawiony w tym rozdziale jest często stosowany w programowaniu gier wideo, jednak nie jest on jedyny. Poza wprowadzaniem oczywistych kolejnych podziałów poszczególnych warstw, takich jak rozdzielenie silnika technologicznego na warstwy obsługujące poszczególne platformy, istnieją też warstwy implementowane równolegle z tymi opisanymi wcześniej. Przykładowo w bardzo wielu projektach, najczęściej średniej i małej wielkości, spotyka się sam kod gry, bez wydzielenia warstwy silnika gry. Choć z punktu widzenia projektowania i przyszłego ponownego użycia kodu takie podejście nie jest najszczęśliwsze, to w wielu wypadkach występuje, gdyż często na samym początku projektu nie do końca wiadomo, które fragmenty będą na tyle uniwersalne, by stanowić część silnika gry.

Kod gry jest warstwą działającą bezpośrednio na silniku technologicznym lub poprzez warstwę silnika gry. Są w niej zawarte implementacje konkretnych mechanizmów charakterystycznych dla konkretnego tytułu, a nie całego typu czy też serii gier. W sytuacji idealnej warstwa ta powinna być niewielka, gdyż większość mechanizmów ogólnych dla gatunku gry powinna się znajdować w silniku gry, takich jak na przykład obsługa kamery czy kolizji w grach typu FPP (z ang. „*First Person Perspecitve*"). Większość specyficznych ustawień, takich jak na przykład parametryzowanie kamery, powinna być realizowana na poziomie narzędzia do tworzenia gry, na przykład przy użyciu skryptów.

Taka sytuacja niestety występuje relatywnie rzadko. Bez bardzo dojrzałego silnika do tworzenia gry, który przeszedł już kilka iteracji danego gatunku gier, warstwa kodu gry zawsze będzie obszerna. Przykładowo, jeśli powstaje gra na silniku technologicznym przystosowanym do tworzenia gier z gatunku FPP, w oparciu o ten silnik programista tworzy zaawansowane AI na potrzeby konkretnej serii gier i umieszcza je w warstwie silnika gry. Implementacja zachowań konkretnej instancji specjalnego przeciwnika, np. bossa, będzie jednak unikalna na potrzeby jednego tytułu i umieszczona w warstwie kodu gry. W przypadku idealnym opisanym wcześniej, kod AI w warstwie silnika gry będzie napisany na tyle ogólnie i będzie na tyle dobrze parametryzowalny, że do stworzenia takiego bossa, wystarczy designerowi zmodyfikowanie parametrów z poziomu edytora bądź skryptów.

W przypadku bardzo specyficznych gatunków gier, typu MMORPG (ang. „*Massive Multiplayer Online Role Playing Game*"), bardzo rozbudowana będzie warstwa sieciowa w silniku gry. Powstają wtedy osobno całe systemy do obsługi tylko tego typu rozgrywki, w całkowitym oderwaniu od takich rzeczy jak np. wyświetlanie grafiki.

Podsumowując, w rzeczywistym projekcie sztywne ramy podziału na warstwy będą rzadko utrzymane, natomiast warstwy kodu będą się w jakiś sposób między sobą zazębiać. Jednak należy owo zazębianie utrzymywać na minimalnym możliwym poziomie.

4

Architektura silnika technologicznego

4.1 Wstęp

Pojęcie silnika w branży gier wideo pojawiło się po raz pierwszy na początku lat 90 wraz ze stworzeniem przez firmę *ID Software* serii gier FPP, zaczynając od Wolfenstein 3D, poprzez bardzo popularnego Doom, aż do wydanego w roku 1996 *Quake'u*. To właśnie *Quake Engine* wyznaczał pierwsze szlaki w sformalizowanym podejściu do tworzenia silników. Jednakże niezależnie od tych wydarzeń wiele gatunków gier w tamtych czasach powstawało bez wydzielonej warstwy silnika. Całkowite zrewolucjonizowanie podejścia do tworzenia kodu gry jako warstwy opartej na silniku, nastąpiło wraz ze wzrostem popularności *Unreal Engine* autorstwa *Epic Games*. Pierwsza wersja tego silnika powstała w roku 1997, zaś od roku 2000 bardzo wiele gier zaczęło powstawać w oparciu o niego i jego kolejne wersje. Patrząc na tę sprawę z punktu widzenia czasu, warto zauważyć, że branża gier komputerowych liczy sobie około 40 lat, a dopiero od około 10 została ugruntowana potrzeba tworzenia silników. Przyczyn takiej kolei rzeczy jest kilka, jednak główną jest ogromny rozwój możliwości sprzętowych w ciągu ostatnich kilkunastu lat, za czym poszło zwiększenie rozmiarów tworzonych gier. Nastała potrzeba tworzenia solidnych fundamentów do tworzonego kodu. Relatywnie młode zagadnienie silników technologicznych sprawia, że nie ma utartych i jedynych prawidłowych praktyk związanych z ich tworzeniem. Choć jest wielka baza wspólnych doświadczeń każdej firmy zajmującej się tworzeniem silników, czy to w celach komercyjnych, czy użycia we własnym produkcie, to jednak silniki znacznie różnią się od siebie architekturą. W tym rozdziale zostaną wypisane pewne elementy, z których musi się składać każdy silnik, jednak sama proponowana architektura jest tylko jedną z wielu koncepcji istniejących na rynku. Wraz z kolejnymi latami na pewno coraz więcej elementów zostanie ustandaryzowanych w idei tworzenia silników, jednak do tego czasu warto zaznajomić się z aktualnie istniejącymi rozwiązaniami i samemu osądzić, które do danej potrzeby będzie najlepsze.

4.2 Zestawienie wymagań zależnie od gatunku gry

Zależnie od gatunku gry jej wymagania względem silnika technologicznego, na którym zostanie oparta, mogą się bardzo różnić. Poniżej znajduje się zestawienie gatunków gier, ze względu na funkcjonalności potrzebne z punktu widzenia silnika. Należy pamiętać, że to zestawienie nie wyczerpuje absolutnie wszystkich gatunków gier, a jedynie przedstawia pewien zestaw uniwersalnych zagadnień, które są ważne z punktu widzenia silnika gry.

- **FPS** (z ang. *„First Person Shooter"*, pol. „Strzelanina z pierwszej osoby") - gry nieodłącznie związane z widokiem typu FPP (z ang. *„First Person Perspective"*, pol. „Perspektywa pierwszej osoby"). W grach FPS zadaniem gracza jest eliminacja przeciwników przy pomocy broni, oglądana oczami postaci bohatera. Kiedy gatunek tak naprawdę dopiero się rozwijał, czyli na początku lat 90, przez takie gry jak *Wolfenstein 3D* czy *Doom*, zawierał w sobie element eksploracji zazwyczaj mocno zamkniętych korytarzy i labiryntów, bez otwartych przestrzeni. Obecnie z punktu widzenia graficznego jest to najbardziej skomplikowany gatunek gier do stworzenia, chociaż nadal najczęściej akcja rozgrywa się w mocno ograniczonym terenie (np. seria *Call of Duty: Modern Warfare*) lub wręcz w zamkniętych pomieszczeniach (np. seria *Bioshock*). Tak wysokie wymagania graficzne stawiane przed gatunkiem FPS biorą się z próby przedstawienia w tego typu grach jak najrealniejszego świata gry, gdyż gracz widzi go właśnie z perspektywy pierwszej osoby. W takim widoku gracz zwraca uwagę na wszystkie odstępstwa od realności umownego świata postawionego przed nim. Przykładowo, grając w strategie czasu rzeczywistego z widokiem z dalekiej perspektywy, gracz nie jest w stanie zauważyć, że postacie nie poruszają ustami podczas rozmowy, a w grze typu FPS takie rozwiązanie będzie mocno widoczne i niekorzystne dla gry. Tak samo, jeśli w platformówce przeciwnik zniknie w chmurce dymu to jest to coś, czego gracz może się spodziewać, lecz w grze typu FPS przeciwnik powinien upaść na ziemię zgodnie z prawami fizyki. Różnica skomplikowania tych dwóch zagadnień z punktu widzenia programowania jest ogromna. Poniżej lista standardowych wymagań stawianych przed silnikami obsługującymi ten gatunek:
  - Bardzo efektywny rendering dużych trójwymiarowych światów, zależnie od nastawienia na zamknięte przestrzenie czy otwarte, technologia ta może się różnić w implementacji;
  - Bardzo wysoka jakość modeli postaci oraz broni występujących w grze, wraz z animacjami;
  - Zaawansowana fizyka ciał sztywnych wraz z efektywnym systemem wykrywania kolizji oraz fizyki typu rag-doll;
  - AI dostosowane do danego typu rozgrywki, inna implementacja będzie dla zamkniętych przestrzeni, a inna dla otwartych;
  - Możliwość rozgrywki sieciowej na mała skalę, zazwyczaj poniżej 64 graczy naraz.

Rysunek 4.1. Call of Duty: Modern Warfare 2 jako przykład gry FPP/FPS

- **Gry akcji** - gatunek charakteryzowany obecnie głównie poprzez widok TPP (z ang. *„Third person perspective"*, pol. Perspektywa trzeciej osoby). W grach akcji gracz koncentruje się na eksploracji terenu i walce z przeciwnikami, często zmuszony jest też znajdować rozwiązania prostych zagadek logicznych. Spopularyzowany przez serię gier *Tomb Raider*. Główną różnicą pomiędzy widokami TPP a FPP, poza sposobem przedstawienia rozgrywki, który z punktu widzenia technologii nie ma większego znaczenia, jest ilość animacji, jaką musi posiadać główna postać. W grach z widokiem TPP główna postać posiada szeroki wachlarz animacji konieczny do wykonywania wszelakiego rodzaju czynności takich jak: skakanie, wspinaczka itp. Animacje przeciwników i otaczającego świata mają inne znaczenie w grach akcji niż w FPS, głównie z powodu oddalenia kamery oraz mniejszej możliwości od strony gracza dostrzeżenia niedociągnięć w grafice. Współcześnie te różnice coraz bardziej się zacierają, z jednej strony gatunek FPS również musi posiadać postać z kompletem animacji na potrzeby gry sieciowej, nie tak jak kiedyś, gdy wystarczyły tylko ręce i broń dobrej jakości. Z drugiej zaś częsta możliwość dowolnego zarządzania kamerą w grach akcji sprawia, że grafika musi być na bardzo wysokim poziomie, by gracz nie miał dużej różnicy w odebraniu gry, gdy przykładowo podejdzie blisko jakiegoś obiektu, by mu się przyjrzeć. Z punktu widzenia potrzeb technologicznych pod gatunek gier akcji z widokiem TPP można podciągnąć wszelkiego rodzaju platformówki 3D (np. seria *Crash*), shootery i przygodowe gry akcji TPP (np. serie *Gears of War* i *Uncharted*). Trzeba jednak zauważyć, że platformówki, które zazwyczaj posiadają pewną formę kreskówkową grafiki, często są mniej wymagające pod względem

graficznym. Poniżej lista charakterystycznych cech, które muszą być uwzględnione w technologii do tego gatunku:

- Zaawansowana interakcja z otoczeniem: drabiny, liny, ruchome platformy itp. Zazwyczaj połączone z zaawansowana fizyką i kolizjami, nie tylko postaci.
- Inteligentna kamera poruszająca się za graczem i zmieniająca swą pozycję zależnie od sytuacji, kiedy przykładowo gracz podejdzie do ściany i obróci się do niej plecami itp. Obejmuje to skomplikowany system kolizji kamery z otoczeniem, by kamera nigdy nie przeniknęła żadnej geometrii.

Rysunek 4.2. *Uncharted 2: Among Thieves*, Gra akcji studia Naughty Dog, wielki sukces na platformę PS3

- **RTS** (z ang. *„Real Time Strategy"*, pol. Strategia czasu rzeczywistego) - za grę, która po raz pierwszy zdefiniowała ten gatunek, uznaje się powszechnie *Dune 2*, stworzoną w 1992 roku przez *Westwood Studios*. Strategie czasu rzeczywistego polegają zazwyczaj na wystawieniu znacznej ilości wojsk przy pomocy jakiegoś systemu ekonomicznego oraz walki z armią przeciwnika, który wystawia swoją armię w ten sam sposób. Takim system ekonomicznym jest najczęściej zbieranie z mapy pewnego zestawu surowców, za które później w zbudowanych przez siebie budynkach można szkolić kolejne jednostki (np. seria *Warcraft* lub *Command & Conquer*). Pierwsze gry RTS robione w 2D przedstawiały rozgrywkę albo całkowicie z góry (np. *Lords of Realms*), albo izometrycznie (np. *Starcraft* lub *Age of Empires*). Obecnie większość gier RTS tworzona jest w pełnym 3D, dając jednak zazwyczaj pewne ograniczenia w sterowaniu kamerą, tak by widok był lekko izometryczny z góry. Gatunek ten jest często łączony ze strategiami turowymi, gdzie bardziej strategiczne podejście makro do sytuacji

wojennej jest rozgrywane turowo, natomiast konkretne bitwy w czasie rzeczywistym (np. seria *Total War*). Prawie każda gra RTS wydawana obecnie posiada tryb rozgrywki sieciowej. Po grach MMOG i FPS jest to najpopularniejszy obecnie gatunek do rozgrywki wieloosobowej. Standardowe wymaganie technologiczne stawiane przed grami tego typu:

- Obsługa bardzo dużej ilości jednostek występujących jednocześnie na mapie, zazwyczaj wiąże się to faktem, że każda jednostka jest relatywnie w niskiej rozdzielczości, czyli posiada małą ilość poligonów i mała teksturę
- Zaawansowany system tworzenia terenów, zazwyczaj z map wysokości
- Możliwość rozgrywki sieciowej, zazwyczaj na małą skalę, rzadko więcej niż 16 graczy naraz
- Duże możliwości 2D na potrzeby zaawansowanego UI (minimapy, wyświetlanie zasobów, dostępnych jednostek i budynków itd.)
- Zaawansowane AI, także np. punktu widzenia znajdowania optymalnej drogi przez grupy jednostek (tak zwany z ang. *Crowd Pathfinding*)
- Możliwość łatwej zmiany dowolnego parametru dowolnej jednostki, czyli udostępnienie pewnego interfejsu skryptowego na potrzeby balansowania rozgrywki (zwłaszcza sieciowej)

Rysunek 4.3. *Starcraft 2: Wings of Liberty* jest przykładem typowej gry z gatunku RTS

- **cRPG** (z ang. *„Computer Role Playing Game"*, pol. komputerowe gry fabularne) - sformułowanie komputerowe jest niedokładne, gdyż gatunek ten obecnie tworzony jest na wszystkie platformy, w tym konsole i telefony. Człon ten został dodany przez środowisko graczy w celu odróżnienia gier

wideo od klasycznych papierowych gier fabularnych, opartych na pewnych systemach i światach (np. system *Dungeon & Dragons*, oraz świat rozgrywany w tym systemie *Forgotten Realms*). Gatunek ten ma bardzo wiele podgatunków, przez co też bardzo trudno go zdefiniować. Wywodzi się on z klasycznych gier fabularnych, stąd jego głównym celem jest zazwyczaj dać graczowi iluzję całkowitej wolności wyboru oraz możliwości wpływu na otaczający go świat oraz zbudować klimat pozwalający mu się zidentyfikować z odgrywaną postacią. Jest to oczywiście pewien ideał, którego nie sposób osiągnąć z powodu technicznych ograniczeń gier wideo, jednak wiele gier starało się do tego ideału dążyć (np. *Fallout 2, Star Wars: Knights of the Old Republic*). Kilka podgatunków cRPG osiągnęło ogromne sukcesy mimo zastosowania wielu uproszczeń, na przykład gatunek *Hack&Slash*, spopularyzowany przez *Diablo*, gdzie głównym elementem gry była walka i usprawnianie swojej postaci poprzez znajdowanie przedmiotów i rozwijanie umiejętności. Gry cRPG charakteryzują się zazwyczaj dosyć długim czasem grania oraz możliwością wykonywania pobocznych zadań czy też misji, niezwiązanych z głównym wątkiem gry. Zależnie od tego, czy gracz zechce jak najszybciej grę zakończyć czy też wykonać jak najwięcej zadań pobocznych, oczekiwany czas rozgrywki może się wahać średnio od 10 aż do 100 godzin grania. Jakość grafiki w grach cRPG rzadko jednak osiąga poziom obecny np. gry FPS. Dzieje się tak w związku ze znacznie dłuższym czasem tworzenia gry cRPG, co daje w efekcie częste zestarzenie się technologii w trakcie tworzenia gry. Oprócz tego ilość wyświetlanej grafiki naraz, na danym fragmencie terenu jest znacznie większa niż w zamkniętych i zazwyczaj przestarzałych grach FPS. W związku z ogromem grafik, tekstów pisanych, tekstów czytanych i dźwięków, jest to jeden z najbardziej rozbudowanych i najtrudniejszych w realizacji gatunków gier wideo,

Rysunek 4.4. *Dragon Age: Origins*, gra RPG studia *BioWare*, była nawiązaniem do tradycji firmy, czyli takich tytułów jak *Baldur's Gate*

kosztami produkcji ustępujący zazwyczaj tylko grom MMOG. Klasyczne cRPG przeważnie nie posiadają trybu rozgrywki sieciowej, gdyż główny nacisk kładziony jest na fabularny rozwój gry, jednak w niektórych podgatunkach występuje on bardzo często, tam gdzie podstawowym elementem rozgrywki jest na przykład walka. Wymagania stawiane przed silnikiem tego typu gry są dosyć duże, gdyż zazwyczaj obejmują dokładnie taki sam zestaw wymagań jak gry FPS czy TPP, zależnie od tego, w jaki sposób w grze przedstawiana jest rozgrywka, oraz kilka dodatkowych, charakterystycznych dla cRPG:

- Bardzo sprawny system dynamicznego zarządzania sceną, często bez rozróżnienia na tereny zamknięte i otwarte, w celu unikania ekranów ładowania przy eksploracji świata gry
- Efekty graficzne urzeczywistniające otaczający gracza świat, takie jak symulacje nocy i dnia, symulacje pogody, roślinności itp.

- **MMOG** (z ang. *Massive Multiplayer Online Game*, pol. Masowa wieloosobowa gra sieciowa) i jak sama nazwa wskazuje, głównym elementem rozgrywki w tych grach jest rozgrywka sieciowa, nastawiona na interakcje z innymi graczami. Gra obsługuje ogromną liczbę graczy jednocześnie, zazwyczaj po kilka tysięcy na pojedynczy serwer. Sam gatunek MMOG definiuje tylko założenia techniczne dotyczące liczby graczy, dzieli się natomiast na wiele podgatunków rozróżnianych sposobem rozrywki jak MMORPG czy MMORTS. Obecnie najpopularniejszą grą tego typu jest MMORPG *World of Warcraft* firmy *Blizzard*, która posiada ponad 10 milionów aktywnych użytkowników. Z tego typu grami prawie zawsze wiąże się jakaś opłata, zazwyczaj w postaci miesięcznego abonamentu. Głównym wyzwaniem pod-

Rysunek 4.5. *World of Warcraft*, obecnie najpopularniejszy MMORPG na świecie

czas tworzenia tego typu gry jest zaplecze sprzętowe, czyli potężna bateria serwerów, na których utrzymywany jest stały świat gry. Pod względem technicznym gry tego typu zawsze są mało wymagające graficznie. Dzieje się tak z dwóch powodów. Po pierwsze, gry te wymierzone są w maksymalną liczbę użytkowników, więc jakość posiadanego przez nich sprzętu nie powinna wpływać na możliwość dołączenia się do gry. Po drugie, wyświetlają czasem ogromną ilość geometrii naraz, zwłaszcza w skupiskach ludności takich jak np. miasta, gdzie porusza się zazwyczaj mnóstwo avatarów graczy.

Podstawowymi wymaganiami silnikowymi dla tego typu gier są:

- Wysoka optymalizacja odwzorowywania aktualnego stanu świata z serwera na platformie klienckiej z minimalnej ilości danych, w celu ograniczenia przesyłu danych między serwerem a klientem.
- Obsługa bardzo dużej ilości bytów graficznych w tym samym czasie, hipotetycznie rzecz biorąc, może się znaleźć w jednym widoku kilka tysięcy graczy.
- Przystosowanie do bardzo częstych zmian i poprawek wprowadzanych do świata gry, z minimalnym czasem konieczności odłączenia serwera.
- Zapewnienie akceptowalnej jakości grafiki przy niskich wymaganiach sprzętowych, w celu maksymalnego poszerzenia kręgu odbiorców.

- **Bijatyki** - Początkowo bardzo popularny gatunek gier 2D (np. seria *Street Fighter*), zaś obecnie tworzony głównie w 3D nawet wtedy, gdy rozgrywka

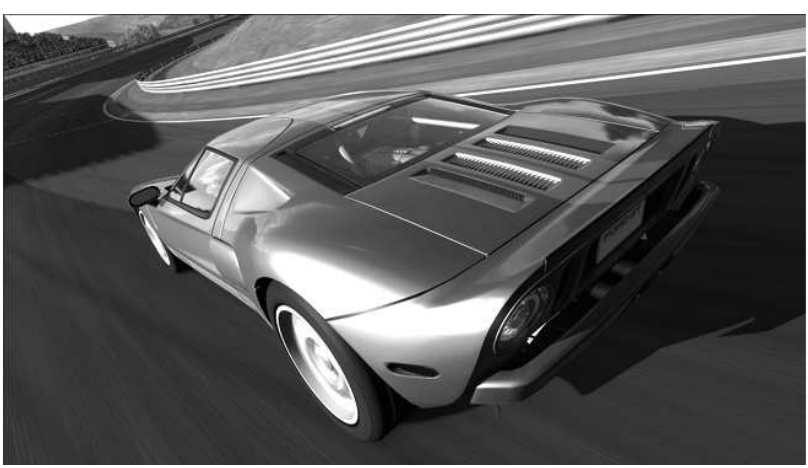
Rysunek 4.6. UFC Undisputed 2010, bijatyka traktująca o zawodach mieszanych sztuk walk

jest spłaszczona do dwóch wymiarów (np. niedawno zapowiedziany nowy *Mortal Kombat*). Z punktu widzenia wydajności jest to relatywnie prosty gatunek w implementacji. Cała akcja gry toczy się przeważnie w jednym miejscu, z w miarę statycznym tłem i małym polem ruchu kamery. Pozwala to na stworzenie bardzo wysoko poligonowych modeli postaci oraz wielu efektów graficznych. Najbardziej skomplikowanym elementem tego typu gier jest ogromna ilość animacji postaci oraz konieczność realizacji płynnych przejść pomiędzy poszczególnymi animacjami, aby ruchy zawodników wyglądały najbardziej naturalnie. Z powodu zapasu możliwości wydajności w tego typu grach często widać technologie graficzne niespotykane w innych gatunkach, jak realistyczne odwzorowanie zachowań odzieży czy włosów (np. *Fight Night Round 4*).

Typowe wymagania stawiane przed silnikiem do gier tego typu:

- Obsługa ogromnej ilość animacji postaci
- Duża dokładność detekcji kolizji ciosu
- Zaawansowana obsługa urządzeń wejścia w celu obsługi wprowadzania skomplikowanych kombinacji

- **Symulatory** - można tak nazwać wiele gatunków gier. Wyścigi samochodowe, symulatory lotu czy nawet symulator pociągu. Główny nacisk w tego typu grach jest kładziony na wygląd pojazdu gracza oraz wierne odwzorowanie zachowania pojazdu. Fizyka oraz kolizje są jednym z kluczowych elementów tego typu gier, obok efektowności graficznej. Mniejszą uwagę poświęca się tutaj na środowisko otaczające rozgrywkę, przykładowo teren na zewnątrz trasy wyścigowej czy na zewnątrz strefy, w której można latać.



Rysunek 4.7. *Gran Turismo 5*, obecnie jeden z najbardziej zaawansowanych graficznie symulatorów jazdy

Typowe problemy technologiczne tego typu gier:

- Duża ilość efektów graficznych nałożona na bardzo wysokiej jakości model pojazdu gracza.
- Zaawansowane AI, szczególnie z punktu widzenia znajdowania optymalnej drogi.
- Fizyka i kolizje na potrzeby sterowania pojazdem przez gracza i AI.
- Triki graficzne potrzebne do ukrycia małej szczegółowości mocno oddalonych elementów środowiska gry, np. LOD (z ang. *Level of Detail*, pol. poziom szczegółów), który w najmniejszej dokładności jest zwykłą bitmapą dwuwymiarową.

Oczywiście gatunków gier jest znacznie więcej: gry sportowe, strategie turowe, przygodowe itd. Jednak z punktu widzenia wymagań technologicznych, o ile każdy gatunek ma pewne unikatowe wymagania, tak przykładowe zestawienie powyżej pokrywa bardzo dużą część z nich.

Bardzo często gry mają w sobie elementy wielu różnych gatunków, wtedy z punktu widzenia silnika technologicznego programista musi mieć do dyspozycji mechanizmy potrzebne w każdym z używanych w grze gatunków. Przykładem może tu być gra *Fallout 3* stworzona przez *Bethesda Game Studios*. Gra gatunkowo głównie jest postrzegana jako cRPG i na pewno posiada wszelkie elementy charakteryzujące ten gatunek, jednak z punktu widzenia technologii gra ma również bardzo wiele wspólnego z FPS, gdyż jej akcja przedstawiona jest z perspektywy pierwszej osoby i jest w niej bardzo dużo walki. Tutaj gra popadła w pewną pułapkę, jako cRPG nie musiała się wykazać na rynku wybitną grafiką konkurującą z najnowszymi FPSami, jednakże z powodu takiego, a nie innego sposobu przedstawienia rozgrywki grafika w tej grze była mimo wszystko porównywana do najlepszych FPSów, które wyszły na rynek w pobliżu jej premiery, czyli np. *Far Cry 2*. Tym samym jedną z głównych wad wymienianych w *Fallout 3* przez recenzentów była właśnie słaba grafika. Dodatkowo *Fallout 3* posiada również niezależny widok TPP, co wiąże się z koniecznością rozwiązania problemów charakterystycznych także dla gier tego typu. Nie zawsze mieszanie gatunków wiąże się ze wzrostem skomplikowania silnika. Przykładowo gra *Heavy Rain* jest często podawana jako zrewolucjonizowanie gatunku gier przygodowych, jednak z punktu widzenia technologii jest to TPP z intensywnie wykorzystywanym QTE. Skomplikowanie tej gry bierze się więc nie z jej gatunku, ale z założenia, że każda z możliwych ścieżek stopnia sukcesu ze strony gracza jest brana pod uwagę, do tego zaś potrzebna jest ogromna ilość animacji przedstawiających wszystkie kombinacje.

Należy pamiętać też o kwestii wykorzystania w grach nowinek technologicznych typu wyświetlacze 3D czy wszelkiego rodzaju systemów rozpoznawania ruchów (Sony PS Move, Microsoft Xbox360 Kinect). W takich przypadkach ważne jest zabezpieczenie obsługi wszystkich tych elementów przez technologię, zanim przystąpi się fizycznie do implementacji mechaniki gry.

4.3 Dostępne rozwiązania

Znając już wymagania, jakie programista może posiadać względem silników technologicznych, warto zastanowić się, jakie gotowe rozwiązania są już dostępne na rynku. Podstawowym kryterium podziału dostępnych rozwiązań jest ich komercyjność: silnik może być stworzony w celach komercyjnych lub do użytku darmowego. Poniżej zestawienie kilku dostępnych silników technologicznych:

- **Silniki darmowe**
  - *Irrlicht* - silnik 3D open source, prosty do opanowania i daje możliwość uzyskania szybkich efektów. Z powodu kilku błędów architektonicznych nie nadaje się do użytku przy dużych projektach. Przeznaczony głównie na platformę PC, jednak jego wieloplatformowość z założenia pozwoliła na kilka hobbistycznych prób konwersji na platformy takie jak iPhone czy XBox360.

    `Link do strony producenta:http://irrlicht.sourceforge.net/`
  - *OGRE* - silnik 3D *open source* to jeden z najpopularniejszych silników z udostępnionymi źródłami, chociaż nie jest tak prosty do opanowania jak Irrlicht i posiada bardzie wiele modułów, które niekoniecznie są potrzebne w każdym produkcie, to jest to jedyny darmowy, tak szeroko wykorzystywany komercyjnie silnik. Największym produktem komercyjnym stworzonym na tym silniku dotychczas jest gra *Torchlight* firmy *Runic Games*, wydana w 2009 roku. Przeznaczony na platformę PC.

    `Link do strony producenta: http://www.ogre3d.org/`

Rysunek 4.8. *Torchlight*, stworzony na silniku OGRE, jest klasyczną grą z gatunku hack& slash

- *Panda3D* - silnik 3D *open source*, przeznaczony głównie do tworzenia gier w języku Python i C++ na platformę PC. Silnik szeroko wykorzystywany do celów komercyjnych przez producentów gier typu *casual*, np. przez *Disney Interactive Media Group*.

  `Link do strony producenta: http://www.panda3d.org/`

- *Unity* - W porównaniu z poprzednimi przedstawionymi darmowymi silnikami, Unity posiada najbardziej rozbudowany zestaw narzędzi, który w teorii pozwala na stworzenie gry z minimalną potrzebą programowania. Wersja podstawowa silnika Unity na platformę PC jest darmowa. Posiada on również wersję rozszerzoną o nazwie Unity Pro oraz wersje na platformy z serii iOs (np. iPhone) i Android, które są już płatne.

  `Link do strony producenta: http://unity3d.com/`

- **Silniki komercyjne**
  - *Blitz* - wieloplatformowy silnik, obsługujący wszystkie obecne konsole siódmej generacji oraz PC. Jest jednym z niewielu współczesnych silników, który swoją obsługę konsoli Nintendo Wii ma zaimplementowaną na wysokim poziomie, co udowodniła firma *Sega*, tworząc przy użyciu tego silnika grę *The House of the Dead: Overkill*.

    `Link do strony: http://www.blitzgamesstudios.com/blitztech/`

  - *CryEngine* - przeznaczony na mocne platformy takie jak: PC, Xbox360 i Playstion 3. Posiada bardzo wysokie możliwości tworzenia gier o wyjątkowej oprawie graficznej, jednak nie jest silnikiem wysoce zoptymalizowanym, dlatego piękno tworzonych na nim gier często jest opłacone wysokimi wymaganiami sprzętowymi, np. gra *Crysis*, stworzona przez producenta silnika na jego drugiej wersji. Choć gra prezentuje bardzo wysoki poziom graficzny, mało który użytkownik komputera osobistego może ją oglądać w pełnej postaci z powodu wymagań sprzętowych.

    `Link do strony producenta: http://www.crytek.com/`

  - *Gamebryo* - jeden z najpopularniejszych silników wieloplatformowych, który specjalizuje się w tworzeniu pełnego zestawu narzędzi do tworzenia gier opartych na źródłach silnika. Na przykład Gamebryo Lightspeed jest zestawem narzędzi, które umożliwiają bardzo szybkie prototypowanie gier. Istnieją też zestawy skierowane pod produkcję konkretnego typu gier, np. *causal*. Silnik ten obsługuje każdą konsolę siódmej generacji oraz platformę PC.

    `Link do strony producenta: http://www.emergent.net/`

  - *Shark 3D* - silnik wieloplatformowy w pełnym tego słowa znaczeniu, ponieważ obsługuje nie tylko konsolę siódmej generacji (prócz Playstation 3, którego obsługa jest w trakcie realizacji) oraz PC, ale również wszystkie platformy oparte na iOS oraz nawet przeglądarki internetowe (IE, Firefox, Chrome, Opera). Najpopularniejsza gra stworzona na tym silniku to *Dreamfall: The Longest Journey*.

`Link do strony producenta: http://www.spinor.com/`

- *Torque Game Engine* - nie jest to jeden zwarty silnik wieloplatformowy, a raczej zestaw modułów do produkcji osobno gier 2D, 3D i na różne platformy. W sumie posiada możliwość tworzenia gier na platformy PC, Nintendo Wii, Xbox360 oraz te oparte na iOS.

  `Link do strony producenta: http://www.torquepowered.com/`

- *Unreal Engine* - silnik skierowany na platformy PC (oparte na DirectX), Xbox360 oraz Playstation 3. Obecnie silnik w wersji 3 został udostępniony darmowo w postaci UDK - czyli zestawu narzędzi do tworzenia gier, bez możliwości fizycznego programowania. Jest to jedna z najbardziej dojrzałych technologii dostępnych na rynku, posiadająca moduły, które pozwalają tworzyć i modyfikować praktycznie natychmiast dowolny fragment gry wideo od grafiki poprzez fizykę, na dźwięku kończąc.

  `Link do strony producenta: http://www.unreal.com/`

Rysunek 4.9. *Gears of War 2*, gra stworzona na silniku Unreal

- *Virtools* - z założenia jest silnikiem wieloplatformowym do tworzenia gier dowolnego rozmiaru na wszystkie konsole siódmej generacji oraz PC. W rzeczywistości jest to jednak silnik przeznaczony raczej do małych produkcji, które mają powstać bardzo szybko. Wizualny edytor bloczkowy pozwala na bardzo szybkie prototypowanie gry, jednak jest bardzo niewydajny przy większych produkcjach. Implementacja i kompatybilność z konsolą Nintendo Wii jest bardziej umowna niż faktyczna.

  `Link do strony producenta: http://www.3ds.com/`

- *Vision Engine* - podobnie jak w przypadku silnika Virtools, oficjalnie jest to silnik wieloplatformowy z pełną obsługą PC i wszystkich konsol siódmej generacji, jednak implementacja na konsolę Nintendo Wii pozostawia wiele do życzenia i jest aktualnie niekompletna. Sytuacja wygląda znacznie lepiej na pozostałych platformach.

  `Link do strony producenta: http://www.trinigy.net/`

Należy pamiętać, że przez silnik technologiczny rozumie się kompletną technologię potrzebną do stworzenia pewnego zestawu funkcjonalności, na jaki będzie się składała gra. Dlatego też zestawienie darmowych silników jest stosunkowo krótkie, gdyż bardzo mało z dostępnych darmowo technologii jest kompleksowych. Przykładem może być tutaj świetna biblioteka *Box2D*, implementująca zaawansowaną fizykę dwuwymiarową do wykorzystania w grach, jednak nie zajmująca się takimi sprawami, jak na przykład rendering.

Na rynku istnieje bardzo dużo innych silników technologicznych, niektóre bardzo znane, takie jak na przykład IW Engine firmy *Infinity Ward*, czy też RAGE firmy *Rockstar Studios*. Pomimo iż możliwości tych silników są ogromne, np. powstałe na silniku RAGE TPP *Red Dead Redemption* mógł poszczycić się wybitną grafiką, to jednak są one wykorzystywane tylko w produkcjach firm, które te silniki stworzyły. Są to tak zwane silniki *in-house'owe*, niedostępne w komercyjnej sprzedaży.

Coraz częściej stosowane są zabiegi firm tworzących silniki mające na celu spopularyzowanie ich dzieł, przykładowo firma *Epic Games* udostępniła darmowo swój silnik *Unreal Engine* pod postacią UDK (*Unreal Developer Kit*) w roku 2009. Oczywiście wydanie gry stworzonej na tym silniku obarczone jest pewnymi restrykcjami, jednak dało to możliwość nauczenia się korzystania z tej, bardzo uznanej, technologii programistom z całego świata.

4.4 Wieloplatformowość

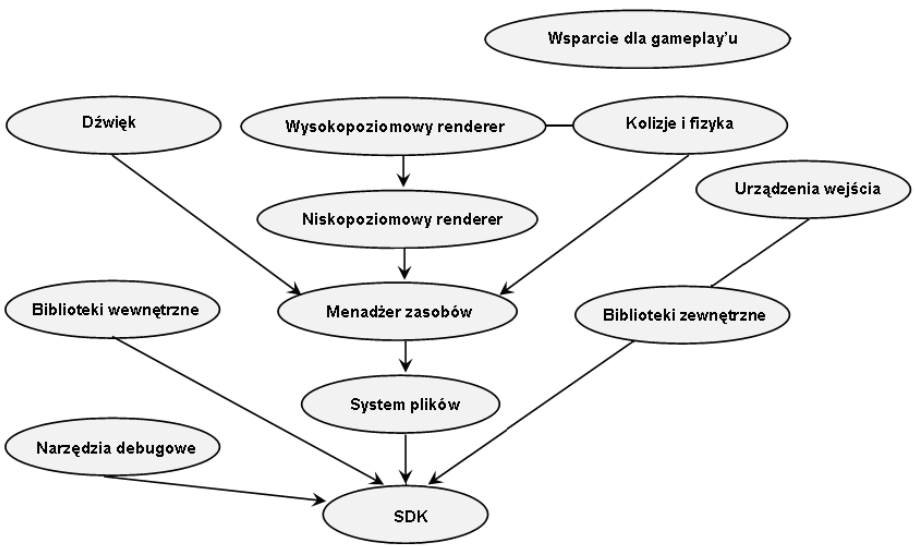
Współcześnie praktycznie każdy silnik jest wieloplatformowy, co widać przykładowo na zestawieniu z poprzedniego podrozdziału. Dzieje się tak z powodu finansowego - coraz mniej opłaca się wydawać grę dedykowaną na pojedynczą platformę, a tworzenie gry od początku na silniku wieloplatformowym jest mniej kosztowne niż portowanie gotowej gry. Oczywiście fakt pisania gry na kilka platform naraz niesie ze sobą kilka problemów, zwłaszcza jeśli różnice sprzętowe między platformami są duże, tak jak przykładowo między konsolami PlayStation 3 a Nintendo DS. Dlatego najczęściej spotyka się dwa rodzaje silników wieloplatformowych: dedykowane na duże platformy oraz dedykowane na wszelkiego rodzaju urządzenia przenośne. Eliminuje to wiele problemów związanych ze znacznymi różnicami w sprzęcie, jednakże w przypadku bardzo intensywnego wykorzystania sprzętu i tak programista napotka wiele problemów wynikających z tych różnic. Choć na początkowym etapie różnica pomiędzy PlayStation 3 i Xbox360 może wydawać się niewielka, tak gdy przyjdzie

czas na solidną optymalizację np. silnika kolizyjnego, gdzie występuje dużo mnożeń macierzy, okaże się, że będzie trzeba napisać oddzielną implementację takich operacji, gdyż różnice w architekturze procesorów tych dwóch platform są przy kilku tysiącach jednoczesnych operacji na tyle duże, że będzie się to opłacało. Im bardziej programista będzie się zbliżał do wykorzystania maksymalnych możliwości z danej platformy, tym bardziej wyspecjalizowana będzie musiała być implementacja silnika pod tę platformę. Niezależnie od problemów, jakie z tego mogą wynikać, warto przy implementacji lub wyborze silnika technologicznego od początku brać pod uwagę wieloplatformowość. Znacznie łatwiej jest wyspecjalizować fragment silnika pod daną platformę niż w trakcie trwania projektu przerabiać jego dolną warstwę, aby w ogóle wieloplatformowość przewidywał.

4.5 Warstwy silnika technologicznego

4.5.1 Wstęp

Silnik technologiczny jest tylko jedną z wielu warstw, na jaką składa się kompletna gra wideo i sam w sobie również jest podzielony na warstwy i moduły. Na rysunku 4.10 pokazano przykładowy rozdział warstw silnika oraz zależności pomiędzy jego modułami. W pionie widać warstwy w silniku, im wyżej, tym większa rola w użyciu tych modułów bezpośrednio w silniku gry. Przykładowo programista gry raczej nie będzie miał bezpośrednio do czynienia z systemem plików, będzie bezpośrednio tworzył przykładowo obiekt dźwiękowy itd.



Rysunek 4.10. Warstwy silnika gry

4.5.2 Menadżer zasobów i system plików

Każda gra składa się z szeregu różnego rodzaju zasobów, jest nim tekstura, model trójwymiarowy, dźwięk czy nawet film wideo. Aby można ich było użyć w grze, potrzebny jest fizyczny dostęp do pliku reprezentującego dany zasób na dysku czy kartridżu, oraz załadowanie go do pamięci. Wyszukiwaniem i wczytywaniem zajmuje się system plików silnika, natomiast zarządzaniem wczytanymi zasobami tzw. menadżer zasobów (ang. *resource manager*).

System plików jest relatywnie prosty w implementacji, jednak uciążliwa bywa obsługa różnic w podejściu do zarządzania plikami między poszczególnymi platformami. Sprowadza się zazwyczaj do zaprojektowania jednego interfejsu dostępu do plików oraz zrealizowania go osobno dla każdej platformy.

Standardowy system obsługi plików powinien posiadać następujące funkcjonalności:

- Składanie i tworzenie ścieżek oraz nazw plików (np. automatyczna obsługa dodawania rozszerzeń dla plików konkretnego typu itp.)
- Otwieranie, zamykanie, czytanie i zapisywanie poszczególnych plików
- Przeszukiwanie folderów
- Asynchroniczny dostęp do plików w celu obsługi dynamicznego wczytywania zasobów (tzw. *streaming*)

Menadżer zasobów musi ściśle współpracować z systemem plików, gdyż przy jego użyciu wczytuje wszelkie zasoby do pamięci. Istnieją zasadniczo dwa podejścia do implementacji menadżera zasobów: scentralizowana i rozproszona. Rozproszony manager zakłada, że za zarządzanie poszczególnymi rodzajami zasobów odpowiedzialne są osobno poszczególne moduły silnika, i tak zasobami graficznymi powinien w takim wypadku zajmować się renderer, dźwiękowymi moduł obsługi dźwięku itd. Do dzisiaj taki sposób realizacji zarządzania zasobami jest stosowany w niektórych silnikach, jednakże w takiej sytuacji trudno jest zapanować nad zużyciem pamięci w trakcie działania gry. W takim podejściu nie istnieje tak naprawdę żadna fizyczna implementacja menadżera zasobów. Architektura opisana poniżej zakłada zcentralizowany menadżer zasobów, który zajmuje się jednocześnie wszystkimi rodzajami zasobów i pozwala na pełną kontrolę zużycia pamięci.

Z punktu widzenia techniki wykonania, jest wiele różnych rodzajów realizacji menadżera zasobów, jedne mogą być oparte na prostym przechowywaniu wskaźników do zasobów, inne na referencjach, jeszcze inne na inteligentnych wskaźnikach. Niezależnie od technicznych aspektów realizacji, każdy menadżer zasobów powinien udostępniać następujące funkcjonalności:

- Możliwość obsługi różnych typów zasobów - idea scentralizowanego menadżera zasobów opiera się na możliwości obsługi wielu różnych typów zasobów. Dla menadżera nie powinna istnieć różnica pomiędzy zarządzaniem plikami dźwiękowymi, teksturami czy meshami.

- Możliwość tworzenia nowych zasobów - użytkownik silnika musi mieć możliwość stworzenia dowolnego zasobu, czyli fizycznego załadowania pliku z dysku do pamięci. Ta funkcjonalność jest silnie związana z systemem plików silnika.
- Możliwość usuwania zasobów - zazwyczaj po usunięciu zasobu przez użytkownika nie jest on jeszcze fizycznie usuwany z pamięci, jedynie oznaczany jako zbędny. W momencie, w którym będzie potrzebna dodatkowa pamięć, zostanie on fizycznie zwolniony. Pozwala to na optymalizację czasów ładowania, gdyż jeśli dodatkowa pamięci nie będzie konieczna w międzyczasie, stworzenie takiego zasobu ponownie będzie praktycznie natychmiastowe.
- Możliwość współdzielenia zasobów - zasób jest niczym innym jak zbiorem danych w pamięci platformy, nic nie stoi na przeszkodzie, by z tych danych korzystało kilka obiektów. Pozwala to przykładowo stworzyć ogromne ilości instancji tego samego modelu przeciwnika na scenie gry, obciążając pamięć tylko jedną instancją.
- Możliwość dodania nowego typu zasobu w prosty sposób - jest to realizowane zazwyczaj poprzez wydzielenie jednego interfejsu, po którym musi dziedziczyć każdy nowo stworzony typ zasobu.

Zrealizowany w sposób ogólny i scentralizowany, menadżer zasobów posiada jeszcze jedną bardzo przydatną cechę - w przeciwieństwie do systemu plików, czy takich elementów silnika jak rendering, menadżer zasobów w założeniach jest na tyle oderwany od postępu technologicznego oraz platformy, że raz dobrze zaprojektowany i zaimplementowany może służyć programiście przez wiele lat, przy wielu różnych wersjach tego samego lub nawet różnych silników.

4.5.3 Urządzenia wejścia

Urządzenie wejścia jest to sposób komunikacji użytkownika z grą wideo. Programista interpretuje dane, które otrzymuje z urządzeń wejścia i przetwarza je na odpowiednią reakcję w świecie gry. Tego typu algorytmy są bardzo różne, od najprostszego przesunięcia obiektu w zależności od wciśnięcia odpowiedniego przycisku aż do skomplikowanego rozpoznawania mowy poprzez porównywanie próbek dźwiękowych.

Każda platforma posiada unikalne dla siebie urządzenia wejścia, mają one jednak wiele elementów wspólnych. W poniższym zestawieniu starano się rozdzielić wszelkie możliwe informacje, jakie może dostać programista, ze względu na rodzaj urządzenia.

Zestawienie pokazane w tabeli, opisuje tylko najbardziej podstawowe urządzenia wejścia. Istnieje jednak duża liczba bardziej złożonych kontrolerów do gier typu: kierownice do gier wyścigowych, czy instrumenty do gier muzycznych. Oprócz tego coraz popularniejsze stają się bardziej zaawansowane metody wykrywa- nia ruchu, przykładowo rozszerzenie do kontrolera Wii Remote o nazwie Wii Motion Plus dodaje żyroskopowe obliczenia do standardowego

Typ urządzenia	Opis	Rodzaj przekazywanych danych
Przycisk	Najbardziej podstawowa forma przekazania informacji przez gracza do gry. Współczesne przyciski w zaawansowanych kontrolerach gier mają często rozróżniane stopnie nacisku, w przypadku zwykłych przycisków jest to zazwyczaj cyfrowe rozróżnienie kilku stopni, jednakże w przypadku tak zwanych spustów interpretacja nacisku może być analogowa	Stan oraz identyfikator przycisku
Wskaźnik	Najbardziej popularny sposób komunikacji użytkownika z platformą komputerów osobistych, zrealizowany pod postacią myszy. Pozwala na bardzo szybką zmianę położenia w dwóch osiach. Wykorzystywane obecnie poza PC m.in. na platformie Wii w postaci wskaźnika na ekranie telewizora	Współrzędne wskaźnika w dwóch osiach
Gałka	Współczesna pochodna joysticka, zwracane wychylenie jest prawie zawsze analogowe, dokładność współczesnych gałek waha się w granicach 10-bitowych wartości wychylenia w osi	Wychylenie w dwóch osiach
Ekran dotykowy	Choć wydaje się najbardziej intuicyjnym sposobem przekazywania informacji, spopularyzowany w grach został niedawno, przez platformy przenośne takie jak Nintendo DS czy iPhone. Rzadko wykorzystywany jest w grach na komputery osobiste, które też mogą być wyposażone w ekrany dotykowe, spowodowane jest to również małą popularnością tego typu urządzeń. Zależnie od technicznych możliwości ekranu dotykowego może on zwracać więcej niż jeden punkt nacisku oraz stopień nacisku	Współrzędne punktów nacisku, ewentualnie stopień nacisku
Akcelerometr	Wykrywa wychylenie kontrolera względem siły grawitacji, dzięki temu może wykrywać również siłę ruchów i analizować przykładowo siły odśrodkowe działające na kontroler. Pozwala to na interpretację imitacji takich ruchów jak np. uderzenie paletką w piłkę. Niestety nie pozwala to na pełne określenie orientacji kontrolera, gdyż nie wykryje obrotu w płaszczyźnie prostopadłej do siły grawitacji - obrót w tej płaszczyźnie w żaden sposób nie wpływa na orientację kontrolera względem grawitacji. Problem tego typu można rozwiązać przy pomocy np. żyroskopu (jest tak np. w akcesorium Wii Motion Plus, które można dołączyć do Wii Remote na konsolę Nintendo Wii)	Wychylenie w trzech osiach względem grawitacji
Kamera	Wykorzystywana do detekcji ruchu i gestów. Prędkość oraz rozdzielczość kamery może się znacznie jednak różnić pomiędzy platformami, przyjęte minimalne parametry do komfortowego wykorzystania kamery w grach to rozdzielczość 640x480 przy prędkości 30 Hz	Strumień danych cyfrowych
Mikrofon	Zazwyczaj wykorzystywany w grach jako urządzenie opcjonalne, głównym problemem użycia mikrofonu w grach wideo jest odróżnienie faktycznych komend wydawanych przez gracza od dźwięków obcych oraz dźwięków wydawanych przez samą platformę, duży nacisk w przypadku tego urządzenia jest położony na algorytmikę interpretacji samych danych, gdyż sposób ich nie zmienia się drastycznie od wielu lat	Strumień danych cyfrowych

akcelerometru, pozwalając nie tylko na wyznaczenie odchylenia w każdej osi, ale również orientacji.

Kolejną formą przekazywania informacji na wejściu jest wcześniejsza interpretacja danych z kilku źródeł i podanie jej programiście w pewnej przetworzonej formie jako funkcje dostępne z poziomu SDK. Firma Microsoft w taki sposób stworzyła urządzenie o nazwie Kinect, które za pomocą kamery, czujnika głębokości oraz czterech mikrofonów ma zapewnić zestaw informacji wejściowych na temat komend głosowych pochodzących z różnych miejsc pomieszczenia oraz wykrywać ruchy szkieletu człowieka. Z kolei firma Sony wypuściła na rynek akcesorium o nazwie PlayStation Move, które łączy funkcjonalność akcelerometru z kamerą śledzącą ruchy tego kontrolera, co ma znacznie poprawić dokładność przekazywanych danych, w porównaniu na przykład z kontrolerem Wii Remote.

Poniżej znajduje się zestawienie najczęściej wykorzystywanych rozwiązań, tym razem z informacją, które platformy udostępniają taką funkcjonalność.

Typ urządzenia	**Domyślnie**	**Opcjonalnie**
Przycisk	• PC (klawiatura, mysz) • PlayStation 3 (14 przycisków na padzie) • Wii (10 przycisków na Wii Remote, 2 przyciski na Nunchucku) • Xbox360 (14 przycisków na padzie)	• PlayStation 3 klawiatura) • Xbox360 klawiatura)
Wskaźnik	• PC (mysz) • Wii (Wii Remote)	• PlayStation 3 PlayStation Move, mysz)
Gałka	• PlayStation 3 (2 gałki na padzie) • PSP (1 gałka) • Wii (1 gałka na Nunchucku) • Xbox360 (2 gałki na padzie)	• PC (pady do PC, Joysticki)
Ekran dotykowy	• DS/DSi	• PC • iPhone
Akcelerometr	• iPhone • Wii (Wii Remote)	
Kamera	• DSi (2 kamery)	• PC • PlayStation 3 Eye Camera) • Xbox360 (Xbox Live Vision)
Mikrofon	• DS/DSi	• PC • PlayStation 3 • Xbox360

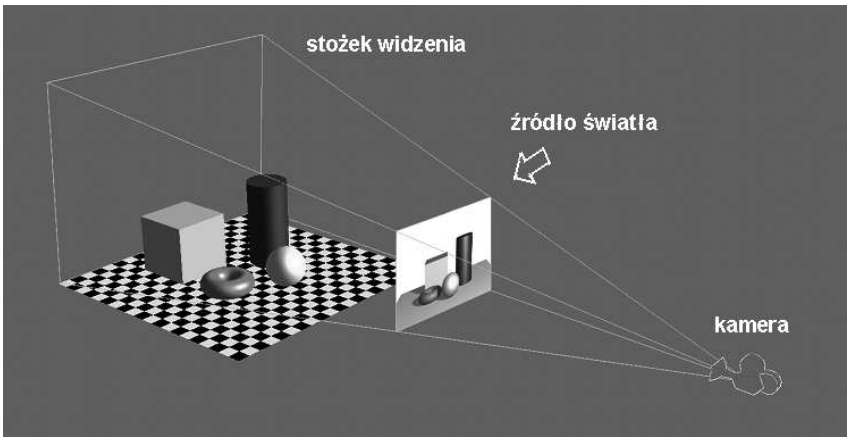
Niezależnie od tego, jak bardzo skomplikowane będzie urządzenie wejścia, z punktu widzenia programisty implementującego silnik istnieją tylko dwa jego rodzaje: analogowe i cyfrowe. Biorąc pod uwagę przekazywane dane, można je interpretować w ten sposób - gałka to urządzenie analogowe, wskaźnik - cyfrowe itd. Przy takim podziale urządzeń wejścia implementacja niezależnego od platformy systemu sterowania w grze wideo jest relatywnie prosta. Wystarczy tylko udostępnić funkcjonalność mapującą odpowiednie przyciski i sygnały, natomiast ich interpretacja może być identyczna, niezależnie od tego, czy jest to przykładowo pad Xbox360 czy PlayStation 3.

4.5.4 Renderer

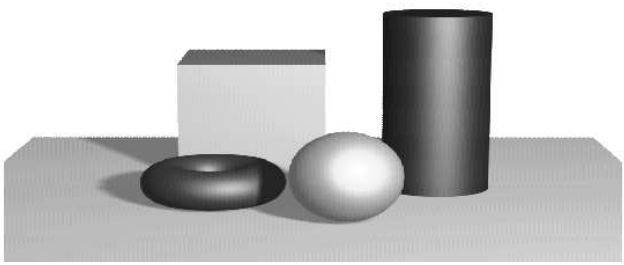
Renderer jest to jedno z najważniejszych, a na pewno najpopularniejszych, zagadnień z zakresu tworzenia gier wideo. W silniku technologicznym renderer odpowiedzialny jest za wyświetlanie grafiki na ekranie. Rendering jest to proces tworzenia obrazu przy użyciu pewnego zestawu danych. W przypadku gier wideo tworzony obraz jest dwuwymiarowy, natomiast dane, z którego ten obraz jest tworzony to matematyczny opis świata gry, często nazywany sceną gry. Zagadnienie renderingu jest jednym z najlepiej opisanych w literaturze związanej z grami wideo. Celem opisu renderingu w tej książce nie jest powielanie tych informacji, jedynie przybliżenie czytelnikowi samego zagadnienia. Rendering do działania potrzebuje zasadniczo czterech rodzajów danych:

- Sceny - opisuje geometrie świata gry, czyli pozycje wierzchołków i połączeń wszystkich *meshy* oraz ich przeźroczystość.
- Kamery - definiują jaki fragment sceny jest faktycznie widoczny dla gracza w danym momencie, fragment ten nazywany jest stożkiem widzenia (ang. *view frustum*). Danych tych używa się również do nierenderowania bez potrzeby elementów zpoza pola widzenia gracza, czyli do tak zwanego *cullingu*.
- Światła - dane te opisują pozycję, orientacje, intensywność, rozproszenie itd. wszystkich źródeł światła na danej scenie.
- Właściwości materiałów - czyli w jaki sposób światło ma wchodzić w interakcji z daną powierzchnią, czy ma pochłaniać światło, odbijać, błyszczeć, być chropowatą itp. W skład danych nt. materiału poszczególnych elementów wchodzi też tekstura koloru, czyli najbardziej podstawowy element materiału pokrywający trójwymiarowe obiekty.

Po zebraniu wszelkich danych potrzebnych do renderingu następuje rozwiązanie równania renderingu, zwanego też równaniem cieniowania - dla każdego fragmentu sceny wewnątrz widzianego obrazu silnik kalkuluje jego kolor na podstawie opisanych wcześniej danych. Istnieje wiele rodzajów algorytmów wykonujących takie równanie. Sztuką w grach wideo jest stworzenie obrazu na tyle dobrze wyglądającego, aby dla gracza był on rzeczywistym, przy tym na tyle małym kosztem obliczeniowym, aby osiągnąć minimalną wymaganą



Rysunek 4.11. Scena 3D z wirtualną kamerą definiującą stożek widzenia, ze źródłem światła, czterema obiektami i podłożem

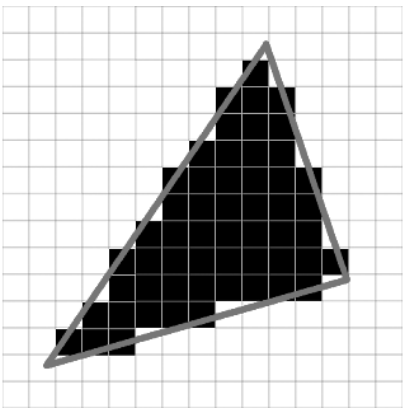


Rysunek 4.12. Render sceny przedstawionej przez rysunek 4.11, w takiej postaci gracz widziałby tę scenę na ekranie

liczbę klatek na sekundę. Należy pamiętać, że renderer musi wykonywać swoje zadania średnio w ciągu 33 milisekund na klatkę, dzieląc ten czas z pozostałymi obliczeniami w grze, takimi jak fizyka, AI itp. Realistyczne przeliczanie oświetlania pojedynczej klatki na wysoko szczegółowej scenie może trwać od kilku godzin do kilku dni, zależnie od mocy sprzętu. Takie metody stosuje się przykładowo przy tworzeniu prerenderowanych przerywników filmowych w grach czy też współczesnych filmów animowanych takich jak np. *Shrek*. Rendering w czasie rzeczywistym, czyli ten obecny w grze podczas jej normalnego działania jest niczym innym jak sprytnym oszukiwaniem widza, poprzez stosowanie algorytmów jedynie przybliżających faktyczne oświetlanie na scenie.

Kiedy renderer posiada już komplet danych gotowych do wyświetlenia, na ekranie następuje rasteryzacja, czyli przetworzenie danych na temat wyrenderowanego dwuwymiarowego obrazu na urządzenie rastrowe, czyli o skończonej

rozdzielczości, takie jak przykładowo monitory. Przykład rasteryzacji przedstawia rysunek 4.13 - szarą linią zaznaczony jest wirtualny trójkąt, na czarno zaznaczone są fizyczne piksele, które zapalił algorytm rasteryzacji, starając się ten trójkąt odwzorować na ekranie.



Rysunek 4.13. Rasteryzacja trójkąta

Powyżej wyjaśniona została podstawowa zasada działania renderera, jednak współcześnie renderer w silnikach technologicznych zajmuje się znacznie bardziej rozbudowanymi zagadnieniami, natomiast sam fakt renderingu sceny na ekran dokonywany jest z poziomu SDKa konkretnej platformy. Funkcjonalności renderera w silniku technologicznym można podzielić na dwie warstwy: niskopoziomową i wysokopoziomową. Renderer niskopoziomowy zajmuje się elementami bardziej technicznymi i działającymi bezpośrednio na SDK, natomiast wysokopoziomowy bardziej efektami graficznymi opartymi na technologiach niskopoziomowych. Poniżej szerszy opis elementów, z jakich się może składać każda z tych warstw:

Niskopoziomowy renderer:

- *Wyrysowanie prymitywów* - czyli poligonów, najczęściej trójkątów. Przy użyciu tego modułu można wyrysować na scenie przykładowo statyczny mesh.
- *Materiały i shadery* - odpowiada za teksturowanie oraz właściwości materiałów poszczególnych obiektów oraz za obsługę shaderów, czyli programów wykonujących masowe obliczenia na wierzchołkach lub pikselach bezpośrednio przy użyciu procesora graficznego, bez obciążania procesora centralnego. Dzięki shaderom można uzyskać ogromną ilość różnych efektów graficznych takich jak wysokiej jakości oświetlenie, odbicia lustrzane, rozmycia itp.
- *Statyczne i dynamiczne oświetlenie* - obsługuje źródła światła wraz z ich parametrami. Statyczne źródła światła wymagają pojedynczego przelicze-

nia ich wpływu na scenę, w przypadku światła dynamicznego wymaga się obliczeń w czasie rzeczywistym, by przykładowo zmieniał się sposób oświetlenia postaci zależnie od tego, pod jakim kątem i w jakiej odległości jest od źródła światła.

- *Kamery* - moduł odpowiadający za obsługę kamery, czyli wirtualnego punktu widzenia gracza.
- *Teksty i czcionki* - system obsługi czcionek i sposobu ich wyrysowywania. Choć system jest relatywnie prosty, to w przypadku obsługi wielu języków posiadających różne alfabety jego skomplikowanie może znacznie wzrosnąć.

Wysokopoziomowy renderer:

- *Animacje szkieletowe* - system oparty w znacznym stopniu na niskopoziomowym module wyrysowywania prymitywów. Odpowiada za możliwość podłączania na zasadzie hierarchii obiektów bądź ich części do kości, czyli matematycznych modyfikatorów, np. pod postacią macierzy. Kości są jedynie obiektami pomocniczymi, nie są renderowane. Dzięki takiemu systemowi możliwe jest przykładowo zrealizowanie fizyki typu *rag-doll*.
- *Rendering GUI* - warstwa GUI jest zazwyczaj w logiczny sposób odseparowana od normalnego renderingu sceny, by nie obejmowały ją systemy kamer sceny czy też kolejności jej renderowania. Moduł GUI powinien zapewniać odpowiedni priorytet w renderowaniu GUI, tak by mógł się zawsze renderować na obrazie sceny. Realizuje się to zazwyczaj poprzez rendering GUI przy użyciu osobnej kamery ortogonalnej.
- *Cienie* - sama niskopoziomowa obsługa światła odpowiada za obliczenie odpowiedniego koloru na poszczególnych elementach sceny, jednak system obliczania cieni jest zazwyczaj od niej odseparowany. Sporadycznie wszystkie obiekty na scenie posiadają dynamiczną obsługę cienia, gdyż byłoby to zbyt obciążające dla współczesnych platform. Dla mniej ważnych elementów stosuje się różnego rodzaju cienie statyczne, jak przykładowo mapy cieni (ang. *shadow maps*), które są wyrysowywane przy renderingu sceny raz, bez dynamicznej aktualizacji.
- *Efekty postprocessingowe* - jest to zbiór efektów, zazwyczaj w postaci zaimplementowanych shaderów, które działają na już wyrenderowanej ramce obrazu. Przykładem takiego efektu może być efekt przepalenia, czyli tzw. *bloom*, który rozjaśnia miejsca, z których dobiega źródło światła, jak pokazuje rysunek 4.14.
- *Efekty cząsteczkowe* - technika renderowania efektów rozmytych takich jak ogień, dym, poruszająca się woda itp. Oparta jest zazwyczaj na emitowaniu dużej ilości bardzo małych dwuwymiarowych i teksturowanych prymitywów poruszających się w określony sposób i z pewnym czasem życia w trójwymiarowym świecie. Każdy taki prymityw nazywany jest cząsteczką, stąd nazwa efektu.

Rysunek 4.14. Efekt przepalenia w grze *Halo 3*

- *Wyświetlanie filmów* - tak zwane FMV (z ang. *Full Motion Wideo*), jest to technologia wyświetlania prerenderowanych filmów w czasie gry, często stosowana przy takich elementach jak intro czy przerywniki filmowe.

4.5.5 Dźwięk

W czasach początkowych gier wideo dźwięk odgrywał drugorzędne znaczenie współcześnie, jako wartość dodana do gry jest bardzo ważny. Odpowiednio dobrana muzyka czy też udźwiękowienie do konkretnej sytuacji w grze wideo może drastycznie zmienić jej odbiór. Znacznie łatwiej stworzyć u gracza konkretne emocje, dodając do akcji odpowiednią muzykę, przykładowo charakterystyczną dla horrorów w momentach zagrożenia itp.

Z punktu widzenia programowania dźwięk jest modułem relatywnie prostym w implementacji. Obecnie zdecydowana większość algorytmiki wymaganej przy realizacji dźwięku jest już dostępna z poziomu SDK dla praktycznie każdej współczesnych platformy. Aktualnie rzadkością jest implementacja zasady działania dźwięku przestrzennego czy efektu Dopplera. Rolą silnika technologicznego jest odpowiednie wystawienie funkcjonalności programiście gry, a w przypadku silnika wieloplatformowego obsługa różnić pomiędzy poszczególnymi SDK. Ogólnie moduł obsługi dźwięku w silnikach można podzielić na następujące elementy:

- Obiekt dźwięku - posiada swoją pozycję i orientację w świecie, co jest bardzo ważne z punktu widzenia dźwięku przestrzennego. Oprócz tego powinien posiadać takie parametry jak zasięg słyszalności, ewentualnie stożek kierunkowy, czyli fragment świata, w którym dźwięk jest lepiej słyszalny. Jest to konieczne, by system dźwiękowy potrafił automatycznie odróżnić sytuacje, w której przykładowo odbiornik radiowy byłby skierowany wprost na gracza lub tyłem do niego. Taki stożek algorytmicznie ma wiele wspólnego ze stożkiem widzenia kamery.
- Bufor dźwięku - to fragment pamięci, w którym zapisany jest rozkompresowany dźwięk, w takiej postaci, w jakiej zostanie bezpośrednio przekazany przez SDKa do DSP (z ang. *„Digital Signal Processor"*), czyli procesora dźwiękowego. Każdy obiekt dźwiękowy jest powiązany z takim buforem.
- Zasób dźwięku - to fizyczny plik dźwiękowy, np. mp3, wczytany do pamięci RAM, może być współdzielony przez poszczególne bufory dźwiękowe. Jest to zazwyczaj element menadżera zasobów.
- Listener - czyli słuchacz, podobnie jak obiekt dźwiękowy również posiada swoją pozycję, orientację, zasięg itd. Odpowiednie przeliczanie parametrów obiektu dźwiękowego i obiektu słuchacza pozwala na rzeczywiste odwzorowanie dźwięku przestrzennego.

Powyższa architektura jest praktycznie tożsama z każdym dźwiękowym SDKa obecnym na rynku, takim jak np. DirectSound czy OpenAL, więc implementacja takiej architektury sprowadza się do odpowiedniego opakowania już dostępnych obiektów i funkcji.

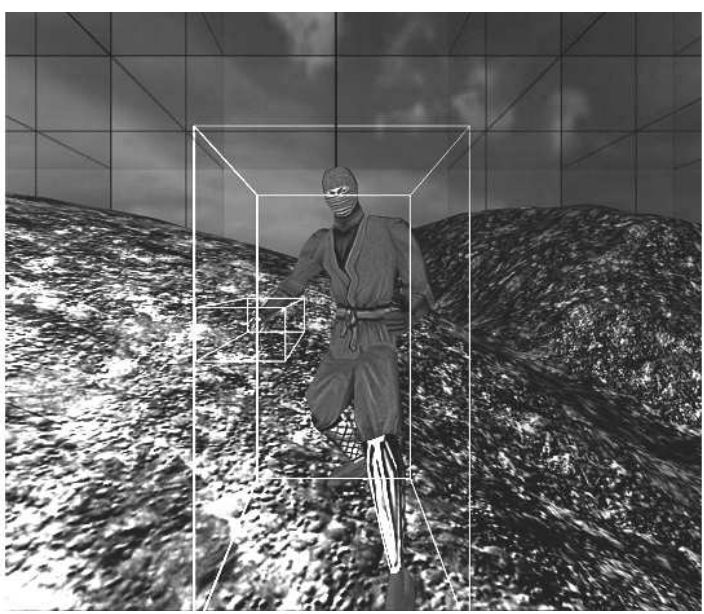
Podkład muzyczny w grze realizowany jest na ogół w oderwaniu od standardowej architektury dźwiękowej. Dzieje się tak z dwóch powodów: po pierwsze muzyka nie powinna być objęta systemem dźwięku przestrzennego, czyli niezależnie od pozycji słuchacza w świecie gry, zawsze powinna być odgrywana w ten sam sposób. Po drugie rozmiar pliku muzycznego jest zazwyczaj tak duży, że marnotrawstwem pamięci byłoby trzymanie go w całości w RAMie. Dlatego też muzyka w grach jest zazwyczaj doładowywana dynamicznie (tzw. *streaming*), natomiast w pamięci jest tylko dostępny aktualnie odgrywany mały jej fragment.

4.5.6 Kolizje i fizyka

Każda gra posiada pewien zbiór zasad rządzący jej światem. Przykładowo, jeśli bohater gry podskoczy, to zazwyczaj użytkownik może się spodziewać, że siłą jakiejś istniejącej w świecie gry grawitacji bohater spadnie zaraz z powrotem na podłoże. Ten relatywnie prosty problem jednak niesie ze sobą wiele pytań w związku ze sposobem implementacji takiego zachowania. Czy postać skacząc, unosi się ruchem jednostajnie opóźnionym? W jaki sposób wykrywany jest powrót do poziomu ziemi? Dawniej wiele gier implementowało tego

typu zachowania pojedynczo, oprogramowując z osobna każdą taką sytuację. Obecnie gry wideo starają się coraz dokładniej odwzorować rzeczywistość, dlatego implementowana jest na potrzeby gry fizyka. Współczesne platformy nie są w stanie w czasie rzeczywistym przetwarzać w pełni realistycznej fizyki, dlatego bardzo ważnym aspektem programowania fizyki w grach wideo jest upraszczanie problemu, na tyle duże, że pozwala na obliczenia w czasie rzeczywistym, a jednocześnie na tyle małe, że użytkownik nie jest w stanie dostrzec uproszczeń, których dokonał programista.

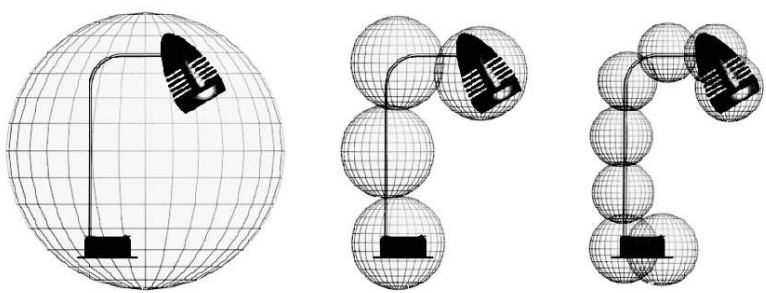
Podstawowymi zagadnieniami modelowania fizyki w grach komputerowych są ruch i detekcja kolizji. Algorytmy detekcji kolizji dostarczają wszystkich potrzebnych informacji dotyczących tego, jakie obiekty zderzały się ze sobą oraz w którym momencie i w jakiej konfiguracji geometrycznej się znajdowały w trakcie kolizji. Fizyka natomiast jest implementacją prawidłowej reakcji na dane dostarczone przez system kolizji. Choć zagadnienia związane z reakcją fizyczną mogą wydawać się skomplikowane, to wszelka wiedza potrzebna do ich implementacji jest ogólnodostępna w postaci już przetworzonych, gotowych do zaimplementowania wzorów. Wbrew pozorom najbardziej skomplikowanym elementem silników fizycznych jest detekcja kolizji, czyli stwierdzenie, czy dwa obiekty w przestrzeni się przecinają. Obiekty w grach są to zazwyczaj modele trójwymiarowe stworzone z wielu poligonów, dokładne obliczanie geometrycznie faktu ich przecięcia jest zadaniem bardzo czasochłonnym



Rysunek 4.15. Prostopadłościany AABB (ang. *Axis Aligned Bounding Box*), czyli objętości ograniczające, zamykające postać oraz jej miecz. W tle widać drzewo podziału przestrzeni. Kadr z ustawień debugowych silnika Irrlicht

i praktycznie niemożliwym do policzenia w czasie rzeczywistym, zwłaszcza gdy należy sprawdzić przecinanie się kilku obiektów o dużej liczbie trójkątów w *meshu* naraz. Zmniejszenie skomplikowania obliczeń sprowadza się zasadniczo do dwóch zagadnień: uproszczenia obliczeń matematycznych oraz minimalizacji liczby testów. Uproszczenie matematyczne osiąga się poprzez zamykanie obiektów w tzw. *bounding volume* (pol. objętość ograniczająca), czyli figury geometryczne, o poziomie skomplikowania znacznie mniejszym niż pełny model obiektu. Sprowadza to analizę kolizji do geometrycznego testu pomiędzy dwoma relatywnie prostymi figurami geometrycznymi, na przykład sferami lub sześcianami (por. rysunek 4.15), zamiast testu kolizji pomiędzy wszystkimi trójkątami, z jakich składa się jeden model, a wszystkimi trójkątami drugiego modelu. Zależnie od żądanej dokładności sprawdzenia kolizji dobiera się objętości ograniczające coraz dokładniej odwzorowujące zamykany obiekt. Często obiekt zamykany jest w kilka takich brył, z rosnącym stopniem dokładności. Jeśli wykryto kolizję przy sprawdzeniu pierwszej, najmniej dokładnej figury, sprawdza się kolizje z następną, aż do ewentualnego wykrycia kolizji z najdokładniejszą objętością, gdzie pozostaje sprawdzenie kolizji na poziomie trójkątów lub zaakceptowanie danej dokładności detekcji.

Zamykanie obiektów w objętości ograniczające przed testem kolizyjnym, by uprościć geometrię modelu, znacznie skraca czas testu. Jednak liczba testów pozostanie ta sama, tak więc zamykanie obiektów w objętości ograniczające przyspiesza proces detekcji o pewną stałą, lecz nie zmienia klasy złożoności czasowej całej detekcji. Ułożenie objętości ograniczających w hierarchię drzewiastą, zwaną BVH (*Bounding Volume Hierarchy*, pol. hierarchia objętości ograniczających), pozwala na redukcję złożoności algorytmu detekcji kolizji do złożoności logarytmicznej, z punktu widzenia wykonanych testów, przykład takiej hierarchii pokazuje rysunek 4.16.



Rysunek 4.16. Kolejne trzy poziomy sferycznej BVH, w którą zamknięty jest model lampy, każdy kolejny poziom ciaśniej otacza model, jednak wymagane jest więcej testów, by stwierdzić kolizję

Minimalizacji liczby testów dokonuje się również poprzez podział przestrzeni świata gry na mniejsze części, tak by tylko ruchome elementy świata, które faktycznie znajdują się na tyle blisko siebie, by istniała szansa interakcji po-

między nimi, wykonywały pomiędzy sobą testy kolizyjne. Zależnie od rodzaju świata gry, sposób konstrukcji drzewa podziału przestrzeni może się różnić, np. w przypadku gier odbywających się w zamkniętych pomieszczeniach naturalne byłoby stworzenie drzewa podziału przestrzeni opartego na pokojach, tak by dwa obiekty znajdujące się w różnych pomieszczeniach nigdy nie sprawdzały ze sobą kolizji. Obecnie przyjęło się, iż zazwyczaj drzewo podziału przestrzeni, niezależnie od sposobu konstrukcji, jest drzewem ósemkowym, gdyż w wielu testach na współczesnych maszynach to drzewo sprawowało się najlepiej. Kiedyś szeroko używano do tego celu drzewa binarnego.

Z wykrywaniem kolizji wiąże się też wiele problemów, takich jak optymalne implementacje matematyczne poszczególnych algorytmów detekcji, przykładowo trójkąt - trójkąt, problem tunelowania, czyli branie pod uwagę funkcji ruchu w czasie przy wyliczaniu kolizji pomiędzy obiektami, czy też problemy związane z dokładnością zmiennoprzecinkową obliczeń oraz wiele innych.

4.5.7 Biblioteki wewnętrzne

Przez biblioteki wewnętrzne rozumiemy wszelkie biblioteki będące częścią silnika technologicznego, SDK czy też systemu operacyjnego platformy, na którą tworzona jest gra. Przykładem może być biblioteka generatora liczb losowych, biblioteka funkcji matematycznych czy biblioteka obsługi urządzeń wejściowych. Liczba takich mechanizmów pomocniczych jest mocno uzależniona od tego, czy silnik jest wieloplatformowy, czy nie. W przypadku silnika dedykowanego pod jedną platformę liczba takich funkcji i klas pomocniczych będzie bardzo mała, gdyż większość potrzeb będą pokrywać funkcje udostępniane przez SDK platformy lub biblioteki zewnętrzne dedykowane pod tę platformę. Przykładowo pisząc grę na komputer osobisty w języku C++, wszelkie funkcje matematyczne będą brane z bibliotek C, np. cmath. Jednak w przypadku silników wieloplatformowych każde SDK działa trochę inaczej i ma różne zestawy funkcjonalności dodatkowych. W takim wypadku istnieją dwa rozwiązania: albo programista implementuje daną funkcjonalność sam, dbając aby kod był niezależny od platformy, albo tworzy tzw. *wrapper funkcjonalności*, czyli udostępnia jeden interfejs, który odpala różne funkcje SDK, zależnie od tego, na jakiej platformie jest uruchamiany.

Poniżej przykład opakowania funkcji zwracającej losową wartość dodatniej liczby całkowitej:

```
// returns a~random integer between 0 and iMax
int RandomInt(int iMax)
{
#ifdef PLATFORM_PC

    // the PC implementation
    srand(time(NULL));
    return (rand() % iMax);

#endif
```

```
#ifdef PLATFORM_DS

    // the DS implementation
    MATHRandContext32 * context;
    MATH_InitRand32(context, (u64)OS_GetTick());
    return MATH_Rand32(context, iMax);

#endif

    // no platform defined
    return -1;
}
```

Dla odróżnienia w tym przykładzie platform użyto osobnych definicji dla każdej platformy w ramach jednego pliku źródowego. Jednak w przypadku dużej liczby platform i klas do opakowania lepszym rozwiązaniem może być wspólny plik nagłówkowy i podłączanie osobnych źródeł dla każdej platformy.

4.5.8 Biblioteki zewnętrzne

Przez biblioteki zewnętrzne rozumiemy wszelkie biblioteki niebędące bezpośrednio ani częścią silnika technologicznego, ani SDK czy też systemu operacyjnego platformy, na którą tworzona jest gra. Biblioteki zewnętrzne mogą pokrywać bardzo dużą gamę funkcjonalności silnika, który w takim wypadku przyjmuje formę szkieletu spinającego w jedną całość elementy pochodzące z wielu miejsc. Poniżej zestawienie najczęstszych elementów silnika, zapożyczanych z innych bibliotek:

- Kodeki wideo (np. *Bink Wideo*)
- Kodeki audio (np. FMOD)
- Silnik fizyczny i kolizyjny (np. *Havok*)
- Wspomaganie programowania (np. *Boost*)
- Generowanie roślinności (np. *Speed Tree*)

Użycie bibliotek zewnętrznych może zaoszczędzić wiele pracy przy rozwoju silnika technologicznego, pod warunkiem że istnieją możliwości finansowe, by kupić licencję na ich użycie. Wszystkie przykłady podane przy powyższym zestawieniu to produkty komercyjne, choć dla niektórych zastosowań istnieją darmowe biblioteki (np. wspomniany już wcześniej Box 2D do fizyki dwuwymiarowej). W przypadku silników wieloplatformowych należy zwracać uwagę na to, czy użyta biblioteka zewnętrzna działa na wszystkich platformach, na jakich powinien działać silnik. Jeżeli tak nie jest, należy zapewnić substytut do każdej platformy. Przykładowo biblioteka audio FMOD obsługuje prawie każdą współczesną platformę poza Nintendo DS, w przypadku użycia FMOD w silniku technologicznym, który ma obsługiwać między innymi konsolę Dual Screen, należy pomyśleć o zastosowaniu innej biblioteki tylko na potrzeby tej konsoli, albo zaprogramowaniu funkcjonalności, jaka będzie wykorzystywana z FMOD na innych platformach, w oparciu o SDK DSa.

4.5.9 Narzędzia debugowe

Przez narzędzia debugowe w silniku technologicznym rozumie się pewien zestaw funkcjonalności pomagający programiście na etapie implementacji. Funkcjonalności te zazwyczaj nigdy nie są częścią finalnego produktu. Poniżej zestawienie kilku przykładów tego typu narzędzi:

- *Konsola debugowa* - to jedno z najbardziej podstawowych narzędzi używanych w praktycznie każdym silniku gry. Konsola debugowa składa się zazwyczaj z wydzielonego pola, na którym jakąś podstawową czcionką wyświetlane są różnego rodzaju informacje. Rodzaj tych informacji może być dwojaki, mogą być to informacje, które silnik wystawia programiście gry, lub mogą być to informacje, które sam sobie wypisuje programista, by kontrolować stan swojej aplikacji w czasie rzeczywistym.

  Sposób realizacji takiej konsoli może się różnić zależnie od platformy. Na komputerach z systemem operacyjnym Windows bardzo popularnym rozwiązaniem jest uruchamianie osobnego okna z konsolą shell, jednak brak przenośności takiego rozwiązania prowadzi zazwyczaj do wspólnej implementacji wyświetlania informacji na ekranie gry w przypadku silników wieloplatformowych.

- *Logi* - to forma przechowania informacji w postaci dynamicznie tworzonego pliku. W logach można przechowywać różnego rodzaju informacje, często mogą być tożsame z informacjami wyświetlanymi na konsoli debugowej. Zaletą logów nad konsolą jest fakt ich pozostania do analizy nawet po zatrzymaniu gry, wadą jest utrudniona analiza w czasie rzeczywistym, dlatego często te narzędzia stosuje się uzupełniająco. Zazwyczaj implementacje pozwalają na stworzenie różnych kontekstów, co daje możliwość, przykładowo trzymania danych pamięci w osobnym pliku niż dane na temat wydajności.

  W pewnych przypadkach liczba danych zapisywana do logów może być bardzo duża, dlatego istnieją również mechanizmy pozwalające kontrolować szczegółowość zapisu do logów. W momencie wywołania funkcji odpowiedzialnej za zapis, programista musi podać poziom ważności danej informacji. Wtedy zależnie od aktualnych ustawień silnika, jeśli dana informacja nie będzie miała wystarczająco wysokiego poziomu ważności, nie będzie zapisana. Często poziomy ważności są stosowane wymiennie z kontekstami, lub są pewną formą mieszaną, jak podany poniżej przykład poziomów logowania:

  - *all* - wszystko jest zapisywane do logów
  - *info* - informacyjne dane
  - *debug* - dane debugowe
  - *warning* - ostrzeżenia, zazwyczaj sugerujące o potencjalnej możliwości wystąpienia błędu

- *error* - błędy w działaniu algorytmów, mogą prowadzić do nieprawidłowych zachowań gry, jednak nie powinny prowadzić do jej zatrzymania
- *fatal* - błąd, który spowodował natychmiastowe zatrzymanie działania gry
- *off* - nic nie jest zapisywane do logów

  Według podanego powyżej przykładu, jeżeli programista ustali poziom logowania na error, do logów powinny trafiać tylko wpisy oznaczone jako fatal i error.

- *Memory Leak Trace* (z ang. śledzenie wycieków pamięci) - to fragment silnika odpowiedzialny za rejestrowanie wszelkich alokacji pamięci w całym kodzie gry oraz usuwanie ich ze spisu w momencie jej zwalniania. Pozwala to na wykrycie pamięci niezwalnianej, przykładowo poprzez zapomnienie przez programistę wywołania operatora *delete* na wcześniej zaalokowanym wskaźniku. Implementacja takiego śledzenia zazwyczaj opiera się na przeciążeniu operatorów *new* i *delete*. Wewnątrz takiego przeciążanie dodawane i usuwane są wpisy z listy aktualnie zajętej pamięci. Forma zapisu to zazwyczaj adres alokowanej pamięci oraz jej kontekst. Inny kontekst będzie miała alokacja wewnątrz silnika i inny przykładowo alokacja programisty gry używającego silnika. Poniżej przykład implementacji takiego przeciążenia, klasa *CMemoryLeakTrace* odpowiada za przechowywanie listy zaalokowanej pamięci:

```
// we overload the memory allocation operator
void * operator new (unsigned int blocksize)
{
    if(blocksize == 0) return NULL;

    void * ptr = malloc(blocksize);

    CMemoryLeakTrace::AddEntry(ptr,blocksize);

    return ptr;
}

// we overload the memory release operator
void operator delete (void * block) throw()
{
    free(block);

    CMemoryLeakTrace::RemoveEntry(block);
}

Prosta implementacja funkcji dodaj/acej wpis mog/laby wygl/ada/c nast/
     epuj/aco:

// adds entry
void CMemoryManager::AddEntry(void *pAddress,unsigned int uiSize)
{
    SMemEntry sMemEntry;
    sMemEntry.pAddress = pAddress;
    sMemEntry.uiSize = uiSize;
    sMemEntry.pszFileName = m_pszFileName;
```

```
    sMemEntry.iLine = m_iLine;
    sMemEntry.eContext = m_eLatestContext;

    m_vEntries.push_back(sMemEntry);
}
```

Oczywiście klasa *CMemoryManager* musi posiadać pole przechowujące wszystkie wpisy, przykładowo zrealizowane na wektorze:

```
static std::vector<SMemEntry> m_vEntries;
```

Zmianę kontekstu można zrealizować poprzez funkcje wywoływane z poziomu klasy *CMemoryLeakTrace*. Konteksty powinny działać na zasadzie stosu, czyli zmieniając kontekst na nowy, wrzucany jest na szczyt stosu historii kontekstów, kiedy skończy się dany kontekst, wywołujemy funkcję odpowiedzialną za przywrócenie poprzedniego. Poniżej przykładowe deklaracje takich funkcji:

```
//! set memory leak context
static void SetMemoryLeakContext(CMemoryLeakTrace::CONTEXT eContext);

//! set previous memory leak context
static void SetPreviousMemoryLeakContext();
```

Klasa *CMemoryLeakTrace* powinna posiadać funkcje statyczne, aby programista mógł odwołać się do nich z dowolnego miejsca kodu, bez tworzenia nowej instancji klasy.

Dzięki takiemu podejściu do kontekstów programista może przykładowo stworzyć kontekst, w którym będzie widział na spisie wycieków pamięć zaalokowaną wewnątrz konkretnego fragmentu algorytmu. Do odczytania aktualnego stanu zaalokowanej pamięci może być użyta konsola debugowa lub zapis do logów.

- Statystyki - jest to pewien zbiór informacji, jakie przechowuje silnik, służące programiście do oceny stanu aplikacji w czasie wykonywania kodu. Poniżej kilka przykładów najczęstszych statystyk dostępnych w silnikach technologicznych:
  - FPS (*„Frames per Second”*, z ang. klatki na sekundę), czyli podstawowy parametr informujący o ogólnej wydajności gry. Podawanie FPS aktualizowanych w czasie rzeczywistym jest mało użyteczne, gdyż czasem duże spadki mogą umknąć uwadze obserwującego, dlatego najczęściej stosuje się wyświetlanie średniej z ostatnich kilku sekund. Osobno można wyświetlać takie informacje jak maksymalna i minimalna wartość, średnia z ostatniej minuty itp.
  - Wolna/Zajęta pamięć RAM.
  - Liczba obiektów/trójkątów/poligonów, które są aktualnie renderowane na scenie.

  Do wyświetlania statystyk najczęściej służy konsola debugowa, ale można też używać zrzutu do logów.

- Wyświetlanie obiektów kolizyjnych - bardzo częsty mechanizm wykorzystywany w silnikach kolizyjnych i fizycznych. Polega na osobnym renderowaniu samych obiektów kolizyjnych, czyli wszelkich objętości ograniczających. Pomaga to w stwierdzeniu, czy faktycznie kolizja powinna zostać zgłoszona czy nie oraz pozwala na ocenę, czy dokładność przylegania objętości do modelu jest wystarczająca. Często również renderowane są drzewa podziału przestrzeni, by użytkownik widział, w jaki sposób silnik kolizyjny podzielił świat gry. Realizacja takiego renderowania zazwyczaj jest bardzo prosta, opiera się tylko i wyłącznie na rysowaniu linii w różnych kolorach, przy użyciu najprostszych metod dostępnych z poziomu SDK platformy. Przykładowo na platformie korzystającej z DirectX 9 służyłby do tego interfejs ID3DXLine.
- *Free Camera* (z ang. wolna kamera) - mechanizm używany tylko w silnikach 3D, polega na włączeniu pod jakimś tymczasowych przyciskiem wolnej kamery podczas działania gry. Przy pomocy zdefiniowanych przycisków do sterowania użytkownik może swobodnie poruszać kamerą we wszystkich płaszczyznach przesunięcia i obrotu. Pozwala to na dostrzeżenie wielu elementów gry, które mogą być trudniej dostępne z kamery używanej w samej grze, przykładowo, jeśli jest to gra typu TPP.

Zależnie od różnych implementacji silnika, może być wiele innych rodzajów narzędzi dostępnych jako pomoc dla programisty. Należy pamiętać, by ta funkcjonalność mogła być w łatwy sposób wyłączona, szczególnie na przykład wtedy, gdy tworzy się finalną grę.

Nie należy mylić narzędzi dostępnych jako część silnika technologicznego z zewnętrznymi narzędziami takimi jak np. aplikacje profilujące wydajność. Tego typu narzędzia omawiane są w rozdziale 6.

4.5.10 Wsparcie dla gameplayu

W przypadku silnika technologicznego, całkowicie oderwanego od silnika gry, trudno wymagać dużego wsparcia dla konkretnych mechanizmów pomagających przy tworzeniu gry. Jednak sytuacja takiego całkowitego oderwania istnieje bardzo rzadko, zazwyczaj każdy silnik technologiczny na początku istnienia tworzony był z myślą o implementacji konkretnej gry. Dlatego dużo silników technologicznych posiada zaimplementowane wiele mechanizmów wspierających dany typ gry. Przykładowo silnik *Irrlicht* posiada zaimplementowany kompletny mechanizm kamery z widoku pierwszej osoby, na modłę gier FPS, połączony z mechanizmem wykrywania kolizji z terenem i innymi obiektami. Pozwala to w sposób praktycznie natychmiastowy stworzyć szkielet gry FPS ze światem, w którym gracz może się poruszać. Zaimplementowano dodatkowo nawet uproszczoną fizykę skoku oraz wykrywanie kąta nachylenia terenu, by gracz nie mógł wejść na zbyt strome zbocze terenu. Choć wsparcie przy

tworzeniu gier od strony silnika technologicznego bywa pomocne, tak im bardziej szczegółowo pomoc nastawiona jest na konkretny element gry, tym mniej jest uniwersalna. Przykładowo tworząc grę RTS w oparciu o *Irrlichta*, mamy wbudowaną niepotrzebnie obsługę kamery FPP. Lepsze podejście prezentuje silnik *Gamebryo*, który zbudowany warstwowo pozwala na dołączenie konkretnych modułów w postaci nakładek na rdzenny silnik technologiczny. Poniżej, przykładowa lista modułów, jakie może posiadać silnik technologiczny:

- **Sterowanie** - wszelkie mechanizmy już wstępnie przetwarzające sygnały z urządzeń wejścia, przykładowo rozpoznawanie konkretnych gestów na ekranie dotykowym
- **Sztuczna inteligencja** - wysoko poziomowe ramy na których może być oparta konkretna implementacja sztucznej inteligencji, np. zestaw podstawowych klas do maszyny stanów lub sieci neuronowych
- **Kontrolki UI** - silnik technologiczny może posiadać podstawową implementację kontrolek interfejsu takich jak przyciski, suwaki itp. Ważne jest by zachowania i wygląd tych elementów był w pełni modyfikowalny
- **Efekty cząsteczkowe** - choć praktycznie każdy silnik posiada możliwość tworzenia efektów cząsteczkowych, silnik może również zapewniać pewnego rodzaju modyfikowalne szablony konkretnych efektów, jak np. ognia.
- **Obsługa kamery** - pewne szablony podstawowych zachowań kamery, np. na potrzeby FPP czy TPP.

Moduły te są na tyle uniwersalne, że nie muszą one składać się na część silnika gry. Głównym problemem z umieszczaniem elementów wspierających tworzenie gry w silniku technologicznych jest brak możliwości modyfikacji tych elementów na potrzeby konkretnej instancji gry. Przykładowo gdyby powstawały równolegle dwie gry FPS na silniku *Irrlicht*, obie korzystałyby z mechanizmu sterowania i kamery FPS wbudowanych w ten silnik. Tak długo jak sterowanie w tych dwóch grach różniłoby się jedynie parametrami, jakie udostępnia *Irrlicht* do tej funkcjonalności, nie będzie problemu. Jednak jeśli w którejś z gier pojawi się potrzeba modyfikacji algorytmu interpretacji stromego zbocza, pojawi się problem. Albo w obu grach będzie trzeba używać tego samego mechanizmu i zmiana jest wprowadzana w *Irrlichtcie*, albo cały mechanizm musi zostać nadpisany własną implementacją. O ile pierwsze wyjście jest zazwyczaj absolutnie nie do przyjęcia, gdyż uniwersalność silnika technologicznego nie powinna być nigdy naruszana z powodu gry, tak drugie wyjście wymaga sporo nadmiarowej pracy, gdyż z początku wydawało się, że prawie cały mechanizm sterowania i kamery jest gotowy. Najbezpieczniejszym rozwiązaniem jest implementowanie mechanizmów gry w warstwie silnika gry w taki sposób, by silnik technologiczny był jak najbardziej uniwersalny. Wyjątkiem może być silna specjalizacja silnika w kierunku tworzenia gier z konkretnego gatunku, jak na przykład *Unreal Engine*, gdzie możliwości dostosowania kamery FPP i sterowania są tak dobrze rozbudowane i parametryzowane, że zadowolą każdą grę z gatunku FPS.

5

Architektura silnika gry

5.1 Wstęp

We wcześniejszym rozdziale opisanych zostało wiele mechanizmów udostępnianych przez silnik technologiczny, takich jak wyświetlanie obiektów przez renderer, obliczanie kolizji i fizyki, odgrywanie dźwięków i pobieranie danych z urządzeń wejścia. Jednak wszystkie te mechanizmy razem nadal nie składają się na grę wideo. To, co definiuje grę, to *gameplay*, a konkretniej: mechanizmy na niego się składające. Mechanizmy te są jak zasady rządzące światem gry, cele stawiane przed graczem, kryteria porażki i sukcesu, składają się na ogólne doświadczenie gry przez gracza. Silnik gry jest tą warstwą, która ma za zadanie udostępnić gotową implementację tych mechanizmów lub ją ułatwić. W przypadku rozbudowanych systemów do tworzenia gier, gdzie oprócz warstwy silnika gry dostępne są narzędzia do jej tworzenia, takie jak edytor, silnik gry ma zadanie zinterpretować dane dostarczone przez edytor i przy użyciu warstwy silnika technologicznego wprowadzić je w życie. Przykładowo, jeżeli w edytorze designer stworzy postać przeciwnika, nadając mu odpowiednie parametry, takie jak punkty życia oraz oskryptuje jego zachowanie, to zadaniem silnika gry będzie interpretacja skryptu zachowania przez moduł AI oraz obsługa mechanizmów związanych w warunkami życia i śmierci danej postaci. Ponadto silnik gry użyje udostępnionych przez silnik technologiczny mechanizmów wyświetlania i animowania modelu postaci.

W przypadku silnika technologicznego mechanizmy zazwyczaj implementowane są z myślą o jak najszerszym zastosowaniu. Jedyne ograniczenia nałożone na silnik technologiczny mogą mieć związek w dedykowaniem go pod konkretną platformę lub konkretny gatunek gry. Silnik gry jest dużo bardziej wyspecjalizowany. Jego podstawowym ograniczeniem jest zazwyczaj dedykowanie pod konkretny gatunek, mało tego, pod konkretny sposób prezentowania rozgrywki. Przykładem takiego silnika jest *Wintermute Engine*. Jest to darmowy silnik, technologicznie oparty na DirectX, który udostępnia kompletny zestaw narzędzi oraz mechanizmów do tworzenia gier przygodowych 2D

Definicja 5.1. Gameplay

Po polsku można określić *gameplay* mianem mechaniki rozgrywki. *Gameplay* jest to zestaw mechanizmów gry, które definiują, w jaki sposób doświadcza jej gracz. Przykładowo, *gameplay* w typowym FPS sprowadza się do sterowania postacią z kamery pierwszej osoby za pomocą klawiatury i myszki, eksploracji terenu, zazwyczaj zamkniętych przestrzeni oraz pokonywania napotkanych wrogów, których zadaniem jest zabić gracza. Często recenzenci oceniając jakość danej gry, używają słowa *gameplay*. Wtedy sam *gameplay* oceniany jest w oderwaniu od jakości grafiki, dźwięku czy fabuły gry. Nie należy mylić *gameplayu* z często używanym w recenzjach słowem grywalność. Grywalność oznacza jakość prowadzonej rozgrywki, a *gameplay* sposób jej prowadzenia. Wybrany *gameplay* może cechować się dużą lub małą grywalnością.

i 2.5D (w grach 2.5D trójwymiarowa postać porusza się po dwuwymiarowych, zazwyczaj prerenderowanych ze scen 3D, tłach). Tak wysoki poziom specjalizacji silnika uniemożliwia stworzenie gry, nawet przygodowej, która reprezentowałaby rozgrywkę w inny sposób, na przykład poprzez pełne 3D.

W tym rozdziale przybliżone zostaną elementy mechanizmów gry wspólne dla wielu gatunków oraz problemy, jakie musi rozwiązywać każdy silnik gry. Należy jednak pamiętać, że istnieją przypadki gatunków gier czy też konkretnych mechanizmów, które niekoniecznie będą wpasowywały się w poniżej przedstawione ramy.

5.2 Świat gry

Zdecydowana większość gier wideo, zarówno 3D, jak i 2D, odbywa się w pewnym zdefiniowanym wirtualnym świecie. Na ten świat składa się wiele różnych elementów, które zazwyczaj dzieli się na dwie podstawowe kategorie: statyczne i dynamiczne. Statyczne elementy to teren, budynki, drogi i wszystkie inne elementy świata gry, które nie wchodzą w aktywny sposób w interakcje z graczem ani z innymi dynamicznymi elementami gry. Elementy dynamiczne to postacie, pojazdy, bronie, oświetlenie, efekty cząsteczkowe itp. Zazwyczaj *gameplay* w grach skoncentrowany jest wokół elementów dynamicznych świata gry. Nie należy mylić elementów statycznych i dynamicznych z elementami ruchomymi i nieruchomymi. Statyczny element rozgrywki może być ruchomy (np. animowany wiatrak, w którym gracz nic nie może zrobić, nie może go zatrzymać, zniszczyć itp.). Wszelkie mechanizmy kontroli nad stanem świata gry odnoszą się do aktualnego stanu wszystkich elementów dynamicznych znajdujących się w świecie gry. Elementy statyczne stanowią pewną stałą, która nie wpływa w znaczny sposób na skomplikowanie algorytmiczne czy też wydajnościowe mechanizmów kontroli stanu gry.

Zależnie od gatunku gry stosunek elementów dynamicznych do statycznych może się różnić. Im większa jest liczba elementów dynamicznych w świecie gry, tym bardziej żywa się ona wydaje. Zazwyczaj z przyczyn wydajnościowych elementów statycznych w grach jest znacznie więcej niż dynamicznych. Stosunek ten ciągle się zmienia na korzyść elementów dynamicznych, co związane jest ze stałym wzrostem możliwości sprzętowych kolejnych platform.

Rozróżnienie elementów statycznych od dynamicznych bywa niekiedy rozmyte. Mogą istnieć elementy z pozoru dynamiczne, takie jak na przykład poruszająca się roślinność. Jednak tak długo jak ta roślinność będzie się poruszać w sposób tylko przez nią zdefiniowany, bez brania pod uwagę zewnętrznych czynników, takich jak zmienny kierunek wiatru czy postać przechodzącą przez nią, wiele silników będzie traktowało te elementy jako statyczne. Silniki mogą różnić się od siebie podejściem do rozróżniania elementów statycznych i dynamicznych. W skrajnym przypadku dla silnika wszystkie elementy mogą być dynamiczne, w sensie takim, że są potencjalnie dynamiczne, czyli można dowolnie zmieniać ich stan w czasie rzeczywistym.

Powodem rozróżnienia elementów statycznych i dynamicznych jest optymalizacja. Mniejsza moc obliczenia zużywana jest na obiekty, o których wiadomo, że ich stan się nie zmieni. Przykładowo współrzędne wierzchołków statycznego *mesha* można przeliczyć na współrzędne świata, przed wykonywaniem kodu, co pozwoli uniknąć przemnażania macierzy transformacji z lokalnego układu współrzędnych modelu do współrzędnych świata, każdego wierzchołka siatki w czasie renderingu.

! Definicja 5.2. Mesh

Mesh (z ang. siatka) jest to zbiór wielokątów połączonych ze sobą krawędziami. Zwykle siatki tworzone są z trójkątów lub czworokątów wypukłych. Siatka jest to podstawowy sposób przedstawiania geometrii trójwymiarowej w grach wideo.

! Definicja 5.3. Lightmap

Lightmap (z ang. mapa oświetlenia) są to dane określające oświetlenie danej powierzchni. Realizacja mapy oświetlenia może być różna, najczęściej stosowane jest przeliczanie koloru każdego wierzchołka na podstawie mapy oświetlenia lub nałożenie jej jako dodatkowej warstwy tekstury. Mapy oświetlenia są generowane dla statycznych obiektów przed ich renderowaniem w czasie rzeczywistym.

Oświetlenie również może być wstępnie policzone w postaci na przykład map oświetlenia (tzw. *lightmapy*). Praktycznie każde obliczenie, które musi

być wykonane w czasie wykonywania programu dla obiektów dynamicznych, może być dobrym kandydatem do wcześniejszego obliczenia lub wręcz pominięcia w przypadku obiektów statycznych.

Gry w których znajduje się wiele elementów zniszczalnych, są dość trafnym przykładem silnego rozmycia się różnicy między elementami statycznymi a dynamicznymi. Zniszczenia, na przykład budynków, realizowane są najczęściej na elementach statycznych, a w momencie zniszczenia podmieniana jest geometria budynku. Ta podmiana wymaga jednak zarządzania charakterystycznego dla elementów dynamicznych gry. Granica między obiektami statycznymi i dynamicznymi może być przesuwana, zależnie od tego, w którą stronę będzie kładziony większy nacisk - czy na optymalizację, czy na swobodę projektowania świata gry.

Zdefiniowanie, jakie elementy są statyczne, a jakie dynamiczne, może stanowić optymalizację nie tylko z punktu widzenia programowania. Przykładowo, statyczna geometria w postaci siatki zazwyczaj tworzona jest przez grafików w programach typu *Maya* lub *3D Studio*. Grafik, który tworzy siatkę danego obiektu, może ją stworzyć jako jedną całość lub w częściach. Takie części mogą być podstawą do instancjonowania geometrii statycznej na scenie. Instancjonowanie służy głównie oszczędzaniu pamięci, polega na wyświetlaniu relatywnie mało skomplikowanej geometrii w wielu miejscach w świecie gry i w różnej orientacji. Typowym przykładem może być roślinność (np. wielokrotne użycie tego samego modelu drzewa), budynki itp. Przykładem ekstremalnym zastosowania tej techniki jest gra *Hellgate: London*. Na potrzeby losowego tworzenia lokacji wszystkie poziomy w tej grze były składane z wcześniej przygotowanych klocków geometrii takich jak podniszczony budynek, fragment ulicy w ławką i latarnia itp.

5.2.1 Przeładowywanie świata gry

Gry często rozgrywają się w bardzo dużych wirtualnych światach i nie ma możliwości przechowywania w pamięci kompletu danych na jego temat w czasie wykonywania programu. Dlatego często świat jest podzielony na fragmenty. Czy rozróżnienie na te fragmenty jest jasno widoczne dla gracza, w postaci na przykład ekranów ładowania pomiędzy mniejszymi lokacjami w grze, jak to ma miejsce np. w grze *Bioshock*, czy wykonywane w tle podczas normalnego działania gry, jak na przykład w *Grand Theft Auto 4* lub *World of Warcraft*, jest tylko kwestią skomplikowania implementacji. Oczywiście łatwiej przeładować w pamięci wszystkie potrzebne dane i w tym czasie wyświetlić ekran ładowania się gry, jednak jest to rozwiązanie mniej korzystne dla gracza. Z drugiej strony, im więcej stara się gra pokazać świata na raz bez jawnego przeładowania, tym mniej ma miejsca na bufor pamięci, którego może używać na dynamiczne przeładowywanie potrzebnych zasobów. Przykładem, gdzie nie do końca udała się implementacja takiego mechanizmu, może być gra *Morrowind*. Świat był w niej przeładowywany dynamicznie, jednak jeżeli gracz poruszał

się zbyt szybko po fragmencie w którym dużo elementów naokoło gracza się zmieniało, bufor na przeładowania dynamiczne się zapełniał i gra na chwilę zatrzymywała się i pokazywała na środku ekranu napis „Loading". Dla gracza nie było jasnego przekazu, dlaczego akurat w tym miejscu gra zdecydowała się na doczytanie czegoś do pamięci i mimo napisu było to równie irytujące jak na przykład chwilowe „zamrożenie" systemu operacyjnego w trakcie procesu dostępu do pliku wymiany na dysku.

Jeżeli programista zdecyduje się na rozwiązanie w postaci ekranów przeładowywania, należy pamiętać, by zoptymalizować czas takiej operacji. Na wielu platformach istnieją konkretne ograniczenia czasowe, jakich nie wolno przekroczyć, aby gra została dopuszczona do sprzedaży. Przykładowo, maksymalny czas ładowania się fragmentu gry na platformie Nintendo Wii to 15 sekund z możliwością zwiększenia tego limitu do 25 sekund, jeżeli ekran ładowania będzie zawierał jakiś element animowany, który będzie upewniał gracza w tym, że gra się nie zawiesiła. Gry, które nie będą spełniały tego warunku, nie będą mogły zostać wydane.

Jednym z podstawowych sposobów optymalizacji czasu przeładowań jest możliwe zmniejszenie objętości wczytywanych danych poprzez odpowiednie przygotowanie formatu, w jakim dane na temat świata gry są przechowywane. Dwa najczęściej używane formaty są następujące:

- **Obrazy binarne** - jest to zapis w postaci pliku binarnego pamięci zajmowanej przez konkretny obiekt w grze. Odtworzenie w grze takiego obiektu jest wtedy bardzo uproszczone, gdyż załadowanie z powrotem do pamięci takiego obrazu powinno dać efekt w pełni działającego ponownie obiektu. Niestety nie jest to takie proste, jak sama idea by wskazywała. Przykładowo, przechowywanie obrazów binarnych klas języka C++ jest utrudnione z kilku powodów, takich jak obsługa wskaźników czy tablic wirtualnych. Bardzo trudno również wprowadzić zmiany do obrazów binarnych, dlatego przechowywanie w tym formacie obiektów ściśle związanych z dynamiczną rozgrywką jest problematyczne. Obrazy binarne są dobrym rozwiązaniem dla statycznych obiektów, takich jak na przykład meshe terenu.
- **Serializacja** - jest sposobem na przechowywanie danych na temat obiektu na dysku w postaci strumienia danych, w których system serializacji będzie próbował odtworzyć obiekt. Dane zazwyczaj są przechowywane w formacie opartym na XML albo w plikach tekstowych. W momencie zapisu obiektu na dysku wywołana jest funkcja odpowiedzialna za wyprodukowanie strumienia danych koniecznych do ponownego odtworzenia obiektu. Załadowanie obiektu do pamięci odbywa się poprzez stworzenie nowej instancji klasy i podanie wcześniej utworzonego strumienia danych na wejściu. W przypadku poprawności serializowanych danych, obiekt po załadowaniu powinien wyglądać identycznie jak obiekt zapisany wcześniej na dysku. Niektóre języki takie jak C# czy też Java udostępniają ustandaryzowane na poziomie klas mechanizmy serializacji do plików tekstowych lub XML. Chociaż język C++ nie posiada żadnego wbudowanego me-

chanizmu serializacyjnego, na szczęście jego implementacja jest relatywnie prosta. Serializację w C++ można zrealizować poprzez stworzenie dwóch wirtualnych funkcji, nazwanych przykładowo *SerializeToDisk()* i *SerializeFromDisk()*. Wszystkie klasy, które będą częścią mechanizmu serializacji, muszą przeciążyć te dwie metody oraz je zaimplementować. Jest to rozwiązanie czasochłonne, jednak pozwala na maksymalną optymalizację ilości serializowanych danych, gdyż to programista podejmuje decyzje, które dane będą koniecznie potrzebne w momencie utworzenia obiektu. Drugim rozwiązaniem może być implementacja uproszczonego mechanizmu refleksji na wzór tej w C#. Da to całkowicie automatyczną serializację wszelkich obiektów języka C++, jednak odbierze programiście możliwość nieserializowania pewnych danych. Z punktu widzenia pamięciowego oraz z punktu widzenia czasu ładowań jest to rozwiązanie gorsze, ale na pewno elegantsze. Choć nie jest proste w implementacji, to przy dużej liczbie obiektów może się opłacać bardziej niż implementowanie kilkuset metod serializacyjnych.

Definicja 5.4. Refleksja

Jest to mechanizm umożliwiający dostęp do danych w postaci opisu wszystkich klas. Opis klasy udostępnia informacje takie jak nazwa klasy, pola, typy pól i nazwy metod danej klasy. Mechanizm refleksji dostępny jest podczas wykonywania programu (*at runtime*). Przykładem implementacji takiego mechanizmu może być język C# i udostępniona w nim funkcjonalność w obrębie przestrzeni nazw *System.Reflection*.

Wadą wysoce zoptymalizowanych systemów wczytywania świata jest konieczność dokładnego odwzorowania struktury danych potrzebnych podczas ładowania w edytorze w którego obiekty są eksportowane. Sprowadza się to do uciążliwego procesu konieczności wprowadzania zmian w kodzie odpowiedzialnym za strukturę obiektów w dwóch miejscach - w kodzie gry i w kodzie edytora. Bardzo wygodnym systemem jest serializacja oparta na plikach XML uzupełniona o jednokrotne przekonwertowanie danych na format o mniejszej zajętości pamięci pod koniec projektu.

Czasami, mimo wysiłków programisty mających na celu skrócenie czasu ładowania gry, nadal jest on zbyt długi. W takich przypadkach można zastosować prawie kompletne przeładowanie gry w czasie rzeczywistym jej działania, a więc już po pokazaniu graczowi świata gry oraz oddaniu mu kontroli. Metoda ta wymaga, żeby lokacje były odpowiednio zaprojektowane i nie obfitowały w elementy zajmujące pamięć. Celem zaś, by bufor pamięci dostępny do przeładowania był maksymalnie duży. Przykładem zastosowania takiego rozwiązania może być gra *Mass Effect*. Czasem, zamiast ekranu przeładowania, gra ta stosowała do przejścia między lokacjami windy. Czas jazdy windą był wyraźnie dłuższy niż powinien na same potrzeby wiernego oddania tej czynności

graczowi, jednak w tym czasie gra przeładowywała wszelkie potrzebne zasoby w tle, choć dla gracza był to po prostu przydługi element normalnej rozgrywki. Urozmaicone to było dyskusjami prowadzanymi przez bohaterów w trakcie jazdy windą i oczywiście było mniej odrywające gracza od wirtualnego świata gry niż ekran przeładowania. Czasem przeładowania w tle mogą być mocno uwzględnione w fabularnym projekcie gry. Przykładem może być gra *Metroid Prime*, gdzie do przeładowania zasobów wykorzystywane były małe pomieszczenia, w których gracz musiał wykonać konkretne czynności, by móc przejść dalej. Interakcje i lokacje były tak zaprojektowane, by zużywały jak najmniej pamięci. O ile w grze *Mass Effect* gracz mógł się jeszcze domyślać, że jazda windą jest pewnym substytutem ekranu przeładowania, to w *Metroid Prime* mechanizm ten był tak zgrany z samym projektem oraz fabułą gry, że gracz nie mógł być tego świadomy. Pomieszczenia służące do przeładowania scen nazywane są *airlock* (ang. śluza powietrzna).

5.3 Przepływ w grach

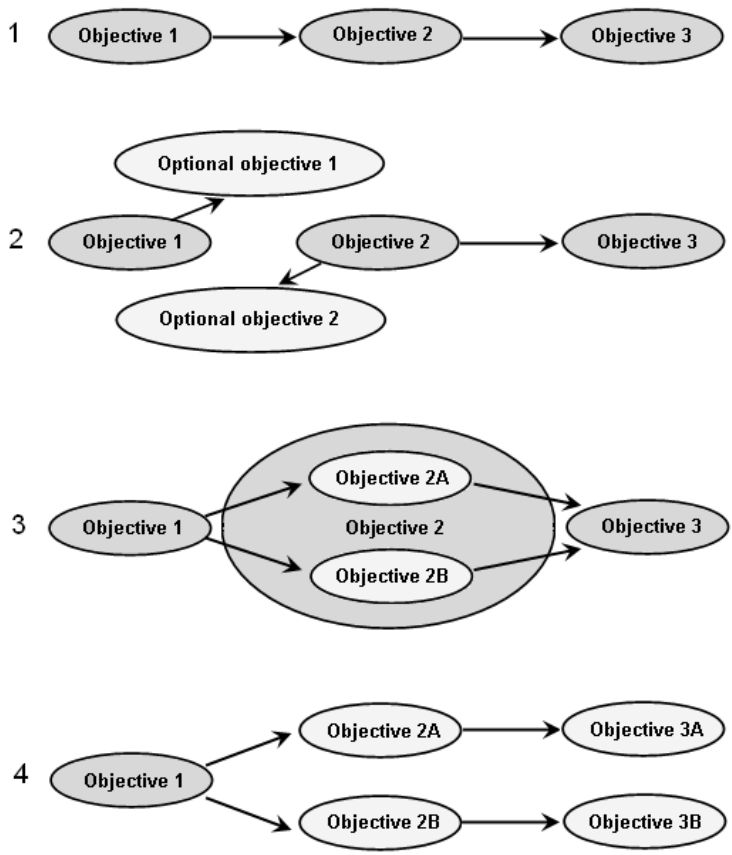
Przez przepływ w grach wideo, z angielskiego zwany popularnie *„flow"*, rozumiany jest pewien zbiór danych sekwencyjnych w postaci drzewa czy też grafu. Dane w tym zbiorze to cele, jakie musi osiągnąć gracz, by rozgrywka posunęła się do przodu. Tym celem może być przejście danego poziomu gry platformowej czy obrona przed falą atakujących przeciwników w grach typu *tower defense*. Zależy to od gatunku gry. W przepływie powinny być również uwzględnione warunki porażki gracza. W przypadku przepływu sekwencyjnego jest to zazwyczaj powtórzenie danego fragmentu gry od początku czy to całego poziomu, czy od tak zwanego *check pointu*.

! Definicja 5.5. Check point

Check point (z ang. punkt kontrolny) jest to miejsce automatycznego zapisu stanu gry. *Check pointy* używane są zazwyczaj w rozgrywkach liniowych, gdzie rozmiar pojedynczego poziomu, na którym toczy się rozgrywka, jest na tyle duży, że nużące dla gracza byłoby przechodzenie całego poziomu od początku w razie niepowodzenia.

Jeżeli zbiór celów do osiągnięcia jest sekwencyjny, czyli nie da się rozegrać żadnego innego fragmentu gry bez przejścia poprzedniego i gracz ma zawsze do obrania tylko jedną możliwą ścieżkę postępu, to nazywa się takie gry liniowymi. Nieliniowymi są takie, w których zbiór celów rozgrywki można przedstawić na drzewie lub grafie. Często w grach nieliniowych porażka czy też błąd gracza jest po prostu jedną ze ścieżek rozwoju rozgrywki i niekoniecznie musi się skończyć powtarzaniem fragmentu gry. Ekstremalnym przykładem

tego typu gry jest *Heavy Rain*, gdzie praktycznie każdy możliwy błąd gracza mimo wszystko popycha fabułę na przód i jest w pewien sposób rozpatrzony. W grze tej nie ma możliwości powtórzenia żadnego fragmentu gry jeszcze raz w pojedynczym jej przejściu, a gracz może kontynuować rozgrywkę nawet w przypadku śmierci jednego w kierowanych przez niego bohaterów.



Rysunek 5.1. Różne rodzaje przepływu w grach: 1.Liniowy 2.Liniowy, w opcjonalnymi zadaniami 3.Okresowo nieliniowy 4.Nieliniowy

Gatunkiem, w którym najczęściej występuje nieliniowość, jest cRPG. Gry tego typu są mocno nastawione na stworzenie graczowi iluzji wolności, co rzadko idzie w parze z podążaniem jedną wyznaczoną ścieżką. Istnieją również hybrydy gier liniowych i nieliniowych - można tego typu gry nazywać okresowo nieliniowymi. Oznacza to, że przepływ w danej grze można przedstawić przykładowo na grafie, na którym co jakiś czas będą istniały węzły pojedyncze, przez które gracz musi przejść by rozgrywka potoczyła się dalej. Dobrym tego

przykładem są gry przygodowe, gdzie często gracz ma dużą dowolność przy przemieszczaniu się po pewnym zamkniętym zbiorze lokacji, zbieraniu przedmiotów i wykonywaniu interakcji, jednak zazwyczaj prowadzi to do jednej głównej interakcji, która otwiera graczowi możliwość przejścia do kolejnego zbioru lokacji itd.

Z punktu widzenia skomplikowania stworzenie gry liniowej jest znacznie prostsze niż gry nieliniowej. Powodów jest kilka, podstawowym jest rozmiar zawartości gry. Tworząc grę nieliniową często tworzy się dużo zawartości w postaci kodu czy też grafik, których gracz może nigdy nie zobaczyć. Bardzo prostym przykładem tego typu dodatkowej zawartości może być gra *Gears of War*. Choć jest to gra z gruntu rzeczy liniowa, co jakiś czas gracz ma możliwość obrania jednej z dwóch ścieżek, drugą pójdzie jego partner sterowany przez AI. Jeżeli gracz nie przejdzie tej gry drugi raz i nie obierze za każdym razem innej ścieżki niż poprzednio, pewnych lokacji stworzonych na potrzeby tej gry nigdy nie zobaczy, mimo iż grę ukończy. W przypadku tak drobnej nieliniowości jak w *Gears of War*, koszta produkcji gry wzrastają nieznacznie. Jednakże w przypadku gier cRPG różnice w kosztach mogą być gigantyczne. Gracz, który wykona w grze *Fallout 3* tylko zadania głównego wątku gry, nie zobaczy znacznej części zawartości gry.

Dla programisty głównym skomplikowaniem podczas implementacji gry z przepływem nieliniowym jest stałe kontrolowanie aktualnego stanu rozgrywki, przykładowo kontrolowanie, jakie dane należy zapamiętywać podczas zapisu gry. Komplikuje się jednak przede wszystkim logika. Im więcej kombinacji designer musi przewidzieć, tym więcej ścieżek gry musi zostać oprogramowanych lub oskryptowanych, co znacznie zwiększa podatność na błędy, zwłaszcza podczas wprowadzania zmian do scenariusza.

Od niektórych gatunków gier tradycyjnie oczekuje się pewnego typu przepływu, jak nieliniowości od cRPG, okresowej nieliniowości od gier przygodowych czy liniowości od gier platformowych. Jednak coraz bardziej zacierają się granice między gatunkami, coraz więcej gier posiada w sobie elementy wielu gatunków i nie trudno znaleźć obecnie FPSa z misjami pobocznymi, jak na przykład *Chronicles of Riddick*.

5.4 Elementy systemu gry

Większość silników gry udostępnia pewien zestaw komponentów, na podstawie których budowana jest gra z programistycznego punktu widzenia. W branży gier wideo nie ma ustandaryzowanego nazewnictwa na ten fragment silnika gry. Niektórzy nazywają go *frameworkiem*, inni natomiast systemem podstaw *gameplayu*. Na potrzeby tej książki wszystkie komponenty tego fragmentu kodu będą nazywane systemem gry. System gry może być w dużej mierze napisany w oderwaniu od gatunku gry, w przeciwieństwie do kompletnego silnika gry.

Zdarzenie, bezpośrednia komunikacja czy pośrednik?

Często programista musi przekazać pewną informację z jednej klasy do drugiej. Zakładając, że w architekturze nie zaplanowano posiadania bezpośredniego wzajemnego dostępu tych klas do swoich instancji, programista stoi przed jedną z trzech możliwości rozwiązania problemu:

- Stworzenie możliwości bezpośredniej komunikacji pomiędzy tymi klasami (np. poprzez przekazanie wskaźnika lub referencji);
- Rozesłanie komunikatu w postaci zdarzenia i odbiór tego komunikatu w drugiej klasie;
- Przekazanie danej przez inną klasę lub nawet zbiór klas, zgodnie z architekturą obecną w systemie gry.

Rozwiązanie pierwsze jest najprostsze, lecz nadużywane prowadzi do sytuacji „wszystko ma wskaźnik na wszystko", czyli całkowitej architektonicznej katastrofy. W przypadku tego rozwiązania należy sobie zadać pytanie „Czy klasa A powinna w ogóle wiedzieć, co to jest klasa B i mieć do niej dostęp?". Jeśli np. potrzebujemy, by HUD wyświetlał liczbę punktów, która się zwiększa w momencie śmierci przeciwnika, to nie oznacza, że przeciwnik powinien mieć wskaźnik na obiekt klasy HUD, gdyż nie ma to logicznego poparcia w architekturze. Ten sposób powinien być stosowany tylko, kiedy istnieje ścisły związek pomiędzy dwiema klasami, np. zawieranie.
Drugi punkt stanowi dużo bardziej elegancki sposób rozwiązania sytuacji, jednak również nadużywany prowadzi do bałaganu, gdyż jest jednoznaczny z ogromem ilości puszczanych w przestrzeni systemu gry zdarzeń, których teoretycznie może słuchać każda klasa. Najlepiej stosować to rozwiązanie dla najważniejszych informacji, które są źródłem zainteresowania więcej niż jednej klasy. Śmierć przeciwnika jest dobrym przykładem sytuacji, w której można by użyć zdarzenia, gdyż na pewno tym faktem będzie zainteresowanych wiele klas.
Najlepszym rozwiązaniem z punktu widzenia projektowania kodu jest odpowiednie przekazanie tej informacji poprzez inną klasę lub zbiór klas. Zależnie od architektury może to być wyspecjalizowana klasa służąca do śledzenia zmian i informowania zainteresowanych klas, czyli wzorzec projektowy obserwator lub prawidłowe przekazywanie informacji z klasy do klasy po drzewie architektury. Kontynuując poprzedni przykład, przeciwnik mógłby poinformować klasę nim zarządzającą, czyli np. managera przeciwników, ten poinformowałby klasę nadrzędną dla niego, czyli np. scenę gry, ta z kolei powinna mieć prosty dostęp do HUD'a. Rozwiązanie to może być uciążliwe w implementacji, jednak długofalowo pozwala zachować porządek w projekcie i ułatwić programiście poruszanie się w nim.

Zawsze jednak w takim systemie będą istniały elementy charakterystyczne tylko dla konkretnego sposobu rozgrywki. W podrozdziale 5.6 opisane zostało rozróżnienie pomiędzy kodem gry a kodem silnika gry. Najbardziej narażoną częścią silnika gry na posiadanie elementów wyspecjalizowanych na potrzeby tylko jednego tytułu jest właśnie system gry.

Każdy silnik gry posiada inną architekturę, tym samym różne jest podejście do implementacji systemu gry. Można jednak wyszczególnić pewne elementy, które są obecne w prawie każdym silniku:

- *Model obiektów gry* - abstrakcyjny model wszystkich obiektów gry, jest to najbardziej rozbudowany element systemu gry. Dokładny opis tego modelu znajduje się w podrozdziale 5.4.1
- *Aktualizacja stanu gry* - podstawowy element systemu gry, który wprawia cały świat gry wideo w życie. W najprymitywniejszej postaci często nazywany jest po prostu główną pętlą gry, jednak rzadko jest to fizycznie pojedyncza pętla programistyczna. Musi ściśle współpracować z aktualizacją silnika technologicznego, by istniała określona kolejność renderowania, aktualizacji kolizji, przeliczania fizyki itp.
- *System przepływu informacji* - moduł odpowiadający za komunikację pomiędzy obiektami w grze. Zależnie od architektury systemu gry może być to system oparty na zdarzeniach (ang. *events*) albo specjalnych klasach odpowiedzialnych za przesyłanie informacji, na przykład w oparciu o wzorzec projektowy *obserwator*.
- *Zarządzanie zawartością* - ten komponent systemu gry odpowiada za ładowanie i zwalnianie wszelkich zasobów świata gry opisanego w podrozdziale 5.2. Musi ściśle współpracować z komponentem zarządzania przepływem, by wiedzieć, które elementy gry zwalniać, a które ładować.
- *Zarządzanie przepływem* - komponent odpowiedzialny za kontrolę przepływu gry, czyli wszystkie elementy opisane w rozdziale 5.3.
- *Skryptowanie* - żeby przekazać możliwość implementacji logiki gry innym osobom niż programistom, system gry musi przewidywać interpretację wysoko poziomowego języka skryptowania, zazwyczaj przekazywanego do gry przez edytor. Język skryptowy może być oparty na tekście, tak jak np. Lua czy Python lub na graficznym interfejsie jak Kismet w silniku Unreal lub podobnym systemie bloczków w silniku Virtools.

5.4.1 Model obiektów gry

Model obiektów gry jest to jeden z najbardziej skomplikowanych elementów systemu gry, który można podzielić na następujące komponenty:

- *Dynamiczne tworzenie i niszczenie obiektów* - u podstaw tego elementu systemu gry leży manager zasobów silnika technologicznego. Sam system musi dawać programiście możliwość tworzenia i niszczenia obiektów gry w cza-

sie wykonywania programu. Nie musi to oznaczać dynamicznego zwalniania i alokacji pamięci. Zależnie od realizacji modelu obiektów gry, może istnieć statyczna liczba maksymalnych obiektów danego typu, a obiekty przedawnione mogą nie być usuwane z pamięci, tylko dodawane do puli obiektów oznaczonych do ponownego użycia, co tworzy tak zwany *object recycling*. Choć jest to rozwiązanie mniej elastyczne niż autentyczne zwalnianie pamięci, pozwala zaoszczędzić na czasach ładowania oraz mieć większą kontrolę nad stanem pamięci RAM w czasie wykonywania programu. Rozwiązanie to również zapobiega w pewnej mierze fragmentacji pamięci.

- *Połączenie z silnikiem technologicznym* - jeżeli obiekt gry nie jest tworem czysto abstrakcyjnym, prawie zawsze będzie zawierał w sobie elementy silnika technologicznego, związane z rendererem w przypadku wizualizacji obiektu, np. w postaci modelu 3D, czy efektu cząsteczkowego lub z innymi elementami silnika technologicznego, jak silnikiem fizycznym czy odgrywaniem dźwięku.
- *Aktualizacja stanu wszystkich obiektów* - system w pewnej części tożsamy z aktualizacją stanu gry, jednak wyodrębniony na aktualizacje jedynie obiektów wchodzących w skład modelu. Aktualizacja obiektów może występować w różnych interwałach, niekoniecznie co każdą klatkę. Może też istnieć ścisła kolejność, w jakiej obiekty modelu gry powinny być aktualizowane, by nie zaistniały przekłamania w logice gry.

Definicja 5.6. Inteligentne wskaźniki

Inteligentny wskaźnik (ang. *smart pointer*) to abstrakcyjny typ danych symulujący zachowanie wskaźnika, ale oferujący dodatkowe funkcjonalności, takie jak sprawdzenie zakresu czy automatyczne usuwanie danych nieużywanych. Głównym zadaniem inteligentnych wskaźników jest zredukowanie błędów związanych z zarządzaniem pamięcią, takich jak wycieki pamięci czy próba odwołania do nieważnych wskaźników. Najpopularniejszą biblioteką udostępniającą implementację wielu różnych inteligentnych wskaźników obok innych funkcjonalności jest darmowy *Boost*.

- *Baza obiektów* - samo tworzenie i niszczenie obiektów nie wystarczy. Musi też istnieć system, który nie tylko przechowuje te obiekty, ale odpowiednio je kataloguje oraz udostępnia. Każdy obiekt w grze powinien mieć swoje unikalne ID, na wzór bazy danych. System przechowywania obiektów powinien umożliwiać odpytanie całego zbioru po ID w celu wyszukania konkretnego obiektu. Mechanizm ten może udostępniać bardziej zaawansowane możliwości sprecyzowania zapytania, przykładowo znalezienie wszystkich przeciwników na planszy z poziomem życia poniżej połowy lub wszystkich obiektów znajdujących się w promieniu 10 metrów od gracza. W jaki spo-

sób obiekt jest przekazywany odpytującemu, zależy od implementacji modelu obiektów gry, może to być zwykły wskaźnik na obiekt klasy lub też bardziej zaawansowane rozwiązanie, takie jak inteligentny wskaźnik.

- *Możliwość definiowania nowych typów obiektów* - pamiętając, że zmiany podczas tworzenia gry wideo będą zawsze, architektura modelu systemu gry musi przewidywać prosty sposób na definiowanie nowych typów obiektów, najczęściej w oparciu o już istniejące lub przynajmniej o część ich funkcjonalności. Dodanie nowego typu obiektu musi być odwzorowane oczywiście w edytorze. W przypadku idealnym tworzenie takiego obiektu jest całkowicie napędzane danymi: designer w edytorze składa w pewnych dostępnych funkcjonalności nowy obiekt i jego kodowa reprezentacja jest automatycznie tworzona w modelu obiektów gry. W praktyce prawie zawsze do wprowadzenia nowego typu obiektu potrzebny jest programista, zazwyczaj z prozaicznego powodu, że nikt wcześniej nie mógł przewidzieć kompletu potencjalnych funkcjonalności, na jakie może się składać obiekt gry.
- *Sieciowa replikacja stanu obiektów* - moduł wymagany jedynie przy obecności rozgrywki sieciowej w danym projekcie. Wszelkie obliczenia aktualizacji stanu obiektów gry w grze sieciowej są zazwyczaj przeprowadzane na jednej maszynie, czyli serwerze, niezależnie od tego, czy jest to serwer dedykowany, czy jedna w platform biorących udział w rozgrywce. Na pozostałych platformach, czyli klientach, musi być możliwość szybkiego i prostego pobierania danych i replikacja rzeczywistości gry w serwera.
- *Zapis i odczyt stanu gry* - większość gier pozwala na zapis i odczyt stanu gry, czy to w dowolnym momencie, czy w postaci punktów kontrolnych lub na zasadzie odblokowywania kolejnych poziomów gry po przejściu poprzedniego. Zależnie od tego model obiektów gry może mieć mechanizmy umożliwiające zapis stanu wszystkich obiektów i jego ponownego odtworzenia, np. poprzez serializację obiektów. Zależnie od rodzaju przepływu i docelowej platformy, na którą jest kierowana gra, może to być komponent bardzo prosty albo bardzo trudny w implementacji. Serializacja skomplikowanego świata gry może sprawić, że rozmiar pliku zapisu osiągnie kilka megabajtów. Na niektórych platformach, zwłaszcza przenośnych, duży rozmiar pliku zapisu gry to poważny problem. W przypadku gier opartych na zapisie stanu do cartridga, a nie do pamięci masowej platformy, sprawa rozbija się o koszty produkcji. Im większa pamięć przeznaczona na zapis stanu gry, tym większy jest koszt cartridga. W takich przypadkach serializacja modelu obiektów jest w zasadzie niemożliwa, dlatego też często w grach na tę platformę nie stosuje się zapisu stanu gry w dowolnym momencie.

5.4.2 Architektura modelu obiektów gry

Każda gra posiada pewien zestaw konkretnych obiektów, które się na nią składają i których designer może użyć, przykładowo w edytorze świata gry, przy

tworzeniu danego poziomu czy też fragmentu gry. Ten zestaw obiektów musi być odwzorowany w pewien sposób w modelu obiektów gry przez programistę w konkretnej architekturze. Istnieje kilka sposobów zaprojektowania modelu obiektów gry, jednak można je podzielić na dwie zasadnicze kategorie:

- Architektura oparta na obiektach;
- Architektura oparta na właściwościach.

Najczęściej wykorzystywana w branży gier wideo jest architektura oparta na obiektach, dlatego też jej zostanie poświęcone więcej uwagi. Warto jednak także wspomnieć o architekturze opartej na właściwościach, gdyż jest to dosyć ciekawe podejście do realizacji modelu obiektów gry.

By lepiej zilustrować, jak wygląda przełożenie obiektów widzianych przez designera czy też gracza na ich reprezentacje w kodzie źródłowym gry, kolejne rozważania na temat architektury modelu obiektów będą się opierały na przykładzie bardzo uproszczonej gry FPS składającej się z następujących obiektów:

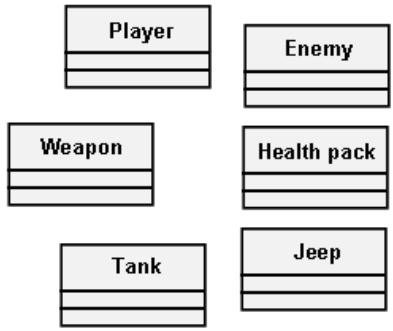
- Postaci gracza;
- Postaci przeciwników;
- Zbieranych broni;
- Zbieranych apteczek;
- Czołgu;
- Jeepa.

Architektura oparta na obiektach

W architekturze opartej na obiektach każdy logiczny element gry ma swoje odzwierciedlenie w instancji konkretnej klasy lub połączenia kilku klas. Możliwości realizacji tego typu architektury jest kilka, po kolei rozważone zostaną różne przypadki począwszy od najprostszego.

- *Prosty model obiektowy* - czyli przeniesienie jeden do jeden bytów logicznych na instancje klas w kodzie źródłowym gry. W zasadzie nie muszą to nawet być klasy. Skoro nie będzie zależności hierarchicznych pomiędzy tymi obiektami, można spokojnie zaimplementować tego typu architekturę w nieobiektowym języku takim jak C.

  W przypadku prostych gier taki sposób przeniesienia obiektów do kodu jest najbardziej intuicyjny. Problemy pojawiają się w momencie, kiedy w implementacjach poszczególnych obiektów powtarzać się będzie kod wykonujący dokładnie to samo. Przykładowo, zarówno czołg, jak i jeep z przykładowego FPSa na pewno będą miały elementy takie jak detekcja kolizji, mechanizm obsługi przez gracza, renderowanie modelu itd. W momencie całkowitego rozdzielenia implementacji tych obiektów programista będzie



Rysunek 5.2. Prosty model obiektowy gry typu FPS, nie ma żadnych zależności pomiędzy poszczególnymi implementacjami

musiał kopiować ogromną ilość kodu pomiędzy poszczególnymi implementacjami. Już na przykładzie maksymalnie uproszczonego FPSa widać, że prosty model obiektowy nie wystarcza.

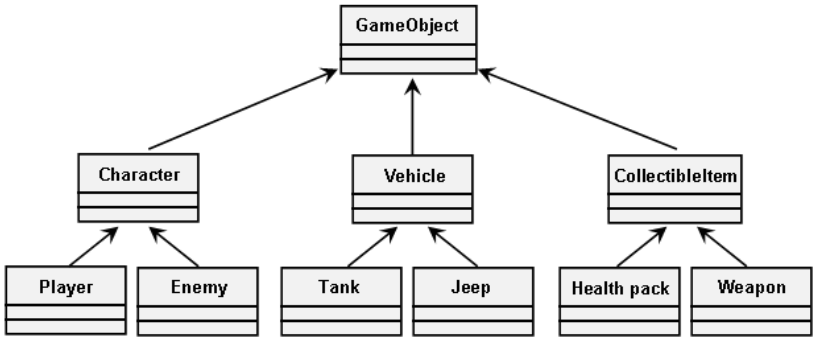
Kopiuj i wklej

Mechanizm kopiuj i wklej to jeden z najwygodniejszych i najbardziej oszczędzających czas elementów wszystkich edytorów. Mechanizm ten również obecny jest we wszystkich współczesnych środowiskach programowania. Należy jednak zachować ostrożność, korzystając z niego, gdyż nadużywanie mechanizmu kopiuj i wklej przez programistów ma pewne konsekwencje:

- Jest głównym źródłem błędów syntaktycznych i semantycznych w kodzie źródłowym. O ile te pierwsze wychwytywane są przez kompilator, to odkrycie złośliwego błędu semantycznego popełnionego poprzez kopiowanie kodu może zająć programiście wiele czasu.
- Kopiowanie większych fragmentów kodu, jak na przykład całych algorytmów w obrębie jednego projektu jest ogromnym błędem projektowym. Trzymanie tożsamego kodu w wielu miejscach sprawia, że zmiany oraz poprawki do niego trzeba wykonywać we wszystkich miejscach równocześnie, co nie tylko jest uciążliwe, ale często o którymś miejscu można zapomnieć.

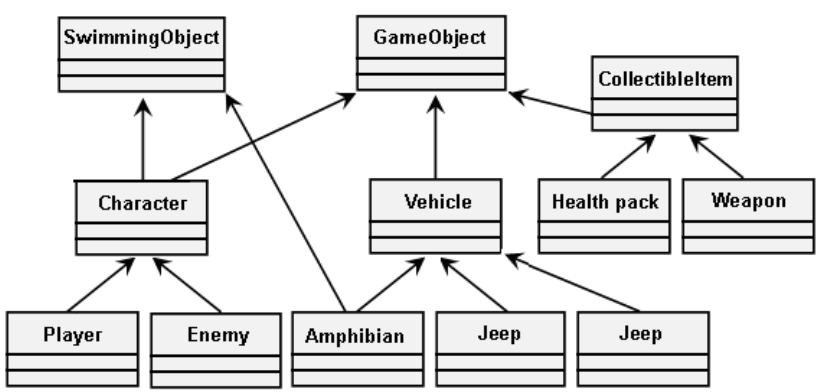
Generalnie dobrą praktyką jest zastanowienie się przed każdym kopiowaniem kodu źródłowego, czy na pewno jest to konieczne.

To, że hierarchia klas jest najpopularniejszym rozwiązaniem, nie znaczy, że jest wolna od wad. Projektowanie takich klas zazwyczaj zaczyna się



Rysunek 5.3. Przykładowa hierarchia klas gry typu FPS

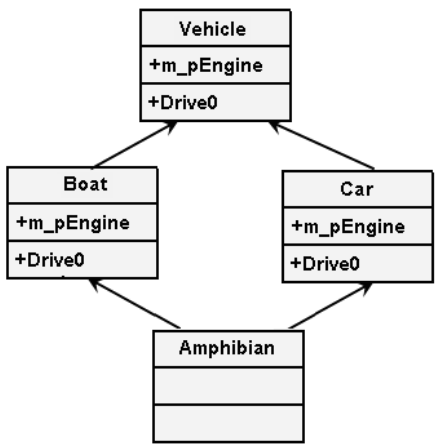
prosto i zwięźle, jednak w miarę rozwoju projektu, zarówno od strony designu, jak i implementacji, hierarchia zaczyna się pogłębiać i poszerzać. Zazwyczaj prowadzi to do tzw. monolitycznej hierarchii klas, czyli sytuacji, w której każda klasa dziedziczy po jednej głównej klasie w całej hierarchii, która staje się bardzo ciężka w utrzymaniu. Jednym z rozwiązań tego problemu jest użycie dziedziczenia wielobazowego, czyli jednego z mechanizmów obecnych na przykład w języku C++. W rozważanej przykładowej grze typu FPS, jeżeli designer stwierdzi, że dobrym pomysłem byłoby dodać pojazd typu amfibia mogący poruszać się po lądzie i wodzie, programista prawdopodobnie zaimplementowałby taką możliwość w klasie `Amphibian`, która będzie dziedziczyła po klasie pojazdu `Vehicle`. Jednak designer może pójść za ciosem i stwierdzić, że skoro teraz niektóre pojazdy będą mogły się poruszać po wodzie, świetnym pomysłem byłoby, gdyby gracz i przeciwnicy mogliby również poruszać się po wodzie. Jed-



Rysunek 5.4. Dodanie klasy dającej możliwość pływania po wodzie obiektom po niej dziedziczącym do hierarchi gry typu FPS

nym ze sposobów jest zaimplementowanie mechanizmu obsługi pływania osobno w klasie `Character` i `Vehicle`, ale byłoby to pogwałceniem zasady niekopiowania fragmentów kodu pomiędzy klasami. Można ten problem rozwiązać poprzez dziedziczenie wielobazowe, jak pokazuje rysunek 5.4.

Choć rozwiązanie to wydaje się eleganckie i rozstrzyga problem dublowania się implementacji mechanizmu pływania, z dziedziczeniem wielobazowym wiąże się tak zwany problem diamentu (ang. *diamond problem*). Nazwa wzięła się od wyglądu podstawowego diagramu klas dziedziczenia wielobazowego, co demonstruje rysunek 5.5.



Rysunek 5.5. Diamentowy diagram klas dziedziczenia wielobazowego

Problem diamentu na przykładzie rys. 5.5 wygląda następująco: zakładając, że klasa `Amphibian` nie przeciążyła funkcji `Drive()`, natomiast klasa Boat i Car tak, jaka funkcja powinna się wywołać w klasie `Amphibian` w momencie wywołania funkcji `Drive()`? Zależnie od języka programowania rozwiązywany jest na różnie sposoby. Języki takie jak Java i C# nie dają w ogóle możliwości dziedziczenia wielobazowego, natomiast C++ w przypadku takiego dziedziczenia stworzyłby w klasie `Amphibian` dwie kopie klasy `Vehicle`, a wywołanie bez odwołania się do konkretnej klasy, w której programista chciałby skorzystać, typu `Boat::Drive()`, zakończyłoby się błędem kompilatora, który nie byłby w stanie stwierdzić, z której kopii klasy `Vehicle` ma skorzystać. Rozwiązaniem jest tu podziedziczenie po klasie `Vehicle` wirtualnie, wówczas stworzy się tylko jedna kopia tej klasy. W bardziej rozbudowanych hierarchiach klas dziedziczenie wielobazowe sprawia wiele problemów, zmieniając w miarę proste drzewo dziedziczenia w skomplikowany graf. Wiele zespołów programistycznych ma wewnętrznie zabronione używanie dziedziczenia wielobazowego lub przynajmniej wprowadzone ograniczenia, jak na przykład zasada, że

klasa może mieć tylko jedną klasę dziadka, ale dowolną liczbę rodziców. Oznacza to, że tylko jeden rodzic może nie być samodzielną klasą. Skoro więc dziedziczenie wielobazowe sprawia zazwyczaj więcej kłopotów niż pożytku, programista zdecyduje się na dodanie interfejsu pływania do klasy `GameObject`. Tylko teraz `CollectibleItem` ma możliwość pływania, mimo iż absolutnie jej nie potrzebuje. Dodatkowo prawdopodobnie pojawi się w klasie bazowej zmienna `bool` o nazwie w stylu `m_bCanSwim`, żeby odróżnić, które obiekty korzystają z tej możliwości, a które nie. Rodzi się tak zwany efekt nadymania (ang. *bubble-up effect*), w którym klasa bazowa posiada wszystkie możliwe podstawowe funkcjonalności obiektów gry takie jak: rendering, kolizje, poruszanie się itd. W efekcie, by stworzyć obiekt gry, programista musi podziedziczyć po obiekcie bazowym, nawet jeżeli nie potrzebuje funkcjonalności renderowania, tylko przykładowo abstrakcyjnego obiektu kolizyjnego. Przykładem efektu nadymania jest klasa `Actor` w silniku *Unreal Engine*. Zajmuje się ona renderingiem, fizyką, animacjami, efektami dźwiękowymi, replikacją sieciową itd. Jak widać, hierarchia klas może przysporzyć programiście wielu kłopotów wraz z rozwojem projektu, pomimo to jej prostota przy początku implementacji jest bardzo atrakcyjna i wielu programistów decyduje się na jej użycie.

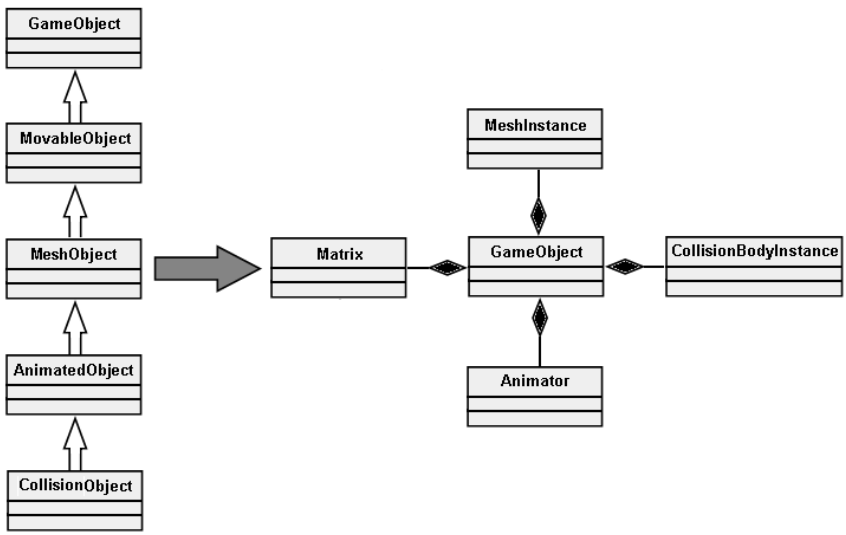
- *Uproszczona hierarchia z kompozycją* - najczęstszą przyczyną nadmiernego skomplikowania hierarchii obiektów jest nadużywanie relacji „jest" z punktu widzenia projektowania obiektowego, czyli nadużywanie dziedziczenia. Przykładowo programista tworząc pojazd może podziedziczyć go po obiekcie kolizyjnym. Na pierwszy rzut oka wszystko jest w porządku, pojazd jest obiektem kolizyjnym, jednak fakt bycia obiektem kolizyjnym nie definiuje go w pełni. Pojazd jest przecież również modelem, czyli siatką trójkątów, która go reprezentuje. Jak w tym momencie rozwiązać dziedziczenie? Dziedziczenie wielobazowe nie jest najlepszym pomysłem, więc może niech obiekt kolizyjny dziedziczy po obiekcie renderowanym? Przecież prawie każdy obiekt kolizyjny jest widoczny. No właśnie: prawie każdy. Programista może w ten sposób udostępnić funkcjonalność renderowania się obiektowi kolizyjnemu, którego zadaniem będzie np. stworzenie pola zasięgu wzroku przeciwnika, czyli obiektowi, który z renderingiem nie powinien mieć nic wspólnego. To rozwiązanie jest prostą drogą do efektu nadyma-

Definicja 5.7. Kompozycja

Kompozycja jest to relacja pomiędzy dwiema klasami, oparta na agregacji, czyli na metodzie tworzenia nowych obiektów składających się z jednego lub więcej innych obiektów. Kompozycja rozszerza agregacje o odpowiedzialność za obiekt zawierany. Jeśli obiekt B jest zawarty w obiekcie A, wtedy obiekt A jest odpowiedzialny za stworzenie i zniszczenie obiektu B. W przeciwieństwie do agregacji, obiekt B nie może istnieć bez obiektu A.

nia wspomnianego przy poprzedniej architekturze. Prawidłowym rozwiązaniem tego problemu jest relacja „ma", czyli kompozycja lub agregacja. Pojazd powinien składać się zarówno z obiektu kolizyjnego, jak też siatki modelu. Odpowiednie zastosowanie kompozycji w połączeniu w dziedziczeniem pozwala na znaczne zmniejszenie szerokości i głębokości hierarchii dziedziczenia oraz pozwala zapobiegać efektowi nadymania.

Na rysunku 5.6 przedstawiono przykład transformacji modelu „jest" na „ma".



Rysunek 5.6. Transformacja hierarchii dziedziczenia na kompozycję. Klasa MovableObject została zamieniona na komponent Matrix, klasa MeshObject na komponent MeshInstance itd

- *Model komponentowy* - kolejnym podejściem do architektury modelu obiektów w grze jest podejście czysto komponentowe. Jest to rozwinięcie poprzedniego modelu przez całkowite usunięcie hierarchii klas z obiektu głównego gry. Wtedy każda klasa będąca komponentem reprezentuję, pewną konkretną funkcjonalność, może się komunikować z innymi komponentami, jednak nigdy nie jest z nimi w relacji uogólnienia, czyli komponenty implementujące różne funkcjonalności nie dziedziczą po sobie.

Jedną z implementacji rozwiązania tego typu jest całkowite przesunięcie funkcjonalności z głównego obiektu gry i przechowywanie w niej pustych wskaźników do wszelkich klas je implementujących, czyli do komponentów. W takim rozwiązaniu klasa posiadająca wszystkie komponenty nazywana jest *hub'em* (z ang. *centrum*).

Kompozycja czy dziedziczenie?
Pewien *Lead Programmer* w branży gier wideo, gdy przeprowadzał rozmowę rekrutacyjną z potencjalnymi programistami, często zadawał następujące pytanie: „Co z punktu widzenia prawidłowego programowania obiektowego jest lepsze: kompozycja czy dziedziczenie?". Już pewnym sukcesem było zrozumienie przez programistę pytania i próba odpowiedzi. Odpowiedzi były różne, każdy programista starał się w jakiś sposób argumentować swój wybór. Jeżeli programista była naprawdę obiecujący, potrafił wymyślić przykład, który niepodważalnie udowadniał, że jedno jest lepsze od drugiego. Odpowiedzią, której ów Lead Programmer oczekiwał było: „To zależy". Można wymyślić sytuację, w której na pewno zawsze jedno rozwiązanie będzie lepsze od drugiego, i na odwrót. Zdaniem autora, kompozycja nie jest lepsza od dziedziczenia, jednak nadużywanie dziedziczenia przez wielu programistów sprawia, że kompozycja wymaga niejako promocji w środowisku programistycznym.

Biorąc przykład kompozycji w rysunku 5.6 nagłówek takiej klasy wyglądałby następująco:

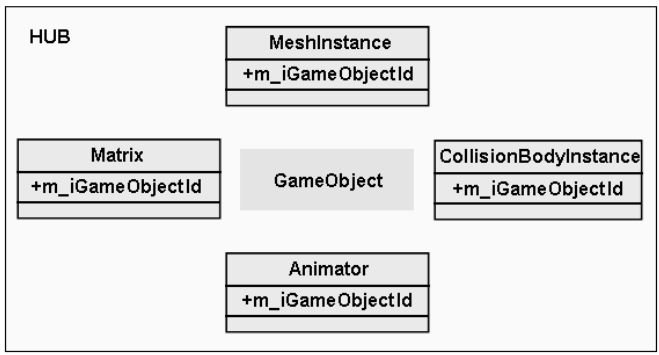
```
// main game object
class CGameObject
{
private:
    // matrix of the object
    CMatrix * m_pMatrix;

    // mesh animator
    CAnimator * m_pAnimator;

    // object mesh
    CMeshInstance * m_pMesh;

    // collision body
    CCollisionBodyInstance * m_pCollisionBody;
};
```

Innym podejściem do implementacji tej architektury jest trzymanie dynamiczne wszystkich komponentów, na przykład jako listy wskaźników, zamiast na sztywno wpisanych wskaźników na każdy typ komponentu z osobna. Pozwala to na dużą elastyczność w używaniu głównego obiektu gry, który w dużej mierze może sobie nie zdawać sprawy, jakie funkcjonalności udostępnia. Taka implementacja wymaga, by każdy komponent dziedziczył po wspólnym interfejsie, aby możliwe było łatwe iterowanie po wszystkich składnikach obiektu gry. Kod nagłówkowy takiego obiektu gry mógłby wyglądać tak:

```
// main game object
class CGameObject
{
private:
    std::list<IComponent*> m_lComponents;
};
```

Oczywiście w takim przypadku wszystkie komponenty takie jak `CMatrix` czy `CAnimator` musiałyby dziedziczyć po interfejsie IComponent. Największą zaletą tego rozwiązania jest możliwość tworzenia nowych typów komponentów bez jakiegokolwiek modyfikowania głównego obiektu gry. Rozwiązanie to jednak jest dosyć trudne w zaprojektowaniu, gdyż wymaga bardzo dużej abstrakcji w implementacji obiektu gry, który nie jest w stanie założyć, z jakich elementów się składa, oraz w implementacji komponentów, które również nie wiedzą, z jakimi komponentami im przyjdzie współpracować.

Ekstremalny przypadek modelu komponentowego to całkowite usunięcie jakiejkolwiek funkcjonalności z obiektu głównego gry oraz przesunięcie jej do zewnętrznych komponentów. W takim przypadku obiekt główny gry byłby jedynie kontenerem komponentów, unikalnym identyfikatorem i listą wskaźników. Jeśli przesunie się teraz to unikalne ID do konkretnych instancji komponentów, usuwając całkowicie główny obiekt gry, otrzyma się tak zwany czysty model komponentowy. Przekształcając kompozycję, którą pokazuje rys. 5.7, otrzyma się w takim wypadku następującą konstrukcję:



Rysunek 5.7. Czysty model komponentowy. Klasa GameObject jest tylko tworem wirtualnym, nie istnieje nigdzie fizycznie w systemie gry

Rozwiązanie to jest bardzo interesujące, lecz nie jest pozbawione wad. Po pierwsze w modelu komponentowym tworzenie obiektów odbywało się poprzez tworzenie instancji klasy GameObject, do której można było dodawać poszczególne moduły. Teraz tworzenie nowych instancji komponentów jest zawieszone w powietrzu, więc trzeba w jakiś sposób to rozwiązać, przykładowo poprzez stworzenie fabryki komponentów. Kolejnym problemem jest komunikacja pomiędzy komponentami. Wcześniej, jak sama jego nazwa wskazywała, komunikacją zajmował się *hub*. Teraz komponent, jeżeli chce się skomunikować z innym komponentem, musi przeszukać po swoim unikalnym ID cały zbiór w celu znalezienia komponentu składającego się na ten sam wirtualny obiekt gry, co jest mało efektywne. Przykładowo, jeżeli komponent `Animator` o uni-

kalnym ID 77 będzie chciał poruszyć siatką modelu, będzie musiał odpytać po kolei zbiór komponentów `MeshInstance`, aż znajdzie ten o ID 77.

Czysty model komponentowy ma jednak również jedną wielką zaletę. Pozwala on na jeszcze większe napędzanie silnika gry danymi niż w przypadku modelu komponentowego z głównym obiektem gry. W odpowiednio zaprojektowanej architekturze i z odpowiednio dużą ilością stworzonych komponentów programista będzie bardzo rzadko wymagany przy tworzeniu zawartości gry.

Architektura oparta na właściwościach

Programowanie obiektowe wyucza w programiście podejście do każdego elementu gry jako instancji konkretnej klasy. Obiekty zawierają atrybuty, czyli dane, i pewne zdefiniowane zachowania, czyli metody. Takie podejście w oparciu o przykład gry FPS wygląda następująco:

- Vehicle
  - Position
  - Orientation
  - Health
- Health pack
  - Position
  - Orientation

Istnieje jednak podejście skupiające się na właściwościach bardziej niż na obiektach. W takiej architekturze definiuje się wszelkie możliwe właściwości, jakie mogą występować oraz tworzy się tablicę składającą się z unikalnych ID obiektów w grze, które posiadają daną właściwość. Przerabiając poprzedni przykład na architekturę opartą na właściwościach otrzymuje się następującą konstrukcję:

- Position
  - Vehicle
  - Health pack
- Orientation
  - Vehicle
  - Health pack
- Health
  - Vehicle

Jak widać na powyższym zestawieniu, pojazd nie posiada już swojej pozycji czy zdrowia. Pojazd w tym podejściu w zasadzie nie istnieje jako fizyczny obiekt. Jednak istnieje ID pojazdu, które jest obecne zarówno w tablicy pozycji, jak i orientacji. Tworzenie obiektu w takim podejściu polega na stworzeniu nowego ID oraz dodaniu go do wszystkich tablic właściwości, z jakich ma się

składać. Jeśli np. obiekt ma mieć swoją reprezentację graficzną w postaci modelu 3D, w tablicy spisującej wszystkie renderowane modele powinno się pojawić ID tego obiektu. W realizacji przypomina to relacyjną bazę danych, w której każdy atrybut jest kolumną, a ID obiektów gry służy jako klucz.

Pojawia się pytanie, w jaki sposób realizowane są konkretne czynności wykonywane przez obiekty, gdyż w podejściu obiektowym istniały nie tylko atrybuty, ale też i metody. Rozwiązania w podejściu opartym na właściwościach są dwa:

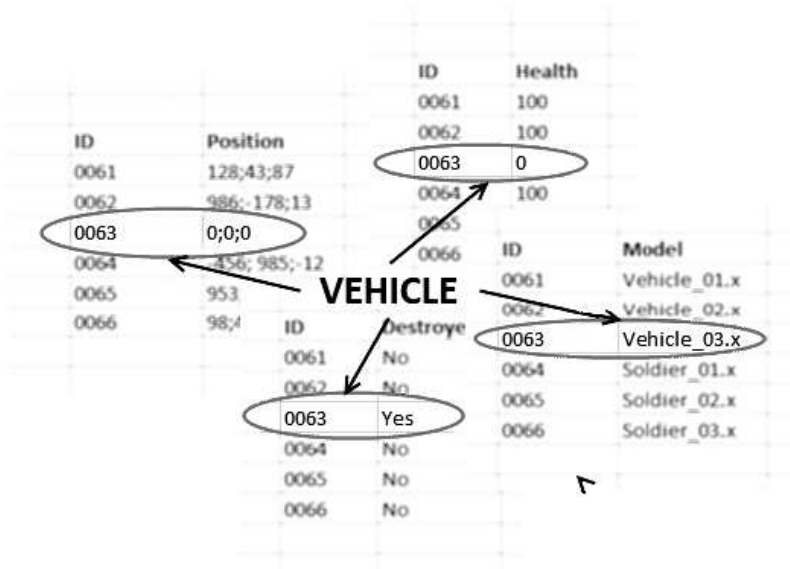
- Poprzez skrypt - stworzenie właściwości przechowującej skrypt, który będzie definiował pewne zachowanie i który będzie interpretowany przez pozostałe moduły modelu systemu gry. Przykładowo moduł AI może pracować na tablicy takich skryptów, wywołując po kolei wszystkie zawarte w niej instrukcje.
- Poprzez klasę właściwości - obiekt na wzór komponentu, jednak całkowicie abstrakcyjny. Klasa właściwości powinna implementować pewną konkretną właściwość obiektu gry. Przykładowo można to zobrazować na klasie właściwości `Destroyable`. Odpowiadałaby za nadanie obiektowi możliwości bycia zniszczonym. Klasa taka posiadałaby ID obiektu, którego właściwość zniszczalności określa i regularnie sprawdzała pod tym ID tablicę właściwości przechowującą punkty życia wszystkich obiektów. W momencie, kiedy punkty spadną poniżej zera, ustawi w tablicy właściwości `Destroyed` wartość `true` dla tego obiektu.

Nie należy mylić architektury opartej na właściwościach z komponentowym podejściem do architektury obiektowej. Choć założenia są dosyć podobne, w podejściu komponentowym nadal istnieje hierarchia klas i pojęcie obiektu jako bytu, nie funkcjonalności. Można się natomiast zgodzić ze stwierdzeniem, że architektura oparta na właściwościach jest bardzo podobna do podejścia czysto komponentowego.

Architektura oparta na właściwościach nie jest tak popularna jak podejście obiektowe, jednak *Deus Ex 2, Thief* czy też *Dungeon Siege* są przykładami dużych gier napisanych w oparciu o tę architekturę. Największą zaletą tego rozwiązania jest naturalne zasilanie tego typu systemu przez dane, co znacznie ułatwia udostępnianie gry do modyfikacji „nieprogramistom" i sprawia, że designer może mieć dużą swobodę przy tworzeniu zawartości gry. Przykładowo może on dowolnie tworzyć różne obiekty składające się w zestawu dostępnych właściwości. Programista jest w tym momencie potrzebny tylko w przypadku wymyślenia nowej, nieistniejącej jeszcze właściwości.

5.5 Elementy mechaniki gry

W rozdziale 4.5.10 wymienione zostały przykładowe elementy silnika technologicznego wspierające implementacje konkretnych mechanizmów gry. W przy-



Rysunek 5.8. Abstrakcyjny obiekt pojazdu w architekturze opartej na właściwościach. Na rysunku widać, że obiekt od ID 0063 posiada swoją pozycję, model grafiki, punkty życia i informacje, że jest zniszczony

padku silnika technologicznego mechanizmy te musiały być wysoce abstrakcyjne, ale w przypadku silnika gry można stworzyć dużo większą specjalizację na potrzeby konkretnego gatunku czy też gry. Elementy wymienione wcześniej w tym rozdziale, takie jak system obiektów gry, są bardzo mocno związane z silnikiem i muszą być w nim obecne.

Poniższe zestawienie jest bardziej modułowe. Pokazuje, jakie elementy mechaniki mogą zostać zaimplementowane, nadal w sposób na tyle abstrakcyjny, by nie były warstwą kodu gry, jednak na tyle wyspecjalizowany, by nadawały się jedynie do konkretnych zastosowań w obrębie jednego gatunku czy też serii gier:

- *Kamera* - w przypadku skonkretyzowanego sposobu przedstawiania rozgrywki, dobrym rozwiązaniem jest implementacja wszelkich mechanizmów obsługi kamery w silniku gry. Silnik technologiczny zazwyczaj udostępnia podstawową funkcjonalność, taką jak ustawienie pozycji, orientacji i wektora definiującego górę horyzontu (tzw. *up vector*). Silnik gry może udostępniać bardziej zaawansowane funkcje obsługi kamery, takie jak poruszanie się po zdefiniowanych ścieżkach, algorytmy wygładzające ruch kamery, zintegrowanie z systemem kolizji, by kamera nie przenikała geometrii w grze, czy w końcu tak zwaną inteligentną kamerę, czyli kamerę próbującą się ustawić w najbardziej dogodny dla gracza sposób zależnie od sytuacji.

- *Multiplayer* - elementy takie jak protokoły, nawiązywanie połączenia, sprawdzanie poprawności przesyłanych paczek danych itp. mogą być rozwiązane w warstwie silnika technologicznego. Na warstwie silnika gry programista zna już konkretne parametry rozgrywki: jakiego typu dane będą przesyłane, czy system będzie się opierał na serwerach dedykowanych, czy kon-solowym matchmakingu, jakie będą typy rozgrywki, jaka będzie liczba graczy itd. Obsługa wszystkich tych elementów oraz architektura obiektów gry przewidująca wykorzystanie w rozgrywce wieloosobowej mogą być częścią warstwy silnika gry.

Definicja 5.8. Matchmaking

Matchmaking (ang. swatanie), w odniesieniu do gier wideo oznacza zautomatyzowany system dobierania graczy do rozgrywki wieloosobowej. Gracz ma zazwyczaj możliwość określenia pewnych parametrów, takich jak typ rozgrywki, w jakiej chce brać udział lub grupę znajomych, z którymi by się chciał znaleźć razem w danej rozgrywce. Jednak sam system poszukiwania dodatkowych graczy odbywa się automatycznie, spośród graczy, którzy szukają tego samego typu rozgrywki. Jest to alternatywne rozwiązanie do dedykowanych serwerów, gdzie gracz sam definiuje wszelkie parametry rozgrywki, a możliwość dostępu do jego gry inni gracze mają poprzez przeglądanie wszystkich aktualnie dostępnych rozgrywek. Matchmaking jest szeroko stosowany na konsolach, natomiast serwery dedykowane są spotykane przede wszystkim w grach na platformę PC.

- *Sterowanie* - silnik technologiczny powinien udostępniać komplet danych płynących z urządzeń wejścia, jednak obróbką tych danych powinien się zająć silnik gry. Zależnie od gry sterowanie może być mocno generyczne, jak na przykład sterowanie w grach FPS lub bardziej skomplikowane, jak w symulatorach lotniczych.
- *Sztuczna inteligencja* - skonkretyzowane AI to idealny przykład modułu tworzonego pod konkretną grę. O ile silnik technologiczny, jak zostało wspomniane to wcześniej, może posiadać pewną architekturę ułatwiającą implementację AI, jak maszynę stanów czy wsparcie dla logiki rozmytej, to zupełnie inaczej będzie wyglądał kod odpowiedzialny za ruchy przeciwnika w bijatyce, a inaczej w strategii turowej. Zaawansowane AI jest jednym w najbardziej skomplikowanych zagadnień w grach wideo, jednak idea AI ciągle opiera się na tych samych zasadach logiczno-matematycznych, co przed wielu laty. Dlatego też jednorazowa dobra implementacja w silniku gry, na przykład algorytmu odnajdywania ścieżki może być przenoszona do kolejnych produkcji wiele razy, co będzie dużym uzyskiem w kontekście pracy programistycznej.

- *UI* - większość silników technologicznych udostępnia pewien zestaw podstawowych mechanizmów interfejsu użytkownika takich jak np.: przyciski, suwaki czy pola tekstowe. W przypadku skonkretyzowanej gry można stworzyć dużo bardziej wyspecjalizowane elementy UI. Przykładowo dla gry z gatunku RTS można zaimplementować pasek narzędziowy z możliwością prostego dodawania do niego jednostek czy też budynków, jakie gracz może stworzyć. W przypadku gier RPG można stworzyć panel do obsługi dialogów, w którym gracz będzie miał opcje wyboru wypowiedzi.

Warto założyć, że będzie multiplayer

Nawet jeżeli w projekcie gry nie jest przewidziana rozgrywka wieloosobowa, to warto w architekturze zarówno silnika, jak i projektu samej gry ją uwzględnić. Bardzo trudno jest w późnej fazie projektu dostosować kod dedykowany pod rozgrywkę jednoosobową do wieloosobowej. Nikt nie jest w stanie na pewno przewidzieć, czy taki pomysł nie wejdzie do realizacji, więc warto się zabezpieczyć. Oczywiście, jeżeli w projekcie gry nie przewidziano trybu wieloosobowego nie ma sensu fizycznie go implementować, jednak sam projekt komunikacji między obiektami i interfejsami powinien taką możliwość przewidywać. Architektura przygotowana pod rozgrywkę wieloosobową jest z natury bardziej zwarta, lepiej określony jest przepływ danych i komunikacja pomiędzy obiektami jest ograniczona do niezbędnego minimum. Są to atrybuty, które zawsze warto posiadać w kodzie, niezależnie od trybu rozgrywki. Przykładem może być implementacja sterowania postacią. Jeżeli zostanie zaprojektowana tak, by nie miało od strony kodu znaczenia, czy steruje nią gracz czy AI, to również nie powinno być problemów, by sterował nią gracz z zewnątrz. Jeżeli natomiast zaszyte będą w sterowaniu postacią przykładowo odwołania do sterowania z klawiatury czy pada, pomijając jedną warstwę abstrakcji, implementowanie obsługi takiego obiektu przez innego gracza będzie znacznie trudniejsze i prawdopodobnie sprowadzi się do dodania brakującej warstwy.

5.6 Silnik gry a kod gry

Zależnie od rozmiaru projektu oraz dojrzałości używanego silnika, pewne elementy gry są na sztywno implementowane przez programistów w warstwie silnika gry. Takie podejście realizowane jest przeważnie w dwóch przypadkach. Pierwszy, kiedy zespół jest relatywnie mały i pracuje nad małym projektem. W takich zespołach zazwyczaj jest więcej programistów niż designerów czy skrypterów, naturalne jest więc, że pewne mechanizmy są implementowane

przez programistę oraz przez niego zmieniane. Nie ma wtedy potrzeby tworzenia narzędzia, którego zazwyczaj głównym zadaniem jest umożliwienie każdemu, nie tylko programistom, zmieniania zawartości gry. Drugim powodem jest nacisk na bardzo szybki efekt od strony osób zarządzających projektem. Programista, który pracuje pod dużą presją czasu, stojąc przed wyborem implementacji narzędzia, a sztywną implementacją w kodzie, wybierze drugą opcję, gdyż będzie szybsza.

W pierwszym przypadku, choć nie jest to do końca zgodne ze sztuką tworzenia gier, rozwiązanie jest logiczne. W drugim przypadku może się bardzo zemścić na programiście, również z dwóch powodów. Przede wszystkim od tego momentu każda zmiana w mechanizmie, który zaimplementował, będzie musiała być wprowadzana przez niego, a w najlepszym przypadku przez innego programistę, ale designer nie będzie miał możliwości, żeby jakąkolwiek zmianę wprowadzić. Z każdą najdrobniejszą zmianą będzie się wiązała dodatkowa praca takiego programisty, a uzyskanie urlopu od przełożonego może być niemożliwe - „jesteś teraz potrzebny, musimy wprowadzić zmiany w mechanice rozgrywki a tylko ty wiesz, gdzie i jak to zrobić". Po drugie, sztywna implementacja mechanizmów na potrzeby danej gry zazwyczaj zaburza warstwowość silnika gry, co sprawi kłopoty w przypadku próby wykorzystania tej technologii w następnej grze lub jeszcze gorzej, w grze, która powstaje równolegle na tym samym silniku. Nacisk na to, by programista nie był zaangażowany w wprowadzanie zmian do zawartości gry, a tylko udostępniał taką możliwość jak najszerszemu gronu „nieprogramistów", bierze się ze specyfiki zespołu tworzącego grę. Programiści zazwyczaj są w mniejszości w stosunku do ludzi tworzących zawartość, takich jak artyści czy designerzy. Programistę da się najczęściej wykorzystać dużo lepiej, tam gdzie ani designer, ani grafik nie będą w stanie programiście pomóc, czyli przy tworzeniu kodu źródłowego do kolejnego elementu gry, który potem będą mogli wszyscy w pewnym stopniu modyfikować.

Pułapka sztywnej implementacji
Implementowanie elementów gry, które mogłyby zostać napisane uniwersalnie, to duży problem nawet profesjonalnych silników gier. Pewna firma tworząca komercyjnie silnik przeznaczony dla gier z gatunku FPS jednocześnie rozwijała ten silnik, tworząc na nim swoją własną grę, w której występowało dwóch głównych bohaterów. Wielkim zaskoczeniem było dla niektórych programistów z zewnątrz, korzystających z tego komercyjnie zakupionego silnika, zastosowanie w teoretycznie abstrakcyjnej warstwie silnika gry instrukcji warunkowych, zawierających imiona bohaterów gry utworzonej wcześniej przez autorów silnika. Jest to przykład bardzo pochopnej i nieprzemyślanej implementacji. Tego typu sztywne konstrukcje bardzo podważają poziom profesjonalizmu danego silnika i należy się ich wystrzegać.

Najlepszym sposobem na odróżnienie, czy dany fragment kodu jest częścią silnika gry czy kodem samej gry, jest zadanie następującego pytania: „Czy można przenieść ten fragment do następnej gry opartej na tym silniku bez żadnej zmiany w kodzie?". Jeżeli odpowiedź będzie twierdząca, znaczy to, że dany fragment jest warstwą silnika gry, jeżeli natomiast będzie negatywna, oznacza to, że dany fragment jest sztywnym kodem gry.

6

Narzędzia

Narzędzia towarzyszą użytkownikowi w bardzo szerokim zakresie w trakcie całego procesie tworzenia gier wideo - od edytorów świata gry, poprzez narzędzia wspomagające artystów, na systemach raportowania błędów i repozytoriach kończąc. W tym rozdziale zostanie opisana większość rodzajów narzędzi, z jakimi programiście przyjdzie pracować lub tworzyć, w branży gier wideo.

6.1 Narzędzia deweloperskie

6.1.1 Edytory

Edytor świata gry to narzędzie umożliwiające tworzenie, modyfikację i kontrolę elementów *gameplay'u* gry. Zakres możliwości edytora jednak może się znacznie różnić pomiędzy poszczególnymi implementacjami. W tym podrozdziale zostanie przybliżona większość zagadnień związanych z edytorami oraz ich implementacją.

Zakres funkcjonalności edytorów

W wersji minimalnej rolę edytora może pełnić edytor tekstowy (np. *Notatnik z systemu Windows*) modyfikujący pliki konfiguracyjne (np. typu INI), w których zapisane są podstawowe ustawienia gry. W wersji bardzo rozbudowanej edytor może udostępniać graficzny interfejs oraz możliwość edycji każdego elementu gry, od rozstawiania modeli w świecie, poprzez implementację skryptów zachowań AI, na edycji efektów cząsteczkowych i dźwiękowych kończąc. Pomiędzy tymi dwiema skrajnościami istnieje cała skala rozwiązań pośrednich. Nie ma w branży gier wideo twardej definicji, jakiego typu funkcjonalności powinien posiadać i jak powinien wyglądać edytor gry. Istnieje jednak wiele utartych praktyk, oraz pewien zestaw funkcjonalności podstawowych, których designerzy oczekują po tym narzędziu.

Plik INI
Jest to format pliku konfiguracyjnego, oryginalnie przeznaczony dla systemu operacyjnego Windows, jednak swego czasu szeroko stosowany we wszelkiego rodzaju aplikacjach z grami włącznie. Obecnie wyparty przez znacznie częściej używany format XML. Najbardziej podstawowy format pliku INI rozróżnia trzy rodzaje wartości: sekcje, parametry i komentarze, jak widać na poniższym przykładzie:

```
; wideo settings section
[WIDEO]
ResolutionX  = 1024
ResolutionY   = 768
```

Jego nazwa wzięła się od słowa inicjalizacja, standardowo jest to plik tekstowy z rozszerzeniem `*.ini`.

Jako że edytor świata gry jest ostatni w łańcuchu warstw oprogramowania i posiada najwyższą specjalizację, najczęściej pisany jest z myślą o konkretnym gatunku gier lub przynajmniej zestawie gatunków o zbliżonych wymaganiach technologicznych, jak np. współczesne gatunki FPS i TPP. Poniżej podane jest zestawienie typowych możliwości edytora świata gry:

- *Tworzenie fragmentów świata* – przez fragmenty świata rozumie się zazwyczaj poziomy (ang. *levels*) lub mapy, zależnie od gatunku gry. Edytor powinien pozwalać tworzyć, edytować, zapisywać i ponownie odczytywać takie fragmenty świata gry. Zazwyczaj z takim poziomem czy też mapą związane jest kilka elementów, które nie są częścią modelu obiektów gry, jak na przykład teren czy *sky box*. Ta część edytora może udostępniać dodatkowe funkcjonalności związane z tymi elementami, jak na przykład tworzenie terenu z mapy wysokości czy ustawianie globalnego oświetlenia na poziomie.
- *Wstawianie obiektów gry* – najważniejszy element edytora świata gry. Pozwala na rozstawianie na poziomie wszelkich obiektów przewidzianych przez edytor. Mogą to być elementy statyczne typu drzewa oraz budynki, ale przede wszystkim elementy dynamiczne: modele postaci, przeciwnicy, oświetlenie itp. Zależnie od sposobu implementacji edytora możliwości edycji obiektów gry mogą się znacznie różnić - od pełnej kontroli praktycznie równej z modyfikowaniem wszelkich właściwości dostępnych z poziomu kodu modelu obiektów gry, po bardzo uproszczony model rozstawiania obiektów z domyślnymi parametrami i możliwością zmiany przykładowo tylko pozycji obiektu. Więcej na temat implementacji modelu obiektów gry po stronie edytora znajduje się w podrozdziale Implementacja.
- *Zarządzanie assetami* – wiele obiektów gry, jakie może wstawiać użytkownik edytora, jest ściśle związanych z assetami takimi jak modele czy tekstury. W edytorze powinna być możliwość łatwego połączenia danego

zasobu graficznego czy muzycznego z konkretnym obiektem gry. Ponieważ większość silników działa na konkretnych formatach plików, musi istnieć sposób na konwersje materiałów z plików źródłowych na te używane przez silnik. Edytor można zintegrować z konwerterem lub przynajmniej dać możliwość łatwego przeglądania elementów już skonwertowanych. Więcej na temat konwersji assetów w rozdziale 6.1.2 Plug-iny.

! Definicja 6.1. Assety

Assety (pol. zasoby) - modele, tekstury, filmy, muzyka i wszystko inne, co jest tworzone przez artystów, jako zasób do użycia w grze przez designerów lub programistów. Nazwa stosowana zazwyczaj przez programistów w formie angielskiej, by nie mylić jej z zasobami takimi, jak pamięć czy też czas procesora.

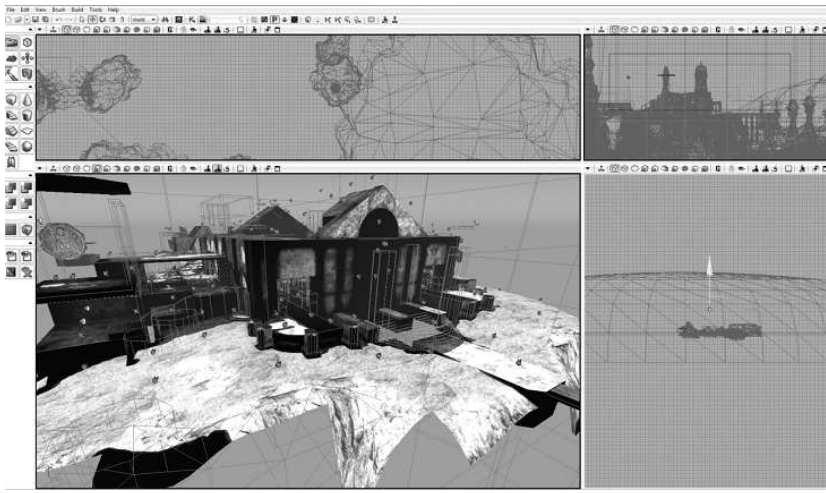
- *Funkcjonalności dodatkowe* – każdy element kodu gry, który wymaga podania konkretnych parametrów definiujących pewien efekt widziany w świecie gry, jest dobrym materiałem na udostępnienie możliwości modyfikacji tych danych w edytorze. Dobrym przykładem takiej dodatkowej funkcjonalności jest edytor efektów cząsteczkowych. Wiele silników udostępnia osobne moduły pozwalające na tworzenie takich efektów z możliwością modyfikacji praktycznie każdego parametru, do jakiego ma dostęp programista bezpośrednio używający technologii silnika. Edytory takie zazwyczaj mają możliwość natychmiastowego podglądu efektów prac, co znacznie ułatwia tworzenie efektów cząsteczkowych i udostępnia tę możliwość na przykład grafikom, niekoniecznie programistom.

Interfejs

Najważniejszym elementem dla użytkownika edytora świata gry jest przejrzysty i funkcjonalny interfejs. Jako że edytor jest narzędziem bardzo intensywnie wykorzystywanym, często przez wiele osób naraz, efektywność jego interfejsu przekłada się w prosty sposób na efektywność zespołu, który na nim pracuje. Poniżej zestawienie podstawowych funkcji, jakie powinien udostępniać edytor świata gry:

- *Wizualizacja* – główną zaletą edytora nad sztywną implementacją w kodzie jest graficzny interfejs, który pozwala na znacznie lepsze odwzorowanie pożądanego efektu w świecie gry. Zależnie od technologii, wizualizacja w edytorach może się różnić. W najpopularniejszych edytorach do gier 3D, najczęstszym rozwiązaniem jest pokazanie perspektywicznego okna z widokiem trójwymiarowym oraz ortogonalne rzuty świata gry z trzech osi, na modłę programów graficznych takich jak 3D Studio. Zależnie od implementacji edytora, widok perspektywiczny może być uproszczony lub

też oparty na silniku gry uruchomionym w oknie. Więcej na ten temat w dalszych rozdziałach książki.



Rysunek 6.1. Okno edytora silnika Unreal

Skróty klawiszowe

Przy implementacji edytora świata gry bardzo ważna jest implementacja skrótów klawiszowych dla najczęściej wykonywanych operacji. Absolutnym minimum jest implementacja skrótów kopiuj/wklej (standardowo przyjęte jako Ctrl + C / Ctrl + V). Skróty klawiszowe znacznie przyśpieszają prace oraz pozwalają unikać niepotrzebnych i powtarzalnych kliknięć, czy też ruchów myszą. Klawiatura jest znacznie szybsza w użyciu niż mysz. Każdy programista powinien nie tylko implementować skróty klawiszowe w aplikacjach GUI tworzonych przez siebie, ale również intensywnie wykorzystywać skróty klawiszowe w aplikacjach z których korzysta, np. ze środowiska programistycznego. W Visual Studio np. znacznie szybciej jest użytkownikowi wcisnąć F7 niż kliknąć w zakładkę „Build", a potem w „Build solution". Jeśli oszczędzimy dzięki skrótowi klawiszowemu tylko 3 sekundy potrzebne na wybranie np. opcji z menu, to zakładając np. 50-krotne użycie w ciągu godziny skrótu klawiszowego zamiast opcji z menu otrzymamy 150 sekund oszczędności. W skali 8-godz. dnia pracy da nam 1200 sekund, a więc 20 min.! W ciągu tygodnia to już 100 min., a patrząc na to w skali roku - będziemy mieli kilkadziesiąt godzin oszczędności. Grafik pracujący w programie *Adobe Photoshop* potrafi używać skrótów klawiszowych dużo częściej niż 50 razy w ciągu godziny...

- *Nawigacja* – użytkownik powinien mieć możliwość swobodnego poruszania się po świecie widocznym w edytorze na zasadzie wolnej kamery, czyli swobody w każdym kierunku i orientacji.
- *Warstwy* – liczba elementów w świecie gry może być ogromna. Możliwość grupowania oraz wyświetlania tylko pewnych konkretnych typów obiektów znacznie upraszcza pracę w takich przypadkach. Przykładowo, w edytorze może być opcja wyłączania i włączania widoku wszelkiej statycznej geometrii obecnej w danym fragmencie świata, lub wyświetlenie tylko efektów cząsteczkowych. Mechanizm szeroko stosowany w programach graficznych, takich jak na przykład w *Photoshopie*.
- *Właściwości* – najważniejszą czynnością po wstawieniu obiektów w edytorze jest edycja ich właściwości. To właśnie tutaj ożywiany jest świat gry. To, jakie właściwości są dostępne do edycji przez designera, bezpośrednio zależy od modelu obiektów gry stosowanego w silniku oraz sposobu implementacji edytora: inne właściwości dostępne są, gdy implementacja jest uproszczona, a jeszcze inne, gdy przeniesione są wszystkie możliwości z silnika gry. Przykładem właściwości obiektu może być parametr typu punkty życia lub skrypt definiujący jego zachowanie.

Języki skryptowe

Większość współczesnych gier wideo stara się zgodnie z MVC oderwać logikę gry od sztywno implementowanych algorytmów. W takim przypadku logika najczęściej jest zapisana w formie skryptowej, co pozwala na tworzenie zawartości gry przez osoby bez zaawansowanych umiejętności programistycznych, na przykład designerów. Zależnie od edytora skrypty mogą być wprowadzane przez jego interfejs, na przykład przez właściwości obiektów lub mogą wymagać osobnej edycji, przykładowo w zwykłym edytorze tekstu. Poniżej zestawienie najpopularniejszych języków skryptowych używanych przy tworzeniu gier wideo:

- **Lua** – jeden z najpopularniejszych języków skryptowych, szeroko stosowany w branży gier wideo. Lua jest językiem z dynamicznym typowaniem, co oznacza, że zmienne nie posiadają typu, tylko konkretne wartości. Główną strukturą danych w tym języku jest tablica asocjacyjna. Syntaktycznie Lua to praktycznie język C - w zasadzie *Lua* została napisana jako biblioteka do języka C. Posiada swoją własną maszynę wirtualną C, co pozwala na odwoływanie się bezpośrednio do konkretnych funkcji napisanych w języku C. Umożliwia to bardzo proste połączenie *Lua* z silnikiem gry. Do gier skryptujących logikę w Lua zaliczają się takie tytuły, jak *Fable 2, Crysis* czy *Wiedźmin*.
- **UnrealScript** – stworzony całkowicie od postaw na potrzeby silnika *Unreal Engine* firmy *Epic Games*. Jego syntaktykę oparto na języku C++, posiada wszystkie jego podstawowe elementy, takie jak klasy, pętle, zmienne i referencje. Nie posiada wsparcia dla wskaźników. Posiada wiele

zaawansowanych mechanizmów, jak redefinicja oraz rozszerzenie natywnego modelu obiektowego języka C++, co pozwala w czasie interpretacji na tworzenie i dziedziczenie klas w oparciu o już istniejące klasy w kodzie C++. Posiada możliwość usypiania funkcji skryptowych, czyli wywołania ich po upłynięciu odpowiedniego czasu, natywne wsparcie dla replikacji sieciowej skryptów oraz możliwość natychmiastowej integracji z edytorem *UnrealEd*. Wszystkie gry stworzone w oparciu o *Unreal Engine* korzystają z tego języka skryptowego. Lista tytułów jest bardzo długa z racji ogromnej popularności tego silnika, m.in. są to takie gry jak: *Mass Effect, Batman: Arkham Asylum* czy *Gears of War*.

Definicja 6.2. Kacze typowanie

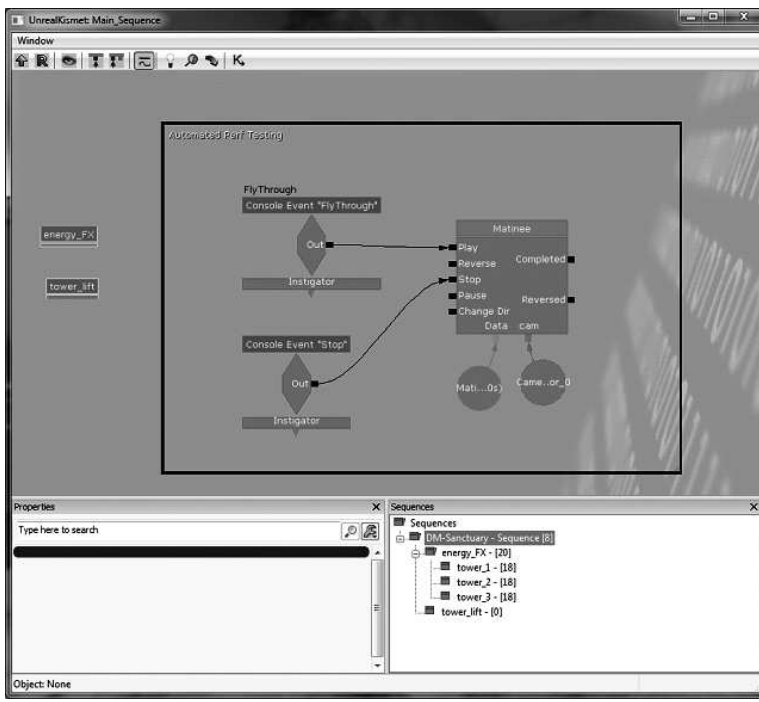
Kacze typowanie (ang. *duck typing*) - jest to rozpoznawanie typu obiektu poprzez sprawdzenie metod, jakie udostępnia, a nie na podstawie jego deklaracji, jak to się dzieje zazwyczaj. Pozwala to przykładowo na dowolne rzutowanie wskaźników pomiędzy dwiema klasami, pod warunkiem, że udostępniają wspólny interfejs, ale nie muszą po nim dziedziczyć. Pozwala to na wykorzystanie polimorfizmu bez korzystania z dziedziczenia. Nazwa wzięła się od powiedzenia: „jeżeli coś chodzi jak kaczka i kwacze jak kaczka, to musi być kaczką".

- **Python** – stworzony przez Guido van Rossuma, swą syntaktyką bardzo przypomina C, jednak znacznie różni się w podejściu do struktur danych. W *Python*ie głównymi strukturami danych jest lista sekwencyjnie indeksowanych atomicznych wartości lub innych list oraz słowników, czyli tablic sparowanych wartości. Każda z tych struktur danych może trzymać w sobie instancje drugiej, co daje możliwości znacznej rozbudowy struktur danych w tym języku. *Python* wspiera kacze typowanie, dzięki czemu można używać w nim polimorfizmu bez konieczności używania dziedziczenia. Do najpopularniejszych gier używających *Python*a jako języka skryptującego logikę gry należą takie tytuły jak *Civilization IV* i *EVE Online*.
- **QuakeC** – stworzony przez Johna Carmacka pracującego wówczas dla firmy *Id Software* na potrzeby gier na silniku *Quake Engine*. *QuakeC* to uproszczona wersja języka C. Nie posiada wsparcia dla wskaźników ani struktur, ale pozwala na bezpośrednie odwołania do silnika *Quake Engine*, czy też manipulowanie bytami (ang. *entities*), czyli głównymi obiektami gry w tym silniku oraz wysyłania i obsługi zdarzeń w grze. *QuakeC* był pierwszym językiem skryptowym masowo używanym przez społeczność graczy do modowania gier opartych na silniku *Quake Engine*. Używany jest we wszystkich grach opartych na *Quake Engine*, czyli na przykład *Half Life* i *Resident Evil 2*.

Definicja 6.3. Modowanie

Modowanie, czyli spolszczenie słowa oznaczającego tworzenie modów do gier wideo. *Mod* to skrót od angielskiego *modification*. Polega na wprowadzaniu zmian do gry wideo przez społeczność graczy lub niezależnych deweloperów. Modowanie jest zazwyczaj procesem w pełni legalnym i takie firmy jak id Software, Valve Software, Bethesda Softwork czy Firaxi wręcz zachęcają do tworzenia *mod*ów, udostępniając szereg narzędzi i dokumentację, by ten proces wspomóc. Mod może być drobny, zmieniający pewne aspekty rozgrywki lub może tworzyć zupełnie inną grę w oparciu o oryginał. Najpopularniejszym modem w historii gier wideo jest *Counter Strike*, mod do gry *Half Life* stworzonej przez *Valve Software*. *Counter Strike* przerósł popularnością oryginał i osiągnął miano gry uprawianej jako e-sport obok takich tytułów jak *Starcraft* czy *Tekken*.

Niektóre edytory posiadają graficzny interfejs skryptowania. *Unreal Engine* np. posiada narzędzie o nazwie Kismet, które pozwala na edycję skryptów języka *UrealScript* pod postacią bloczków logicznych, jak pokazuje rys. 6.2.

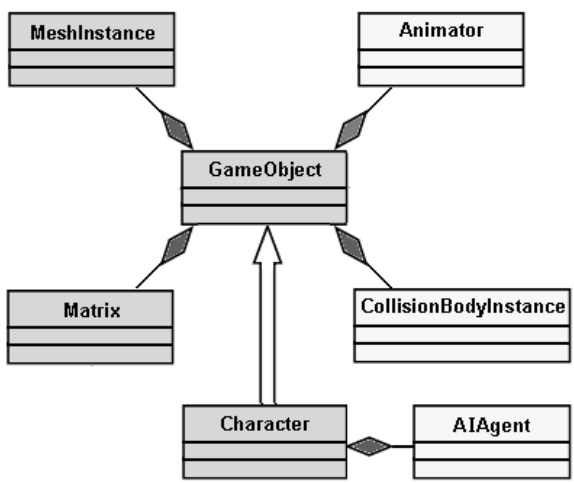


Rysunek 6.2. Okno graficznego interfejsu skryptowania Kismet w silniku Unreal

Implementacja

W podrozdziale 6.1.1.2 zostało opisanych kilka zagadnień związanych z implementacją interfejsu edytora, czyli tego, co dla użytkownika edytora jest najważniejsze. Jednak edytor pod warstwą interfejsu musi mieć ukrytą warstwę odpowiedzialną za translację danych dostarczanych przez użytkownika na faktyczne obiekty znajdujące się w kodzie silnika gry. Można to rozwiązać na dwa sposoby: w oderwaniu od silnika gry lub w oparciu o niego. Poniżej zostaną wyjaśnione oba podejścia.

- **Implementacja niezależna** – w tym podejściu programista przedstawiając efekty pracy designera nie korzysta z silnika gry, tylko z własnej uproszczonej implementacji świata gry lub w oparciu o istniejące oprogramowanie pozwalające na podstawowe przedstawienie świata i manipulowanie obiektami w przestrzeni, jak np. programy graficzne typu *Maya* czy *3D Studio*. W takim podejściu lista obiektów udostępniona przez edytor designerowi prezentuje uproszczony model obiektów gry, który jest wzorowany na modelu w warstwie silnika. Przykładowo, jeżeli postać w edytorze jest przedstawiona jako obiekt o nazwie `Character`, do którego można z listy dobrać odpowiedni model graficzny, oraz uzupełnić skrypt zachowania, to po stronie modelu obiektów gry w silniku będzie to klasa `Character` składającą się z takich komponentów jak model, fizyka i AI. Na tym przykładzie widać różnice w module fizyki: przykładowo designer nie musi definiować obiektu kolizyjnego dla postaci, gdyż zostanie on automatycznie stworzony na podstawie dobranej siatki modelu. Różnice w tych modułach pokazuje rys. 6.3. Wyszarzone zostały klasy, które nie wymagają implementacji po stronie edytora. Designer musi mieć możliwość przemieszczania obiektu (Matrix)



Rysunek 6.3. Model obiektów gry w silniku i w edytorze

oraz dobrania mu modelu (MeshInstance), natomiest pozostałe elementy muszą być obecne tylko w czasie wykonywania się kodu, niekoniecznie w edytorze.

Przy tym sposobie implementacji, pomostem łączącym te dwa modele są dane. Model obiektów gry po stronie silnika gry musi mieć możliwość importowania danych eksportowanych przez edytor. Zazwyczaj odbywa się to dwufazowo. W pierwszym etapie eksportowane są łatwe do zmiany i podglądu przez człowieka formaty plików, na przykład XML. W drugim etapie konwertowane są one zazwyczaj na format binarny, by zminimalizować czas tworzenia obiektów w czasie działania gry. Ten sposób implementacji edytora jest bardzo popularny, gdyż zazwyczaj uproszczona implementacja edytora sprawia, że jest on bardzo prosty w obsłudze i umożliwia stabilną pracę designerom przy jednoczesnej możliwości wprowadzania zmian do kodu silnika gry programistom. Całkowite oderwanie gry od edytora sprawia, że każda zmiana w założeniach musi być odwzorowana w dwóch miejscach, co utrudnia jego utrzymanie. Również fakt niedokładnego odwzorowania świata gry w edytorze sprawia, że designerzy nie będą mieli pełnej kontroli nad faktycznym stanem gry, dopóki jej nie uruchomią. Przykładem takiego silnika może być *Blender Game Engine*, który jest oparty na programie graficznym *Blender*.

- **Implementacja w oparciu o silnik gry** – w tym rozwiązaniu widok edytora jest generowany przez ten sam silnik, jaki jest wykorzystywany w grze. Nie istnieje żadna pośrednia warstwa abstrakcji pomiędzy obiektem umiejscowionym w edytorze przez designera a fizycznym obiektem modelu gry. Ograniczenie w modyfikacji właściwości obiektów programista może narzucić jedynie przez ograniczoną implementację interfejsu edytora. W przykładzie z poprzedniej implementacji użytkownik stworzyłby pełną klasę `Character` w momencie wstawienia jej do fragmentu świata gry w edytorze, jednak właściwości, jakie mógłby ustawiać designer, mogłyby być ograniczone do wyboru modelu postaci oraz zmiany pozycji i orientacji. Największą zaletą tego rozwiązania jest wierniejsze odzwierciedlenie możliwości silnika gry w edytorze. Łatwiejsze jest również utrzymanie takiego rozwiązania – zmiany w obiektach gry wprowadzane są w jednym miejscu. Może to być również postrzegane jako wada w pewnych przypadkach. Przy większych zmianach w silniku gry stabilność edytora może być mocno zagrożona. Trudniej utrzymać ciągły development przy jednoczesnym zachowaniu stałej funkcjonalności edytora, który w pełni polega na silniku gry.

Niezależnie od sposobu implementacji edytora gry, zawsze warto w nim dodać obsługę natychmiastowego uruchomienia danego fragmentu gry. W przypadku implementacji opartej na silniku jest to bardzo proste, w przeciwnym wymaga dodatkowej pracy w postaci implementacji automatycznego eksportu danych z edytora oraz uruchomienia gry z importem tych danych.

6.1.2 Plug-iny

Plug-in (pol. wtyczka) jest to dodatek do istniejącego programu rozszerzający jego możliwości. W branży gier wideo wtyczki stosuje się głównie po to, żeby do aplikacji niededykowanych do tworzenia gier dodać funkcjonalności, które będą ułatwiały współpracę przy tym rodzaju projektu. Najczęściej stosowaną wtyczką jest eksporter.

Zdecydowana większość silników gier obsługuje konkretny, często autorski, format wszelkich assetów. Konwersja z formatu oryginalnego, przykładowo modelu stworzonego w programie *3D Studio Max*, może się odbyć na dwa sposoby. Albo silnik posiada funkcjonalność importu modeli z pewnego zestawu formatów, wtedy przeważnie jest to narzędzie zintegrowane z edytorem gry, albo też do programu graficznego można dodać wtyczkę odpowiedzialną za eksport modelu do odpowiedniego formatu. Przykładem takiej wtyczki może być *Panda*, która pozwala na konwersję modeli stworzonych w *3D Studio Max* na format modeli *DirectXa*. Ekstremalnym przykładem wtyczki do programu graficznego jest edytor stworzony w oparciu o ten program, jak na przykład wcześniej wspomniany *Blender Game Engine*.

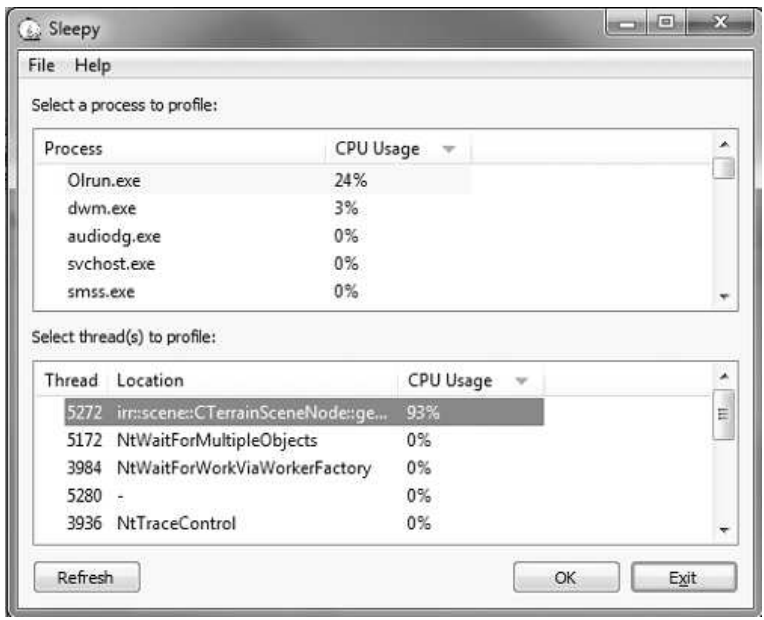
Jeden silnik, jeden format

Programista może sobie zadać pytanie, dlaczego większość silników posiada swój własny format wczytywania modeli czy też muzyki lub co najwyżej obsługuje jeden zewnętrzny, zamiast być zgodnymi z większością standardów przyjętych na rynku? Istnieją konkretne powody, dla których jest to rozwiązanie optymalne.
Po pierwsze, obsługa kilku formatów w czasie rzeczywistego wykonywania programu wymagałaby stworzenia kodu implementującego interpretację każdego z nich, kod ten wchodziłby w skład pliku wykonawczego i niepotrzebnie zwiększał jego rozmiar. Po drugie i najważniejsze, wczytywanie różnego rodzaju formatów i tak musiałoby wewnątrz silnika skończyć się konwersją na jeden zunifikowany format, w przeciwnym przypadku moduły przykładowo wykorzystujące informacje z siatki modelu, musiałyby implementować różne sposoby odczytywania danych, jak na przykład silnik kolizyjny - dane na temat poszczególnych wierzchołków. W takim przypadku przeprowadzanie konwersji w czasie wykonywania programu jedynie niepotrzebnie zwiększy czasy ładowania się świata gry.
Dlatego też, choć z początku wydaje się, że obsługa kilku formatów z poziomu kodu silnika gry, jak to na przykład jest zrealizowane w silniku *Irrlicht*, jest wygodna dla programisty, to jednak jest rozwiązaniem nienajlepszym, którego należy unikać.

6.1.3 Inne

Poza opisanymi wcześniej edytorami i plug-inami istnieje cała gama narzędzi dodatkowych, które mogą wykorzystywać zespoły developerskie przez wsparcie pracy designerów, artystów czy też programistów. Na uwagę od strony programistycznej zasługują dodatkowo dwa typy aplikacji:

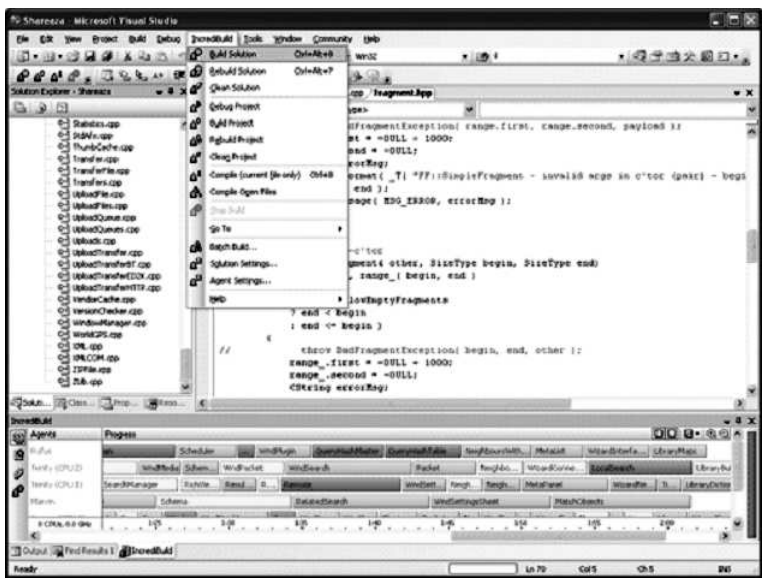
- **Profiler** – narzędzie udostępniane zazwyczaj przez producenta danej platformy oraz jej SDK, służy do sprofilowania obciążenia poszczególnych podzespołów platformy (procesor centralny, graficzny, dysk, pamięć itp.) przez uruchomioną grę. Profiler uruchomiony na odpowiednio skomplikowanym projekcie potrafi wskazać konkretne obiekty czy funkcje, które najbardziej obciążają dany zasób oraz całościowo pokazać, które elementy aktualnie w jakikolwiek sposób oddziaływają na wydajność. Większość tego typu aplikacji udostępnia dane również w postaci wykresów czasowych.



Rysunek 6.4. Okno profilera Sleepy, uruchomionego na grze napisanej na silniku Irrlicht

- **Rozproszony system kompilacji** – większość współczesnych platform używanych do programowania posiada procesor wielordzeniowy, zaś większość kompilatorów, na przykład Visual Studio, stara się maksymalnie ten fakt wykorzystać. Niestety, w przypadku dużej gry wideo kompilacja oparta na wielordzeniowości, to często za mało. Dla projektów składających się z kilkuset tysięcy linii, niejednokrotnie stosujących różne obciążające kom-

pilator konstrukcje, czas pełnej kompilacji na pojedynczym komputerze może sięgać kilkudziesięciu minut. W takich przypadkach rozproszone systemy kompilacji są bardzo pomocne. Pozwalają bowiem na jednoczesną kompilację projektu na kilku maszynach naraz. Oczywiście przyspieszenie kompilacji w stosunku do ilości maszyn biorących udział w kompilacji nie jest liniowe, ale przykładowo możliwe jest osiągnięcie czterokrotnego przyspieszenia przy użyciu sześciu maszyn kompilujących. Oczywiście nie wymaga to zakupu dodatkowego zestawu komputerów dla każdego programisty. Odpowiednie oprogramowanie, np. *IncrediBuild*, instalowane jest na komputerze każdego członka zespołu i wolne moce przerobowe procesora na każdym z nich są automatycznie przydzielane do wspomożenia kompilacji rozproszonej.



Rysunek 6.5. IncrediBuild zintegrowany ze środowiskiem Visual Studio. Kolorowe bloczki na dole reprezentują fragmenty kodu kompilowane równolegle na kilku komputerach

Często wiele narzędzi powstaje na potrzeby konkretnego projektu i wówczas zazwyczaj są one zintegrowane z edytorem świata gry. Warto jest identyfikować czasochłonne procesy manualne w projekcie i starać się je zautomatyzować. Przykładowo, jeżeli grafik za każdym razem kopiuje stworzoną grafikę do konkretnego folderu, nazywając ją według odpowiedniej konwencji, tak żeby była zgodna ze standardem ustanowionym przez programistów w edytorze świata gry, warto napisać krótki program, który będzie tę operację automatyzował. Usuwanie elementów, które można potencjalnie zautomatyzować, ma

jeszcze jedną ogromną zaletę. Do automatyzacji konieczna jest powtarzalność danej czynności, każda tego typu czynność wykonywana manualnie jest bardzo podatna na błędy. Jeśli kopiowane jest ręcznie sto plików i każdemu z nich jest ręcznie zmieniana nazwa, na pewno w kilku z nich człowiek popełni błąd, który na późniejszym etapie będzie musiał być wychwycony i poprawiony.

6.2 Narzędzia wspomagające prowadzenie projektu

6.2.1 Systemy śledzenia błędów

System śledzenia błędów to aplikacja stworzona do pomocy testerom i programistom przy kontroli zgłaszanych błędów do oprogramowania. Zgłaszający błąd ma zazwyczaj możliwość uzupełnienia takich pól jak opis błędu, kroki potrzebne do jego odtworzenia, priorytet oraz osobę, do której dany błąd chce przypisać, czyli zazwyczaj do programisty odpowiedzialnego za daną funkcjonalność. Dodatkowo można dołączyć załącznik do zgłaszanego błędu, na przykład w postaci zrzutu ekranu. Większość systemów śledzenia błędów, jak na przykład *Mantis Bug Tracker* czy *Bugzilla* jest aplikacjami zdalnymi działającymi w oparciu o przeglądarkę internetową, natomiast po stronie serwera o bazę danych.

W branży gier wideo rzadko stosowane są zaawansowane metodologie zarządzania typu PRINCE2 i związane z nimi narzędzie, charakterystyczne dla dużych korporacji IT. Wynika to głównie z innej specyfiki projektów będących grami wideo. Przy naprawdę małych projektach do celu przydzielania zadań może wystarczyć komunikacja ustna i e-mailowa, jednak w praktyce prowadzi ona w prostej linii do chaosu. Trudno jest zachować kontrolę nad wykonaniem i przydzielaniem setek czy tysięcy zadań, opierając się tylko na e-mailach czy komunikacji ustnej. Okazuje się, że systemy śledzenia błędów w wielu firmach tworzących gry służą z powodzeniem jako sposób na przekazanie nie tylko błędów, ale też uwag i nowych pomysłów, w skrócie – zadań. W razie potrzeby wiele takich systemów posiada możliwości rozbudowy i dowolnej implementacji wtyczek, które mogą usprawnić i dostosować system śledzenia błędów do potrzeb rozdzielania zadań. Są przykładowo wtyczki do *Mantis Bug Trackera*, które dostosowują go do rozdzielania zadań w systemie zwinnego zarządzania SCRUM.

Systemy śledzenia błędów wraz z systemem kontroli wersji opisanym poniżej są dwoma podstawowymi narzędziami, z których przyjdzie korzystać praktycznie każdemu programiście pracującemu w branży nie tylko gier vieo, ale ogólnie IT.

6.2.2 System kontroli wersji

System kontroli wersji, również nazywany systemem kontroli rewizji, jest oprogramowaniem zarządzającym zmianami wprowadzanymi do plików. Główny

nacisk położony jest na pliki tekstowe, takie jak kody źródłowe programów, ale w istocie można go używać do przechowywania plików każdego typu - grafik, dźwięków, dokumentów czy plików wykonywalnych. Każda zmiana wprowadzona do pliku oznaczona jest unikalnym kodem, zwanym numerem rewizji. System taki jest konieczny w momencie, w którym nad projektem pracuje więcej niż jedna osoba, ale nawet jeżeli jest tylko jedna, to przydaje się jako forma kopii zapasowej plików. W przypadku programistów głównym problemem, jaki rozwiązuje system kontroli wersji, jest praca równoległa nad tym samym projektem, modułem a nawet nad tym samym plikiem kodu źródłowego, wykonywana przez kilku programistów jednocześnie. Ciężko jest sobie wyobrazić programistów wzajemnie przekazujących sobie co chwilę fragmenty kodu e-mailem lub na nośnikach wymiennych w celu uaktualnienia projektu na poszczególnych stacjach roboczych. Systemy kontroli wersji dbają o istnienie spójnego miejsca w którym przechowywane są pliki projektu. Miejsce to nazywane jest repozytorium. Istnieją dwa podejścia do technicznej realizacji takiego systemu:

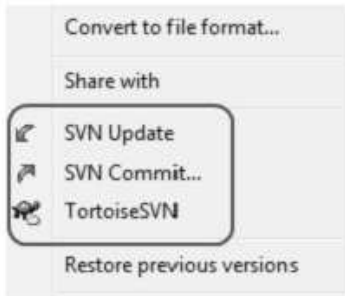
- **Scentralizowane** – oparte na architekturze klient–serwer. Na serwerze istnieje jedno repozytorium, z którym wszyscy klienci synchronizują swoje zmiany. Jest to najpopularniejsze rozwiązanie zastosowane w takich systemach kontroli wersji jak na przykład Subversion, znany również jako SVN.
- **Rozproszone** – oparty na architekturze P2P, gdzie każdy użytkownik repozytorium jest jedną z niezależnych gałęzi (ang. *branch*), które można synchronizować wzajemnie, jednak bez jednoznacznego określenia jednej gałęzi najbardziej aktualnej. Przykładem takiego systemu kontroli wersji jest *Bazaar*.

W branży gier wideo najczęściej korzysta się z architektury scentralizowanej, gdyż musi istnieć jedno konkretne miejsce, w którym znajduje się najbardziej aktualna wersja kodu gry. W praktyce najpierw administrator powinien zainstalować system kontroli wersji na serwerze, a następnie na każdej stacji projektowej należy zainstalować oprogramowanie klienckie. Większość takich klientów instalowanych jest jako rozszerzenie do systemu operacyjnego, na przykład *Tortoise SVN* jest rozszerzeniem powłoki *shell*. Po instalacji programista będzie miał w obrębie menu kontekstowego eksploratora Windows możliwość wykonania na folderze różnych operacji, co widać na rys. 6.6-6.7.

Najważniejsze operacje związane z obsługą systemu kontroli wersji są następujące:

- **Checkout** – Połączenie konkretnego folderu na dysku z repozytorium będącym na serwerze. Od tego momentu będą na tym folderze dostępne wszystkie pozostałe opcje systemu kontroli wersji.
- **Update** – Ściągnięcie wszystkich danych z repozytorium do lokalnego folderu SVN, czyli aktualizacja projektu o wszystkie zmiany, które zostały

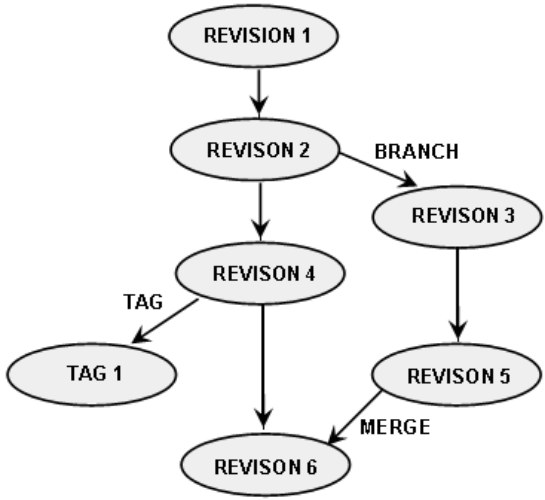
Rysunek 6.6
Opcja Checkout dostępna na niezwiązanym jeszcze z żadnym repozytorium folderze



Rysunek 6.7
Opcje Update i Commit, dostępne na folderze związanym z konkretnym repozytorium

wysłane na serwer. Ściągane będą fizycznie tylko te dane, które się zmieniły lub są nieobecne w folderze lokalnym klienta. Istnieje też możliwość ściągnięcia danych z konkretnej, wcześniejszej rewizji. Podczas wykonywania update'u rozwiązywane są wszelkie konflikty. Jeśli na wersji lokalnej zostały wprowadzone zmiany w pliku, który również ktoś inny zmodyfikował i uaktualnił na serwerze, SVN spróbuje te pliki połączyć. W przypadku plików tekstowych wykona porównanie linijka po linijce. Jeżeli zmiany występują w różnych miejscach pliku, połączenie zostanie przeprowadzone automatycznie. Jeżeli zmiany występują w tym samym miejscu - to miejsce zostanie oznaczone i konflikt będzie musiał rozwiązać użytkownik. W przypadku plików binarnych, takich jak dokumenty czy grafiki, nie ma takiej możliwości i cała operacja jest sterowana fizycznie przez użytkownika. Generalnie SVN nie radzi sobie dobrze z plikami innymi niż tekstowe i trzeba bardzo uważać, by na repozytorium nie zrobił się bałagan, jeżeli jest ono używane na przykład do grafik.

- **Commit** – Uaktualnienie plików serwera o zmiany dokonane lokalnie na komputerze i zwiększenie licznika rewizji o jeden. Zawsze przed commitem należy zrobić update, by się upewnić, że nie będzie żadnych konfliktów. Jest to operacja atomowa w przypadku SVN (w starszej wersji, w CVS, jeszcze nie była). Oznacza to, że w razie kłopotów - np. utraty połączenia z serwerem - nastąpi wycofanie wszystkich zmian do stanu sprzed próby połączenia z serwerem. Przypomina to system transakcji z SQL.
- **Branch** – Rozszczepienie repozytorium. Od tego momentu uaktualnienia wysyłane z tej konkretnej gałęzi będą trafiały w osobne miejsce na serwerze. Pozostali użytkownicy uaktualniając swój projekt, nie będą ściągać tych plików, chyba że fizycznie wymuszą update z brancha. Umożliwia to przykładowo rozwój dwóch niezależnych koncepcji projektu równolegle. Później można np. połączyć gałęzie repozytorium przy pomocy merge.



Rysunek 6.8. Przykładowy graf rewizji

- **Tag** – Oznaczenie aktualnej wersji na serwerze oraz skopiowanie jej w osobne miejsce. W bardzo łatwy sposób można potem wykonać checkout na konkretnym tagu. Służy do rozróżnienia na serwerze ważniejszych rewizji, na przykład takiej, z której powstała wersja beta gry lub demo jakiejś funkcjonalności.

Warto robić tagi

Biorąc pod uwagę liczbę zmian i częstotliwość, z jaką są wprowadzane do projektu gry wideo, nieocenione jest posiadanie konkretnych rewizji, które są dobrze przetestowane i pokazują pewne konkretne funkcjonalności. Przykładowo, jeżeli ktoś - np. producent - poprosi o wgląd do aktualnego stanu gry, który akurat nie będzie zbyt stabilny z powodu trwających prac, można pokazać grę zbudowaną w oparciu o ostatniego taga. Między innymi z tego powodu warto robić tagi często, raz na tydzień - dwa, gdyż tag sprzed miesiąca może się bardzo różnić od aktualnego stanu gry.

- **Lock** – zablokowanie edycji pliku na serwerze. Nawet jeżeli ktoś zmodyfikuje ten plik w swoim lokalnym katalogu SVN, nie będzie miał możliwości wykonania *commita* na tym pliku. Najczęściej służy to do tymczasowego zablokowania pliku na okres intensywnych zmian w nim, by uniknąć potem problemów z łączeniem po zmianach. Mechanizm ten może też służyć do ochrony plików uznanych za finalne przed modyfikacją przez osoby niepowołane.

Oprócz umożliwienia pracy równoległej wielu osobom system kontroli wersji ma jeszcze jedną bardzo ważną zaletę: repozytorium na serwerze może służyć jako kopia zapasowa całego projektu. Ułatwia też robienie twardych kopii w postaci wypalonych płyt czy też dysków zewnętrznych, wystarczy jedynie skopiować odpowiednio wyeksportowaną bazę danych repozytorium.

System kontroli wersji jest nieodzownym narzędziem przy tworzeniu nie tylko gier wideo, ale jakiegokolwiek profesjonalnego oprogramowania, dlatego warto się nauczyć z niego korzystać, nawet przy bardzo małych, jednoosobowych projektach.

6.2.3 Dokumentacja

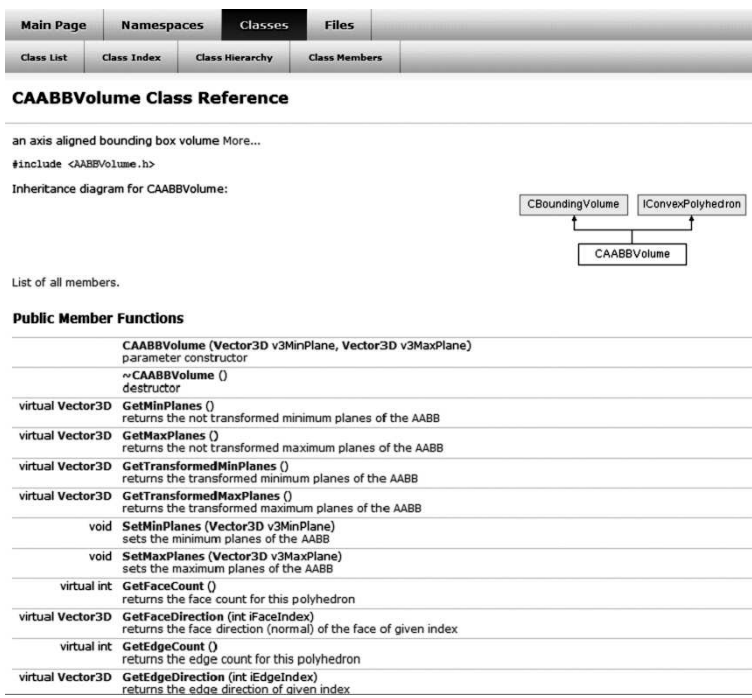
W branży gier wideo dokumentacja ma znacznie mniejsze znaczenie niż w standardowej firmie tworzącej aplikacje biznesowe. Przyczyn takiego podejścia jest kilka, w przypadku programistów najczęstszą przyczyną jest brak narzucenia obowiązku tworzenia dokumentacji przez osoby zarządzające projektem. Jako że programiści znani są ze swojej niechęci do tworzenia dokumentacji, trudno jest oczekiwać, że zdyscyplinują się do niej sami. Kolejną znaczącą przyczyną nietworzenia dokumentacji jest konieczność utrzymania jej zaktualizowanej w stosunku do kodu źródłowego gry wideo, który zmienia się bardzo dynamicznie. Zależnie od liczby zmian w projekcie, a jest ich zwykle dużo, cała architektura kodu może ulec kilkukrotnej przebudowie podczas cyklu życia projektu. Te dynamiczne zmiany wymagałyby natychmiastowego odwzorowania w dokumentacji, by jej istnienie miało sens. Dlatego często rezygnuje się całkowicie z jej tworzenia.

Fakt powszechności bagatelizowania dokumentacji nie oznacza, że jest to zjawisko dobre. Istnieje konkretny powód, dla którego warto o istnienie dokumentacji zadbać. Dobrze prowadzona dokumentacja jest źródłem wiedzy na temat kodu gry, która się przyda każdemu programiście, który danego fragmentu kodu nie stworzył. Powszechne efekty braku dokumentacji są następujące:

- Powrót do danego fragmentu kodu po nieco dłuższym okresie - czasem wystarczy parę tygodni - jest trudny nawet dla samego autora, który zapomina już „jak to wszystko działało".
- Proces poszukiwania błędów bywa dużo bardziej czasochłonny.
- W przypadku nieobecności innego programisty, który pracował nad fragmentem kodu wymagającym modyfikacji lub poprawy, programista musi poświęcić dużo czasu na wdrożenie się w problem.
- Programiści niezaznajomieni z kompletem funkcjonalności, jakie oferuje aktualny stan kodu gry, często implementują swoje własne wersje elementów, które już istnieją w architekturze gry, a więc wykonują niepotrzebną pracę.

- Przekazanie oprogramowania całemu zespołowi, przykładowo w celu zrobienia kolejnego produktu na silniku stworzonym przez inny zespół, wiąże się z bardzo długim czasem wdrożenia lub szkolenia. Dla silników komercyjnych taka sytuacja jest nie do zaakceptowania, dlatego w takich przypadkach prawie zawsze istnieje dokumentacja.
- Wdrożenie do zespołu nowego programisty wymaga intensywnego szkolenia przeprowadzonego przez programistę znającego kod projektu. Wdrożenie się zajmie nowemu programiście bardzo dużo czasu.

Istnieje sposób tworzenia dokumentacji, który od programisty wymaga jedynie dyscypliny przy tworzeniu komentarzy. Waga prawidłowych komentarzy została dosyć jasno podkreślona w rozdziale 2, tam też zostało wspomniane automatyczne tworzenie dokumentacji. W przypadku tworzenia kodu gry w takich językach jak C# czy Java sprawa jest prosta – istnieją tam wbudowane mechanizmy tworzenia dokumentacji. W przypadku języka C++ jest to trochę bardziej skomplikowane, należy w takim wypadku użyć zewnętrznego narzędzia. Dobrym przykładem jest tutaj opisany w rozdziale 2 *Doxygen*, który wymaga jedynie specjalnego oznaczenia komentarzy, które mają trafić do dokumentacji. Dokumentacja w takiej formie nie tylko zawiera opisy wszystkich klas projektu pobrane z komentarzy, ale również może wygenerować diagramy



Rysunek 6.9. Fragment dokumentacji wygenerowany narzędziem Doxygen

zależności, które bardzo pomagają w zrozumieniu architektury udokumentowanego w ten sposób kodu źródłowego.

Oczywiście w tym podrozdziale uwaga poświęcona była dokumentacji programistycznej. Przy tworzeniu gier wideo istnieje mnóstwo innych fragmentów dokumentacji: rysunki konceptowe, dokumenty opisujące ogólne założenia gry, czyli tzw. *pitch* gry, czy wreszcie dokumenty projektu całej gry (ang. *Game Design Doc*, często używane w postaci skrótu GDD), w którym rozpisana jest dokładnie cała mechanika i przepływ gry. W przeciwieństwie do dokumentacji czysto programistycznej dokumentacja projektowa, tworzona zazwyczaj przez designerów, musi istnieć w jakiejś, choćby minimalnej formie, zawsze.

Literatura

1. Bergen, G. v. (2004). Collision Detection in Interactive 3D Envirionments. San Francisco: Morgan Kaufmann Publishers.
2. Cockburn, A. (2007). Agile Software Development. Gra zespołowa. Wydanie II. Gliwice: Helion.
3. Eckel, B. (2002). Thinking in C++. Gliwice: Helion.
4. Eric Freeman, E. G. (2005). Head First Design Patterns. Gliwice: Helion.
5. Gregory, J. (2009). Game Engine Architecture. Wellesley: A K Peters, Ltd.
6. Hamiltion, R. M. (2007). UML 2.0 Wprowadzenie. Gliwice: Helion.
7. Keith, C. (2010). Agile Game Development with Scrum. Boston: Addison-Wesley Professional.
8. LaMonthe, A. (2004). Triki Najlepszych Programistów Gier 3D. Gliwice: Helion.
9. McShaffry, M. (2009). Game Coding Complete Third Edition. Boston: Course Technology, a part of Cengage Learning.
10. Sherrod, A. (2008). Game graphics programming. Boston: Course Technology, a part of Cengage Learning.
11. Weisfeld, M. (2010). Myślenie obiektowe w programowaniu Wydanie III. Gliwice: Helion.

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

ISBN 978-83-63103-02-6

Egzemplarz bezpłatny