Politechnika Poznańska
Wydział Informatyki
Instytut Informatyki

Praca dyplomowa inżynierska

PROJEKT I IMPLEMENTACJA GRY KOMPUTEROWEJ Z WYKORZYSTANIEM TECHNOLOGII MICROSOFT DIRECTX 11

Krzysztof Marciniak, 106574
Piotr Przybysz, 106602
Mikołaj Szychowiak, 106580
Ryszard Wojtkowiak, 106609

Promotor
dr inż.Witold Andrzejewski

Poznań, 2015 r.

Tutaj przychodzi karta pracy dyplomowej;
oryginał wstawiamy do wersji dla archiwum PP, w pozostałych kopiach wstawiamy ksero.

Spis treści

Rozdział 1

Wstęp

1.1 Cel i zakres pracy

Rozdział 2

Przegląd zagadnień teoretycznych

2.1 Zarys działania Microsoft DirectX API

2.1.1 Struktura i podstawowe pojęcia

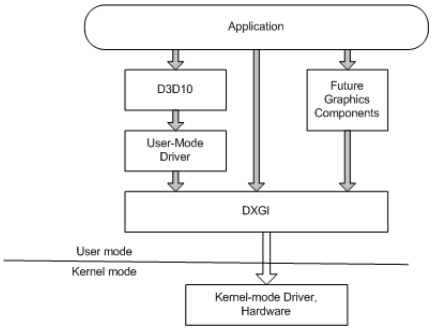
Microsoft DirectX to interfejs programistyczny (API, ang. *application programming interface*) do tworzenia aplikacji multimedialnych. Składają się na nie przede wszystkim:

- DirectDraw, Direct2D/Direct3D (komponenty odpowiedzialne za rysowanie grafiki),
- DirectSound, DirectMusic (obsługa dźwięków),
- DirectInput (obsługa wejścia - myszy, klawiatury, kontrolerów itp.).

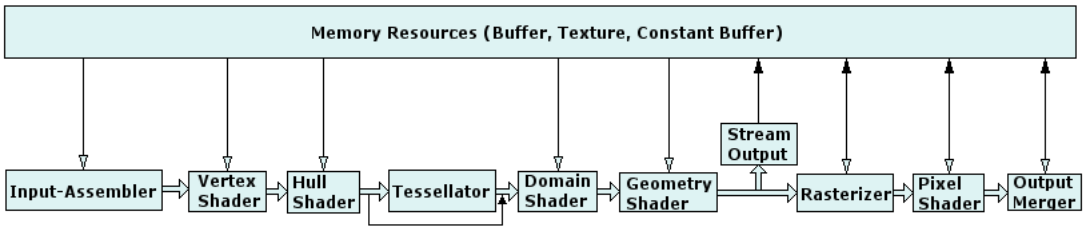
DirectX pozwala na tworzenie aplikacji w trzech językach - C#, Visual Basic oraz C++, przy czym w praktyce spotyka się głównie aplikacje stworzone w C# oraz C++. Jak widać na rysunku 2.1, struktura DirectX jest wielopoziomowa i, poza odwołaniami do API oraz komponentami jak na przykład Direct3D 11 (D3D11), obejmuje także DXGI (Infrastruktura Graficzna DirectX, ang. *Microsoft DirectX Graphics Infrastructure*), które – jako najniższa warstwa – komunikuje się bezpośrednio ze sterownikiem znajdującym się w przestrzeni jądra systemu operacyjnego. Wykorzystanie komponentów COM (ang. *Component Object Model*) pozwala na łatwą rozbudowę oraz jasny podział funkcjonalności.

Podstawowymi i koniecznymi do zrozumienia działania DirectX są koncepcje urządzenia (ang. *device*) oraz jego kontekstu (ang. *device context*).
Urządzenie, reprezentowane w wersji 11. przez interfejs ID3D11Device, reprezentuje kartę graficzną i może służyć do tworzenia zasobów oraz pobierania informacji o jej możliwościach (ang. *capabilities*), tj. oferowanych przez nią funkcjonalnościach takich jak np. obsługa podwójnej precyzji w programach cieniujących.

Kontekst urządzenia z kolei, jak wskazuje nazwa, określa kontekst użycia urządzenia, co jednoznacznie pokazuje, iż do prawidłowej pracy z urządzeniem powiązany powinien być co najmniej jeden kontekst. Pozwala on głównie na ustawienie stanu w procesie renderingu oraz prawidłowych komend wykorzystujących zasoby karty graficznej do wygenerowania obrazu (bezpośrednio



Rysunek 2.1: Architektura DirectX



Rysunek 2.2: Proces generowania obrazu w DirectX

na ekran lub do pośredniczącej tekstury, która może zostać wykorzystana później). Wyróżniamy dwa rodzaje kontekstów: bezpośredni/zwykły (*forward*) oraz opóźniony (*deferred*). Bezpośredni wykonuje komendy od razu gdy zostają wywołane, podczas gdy opóźniony pozwala na zapisanie ich na odpowiedniej liście, dzięki czemu mogą zostać wykonane później (jest to przydatne zwłaszcza w przypadku aplikacji wielowątkowych).

Programy cieniujące (ang. *shader*) to aplikacje działające na karcie graficznej, współcześnie pisane w językach wysokiego poziomu jak HLSL (dla DirectX) czy GLSL (dla OpenGL). Dwa najważniejsze to Vertex Shader (odpowiada on za obliczenie oświetlenia dla pojedynczych wierzchołków modelu) oraz Pixel Shader (w OpenGL nazywany Fragment Shader, interpoluje wartości wyjściowe z etapu pośredniego [Geometry Shader] i oblicza ostateczny kolor piksela).

2.1.2 Proces renderingu

Przebieg wygenerowania grafiki (renderingu) opisywany przez tzw. *pipeline* jest złożony z wielu etapów (ang. *stage*), z których część może być konfigurowana jedynie przez wywołanie odpowiednich komend API poprzedzonych prefiksem będącym skrótem nazwy danego etapu (np. *IA* dla etapu *Input-Assembler*), a część opisywana jest przez programy cieniujące (ang. *shader*).
W pipeline DirectX 11 wyróżnia się następujące etapy:

- Input-Assembler - odpowiada za wczytanie wierzchołków w sposób opisany przez prymityw (trójkąt, czworokąt itp.),
- Vertex Shader - cieniowanie wierzchołków, ustalanie dla nich wartości początkowych zmiennych jak np. kolor, wektor normalny itp. ,
- Hull Shader - pierwszy etap teselacji (zagęszczania siatki), przygotowuje siatkę do zagęszczenia przez ustalenie punktów kontrolnych,
- Tessellator - zagęszcza siatkę wprowadzając dodatkowe prymitywy zastępujące podstawowy i zwraca nowe współrzędne teksturowania,
- Domain Shader - generuje nowe pozycje wierzchołków na podstawie dwóch poprzednich etapów,
- Geometry Shader - pozwala zastąpić prymityw wejściowy innym prymitywem,
- Rasterizer - odpowiada za konwersję wektorowej postaci model na postać rastrową, czyli zbiór pikseli na ekranie,
- Pixel Shader - pozwala zmieniać kolor piksela w wynikowym obrazie,
- Output Merger - generuje końcowy obraz łącząc w odpowiedni sposób (opisany przez komendy takie jak np. OMSetRenderTargets) informacje z poprzedniego etapu.

Wszystkie te etapy przedstawione zostały na Rys. 2.2. Strzałki określają czy dany etap korzysta jedynie z odczytu danych z karty, czy też pozwala na ich zapis (lub oba równocześnie).

Ostatnim zasługującym na uwagę jest fakt, iż w większości aplikacji wraz z DirectX wykorzystywana jest technologia WinAPI pozwalająca na tworzenie aplikacji graficznych pod platformę

Windows. Głównym jej mechanizmem jest pętla komunikatów, których odebranie warunkuje sposób przetwarzania informacji w aplikacji (np. wciśnięcie klawisza powoduje przesunięcie modelu). Komunikaty mogą zostać odczytane w sposób blokujący (przez funkcję *GetMessage*) lub nieblokujący (*PeekMessage*). Z oczywistych względów (tj. narzutu czasowego), w aplikacjach generujących obraz w czasie rzeczywistym wykorzystywana jest wyłącznie funkcja nieblokująca.

2.2 Deferred Shading

Nawiązując do [Owe13], podstawowym sposobem obliczania oświetlenia, uwzględniając opisany wcześniej proces generacji obrazu, jest wyliczenie oświetlenia bezpośrednio dla każdego obiektu. Wartości takie jak wektory normalne, współrzędne teksturowania i inne używane we wspomnianym procesie są wczytywane w shaderze wierzchołków, a następnie interpolowane w pixel shaderze. W przypadku wielu źródeł oświetlenia (na przykład 2000 lub więcej), dla każdego obiektu (niezależnie od tego, czy jest widoczny) należy sprawdzić wszystkie źródła światła, co oznacza złożoność

$$O(\ liczba\_pikseli\_per\_obiekt\ *\ liczba\_zrodel\_swiatla\ ) \tag{2.1}$$

co z kolei w wielu przypadkach jest nie do zaakceptowania (z 2.1 wynika, iż zwiększanie liczby obiektów zmniejsza liczbę źródeł światła, których można użyć na scenie).
Jeśli jednak proces obliczania oświetlania przesuniemy do oddzielnego etapu następującego po wyliczeniu tego, które obiekty są aktualnie widoczne, otrzymamy złożoność

$$O(\ liczba\_pikseli\ *\ liczba\_zrodel\_swiatla\ ) \tag{2.2}$$

Jak z kolei widać z 2.2, podejście to nie wprowadza już zależności między liczbą obiektów a liczbą źródeł światła. Pozwala to uzyskać o wiele lepszą wydajność w przypadku scen z wieloma złożonymi obiektami oraz złożonym oświetleniem. Możliwe jest również wprowadzenie wielu uprawnień, jak na przykład podział obrazu na wiele "kafelków"(ang. *tile*), z których każda może zostać obliczona przez wątki na karcie graficznej - technika ta nosi nazwę *tiled deferred rendering* i jest powszechnie stosowana w popularnych silnikach graficznych.
Uzyskanie tego efektu wymaga utworzenia kilku tekstur pośredniczących, które łącznie noszą nazwę *G-Buffera*. Jest on wykorzystywany do przetrzymywania efektów pośrednich procesu renderingu oraz wyliczenia obrazu końcowego. Format G-Buffera zmienia się w zależności od zastosowania oraz algorytmu obliczania oświetlenia i osoby odpowiadającej za jego implementację, jednak najczęściej wykorzystywane są tekstury:

- wektorów normalnych,
- koloru/albedo,
- głębokości.

Pomimo wielu zalet, deferred shading ma jednak swoje wady. Obliczanie oświetlenia z gotowego obrazu uniemożliwia łatwe obliczenie kolorów w przypadku przezroczystych obiektów, zaś wyświetlanie obrazu końcowego w postaci tekstury wymaga wykorzystania bardziej skomplikowanych algorytmów obliczania antyaliasingu, co z kolei obciąża kartę graficzną. Mimo to deferred shading, w zmodyfikowanych odmianach, jest powszechnie wykorzystywany w wielu popularnych grach komputerowych oraz innych aplikacjach renderujących obraz w czasie rzeczywistym.

2.3 Podpowierzchniowe Rozproszenie Światła/Rozpraszanie Podpowierzchniowe (SubSurface Scattering)

Kolory obserwowane na co dzień są efektem odbicia fotonów od powierzchni obserwowanego obiektu (oraz innych obiektów, co skutkuje efektem tak zwanego "krwawienia kolorów"(ang. *color bleeding*, czyli zabarwienia obiektu kolorem światła odbitego). Jeśli jednak światło zamiast odbić się bezpośrednio dostanie się pod powierzchnię obiektu, odbije kilka razy wewnątrz i wyjdzie w innym punkcie, to zobaczymy, iż obiekt ten jest półprzezroczysty - światło dociera jedynie na pewną głębokość określaną mianem promienia rozproszenia światła (ang. *scattering radius*), gdzie zostaje zabarwione na nowy kolor. Przykładem materiałów zachowujących się w ten sposób mogą być wosk, mleko, marmur czy skóra.

Efekt ten jest trudny do obliczenia w czasie rzeczywistym, w związku z czym w większości przypadków stosuje się jedynie pewne przybliżenia. Przykładem algorytmu dającego przybliżone lecz skuteczne rozwiązanie jest wykorzystanie map głębokości. Opiera się on na zasadzie podobnej do obliczania map cieni (ang. *shadow maps*), jednak w tym wypadku z punktu widzenia źródła światła zapisuje się bardziej skomplikowaną informację - drogę, jaką musi pokonać światło przechodząc przez obiekt, a więc odległość pomiędzy dwoma punktami leżącymi na jego powierzchni z obu stron obiektu. Podczas kolejnego przejścia procesu renderingu można wykorzystać tę informację do oszacowania koloru wynikowego.

2.4 Głębia Ostrości (Depth of Field)

Ostatnim teoretycznym zagadnieniem objaśnianym w tym rozdziale jest zjawisko głębi ostrości, znane powszechnie między innymi z fotografii. Jak opisano w [JD07], polega ono na skupieniu ostrości na pewnym obiekcie lub grupie obiektów w zależności od ogniskowej soczewki, przesłony oraz tak zwanego "krążka rozmycia"(ang. *circle of confusion*; jest to parametr wpływający na ogólną ostrość/rozmycie obrazu).
Ponieważ istnieje wiele algorytmów obliczania głębi ostrości, a sam proces wyznaczania potrzebnych wartości jest skomplikowany, nie będzie on opisywany w niniejszej pracy.

Rozdział 3

Przegląd narzędzi

Jedną z najważniejszych kwestii podczas tworzenia gry komputerowej jest prawidłowy dobór narzędzi, ponieważ decyduje to nie tylko o komforcie pracy, ale także o jakości końcowego produktu. Podczas tego procesu szczególny nacisk powinien zostać położony na doborze silnika graficznego, między innymi ze względu na dużą dywersyfikację narzędzi należących do tej kategorii oraz mnogość funkcjonalności przez nie oferowanych.

Ze względu na specyfikę tego rozdziału, w kolejnych punktach opisane zostaną przetestowane rozwiązania wraz z krótkim podsumowaniem w formie listy jego zalet i wad. Warto w tym miejscu również wspomnieć, iż wszystkie analizowane narzędzia oferują możliwość wykorzystania DirectX 11 API, choć niekoniecznie bezpośrednio.

3.1 Własny silnik graficzny w języku C++

Proces doboru narzędzi rozpoczęto od zaprojektowania i stworzenia własnego silnika graficznego w języku C++ w wersji 11 z wykorzystaniem środowiska Microsoft Visual Studio 2012 na platformę Windows. Pozwoliło to nie tylko na praktyczne wykorzystanie umiejętności nabytych podczas uczestnictwa w zajęciach z Inżynierii Oprogramowania, ale także poznać w praktyce wykorzystanie Microsoft DirectX API w wersji 11 na najniższym dostępnym poziomie. Tworzenie tego rodzaju oprogramowania wymaga jednak nie tylko odpowiedniej ilości czasu, ale także dobrego zaprojektowania interakcji między klasami oraz zrozumienia wielu zagadnień z zakresu grafiki komputerowej, w większości takich, które wykraczają poza program przedmiotu Grafika Komputerowa i Wizualizacja. Pomimo iż stworzenie własnego silnika oferuje największą swobodę w tworzeniu gry, to podejście okazało się zbyt pracochłonne, aby można je było wykorzystać do realizacji zadanego tematu pracy inżynierskiej w dostępnym czasie. Na czasochłonność miała wpływ m.in. złożoność tworzonego oprogramowania oraz konieczność wdrażania się członków zespołu w stworzony przez siebie nawzajem kod.
Zalety:

- większe możliwości w zakresie wykorzystania DirectX API,
- lepsza znajomość możliwości oferowanych przez oprogramowanie,
- brak kosztów.

Wady:

- czasochłonność,
- wysoki próg wejścia (znajomość m.in. C++ oraz DirectX API),
- konieczność wytworzenia edytora poziomów i dodatnia odpowiednich funkcjonalności.

3.2 Unreal Engine 4

Unreal Engine 4 (UE4) jest jednym z najpopularniejszych silników graficznych dostępnych na rynku, co jest złożeniem wielu czynników. Pierwszym z nich jest z pewnością niska cena płatnej licencji (subskrypcja miesięczna to koszt 19$) oraz darmowy dostęp dla studentów zarówno w ramach licencji edukacyjnej (należy w tym wypadku zgłosić chęć wydania licencji w ramach

przedmiotu prowadzonego na uczelni) jak i w ramach GitHub Developers Pack (należy jedynie potwierdzić studencki adres e-mail oraz wykorzystać [ang. *redeem*] licencję dostępną na odpowiedniej podstronie serwisu GitHub w formie kodu [ang. *serial code*]). Drugim jest jakość generowanych (renderowanych) obrazów – wykorzystanie algorytmu Light Propagation Volumes (algorytm rozwiązywania zagadnienia globalnego oświetlenia w czasie rzeczywistym) pozwala uzyskać niemal fotorealistyczną grafikę, jednak kosztem wysokich wymagań sprzętowych. Unreal Engine, zarówno w wersji 3 jak i 4, oferuje dostęp do kodu źródłowego w języku C++, co - po poznaniu API udostępnianego przez twórców - pozwala szybko i wygodnie rozwijać logikę gry. Umiejętność programowania nie jest jednak wymagana do tego ze względu na obecność mechanizmu blueprintów, który pozwala tworzyć kod z wykorzystaniem bloków oferowanych bezpośrednio w silniku graficznym. Ze względu na wysokie wymagania sprzętowe (brak możliwości uruchomienia na komputerach laboratoryjnych oraz komputerach 75% zespołu) ostatecznie odrzucono to rozwiązanie.

Zalety:

- tania (darmowa) licencja,
- wygodny edytor i dostęp do API w języku C++,
- niemal fotorealistyczna grafika.

Wady:

- konieczność poznania API,
- wysokie wymagania sprzętowe.

3.3 Unity

Unity Engine to obecnie najpopularniejszy silnik graficzny wśród twórców gier niezależnych. Poza płatną licencją oferuje także licencję darmową z okrojoną listą funkcjonalności, które jednak nadal pozwalają na tworzenie dość zaawansowanych gier (wycięte funkcjonalności związane są głównie z jakością grafiki). Ma on także niski próg wejścia ze względu na zastosowanie w skryptach języka C# lub JavaScript oraz oferuje 30-dniową wersję próbną. Odrzucono go głównie ze względu na wysoką cenę wersji Pro oraz brak pewnych istotnych funkcjonalności (brak miękkich cieni, brak cieni dla źródeł światła innych niż punktowe itp.)

Zalety:

- darmowa (choć okrojona) wersja,
- niski próg wejścia,
- 30-dniowa wersja próbna.

Wady:

- gorsza wydajność w stosunku do innych silników (języki C# i JavaScript oraz wykorzystanie dynamicznych komponentów w trakcie działania powodują powstanie dodatkowego narzutu czasowego),
- wysoka cena licencji Pro (1500$),
- brak pewnych istotnych dla jakości funkcjonalności.

3.4 OGRE

Object-Oriented Graphics Rendering Engine to elastyczny silnik graficzny o otwartym kodzie źródłowym napisany w języku C++. Pozwala na korzystanie z dwóch API: OpenGL oraz Microsoft DirectX, w tym DirectX 11, przez co został wybrany jako alternatywa dla DirectX 11. Niestety nie jest on rozwijany na bieżąco, przez co wsparcie dla DirectX 11 jest jedynie częściowe i w większości wypadków iluzoryczne, ponieważ próba stworzenia urządzenia kończy się rzuceniem wyjątku wewnątrz biblioteki. Nie posiada on także edytora, co znacząco utrudnia tworzenie gier i ogranicza

jego zastosowania do tworzenia dem technologicznych i testowania nowych technik usprawniających rendering. Ze względu na wspomniane problemy z obsługą DirectX 11 został on odrzucony po licznych próbach uruchomienia własnej aplikacji.

Zalety:

- bezpośredni dostęp do DirectX API (otwarty kod źródłowy silnika),
- (teoretyczne) wsparcie dla DirectX 11.

Wady:

- brak edytora,
- problemy ze wsparciem dla DirectX 11,
- kolejne wersje dystrybuowane są wyjątkowo rzadko (ostatnia wersja sprzed dwóch lat).

3.5 CryEngine 3 SDK

3.6 Unreal Engine 3

3.7 Unreal Development Kit (UDK)

3.8 Autodesk Maya

3.9 Autodesk 3D Studio Max

3.10 Blender

3.11 Gimp

3.12 Adobe Flash - Scaleform

3.13 Git

W projektach nad którymi pracuje kilka osób niemożliwa jest praca bez systemu kontroli wersji. Początkowo, ze względu na pracę z kodem źródłowym, skorzystano z systemu Git ze względu na łatwość użycia oraz możliwość utworzenia prywatnego repozytorium na koncie studenckim w serwisie GitHub. Niestety ze względu na przejście na Unreal Development Kit konieczne było skorzystanie z narzędzia Perforce.

Zalety:

- łatwość obsługi,
- dobra obsługa plików tekstowych,
- prywatne repozytorium w serwisie GitHub dla studentów.

Wady:

- problemy z obsługą dużych plików binarnych,
- brak integracji z Unreal Development Kit.

3.14 Perforce

Ostatnim opisywanym narzędziem jest system kontroli wersji Perforce, który dobrze obsługuje duże pliki binarne, dzięki czemu jest popularny w firmach zajmujących się wytwarzaniem oprogramowania (w tym gier). Pozwala on również – w przeciwieństwie do gita – na integrację z edytorem Unreal Development Kit, co znacząco przyspiesza i ułatwia pracę. Niestety ma on również swoje wady, które ujawniają się dopiero z upływem czasu, na przykład problemy z obsługą usuniętych plików (nadal uwzględniane są w listach zmian) czy nie zapisywanie zmian wprowadzonych w plikach na wspomniane listy.
Zalety:

- dobra obsługa plików binarnych,
- integracja z Unreal Development Kit.

Wady:

- niespodziewane problemy wynikające z niewłaściwej obsługi zmian,
- konieczność zainstalowania dedykowanego oprogramowania serwerowego i utrzymania serwera.

Rozdział 4

Praca własna

Proces twórczy został podzielony na dwa główne etapy: projektowanie i implementację. Ponieważ tworzenie gry wymaga przygotowania oraz doboru odpowiedniego środowiska, etapy te zostały poprzedzone analizą problemu oraz burzą mózgów na której zarysowały się wstępne wymagania funkcjonalne. Zdefiniowane zostały również główne wymagania pozafunkcjonalne, takie jak docelowa platforma obsługująca grę. W celu zapewnienia spójnej wizji gry wśród członków zespołu, postanowiono skonstruować dokument, zawierający opis wszystkich decyzji podjętych podczas tworzenia projektu oraz wyjaśnienia dotyczące wszystkich wymagań funkcjonalnych oraz pozafunkcjonalnych – Game Design Document (GDD), stanowiący jednocześnie załącznik 1 do niniejszej pracy. Pomysły zebrane podczas burzy mózgów zostały poddane wnikliwej analizie, co pozwoliło na stworzenie spójnej wersji projektu.

4.1 Projektowanie

Projektowanie jest bardzo ważnym etapem prac nad każdym projektem informatycznym. Pozwala uspójnić wizję gry w zespole oraz zdefiniować zadania, które będą wykonywane podczas implementacji. Dlatego dobrze, gdy na tym etapie pracy w proces twórczy zaangażowany jest każdy członek zespołu.
Istnieje wiele sposobów na projektowanie gry komputerowej. Podejściem, które ułatwia stworzenie gry, która bawi, jest projektowanie zorientowane na gracza. Zgodnie z charakterystyką przedstawioną w [Ada11], stosując tę metodę, twórcy powinni wyobrazić sobie typowego gracza, a więc osobę, dla której ta gra jest tworzona. Aby użytkownik był zadowolony, należy zapewnić mu rozrywkę – jest to podstawowa funkcja gry. W tym celu należy utożsamić się z graczem. Pozwoli to wybrać funkcje, które uczynią grę atrakcyjną.
Istotnym elementem tworzenia projektu gry jest wnikliwa analiza wymagań funkcjonalnych oraz uszeregowanie ich według wagi. Każdy z członków zespołu posiadał własną wizję gry. Połączenie ich wszystkich zaowocowało spójnym projektem, opisanym w załączniku 1 – GDD.
Aby zapewnić, że projekt będzie spójny, konieczne jest stworzenie dokumentów projektowych. Powstrzymuje to programistów przed puszczaniem wodzy fantazji i tworzeniem funkcjonalności niezgodnych z projektem.
Na tym etapie przydatna okazała się nie tylko teoretyczna wiedza na temat tworzenia zaawansowanych projektów informatycznych, ale przede wszystkim doświadczenia innych twórców gier, które zespół poznał przy okazji udziału w konferencjach takich jak Zjazd Twórców Gier (ZTG) czy World of Gamedev Knowledge (WGK). Zdobyto wiedzę potrzebną między innymi do stworzenia poziomów ciekawych dla graczy, wyboru funkcjonalności nie wymagających skomplikowanych i często zawiłych implementacyjnie elementów (co często powoduje błędy i trudności w dalszym rozwoju projektu), jednocześnie będących skomplikowanymi z punktu widzenia gracza.
Dobrą praktyką przy tworzeniu gry jest również częste testowanie graficznego interfejsu użytkownika, jego czytelności i łatwości użycia kluczowych funkcji w ferworze walki. Pozwala to zaprojektować interfejs przyciągający wzrok oraz funkcjonalny. Projektując HUD [ang. *Head-Up Display*], a więc wszystkie istotne w trakcie rozgrywki wskaźniki, mapkę, poziom życia; postanowiono rozmieścić interesujące dla graczy informacje analogicznie do popularnych gier z gatunku strzelanek. Jest to dla nich atrakcyjne, ponieważ nie muszą zmieniać swoich przyzwyczajeń by sprawdzić poziom życia.

Jednak projektowanie to nie tylko uspójnienie rozgrywki, ale również specyfikacja dotycząca środowiska, w którym gra powstanie. W tym celu należało wybrać odpowiednie narzędzia, co zostało opisane w Rozdziale 3: Przegląd narzędzi.

4.1.1 Projekt silnika graficznego

W ramach pracy jako pierwsze przetestowano podejście z tworzeniem własnego silnika graficznego w języku C++. Jak zostało wspomniane w poprzednim rozdziale, tworzenie tego rodzaju oprogramowania wymaga pewnej określonej wiedzy dotyczącej zarówno architektury silników graficznych i inżynierii oprogramowania, jak i grafiki komputerowej oraz Microsoft DirectX API. Jako że wiedza teoretyczna wraz z podstawowymi pojęciami zostały zarysowane w rozdziale 2., tutaj opisane zostaną kwestie dotyczące architektury silnika oraz usprawnień, które mogłyby zostać wprowadzone w przyszłości.
W silniku graficznym można zasadniczo wydzielić 3 podstawowe składowe:

- renderer - przeprowadza proces renderingu, tj. generowania grafiki,
- menedżer obiektów - zawiera listę obiektów i/lub listę ich grup,
- menedżer zasobów - odpowiada za alokację i zwalnianie zasobów takich jak tekstury, dźwięki itp.

W ramach menedżera obiektów zastosowano wzorzec kompozyt ze względu na jego idealne dopasowanie do problemu. Jak widać każda z tych składowych może stanowić pewną jednostkę (ang. *entity*) budującą system - w tym wypadku system graficzny (wyróżnia się również na przykład systemy fizyki). Jednostki można również podzielić na mniejsze komponenty, które można dynamicznie dodawać i usuwać w trakcie działania - podejście takie zgodne jest wzorcem *Entity-Component-System*. Jak zauważono podczas testowania opisywanego oprogramowania, poza wygodą oraz nowymi możliwościami rozwoju wykorzystanie dynamicznych komponentów wprowadza także narzut czasowy ze względu na wykorzystanie metod wirtualnych oraz nieoptymalne wykorzystanie pamięci podręcznej procesora. W tym wypadku możliwym usprawnieniem mogłoby być budowanie aplikacji z komponentów w edytorze, a następnie generowanie śtałych"klas i kompilowanie zmodyfikowanego w ten sposób kodu do pliku wykonywalnego, który mógłby być dystrybuowany jako gotowa aplikacja.
Nieoptymalne wykorzystanie pamięci podręcznej procesora dotyczy nie tylko wykorzystania komponentów, gdyż jest wąskim gardłem większości zarówno amatorskich jak i profesjonalnych silników graficznych. W celu zniwelowania tego problemu można utworzyć pewien stały obszar pamięci (w języku C++ w wersji 11. w tym celu wykorzystać można operator *placement new*) i utworzyć pulę obiektów, w ramach której mogłyby one być używane ponownie bez konieczności zwalniania zajmowanej przez nie pamięci.
Ostatnim usprawnieniem, które można byłoby wprowadzić celem zwiększenia efektywności procesu renderingu jest wielowątkowość, która jednak wymagałaby wprowadzenia synchronizacji między wątkami aplikacji (np. zamki) oraz użycia metod DirectX obsługujących wielowątkowość (np. operować na opóźnionym kontekście urządzenia). Jej użycie utrudniałoby też skuteczną naprawę błędów ze względu na brak możliwości odpluskwiania kodu w środowisku Microsoft Visual Studio, w którym tworzone było to oprogramowanie.
Dla silnika została napisana i wygenerowana z wykorzystaniem aplikacji Doxygen dokumentacja, jednak ze względu na swoją objętość (150 stron) nie została tutaj załączona.

4.1.2 Modele 3D i animacje

4.1.3 Projekt poziomu

Do stworzenia poziomu wykorzystano środowisko Unreal Development Kit (w skrócie UDK), do którego zaimportowano wcześniej stworzone modele. Niewątpliwą zaletą UDK jest możliwość szybkiego stworzenia poziomu poprzez przenoszenie obiektów z panelu edytora prosto na tworzoną mapę. A także oprogramowanie wydarzeń występujących na niej zgodnie z zasadami programowania wizualnego, za pomocą komponentów, układanych obok siebie i łączonych liniami z wykorzystaniem interfejsu graficznego.
W tworzeniu poziomu istotne jest jego umiejscowienie w realiach rozgrywki. Poziom powinien być powiązany z fabułą i światem przedstawionym. W związku z tym istotne było określenie

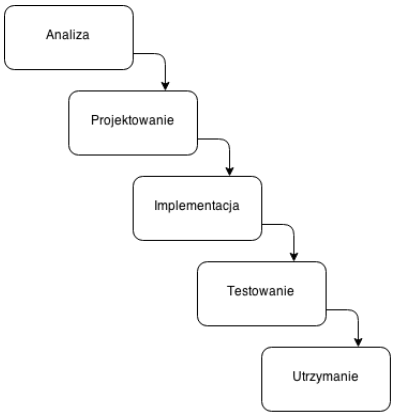
czasu i miejsca akcji. Ponieważ podczas burzy mózgów ustaliliśmy, że fabuła gry będzie polegać na potyczkach żelków to istotne stało się dopasowanie miejsca aby zwiększyć immersję a co za tym idzie przyjemność z gry. W związku z tym, że żelki są jedzeniem a jedzenie jest powiązane z kuchnią, oczywisty stał się wybór tego pomieszczenia jako miejsce starcia. Stworzono, więc wstępny projekt poziomu. Stworzono go w programie Gimp. Jego celem był wstępny zarys, zorientowanie się jak co powinno wyglądać a także przekazanie informacji, jakie tekstury wykonać. Wstępny projekt został stworzony szybko i zawiera jedynie kontury obiektów, gdyż służy głównie do zorientowania się w koniecznych do stworzenia modelach oraz skonsultowania z członkami grupy rozmieszczenia elementów w celu zwiększenia przyjemności z gry, gdyż każdy z nich grał wcześniej w tego typu gry i ma doświadczenie z gry na wielu planszach i może pomóc lepiej zaprojektować poziom. Istotną bowiem sprawą w tworzeniu poziomu jest takie jego zaprojektowanie by był sprawiedliwy i dawał graczom dużo możliwości, różnego poprowadzenia rozgrywki. Dlatego obmyślono stworzenie stołu po którym gracz będzie mógł się wspinać i zaskakiwać przeciwników z wysokości a także porozrzucanie po pomieszczeniu (w tym przypadku kuchni), obiektów służących jako osłony, dzięki czemu łatwiej będzie się schować i zaskoczyć oponenta, tym samym zmniejszając wielkość otwartych przestrzeni, które na podstawie doświadczeń grupy w zbyt dużej liczbie obniżają przyjemność płynącą z grania poprzez zmniejszenie możliwych taktyk.
Następnym krokiem było stworzenie poziomu w edytorze. W tym celu wykorzystano gotowe modele i zaprojektowano mieszkanie 3 pokojowe z korytarzem. Zwiększenie liczby pomieszczeń miało na celu zwiększenie możliwych punktów startu a także oddalenie ich od siebie w celu nie natrafienia od razu na przeciwnika. W innym przypadku mogłoby wyniknąć niezadowolenie z rozgrywki, z powodu różnic komputerów biorących udział w rozgrywce, ładują one poziom w różnym czasie, więc możliwe są drobne różnice czasu. A ponieważ strzelanki to gatunek gier szybkich, nawet niewielka różnica może zadecydować o zwycięstwie w potyczce. Dodatkowo w poziomie uwzględniono możliwość wspinania się (oprócz stołów) na niektóre obiekty, a także wykorzystanie kilku szklanych przeszkód (niezależnie od normalnych przeszkód), przez które widać przeciwnika, a których nie można zniszczyć. Umożliwia to bowiem ukazanie możliwości Directx 11 a także urozmaica rozgrywkę.
Poziom został stworzony z myślą o grze wieloosobowej w trybie każdy na każdego, w związku z tym nie było potrzeby tworzenia specjalnych sektorów dla każdej drużyny, a co za tym idzie jest więcej punktów startowych i w kolejnych rozgrywkach możliwe jest wylosowanie innego, więc konieczne są modyfikacje taktyki i sposobu gry, wynikające z innego otoczenia startowego.

4.2 Implementacja

4.2.1 Wykorzystanie technologii DirectX 11 w UDK

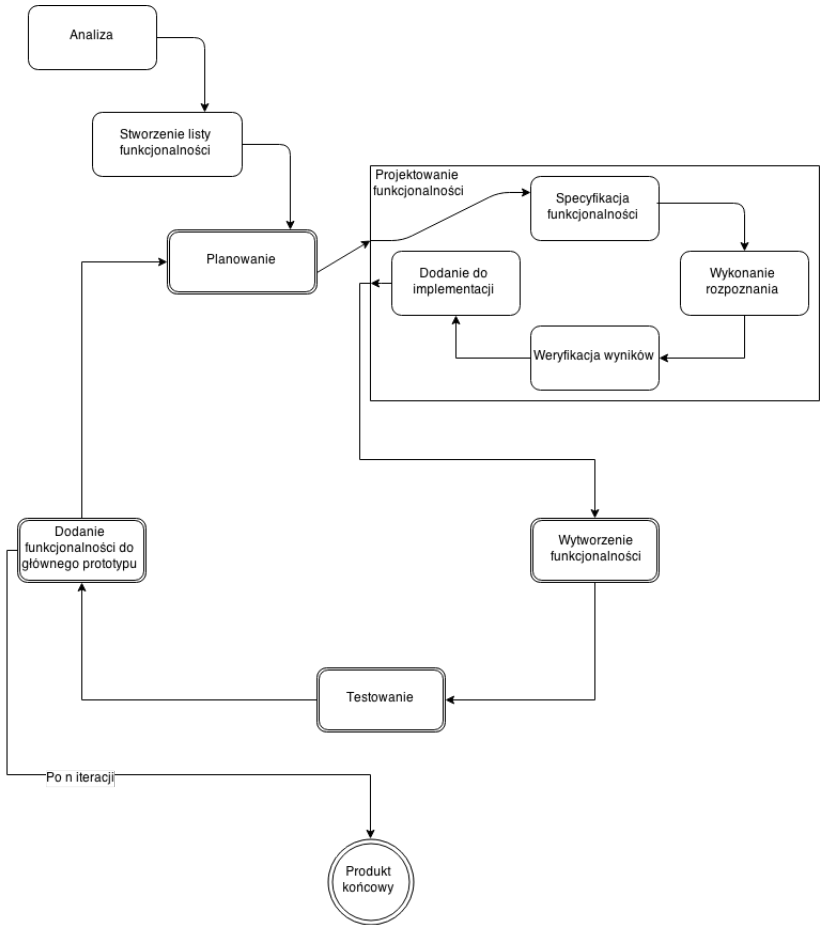
4.2.2 Zarządzanie projektem

W celu sprawnej organizacji pracy w zespole wykorzystano metody zarządzania projektami. Uspójnienie projektu gry podczas fazy projektowania pozwoliło na zdefiniowanie i wyspecyfikowanie zadań, realizowanych podczas implementacji. Znając liczbę zadań, można było podzielić projekt na kolejne przyrosty. Projekt miał być realizowany z wykorzystaniem metod zwinnych. Trudnością w wykorzystaniu typowej zwinnej metody, takiej jak programowanie ekstremalne czy Scrum okazał się charakter projektu, będącego pracą dyplomową. Z tego względu narzucony został ostateczny termin ukończenia produktu. Jest to sprzeczne z manifestem zwinności, dlatego zdecydowano się na inne rozwiązanie.
Początkowo użyto modelu kaskadowego. Jego zaletą jest sekwencyjność, pozwalająca oddzielić procesy analizy problemu, projektowania, implementacji oraz testowania i późniejszego utrzymania projektu, co przedstawiono na Rys. 4.1. Metoda Waterfall, wykorzystująca ten model, nie jest jednak pozbawiona wad. Dotyczą one głównie dużych projektów, a więc nie miały miejsca w przypadku oprogramowania tworzonego w czteroosobowym zespole programistów. Korzystając z tej metody oszczędza się czas na planowaniu. Jak opisano w [Kac14], faza ta zajmuje zaledwie 25% czasu pracy nad projektem. Rezultat końcowy zostaje ustalony jeszcze przed rozpoczęciem implementacji, podobnie jak poszczególne przyrosty implementacji. Każdy przyrost miał trwać 2 tygodnie i zawierał określone zadania. Rezultatem końcowym był produkt posiadający wartość biznesową, w tym przypadku - w pełni działająca gra. Co istotne, wartość biznesową produkt miał zyskać dopiero w przedostatnim przyroście.



Rysunek 4.1: Model kaskadowy

Problemem w wykorzystaniu modelu kaskadowego są błędy, wykryte podczas fazy testowania. Ponieważ w momencie rozpoczęcia testów cały produkt jest już zaimplementowany, poprawka może spowodować powstanie kolejnych błędów. Może jednak się zdarzyć, że błąd powstał na etapie projektowania, w takim przypadku należy powtórzyć tę fazę, jak również całą fazę implementacji. Powoduje to wydłużanie czasu realizacji oraz wzrost kosztów wytworzenia produktu.
W niedługim czasie po zaplanowaniu prac nad projektem okazało się, że metoda ta nie jest wystarczająca, zaczęło się pojawiać opóźnienie w pracach, które mogło spowodować pogorszenie jakości produktu. Zdecydowano się więc skorzystać z metod zwinnych. Metodyki Agile charakteryzują się stałą jakością, a sterowane są zakresem. W omówionej wcześniej metodzie Waterfall zakres był stały, zmieniała się jedynie jakość, a chęć utrzymania wysokiej jakości powodowała opóźnienie względem planu.
Ostatecznie zastosowano metodykę zwinną zbliżoną do metody Feature-Driven Development, wplatając w nią elementy metodologii Scrum. Zadania podzielono na część dotyczącą rozpoznania danego zagadnienia i część implementacyjną.
W części dotyczącej rozpoznania zastosowano cykl PDCA (*Plan – Do – Check – Act*), stworzony przez Williama Edwardsa Deminga. Pierwszy etap cyklu – planowanie – polegało na zapoznaniu się z opisem wybranej funkcjonalności, zawartym w Game Design Document. Następnie, w fazie wykonania, zbierano informacje na temat możliwości implementacji funkcjonalności wykorzystując UDK oraz próbowano stworzyć wybraną funkcję. W trzeciej fazie sprawdzano czy zaimplementowana wersja funkcji odpowiada założeniom projektu. Jeśli wynik był pozytywny, funkcjonalność została wybierana do wykonania w kolejnym przyroście.
Realizowane były tygodniowe sprinty (przyrosty). Każdy sprint rozpoczynał się spotkaniem zespołu, na którym omówiono pozostałe zadania oraz zaplanowano jakie funkcjonalności będą implementowane w kolejnym przyroście. W efekcie udało się zachować jakość wykonania oraz zakończyć prace przed upływem niezmiennego terminu ukończenia, modyfikując nieznacznie zakres dostępnych funkcjonalności. Zespół tworzący grę był wskroś-funkcjonalny, co oznacza, że członkowie zespołu wykonywali zadania dotyczące różnych zagadnień. Począwszy od tworzenia prostych modeli 3D po skryptowanie. Jest to cecha charakterystyczna metody Scrum. Z kolei sposób planowania kolejnych przyrostów, a więc planowanie ze względu na funkcjonalność, zaczerpnięto z metody Feature-Driven Develppment.
Na koniec każdego przyostu otrzymywano prototyp posiadający pewną funkcjonalność. Jeśli funkcjonalność działała prawidłowo podczas testów, dołączano ją do głównego prototypu, który z każdym przyrostem stawał się bardziej funkcjonalny, aż w końcu stał się wersją produkcyjną. Ze względu na wcześniejszy nieudany eksperyment z metodą Waterfall wartość biznesową projekt zyskał dopiero po kliku przyrostach. Oznacza to, że pierwsze efekty nie były satysfakcjonujące, jednak prezentowały postęp prac.
Trzy główne fazy procesu wytwarzania gry, występujące w modelu kaskadowym, a więc: projektowanie, implementacja i testowanie, w metodach zwinnych są wykonywane w każdym sprincie, co znacznie zmniejsza ryzyko powstania krytycznych błędów po zaimplementowaniu całego produktu. W przypadku wykorzystania Feature-Driven Development proces naprawy



Rysunek 4.2: Etapy procesu wytwórczego gry przy wykorzystaniu metody Feature-Driven Development i cyklu PDCA

błędów również obarczony jest mniejszym ryzykiem czasowym. Zwłaszcza, gdy błąd powstał na etapie projektowania danej funkcjonalności. W skrajnym przypadku, wadliwa funkcjonalność nie trafi do głównego produktu.

Rozdział 5

Podsumowanie

Rozdział 6

Literatura

[Ada11] Ernest Adams. *Projektowanie gier. Podstawy. Wydanie II.* Wydawnictwo HELION, ul. Kościuszki 1c, 44-100 Gliwice, 2011.

[JD07] NVIDIA Joe Demers. Depth of field: A survey of techniques. [on-line] `http://http.developer.nvidia.com/GPUGems/gpugems_ch23.html`, 2007.

[Kac14] Krystian Kaczor. *Scrum i nie tylko. Teoria i praktyka w metodach Agile.* Wydawnictwo Naukowe PWN SA, Warszawa, 2014.

[Owe13] Brent Owens. Forward rendering vs. deferred rendering. [on-line] `http://gamedevelopment.tutsplus.com/articles/forward-rendering-vs-deferred-rendering--gamedev-12342`, 2013.

Rozdział 7

Dodatki

© 2015 Krzysztof Marciniak, Piotr Przybysz, Mikołaj Szychowiak, Ryszard Wojtkowiak

Instytut Informatyki, Wydział Informatyki
Politechnika Poznańska

Skład przy użyciu systemu LaTeX.

BibTeX:

```
@mastersthesis{ key,
    author = "Krzysztof Marciniak \and Piotr Przybysz \and Mikołaj Szychowiak \and
Ryszard Wojtkowiak",
    title = "{Projekt i implementacja gry komputerowej z wykorzystaniem technologii
Microsoft DirectX 11}",
    school = "Poznan University of Technology",
    address = "Pozna{\'n}, Poland",
    year = "2015",
}
```