POLITECHNIKA ŚLĄSKA

WYDZIAŁ AUTOMATYKI, ELEKTRONIKI I INFORMATYKI

KIERUNEK INFORMATYKA

Projekt inżynierski

Wysokowydajny silnik modularnych efektów cząsteczkowych

Autor: Leszek Godlewski

Kierujący pracą: dr inż. Agnieszka Szczęsna

Konsultant: mgr inż. Jakub Stępień

Gliwice, styczeń 2013

Spis treści

1. Wstęp....5
1.1. Wprowadzenie do dziedziny problemowej....5
1.2. Cel pracy....6
1.3. Rozwiązania istniejące....7
1.3.1. Modularne i monolityczne systemy cząstek....7
1.3.2. Interpretacja w locie....8
1.3.3. Kompilacja formuł matematycznych....9
1.4. Rozwiązania alternatywne....10
1.4.1. Program GPGPU....10
1.4.2. Symulacja cząstek z uwzględnieniem wykrywania kolizji....11
2. Założenia i wymagania projektu....12
2.1. Punkt wyjścia – projekt zaliczeniowy....12
2.2. Wymagania funkcjonalne i niefunkcjonalne....14
2.2.1. Wymagania funkcjonalne....14
2.2.2. Wymagania niefunkcjonalne....15
2.3. Propozycja modelu....15
2.3.1. Podział na interfejs i zaplecze....16
2.3.2. Zaplecze asemblera x86....16
3. Specyfikacja zewnętrzna....17
3.1. Interfejs programowania....17
3.1.1. Typy proste....17
3.1.2. Identyfikatory i typy wyliczeniowe....18
3.1.3. Emiter....19
3.1.4. Moduły....20
3.1.5. Pozostałe struktury i unie....21
3.1.6. Punkt wejścia i interfejs....24
4. Organizacja wewnętrzna....26
4.1. Schemat architektoniczny....26
4.2. Przegląd kluczowych klas zaplecza x86....26
4.2.1. Klasa X86Generator....26
4.2.2. Klasa X86Launcher....27
4.2.3. Interfejs X86ModuleInterface i klasy generujące kod modułów....27
5. Wybrane zagadnienia implementacji....28
5.1. Uruchamianie procesów potomnych....28
5.2. Preprocesory i translatory kodu....29
6. Przykład wykorzystania....29

6.1. Integracja z aplikacją kliencką....30
6.2. Przykładowa konfiguracja emitera....32
7. Testowanie i eksperymenty....34
7.1. Metodyka testowania....34
7.2. Wyniki eksperymentów....35
7.2.1. Maszyna desktop, tryb x86 (32-bit)....35
7.2.2. Maszyna desktop, tryb x86-64 (64-bit)....36
7.2.3. Maszyna laptop, tryb x86 (32-bit)....37
7.2.4. Wykres....38
7.2.5. Wnioski....38
8. Przebieg i wyniki prac....39
8.1. Oczekiwana a otrzymana wydajność....39
8.2. Napotkane problemy....40
8.2.1. Adresowanie w trybie 64-bit....40
8.2.2. Czerwona strefa....41
8.2.3. Przewidywanie ponownego wykorzystania....42
8.3. Możliwe kierunki rozwoju....43
8.3.1. Optymalizacje....43
8.3.2. Usunięcie zależności od NASM....43
8.3.3. Więcej zapleczy....44
8.3.4. Graficzny edytor konfiguracji emiterów....44
8.3.5. Dodatkowa funkcjonalność....44
9. Podsumowanie....45
Bibliografia....46
Indeks ilustracji....46
Indeks tabel....47
Załącznik 1. Opis zawartości dołączonej płyty CD....48

1. Wstęp

Grafika komputerowa jest jedną z najbardziej satysfakcjonujących dziedzin informatyki: estetyczne efekty można uzyskiwać stosunkowo małym nakładem pracy. Jej zastosowania obejmują wizualizacje artystyczne i użytkowe rzeczywistości realistycznych i fantastycznych. Dzisiaj, w czasach wykonywanych w czasie rzeczywistym technik takich, jak śledzenie promieni i oświetlenie globalne[1], zadomowiła się już na dobre w wielu dyscyplinach: animacji, kinematografii, gier komputerowych czy też wizualizacjach architektonicznych.

Sceny grafiki komputerowej składają się z różnorodnych obiektów, i nie dla wszystkich optymalną reprezentacją jest typowy model będący siatką wielokątów. Obiekty tworzące sceny graficzne można pogrupować w różne klasy w zależności od rozmiaru, charakterystyki ruchu i innych cech. Grupy te różnią metody rysowania, a także obliczenia niezwiązane bezpośrednio z rysowaniem: rozwiązywanie równań kinematyki i dynamiki, wyznaczanie pozy szkieletu itd.

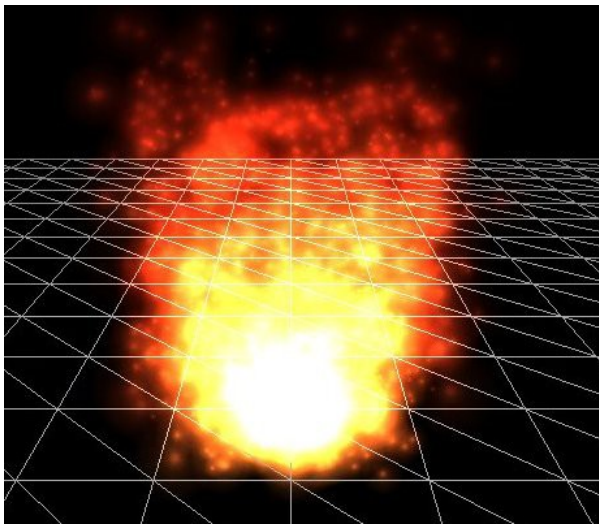
Niniejsza praca stanowi propozycję rozwiązania dla grupy obiektów określanych mianem **systemów cząsteczkowych**.

1.1. Wprowadzenie do dziedziny problemowej

Efekty (systemy) cząsteczkowe to technika grafiki komputerowej, stosowana szeroko do symulacji zjawisk złożonych z wielu mniejszych elementów, jak np. płomieni (cząstek rozżarzonego węgla), obłoków dymu (cząstki pary wodnej i stygnących produktów reakcji spalania), eksplozji czy rozbryzgów wody, a także wszelkiego rodzaju innych efektów specjalnych, często fantastycznych – przykładowo, działania czarów w światach *fantasy*, czy też broni energetycznej w pracach typu *science fiction* (patrz Ilustracja 1).

1 Śledzenie promieni (ang. *ray tracing*), oświetlenie globalne (ang. *global illumination*) – techniki grafiki komputerowej charakteryzujące się wiernym odzwierciedleniem zjawisk zachodzących w rzeczywistości, a co za tym idzie – wysoką złożonością obliczeniową.

Cząstki posiadają pewne właściwości wizualne (kolor, rozmiar) oraz fizyczne (położenie, rozmiar, prędkość, przyspieszenie). Proces tworzenia cząstek składających się na taki efekt nazywamy **emisją**, a abstrakcyjne obiekty (struktury danych), które te cząstki tworzą – **emiterami**. Cząstka po wyemitowaniu **istnieje** (jest widoczna, a jej właściwości podlegają przetwarzaniu) przez określony czas – zwykle stosunkowo krótki, nieprzekraczający kilku sekund – a następnie jest usuwana. Można więc wyodrębnić dwie fazy bytu takich cząstek – emisji (tworzenia) i istnienia (przetwarzania). Same obliczenia własności cząstek, bez ich renderowania, określa się mianem **symulacji** [1].

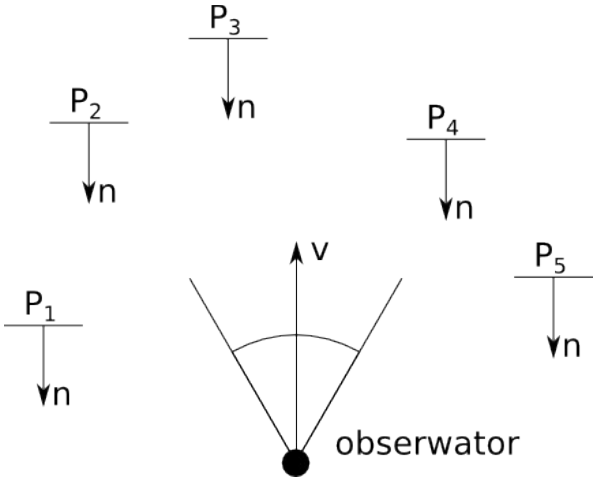


Ilustracja 1: System cząstek symulujący płomień ognia; Źródło: Wikipedia (http://en.wikipedia.org/wiki/File:Particle_sys_fire.jpg)

Cząstki w czasie swojego istnienia są renderowane wraz z resztą sceny graficznej, na jakiej zostały umieszczone. Najpopularniejszą techniką ich rysowania jest tzw. plakatowanie (ang. *billboarding*), czyli renderowanie w postaci prostokątów, których normalna jest zawsze równoległa do osi wzroku obserwatora sceny (patrz Ilustracja 2). Jest to metoda niepozbawiona wad, ale jej zaletą jest stosunkowo niski koszt obliczeniowy.

1.2. Cel pracy

W celu osiągnięcia wiarygodności, a często także zwyczajnie estetycznie zadowalającego efektu, emitery muszą emitować bardzo duże liczby cząstek. Mowa tu o kilkudziesięciu do nawet kilku tysięcy cząstek na pojedynczą instancję efektu, który w całej scenie może



Ilustracja 2: Billboarding; Normalne cząstek P_n równoległe z osią wzroku obserwatora v

występować w liczbie od kilkunastu do kilkuset instancji. Oczywiście, pojawiają się efekty różnego rodzaju. W sumie scena może zawierać setki tysięcy, a nawet przeszło milion cząstek.

W zespołach tworzących gry komputerowe lub efekty specjalne, tworzeniem takich efektów przy użyciu dedykowanego oprogramowania zajmuje się wyznaczony do tej roli **artysta**. Jest to zwykle osoba wykształcona pod kątem sztuk pięknych, a nie nauk ścisłych, toteż przy projektowaniu tworzonych na jej użytek narzędzi należy brać na to poprawkę i czynić je możliwie **prostymi i intuicyjnymi** – nie można przykładowo oczekiwać od niej znajomości matematyki wyższej.

Celem niniejszego projektu inżynierskiego jest **zbudowanie biblioteki narzędziowej** (ang. *middleware*) do wykorzystania w budowie innych aplikacji, zajmującej się symulacją cząstek, łącząc elastyczność wobec wizji artysty z wysoką wydajnością.

1.3. Rozwiązania istniejące

1.3.1. Modularne i monolityczne systemy cząstek

Do problemu konstrukcji efektów cząsteczkowych można podejść w dwojaki sposób, zależnie od pożądanego stopnia elastyczności.

W sytuacji, gdy taka elastyczność nie jest nam potrzebna, lub gdy potrzebujemy jedynie zmieniać niektóre parametry z efektu na efekt, możemy zaprogramować na stałe konkretny algorytm symulacji cząstek (chociażby prostą dynamikę Newtona) i umożliwić zmianę tych parametrów (w przypadku dynamiki Newtona – np. masę). Takie przypadki można sklasyfikować jako monolityczne – nie ma możliwości głębokiej ingerencji w sam proces emisji bądź przetwarzania cząstek.

Wreszcie można pozwolić artyście na dowolne kształtowanie efektów cząsteczkowych poprzez układanie ich z pewnej puli dostępnych przekształceń w całość, podobnie jak ma to miejsce z efektami DSP[2], które transformują sygnał odebrany na wejściu i wystawiają zmieniony sygnał na wyjściu. W podobny sposób można poddawać obróbce cząstkę, zmieniając jej właściwości (patrz Ilustracja 3).

2 *Digital Signal Processing* – ang. cyfrowe przetwarzanie sygnału.

To ostatnie podejście Autor zastosował w swojej pracy inżynierskiej, gdyż zapewnia największą swobodę artyście oraz odciąża programistę, który w przeciwnym razie musiałby udzielać na bieżąco pomocy artyście przy tworzeniu efektów.

Ilustracja 3: Przykład modularnego systemu cząstek; Na czerwonym tle moduły wykonywane w czasie emisji; Na żółtym – w czasie istnienia; Na zielonym – specjalny moduł symulacji, który wykonuje równania kinematyki Newtona

1.3.2. Interpretacja w locie

Tradycyjne podejście do symulacji cząstek polega na iteracyjnym przetwarzaniu wszystkich emiterów i wszystkich cząstek istniejących w scenie, odczytując konfigurację emitera w czasie wykonania.

W fazie emisji program przechodzi listę emiterów, a w każdym z nich – listę modułów, i za pomocą instrukcji warunkowych rozgałęzia się: następuje wybór odpowiedniej dla danego modułu ścieżki kodu. Nowo utworzone cząstki trafiają do pewnego ciągłego bufora, najczęściej luźnej tablicy[3]. Mamy tu do czynienia z problemem podobnym do tablicy mieszającej – możliwością kolizji. Wybór indeksu tablicy, od którego zaczynamy poszukiwanie wolnego miejsca dla nowo stworzonej cząstki ma wpływ na czas wykonania algorytmu. Często stosuje się losowanie początkowego indeksu, aby uzyskać jednorodne prawdopodobieństwo kolizji – pozwala to uniknąć fluktuacji czasu wykonania programu, narastających wraz z czasem działania.

Analogicznie, w fazie przetwarzania program przechodzi wspomnianą luźną tablicę cząstek, i listę modułów emitera w każdej z nich. Przy przejściu do każdego kolejnego modułu następuje rozgałęzienie programu (instrukcje warunkowe).

3 Luźnej, tj. nie wszystkie jej elementy muszą być wykorzystane; niewykorzystane elementy tablicy są odpowiednio oznaczane.

Metoda taka ma szereg wad. Program bardzo często – kilkukrotnie w ramach każdej przetwarzanej cząstki – rozgałęzia się na podstawie niedeterministycznego parametru, jakim jest rodzaj modułu stanowiącego dany element listy w danej cząstce (elemencie tablicy). W efekcie mechanizmy przewidywania rozgałęzień procesorów stają się nieskuteczne: ich decyzje często okazują się błędne i konieczne staje się opróżnienie potoku rozkazów. Notorycznie ma również miejsce zjawisko chybiania pamięci podręcznej – skoki następujące przy przechodzeniu modułów w konsekwencji powodują konieczność odczytu danych spod różnych adresów, co skutkuje ciągłą wymianą linii pamięci podręcznej i kolejnymi opóźnieniami.

Implementację takiej metody Autor również zawarł w pracy celem porównania z zasadniczą metodą zastosowaną w projekcie.

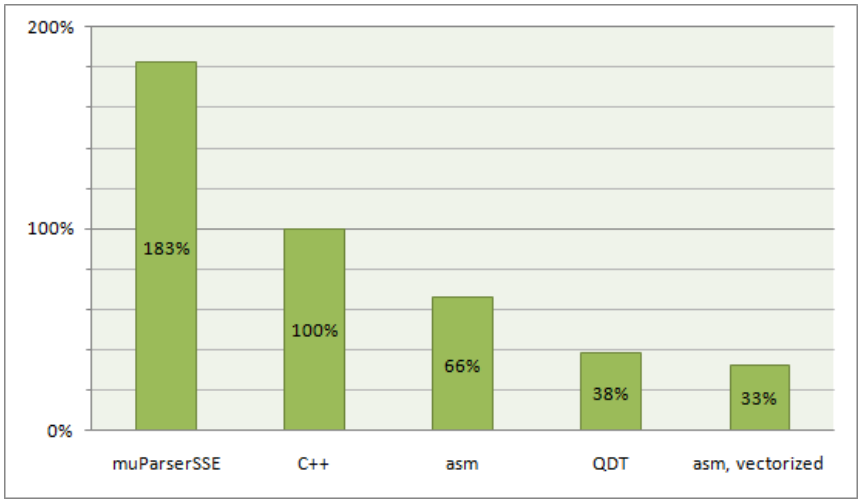
1.3.3. Kompilacja formuł matematycznych

Interesujące podejście do tematu zaproponował Tomasz Dąbrowski w swoim artykule *Quick Data Transformation* [2]: postanowił on bowiem sterować zachowaniem cząstek poprzez formuły matematyczne, które mogą zastępować równania kinematyki Newtona, wyliczając – w zależności od czasu – drogę, prędkość lub przyspieszenie. Celem, jaki chciał osiągnąć, była elastyczność projektowa systemów cząstek przy jednoczesnym zachowaniu wysokiej wydajności rozwiązania.

Dąbrowski pierwotnie próbował korzystać z biblioteki *muParser* [3], która parsuje, interpretuje i wykonuje wyrażenia matematyczne; jej wydajność była jednak niezadowalająca. Zbadał również możliwość wykorzystania projektu pochodnego o nazwie *muParserSSE* [4], który w przeciwieństwie do swojego przodka nie interpretuje wyrażeń w czasie wykonania, a kompiluje je do kodu wykonywalnego z wykorzystaniem instrukcji SSE[4]; jest więc to podejście ideowo podobne do zastosowanego w niniejszej pracy. Niestety, mimo kompilacji do kodu natywnego, wydajność rozwiązania nadal była niewystarczająca. Diagnoza wystawiona przez autora artykułu mówi o braku wektoryzacji (biblioteka wykorzystywała jedynie skalarne warianty instrukcji SSE) oraz strumieniowego przetwarzania danych.

4 *Streaming SIMD Extensions* – ang. rozszerzenia instrukcji pojedynczego strumienia instrukcji przy wielu strumieniach danych; zestaw instrukcji wprowadzony po raz pierwszy w procesorach Intel Pentium III, umożliwiający przede wszystkim szybkie działania na całych 4-elementowych wektorach liczb zmiennoprzecinkowych, używając do tego pojedynczych rozkazów.

Zdecydował więc o stworzeniu własnego narzędzia tego typu i osiągnął bardzo interesujące rezultaty (patrz Ilustracja 4).



Ilustracja 4: Czas wykonania prostego efektu cząsteczkowego względem implementacji w C++ (im mniej tym lepiej); Słupki "asm" i "asm, vectorized" to ręcznie zoptymalizowany dla przypadku testowego kod asemblerowy działający odpowiednio na skalarach i wektorach; QDT to omawiane w artykule autorskie rozwiązanie Tomasza Dąbrowskiego [2]

Mimo, że na potrzeby Dąbrowskiego jest to rozwiązanie wystarczające, to nie spełnia ono wymogów, które Autor postawił przed swoim projektem: jest ono nieprzystępne dla artysty, gdyż wymaga od niego układania formuł matematycznych. Nie przewiduje ono również manipulacji własnościami wizualnymi cząstek, czyli np. kolorem.

1.4. Rozwiązania alternatywne

1.4.1. Program GPGPU[5]

Wraz ze wzrostem mocy obliczeniowych procesorów graficznych oraz popularności technologii takich, jak CUDA i OpenCL, pojawiły się implementacje efektów cząsteczkowych w formie programów typu GPGPU. Wiele takich rozwiązań

5 *General Purpose computing on Graphics Processing Units* – ang. obliczenia ogólnego przeznaczenia z wykorzystaniem procesorów grafiki.

można zaobserwować na tzw. demoscenie[6] w multimedialnych prezentacjach (demach) przygotowywanych na platformę PC.

Z racji natury programów GPGPU, przy projektowaniu silnika efektów cząstek wykorzystującego taki procesor konieczne jest przygotowanie danych cząstek w odpowiedniej formie, czyli macierzy liczb. Dlatego też mimo, iż obliczenia strumieniowe na GPGPU są dziedziną bardzo przyszłościową i drzemie w nich ogromny potencjał wydajnościowy, Autor postanowił nie podejmować się badania tematu od tej strony, gdyż wymagałoby to gruntownej rewizji większości założeń projektowych.

1.4.2. Symulacja cząstek z uwzględnieniem wykrywania kolizji

Należy mieć na uwadze, że wszystkie omawiane powyżej rozwiązania traktują cząstki jako obiekty istniejące w pewnej izolacji od reszty sceny, w której się znajdują: nie przewidują one wykrywania kolizji cząstek z pozostałymi jej elementami. Jest tak z prozaicznych, wydajnościowych powodów – wykrywanie kolizji nie jest tanią operacją, gdyż wymaga zwykle przejścia przynajmniej fragmentu grafu sceny[7] oraz wykonania testów kolizji między cząstką a innymi obiektami. Dla tak licznych bytów, jak cząstki, taki dodatkowy nakład pracy mógłby w znaczącym stopniu wydłużyć czas renderowania sceny.

Istnieją jednakże platformy programistyczne do obliczeń fizycznych takie, jak *Nvidia APEX PhysX*, które oferują taką funkcjonalność. Z racji znacznego obciążenia obliczeniowego zastosowanie poza aplikacjami-demonstratorami technologii, czyli np. w grach komputerowych, ograniczone jest do kilku-kilkunastu kluczowych

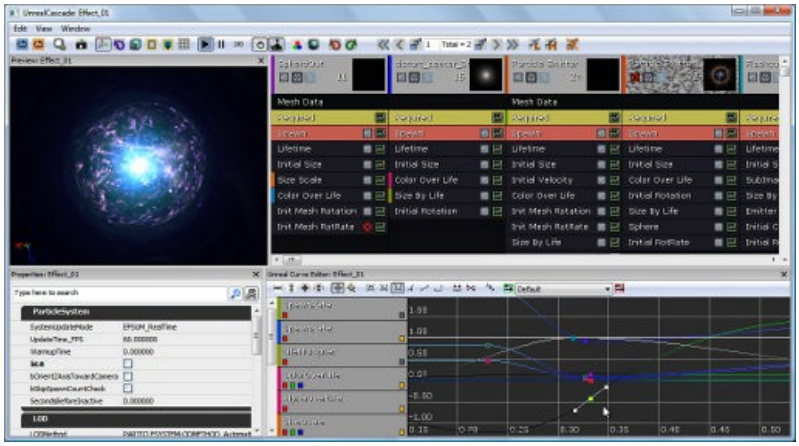
6 Demoscena to społeczność pasjonatów sztuki tworzonej z użyciem komputerów. Przybiera ona głównie formę audiowizualnych prezentacji, zwanych demami, które służą popisom umiejętności programistycznych, graficznych oraz muzycznych poszczególnych członków zespołów (określanych mianem grup demoscenowych). Dema znane są z tego, że wykorzystują moc obliczeniową platform sprzętowych – także historycznych, jak np. komputerów 8-bitowych – do granic możliwości.

7 Graf sceny – struktura danych, najczęściej drzewo binarne, czwórkowe lub ósemkowe. Szeroko stosowany w grafice komputerowej, opisuje hierarchię obiektów istniejących w przedstawianej scenie. Wykorzystuje się go do składania przekształceń algebraicznych, szybkiego ograniczania zbioru obiektów znajdujących się w polu widzenia obserwatora, a także zbiorów potencjalnie kolidujących ze sobą obiektów przy wykrywaniu kolizji.

efektów. Do tego jest ono najczęściej opcjonalne – mimo, iż technicznie możliwe jest uruchamianie takiej symulacji na głównym CPU komputera, to jest on zwykle już aż nazbyt obciążony innymi obliczeniami; a więc wykrywanie kolizji cząstek uaktywnia się wtedy, gdy w systemie dostępna jest jednostka GPGPU wspierająca architekturę CUDA [5].

2. Założenia i wymagania projektu

2.1. Punkt wyjścia – projekt zaliczeniowy



Ilustracja 5: Unreal Cascade podczas edycji efektu cząsteczkowego; emitery w prawej górnej części okna są reprezentowane przez łańcuchy modyfikatorów; w prawej dolnej części ekranu znajduje się edytor wykresów, służący do kontrolowania zmian wartości w czasie

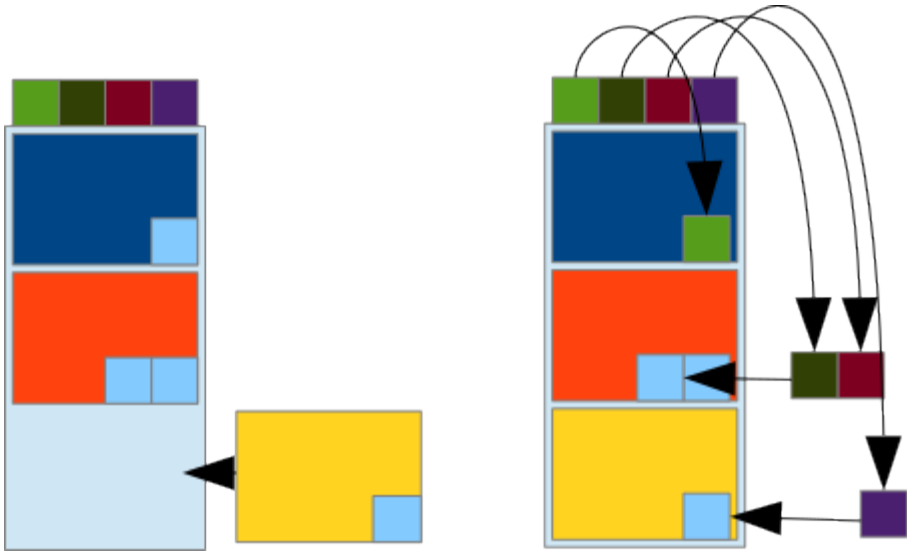
Pierwszą implementację podobnego silnika Autor stworzył w ramach projektu przedmiotowego z Języków Asemblerowych na semestrze V pod opieką dr inż. Piotra Czekalskiego z Zakładu Mikroinformatyki i Teorii Automatów Cyfrowych. Jej treść jest w całości dostępna w internecie [6]. Biblioteka stworzona w ramach projektu przyjęła nazwę *Particlasm*. Założenia, które zostały wówczas przyjęte brzmiały następująco (poza wykorzystaniem asemblera procesorów x86):

- silnik (biblioteka) produkująca dane gotowe do wyświetlenia z użyciem OpenGL[8] bez dodatkowego przetwarzania,
- szerokie wykorzystanie instrukcji wektorowych SSE,
- forma dynamicznie łączonej biblioteki systemu Linux.

8 OpenGL jest jednym z dwóch dominujących API (interfejsów programowania) grafiki komputerowej.

Architektura tej biblioteki była inspirowana narzędziem *Unreal Cascade*, które jest częścią darmowego pakietu do tworzenia gier komputerowych – *Unreal Development Kit* (patrz: Ilustracja 5). Zbudowana jest wokół koncepcji **modułów**, czyli modyfikatorów własności cząstki układanych w łańcuchy i **rozkładów**, czyli struktur danych opisujących w ogólny sposób zmiany wartości (skalarów, wektorów, kolorów) w czasie. Pojedynczy łańcuch **modułów** tworzy **emiter**. Moduły mogą wchodzić w interakcję z cząstkami na dwa sposoby: w fazie **emisji**, lub też podczas jej **istnienia**. Moduły wykorzystują **rozkłady** do wyznaczania konkretnych wartości parametrów. Rozkłady mogą przyjmować rozmaite formy – od wartości stałej, poprzez losowy równomierny rozkład, po interpolację wzdłuż krzywej, a także sterowanie wartością z zewnątrz biblioteki.

W formie, jaką Autor złożył jako projekt zaliczeniowy z JA, biblioteka była w całości asemblowana do kodu maszynowego. Rozkazy procesora w kodzie modułów i rozkładów, których argumentami były adresy w pamięci, zawierały puste adresy bezwzględne. Po odczytaniu konfiguracji emitera, biblioteka „wycinała” z własnego kodu maszynowego fragmenty odpowiadające poszczególnym modułom i rozkładom, a następnie „wklejała” te fragmenty do wcześniej specjalnie do tego przygotowanego bufora. Następnie, w procesie analogicznym do uzupełniania relokacji w pliku wykonywalnym przez program łączący, owe puste adresy były zastępowane właściwymi. Obrazowo algorytm przedstawia Ilustracja 6.



Ilustracja 6: Schemat konstrukcji kodu emitera w pierwotnej wersji biblioteki; w pierwszym kroku (strona lewa) w buforze umieszczane są dane (kolorowe kwadraty na górze schematu) i moduły z pustymi adresami; w kroku następnym (strona prawa) adresy w modułach są wypełniane

Projekt został ukończony, a rozwiązanie uzyskało średni przyrost wydajności 358% wobec analogicznej implementacji w języku C++[9]. Nie było ono jednak pozbawione wad; za największe Autor uznał skrajnie trudny w utrzymaniu kod oraz wsparcie ograniczone tylko do 32-bitowych procesorów z rodziny x86. Stąd też pomysł powtórnego rozpatrzenia tematu w ramach pracy inżynierskiej.

2.2. Wymagania funkcjonalne i niefunkcjonalne

Przystępując do pracy inżynierskiej Autor początkowo zamierzał zwyczajnie rozwinąć wcześniejszy projekt i usunąć jego wady. Napotkał jednak pewne problemy (szerzej opisane w rozdziale *Napotkane problemy*, s. 40), przez które postanowił zacząć praktycznie od nowa.

Podejście drugie do tematu zaczęto od rewizji założeń projektu i ich redukcji do najbardziej podstawowych wymogów funkcjonalnych i niefunkcjonalnych. Wcześniej rozwijany kod został wyodrębniony do osobnej gałęzi w repozytorium, biblioteka uzyskała numer wersji 2, a wymagania przyjęła poniższą formę

2.2.1. Wymagania funkcjonalne

- FREQ 1: Silnik powinien potrafić odczytać z poprawnie przygotowanej konfiguracji emitera listę modułów oraz ich rozkłady.
- FREQ 2: Silnik powinien wyodrębnić z zadanej konfiguracji emitera blok danych.
- FREQ 3: Silnik powinien przetworzyć zadaną konfigurację emitera na blok natywnego kodu wykonywalnego procesu emisji cząstek.
- FREQ 4: Silnik powinien przetworzyć zadaną konfigurację emitera na blok natywnego kodu wykonywalnego procesu przetwarzania cząstek.
- FREQ 5: Silnik powinien wywołać wygenerowany przez siebie blok kodu emisji cząstek w przezroczysty dla oprogramowania-klienta sposób.
- FREQ 6: Silnik powinien wywołać wygenerowany przez siebie blok kodu przetwarzania cząstek w niewidoczny dla oprogramowania-klienta sposób.

9 Pomiar wyników z użyciem dedykowanej aplikacji, której funkcjonalność ogranicza się do samej symulacji cząstek, bez rysowania sceny. Aplikacja wykonuje 1800 iteracji symulacji cząstek, zliczając czas potrzebny na ich wykonanie. Pełne wyniki dostępne w pracy [6].

- FREQ 7: Silnik powinien w wyniku przetwarzania cząstek wyemitować do wskazanego bufora gotowe do renderowania wierzchołki billboardów przedstawiających istniejące cząstki.

2.2.2. Wymagania niefunkcjonalne

- NFREQ 1: Silnik powinien mieć formę dynamicznie dołączanej biblioteki.
- NFREQ 2: Silnik powinien działać na platformach Windows i Linux.
- NFREQ 3: Silnik powinien działać na procesorach architektury x86 (32-bit) i x86-64 (64-bit).
- NFREQ 4: Silnik powinien wykorzystywać wszelkie dostępne mechanizmy akceleracji sprzętowej (w wypadku x86 – instrukcje SSE).
- NFREQ 5: Silnik powinien emitować ciągłe tablice wierzchołków zdatne do renderowania wsadowego jednym wywołaniem funkcji (np. `glDrawElements()` w OpenGL).
- NFREQ 6: Silnik powinien osiągać znacząco wyższą wydajność, niż analogiczna implementacja w języku wysokiego poziomu (>= 358% w warunkach identycznych do eksperymentu z pracy [6]).

2.3. Propozycja modelu

Poza zdefiniowanymi na nowo wymaganiami funkcjonalnymi i niefunkcjonalnymi, w wyniku problemów opisanych w dalszej części pracy Autor postanowił również zmienić ogólną zasadę działania silnika. Mianowicie zamiast „sklejania” kodu emitera z wyciętych z własnego kodu maszynowego kawałków, biblioteka miałaby jedynie generować kod w asemblerze x86, który byłby następnie asemblowany przez właściwego asemblera (w przypadku niniejszego projektu – NASM). Takie rozwiązanie pozwala na napisanie w języku wysokiego poziomu (C++) sporej części kodu biblioteki, która w poprzedniej inkarnacji była z konieczności napisana w asemblerze, co przyczynia się do znacznej poprawy niezawodności i czytelności kodu. Ułatwieniu ulega również proces debugowania, gdyż analizę można prowadzić na wygenerowanym kodzie asemblerowym opatrzonym

komentarzami, nie zaś na zdeasemblowanym kodzie maszynowym, jak miało to miejsce w pierwszej wersji biblioteki.

2.3.1. Podział na interfejs i zaplecze

O ile w ramach niniejszego projektu inżynierskiego Autor postanowił nie zgłębiać technik symulacji cząstek jako programów GPGPU, to zasadne wydaje się zostawienie w architekturze biblioteki odpowiedniej furtki, aby w przyszłości móc to zrobić. Funkcję takiej furtki pełni podział na abstrakcyjny interfejs oraz konkretne zaplecza.

Zaplecza (ang. *back-end*) stanowią implementacje środowisk konstrukcyjnych i uruchomieniowych dla różnych maszyn – rzeczywistych (np. różne architektury CPU, GPGPU) lub wirtualnych. To one generują odpowiedni program na podstawie konfiguracji emitera, a następnie go uruchamiają.

Natomiast interfejs (ang. *front-end*) obejmuje tę część kodu biblioteki, która jest niezależna od maszyny, która będzie wykonywać właściwy kod emiterów. Zapewnia ona API dla programisty aplikacji-klienta i ukrywa przed nią działanie zaplecza. Co ważne, możliwy jest wybór zaplecza, które ma wykonywać pracę.

2.3.2. Zaplecze asemblera x86

Podobnie jak w pierwotnej wersji biblioteki, Autor zdecydował się zaimplementować przede wszystkim zaplecze wykorzystujące natywny kod maszynowy procesorów rodziny x86. Należy traktować ją jako przykład, gdyż – jak wspomniano w poprzednim rozdziale – możliwe jest uzupełnienie biblioteki o inne zaplecza.

To konkretne zaplecze składa się z generatora, który na podstawie zadanej konfiguracji emitera zapisuje do pliku tymczasowego kod asemblerowy. Kod ten jest składany z małych fragmentów, wcześniej przygotowanych przez Autora i będących wbudowanymi w kod biblioteki. Gotowy tymczasowy plik źródłowy trafia do asemblera NASM (wywoływanego jako proces potomny), który z kolei przetwarza go na plik binarny z kodem maszynowym. Ten ostatni wczytywany jest do bloku pamięci z prawami wykonania i w ten sposób kończy się proces kompilacji kodu emitera.

3. Specyfikacja zewnętrzna

Jako, że produktem prac jest biblioteka programistyczna, jej użytkownikiem może być jedynie inny programista. Dlatego też specyfikacja zewnętrzna projektu sprowadza się do opisu jego API (interfejsu programowania).

3.1. Interfejs programowania

Biblioteka została napisana głównie w języku programowania C++; jednakże w celu zapewnienia możliwie dużej interoperacyjności główny interfejs biblioteki zbudowano w języku C i jego deklaracja składa się z jednego pliku – `libparticlasm2.h`. Wszelkie symbole związane z biblioteką posiadają prefiks `ptc` (deklaracje typów, wyliczeń, struktur, unii) lub `PTC_` (definicje preprocesora).

Aby wykorzystać silnik w swoim programie, należy dołączyć do niego ów plik, a następnie załadować dynamicznie bibliotekę (szczegóły w rozdziale *Punkt wejścia i interfejs*, s. 24). Na maszynie kompilującej aplikację konieczna jest dostępność standardowych plików nagłówkowych `inttypes.h` oraz `stddef.h`.

Zaplecze asemblera x86 wymaga w czasie działania dostępności w ścieżce wyszukiwania asemblera NASM (tj. w bieżącym katalogu bądź w którymś z katalogów wymienionych w zmiennej środowiskowej PATH). Dostarczenie asemblera jest odpowiedzialnością aplikacji klienckiej; stosowanie się do tego szczególnego wymogu nie jest jednak problematyczne, gdyż pakiet dystrybucyjny NASM ogranicza się do jednego pliku wykonywalnego o rozmiarze ok. 600 kB.

W dalszej części rozdziału następuje techniczny opis interfejsu.

3.1.1. Typy proste

typedef float	**ptcScalar**
	Zmiennoprzecinkowa liczba skalarna.
typedef **ptcScalar**	**ptcVector** [3]
	Trójwymiarowy wektor (XYZ).
typedef **ptcScalar**	**ptcColour** [4]
	4-komponentowy kolor (RGBA).

3.1.2. Identyfikatory i typy wyliczeniowe

```
enum ptcModuleID {
  ptcMID_InitialLocation = 0,
  ptcMID_InitialRotation,
  ptcMID_InitialSize,
  ptcMID_InitialVelocity,
  ptcMID_InitialColour,
  ptcMID_Velocity,
  ptcMID_Acceleration,
  ptcMID_Colour,
  ptcMID_Size,
  ptcMID_Gravity
}
```

Typ wyliczeniowy określający rodzaj modułu. Szczegóły w dokumentacji poszczególnych struktur (*ptcMod_<nazwa modułu>*).

```
enum ptcDistributionID {
  ptcDID_Constant = 0,
  ptcDID_Uniform,
  ptcDID_BicubicInterp
}
```

Typ wyliczeniowy określający rodzaj rozkładu. Szczegóły w dokumentacji poszczególnych struktur (*ptc<prefiks typu danych>Distr_<nazwa rozkładu>*).

```
enum ptcTarget {
  ptcTarget_RuntimeInterpreter,
  ptcTarget_x86,
  ptcTarget_x86_64
}
```

Typ wyliczeniowy określający wspierane maszyny docelowe.

```
typedef uint32_t ptcID
```

Typ prosty będący uogólnieniem liczb identyfikujących.

```
enum ptcColourFlags {
  ptcCF_SetRGB = 1,
  ptcCF_SetAlpha = 2
}
```

Typ wyliczeniowy służący do konstrukcji flag bitowych do użycia z modułem *ptcMod_Colour*. Jego wartości można sumować ze sobą bitowo.

- *ptcCF_SetRGB* – jeśli bit ustawiony, zmiany obejmą komponenty RGB,
- *ptcCF_SetAlpha* – jeśli bit ustawiony, zmiany obejmą komponent kanału alfa (przezroczystość).

enum	**ptcGravityFlags** {
	ptcGF_AxisX = 1,
	ptcGF_AxisY = 2,
	ptcGF_AxisZ = 4,
	ptcGF_LinearAtt = 8
	}

	Typ wyliczeniowy służący do konstrukcji flag bitowych do użycia z modułem *ptcMod_Gravity*. Jego wartości można sumować ze sobą bitowo. • *ptcGF_AxisX* – jeśli bit ustawiony, grawitacja działa wzdłuż osi X, • *ptcGF_AxisY* – jeśli bit ustawiony, grawitacja działa wzdłuż osi Y, • *ptcGF_AxisZ* – jeśli bit ustawiony, grawitacja działa wzdłuż osi Z, • *ptcGF_LinearAtt* – jeśli bit ustawiony, stosowany jest model tłumienia liniowego zamiast kwadratowego (tzn. siła jest odwrotnie proporcjonalna do odległości między ciałami, nie do jej kwadratu).

3.1.3. Emiter

Za alokację, inicjalizację i zwalnianie struktury emitera (`ptcEmitter`) odpowiedzialna jest aplikacja kliencka. Zawiera ona konfigurację emitera oraz metadane potrzebne bibliotece w czasie wykonania. Jej skład wygląda następująco:

ptcEmitterConfig	**Config**
	Niezależne od platformy merytoryczne zmienne konfiguracyjne emitera. Wypełniane przez klienta.
ptcScalar	**SpawnTimer**
	Zarezerwowana zmienna robocza, służąca do śledzenia, kiedy emitować nowe cząstki. Tylko do odczytu - wypełniane przez bibliotekę.
size_t	**NumParticles**
	Bieżąca liczba istniejących cząstek. Tylko do odczytu - wypełniane przez bibliotekę.
size_t	**MaxParticles**
	Maksymalna liczba istniejących cząstek (rozmiar bufora cząstek). Wypełniane przez klienta.
void *	**InternalPtr1**
	Zarezerwowany, wewnętrzny wskaźnik biblioteki. Tylko do odczytu.
void *	**InternalPtr2**
	Zarezerwowany, wewnętrzny wskaźnik biblioteki. Tylko do odczytu.
void *	**InternalPtr3**
	Zarezerwowany, wewnętrzny wskaźnik biblioteki. Tylko do odczytu.

ptcModule *	**Head**
	Wskaźnik na głowę listy modułów.
ptcParticle *	**ParticleBuf**
	Wskaźnik na zaalokowany przez klienta bufor, w którym biblioteka ma przechowywać istniejące cząstki. Wypełniane przez klienta.

Struktura konfiguracyjna `ptcEmitterConfig`, będąca pierwszym polem struktury `ptcEmitter` zawiera merytoryczną konfigurację emitera. Ma ona następujący skład:

ptcScalar	**SpawnRate**
	Szybkość emisji cząstek. Jednostką jest liczba serii na sekundę.
uint32_t	**BurstCount**
	Wielkość jednej serii emisji (liczba cząstek w jednej serii).
ptcScalar	**LifeTimeFixed**
	Stała część czasu istnienia cząstki. Całkowity czas istnienia to suma części stałej i części losowej przemnożonej przez losową liczbę z przedziału <0; 1>.
ptcScalar	**LifeTimeRandom**
	Zmienna (losowa) część czasu istnienia cząstki. Całkowity czas istnienia to suma części stałej i części losowej przemnożonej przez losową liczbę z przedziału <0; 1>.

3.1.4. Moduły

Moduły tworzą listę linkowaną. Wspólny interfejs wszystkich modułów jest realizowany przez unię `ptcModule`, składającą się z nagłówka modułu (`ptcModuleHeader`) oraz struktur wszystkich obsługiwanych typów modułów. Każda struktura modułu rozpoczyna się nagłówkiem, czyli strukturą `ptcModuleHeader` o następującym składzie:

ptcID	**ModuleID**
	Identyfikator modułu. Musi być jedną z wartości typu wyliczeniowego *ptcModuleID*.
ptcModulePtr	**Next**
	Wskaźnik na następny moduł w liście (NULL, jeśli jest to ostatni element listy).

3.1.5. Pozostałe struktury i unie

ROZKŁADY

Dla każdego z typów danych – skalarów (`ptcScalar`), wektorów (`ptcVector`) i kolorów (`ptcColour`) istnieją identyczne rodzaje rozkładów. Autor postanowił więc uniknąć niepotrzebnej redundancji i zunifikować ich dokumentację. Różnią się jedynie literą-prefiksem w nazwie symbolu (w poniższych wpisach – zastąpioną literą X) oraz właściwym typem (zastąpionym literą T).

W przypadku wektorów i kolorów, rozkład jest po prostu wyliczany dla każdego z komponentów osobno jak dla skalara.

`ptcXDistr_Constant` – stała wartość zmiennej.

ptcID	**DistrID**
	Identyfikator rozkładu. Musi być jedną z wartości typu wyliczeniowego *ptcDistributionID*.
T	**Val**
	Wartość.

`ptcXDistr_Uniform` – równomierny losowy rozkład zmiennej.

ptcID	**DistrID**
	Identyfikator rozkładu. Musi być jedną z wartości typu wyliczeniowego *ptcDistributionID*.
T	**Range** [2]
	Przedział, z którego losowane będą wartości: od Range[0] do Range[1] włącznie.

`ptcXDistr_BicubicInterp` – rozkład zmiennej według interpolacji dwusześciennej.

ptcID	**DistrID**
	Identyfikator rozkładu. Musi być jedną z wartości typu wyliczeniowego *ptcDistributionID*.
ptcVector	**TargVal**
	Wartość, do której dąży zmienna interpolowana.

`TDistr` – unia obejmująca wszystkie rodzaje rozkładów danego typu danych.

ptcID	DistrID
	Identyfikator rozkładu. Musi być jedną z wartości typu wyliczeniowego *ptcDistributionID*.
ptcXDistr_Constant	Constant
ptcXDistr_Uniform	Uniform
ptcXDistr_BicubicInterp	BicubicInterp

MODUŁY PROSTE

Podobnie, jak w przypadku rozkładów, również większość modułów współdzieli niemalże identyczną strukturę, gdyż ich rolą jest proste ustawienie wartości pewnej własności cząstki. Ponownie celem uniknięcia redundancji, Autor zastosował w ich dokumentacji podobny zabieg, zastępując literę-prefiks literą X oraz właściwy typ – literą T.

Poniższa struktura dotyczy następujących modułów:

- `ptcMod_InitialLocation` – ustawia początkowe (w momencie emisji) położenie cząstki w przestrzeni,
- `ptcMod_InitialRotation` – ustawia początkowy kąt obrotu (w radianach) cząstki wokół jej normalnej,
- `ptcMod_InitialSize` – ustawia początkowy rozmiar cząstki (długość boku kwadratu billboardu),
- `ptcMod_InitialVelocity` – ustawia początkową prędkość cząstki,
- `ptcMod_InitialColour` – ustawia początkowy kolor cząstki,
- `ptcMod_Velocity` – kontroluje prędkość cząstki w czasie jej istnienia,
- `ptcMod_Acceleration` – kontroluje przyspieszenie cząstki w czasie jej istnienia,
- `ptcMod_Size` – kontroluje rozmiar cząstki w czasie jej istnienia.

ptcModuleHeader	Header
	Nagłówek modułu.
TDistr	Distr
	Rozkład wartości.

Moduły pozostałe

`ptcMod_Colour` – kontroluje kolor cząstki w czasie jej istnienia.

ptcModuleHeader	**Header**
	Nagłówek modułu.
ptcColourDistr	**Distr**
	Rozkład wartości.
uint32_t	**Flags**
	Flagi koloru. Patrz *ptcColourFlags*.

`ptcMod_Gravity` – przykład modułu o złożonej funkcjonalności. Modeluje on siłę ciążenia klasycznej mechaniki newtonowskiej. Dla uproszczenia przyjmuje się, że cząstka jest punktem ($r = 0$) o masie jednostkowej ($m = 1$).

ptcModuleHeader	**Header**
	Nagłówek modułu.
ptcVectorDistr	**Centre**
	Położenie środka masy obiektu przyciągającego.
ptcScalar	**Radius**
	Promień obiektu przyciągającego. W jego wnętrzu siła przyciągania jest wprost proporcjonalna do odległości od środka obiektu.
ptcScalar	**SourceMass**
	Masa obiektu przyciągającego.
uint32_t	**Flags**
	Flagi modułu. Patrz *ptcGravityFlags*.

Pozostałe struktury

`ptcParticle` – pojedynczy wpis w tablicy (buforze) cząstek.

uint32_t	**Active**
	Flaga oznaczająca, czy w tym elemencie tablicy istnieje cząstka. Wartość zerowa – miejsce wolne; wartość niezerowa – miejsce zajęte przez istniejącą cząstkę.
ptcScalar	**TimeScale**
	Współczynnik skalujący krok symulacji dla tej cząstki. W efekcie kontroluje jej czas istnienia.
ptcScalar	**Time**
	Znormalizowany czas istnienia cząstki, tj. leżący w przedziale <0; 1>. 0 –

	emisja, 1 – śmierć cząstki.
ptcColour	**Colour**
	Kolor cząstki.
ptcVector	**Location**
	Położenie cząstki w przestrzeni.
ptcScalar	**Rotation**
	Obrót cząstki wokół jej normalnej w radianach.
ptcScalar	**Size**
	Rozmiar cząstki (długość boku kwadratu billboardu).
ptcVector	**Velocity**
	Prędkość chwilowa cząstki.
ptcVector	**Accel**
	Przyspieszenie chwilowe cząstki.

`ptcVertex` – pojedynczy wierzchołek billboardu emitowany przez silnik.

ptcColour	**Colour**
	Kolor wierzchołka.
ptcVector	**Location**
	Położenie wierzchołka.
int16_t	**TexCoords** [2]
	Współrzędne tekstur wierzchołka.

3.1.6. Punkt wejścia i interfejs

Silnik jest przeznaczony do dynamicznego ładowania za pomocą mechanizmu właściwego dla danej platformy[10] – należy pobrać uchwyt do biblioteki, a następnie wskaźnik do funkcji punktu wejścia[11]. Nazwę symbolu, który należy wydobyć definiuje makro `PTC_ENTRY_POINT`. Po zakończeniu pracy z biblioteką należy ją zwolnić[12]. Funkcja punktu wejścia posiada następującą sygnaturę:

```
uint32_t ptcGetAPI(uint32_t clientVersion, ptcAPIExports *API);
```

10 Funkcją `LoadLibrary()` (system Windows [7]) lub `dlopen()` (system zgodny z POSIX [8]).

11 Funkcją `GetProcAddress()` (Windows) lub `dlsym()` (POSIX).

12 Funkcją `FreeLibrary()` (Windows) lub `dlclose()` (POSIX).

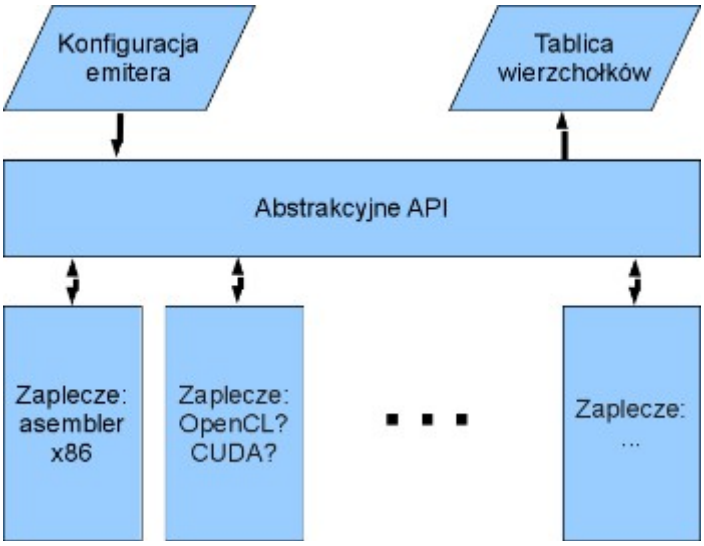
Należy ją wywołać, jako parametr `clientVersion` podając makro `PTC_API_VERSION`. Parametr ten służy weryfikacji zgodności wersji biblioteki, z jaką skompilowano aplikację-klienta, z wersją dostępną w czasie uruchomienia. Jako parametr `API` należy przekazać wskaźnik na zaalokowaną przez klienta strukturę `ptcAPIExports`. Przy powodzeniu wstępnej inicjalizacji (sygnalizowanej zwróceniem niezerowej wartości przez funkcję wejścia; zerowa oznacza niezgodność wersji), struktura ta jest wypełniana wskaźnikami do właściwych funkcji silnika:

uint32_t(*	**QueryTargetSupport**)(**ptcTarget** target)
	Odpytuje bibliotekę o dostępność zaplecza obsługującego zadaną maszynę. Zwraca niezerową wartość, jeśli zaplecze jest dostępne; zero w przeciwnym razie.
uint32_t(*	**InitializeTarget**)(**ptcTarget** target, void *privateData)
	Inicjalizuje zadane zaplecze. Parametr *privateData* jest opcjonalny i jego znaczenie zależy od konkretnego zaplecza. Zwraca niezerową wartość, jeśli inicjalizacja zaplecza się powiedzie; zero w przeciwnym razie.
void(*	**ShutdownTarget**)(void *privateData)
	Zwalnia aktywne zaplecze. Parametr *privateData* jest opcjonalny i jego znaczenie zależy od konkretnego zaplecza.
uint32_t(*	**CompileEmitter**)(**ptcEmitter** *emitter)
	Kompiluje program dla zadanego emitera. Zwraca niezerową wartość, jeśli kompilacja się powiedzie; zero w przeciwnym razie.
uint32_t(*	**ProcessEmitter**)(**ptcEmitter** *emitter, **ptcScalar** step, **ptcVector** cameraCS[3], **ptcVertex** *buffer, uint32_t maxVertices)
	Emituje nowe cząstki i przetwarza istniejące zgodnie z konfiguracją emitera (tj. zaplecze uruchamia jego program). Zwraca liczbę wyemitowanych do bufora wierzchołków. Znaczenie parametrów: • *emitter* – wskaźnik do przetwarzanego emitera, • *step* – krok symulacji w sekundach, • *cameraCS* – macierz 3x3 opisująca lokalny układ współrzędny kamery; służy do poprawnej emisji wierzchołków billboardów, • *buffer* – wskaźnik do zaalokowanego przez klienta bufora, w którym mają znaleźć się wyemitowane wierzchołki, • *maxVertices* – pojemność bufora wierzchołków.
void(*	**ReleaseEmitter**)(**ptcEmitter** *emitter)
	Zwalnia wszystkie zaalokowane przez bibliotekę zasoby związane z zadanym emiterem.

4. Organizacja wewnętrzna

4.1. Schemat architektoniczny

Ilustracja 7 przedstawia wysokopoziomowy schemat architektury biblioteki. Wyraźnie zarysowany jest podział na interfejs i zaplecze.



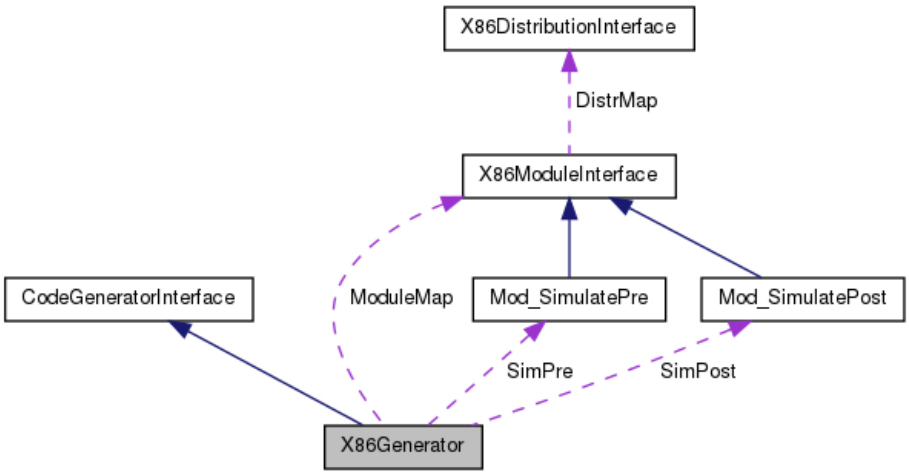
Ilustracja 7: Ogólny schemat architektury silnika

4.2. Przegląd kluczowych klas zaplecza x86

Dla działania całości systemu kluczowe są klasy generatorów, konstruujące programy emiterów – implementujące interfejs `CodeGeneratorInterface` – oraz uruchamiające te programy – implementujące `LauncherInterface`.

4.2.1. Klasa X86Generator

Klasa `X86Generator` implementuje dwie fazy konstrukcji programu emitera – generację kodu źródłowego (metoda `Generate()`) oraz – jeśli generacja się powiedzie – konstrukcję kodu wykonywalnego (metoda `Build()`). Jako klasa generatora kodu jest czymś w rodzaju singletona – w dowolnym momencie może



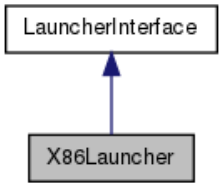
Ilustracja 8: Diagram współpracy klasy X86Generator

istnieć tylko jedna jej instancja, jednakże obiekt nie jest tworzony statycznie, a dynamicznie, na żądanie aplikacji klienckiej podczas inicjalizacji zaplecza.

Jak widać na diagramie współpracy (Ilustracja 8), klasa zawiera statyczne pole `ModuleMap`, czyli tablicę referencji do statycznych instancji wszystkich klas modułów. Podczas przechodzenia listy modułów w konfiguracji emitera wybierany jest z tej tablicy odpowiedni obiekt generujący kod modułu.

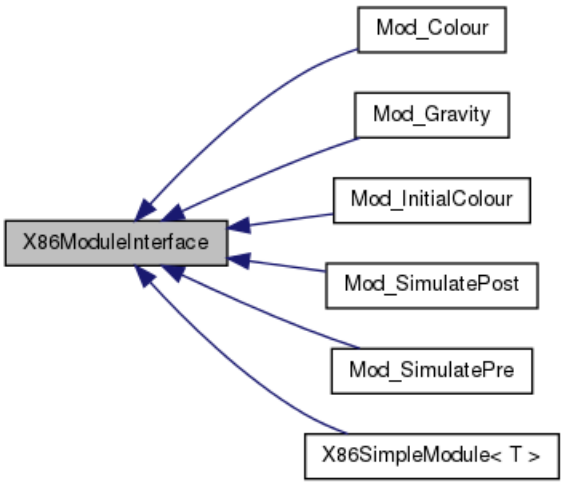
4.2.2. Klasa X86Launcher

Klasa `X86Launcher` (Ilustracja 9) implementuje proces uruchamiania programu, poczynając od zarządzania pamięcią z uprawnieniami uruchamiania, kończąc na odpowiednim wywoływaniu wygenerowanego kodu i przekazywaniu do niego argumentów przez wstawki asemblerowe.



Ilustracja 9: Diagram współpracy klasy X86Launcher

4.2.3. Interfejs X86ModuleInterface *i klasy generujące kod modułów*



Ilustracja 10: Diagram dziedziczenia klasy X86ModuleInterface

Statyczne pole `ModuleMap` klasy `X86Generator` to tablica referencji typu `X86ModuleInterface` (patrz Ilustracja 10). Jest to klasa-interfejs obiektu generującego kod konkretnego modułu (patrz sekcje *Moduły proste* i *Moduły pozostałe*, s. 22). Na podstawie zadanego kontekstu generacji (pomocnicza struktura danych) i wskaźnika do struktury modułu (`ptcModule`, patrz rozdział *Moduły*, s. 20) emitują odpowiedni kod źródłowy.

Na uwagę zasługuje tutaj szablon klasy `X86SimpleModule` – reprezentuje on moduły proste (ponownie, sekcja *Moduły proste*, s. 22), których działanie ogranicza się do sterowania jedną własnością cząstki. Autor postanowił zunifikować ich kod ze względu na brak istotnych różnic w ich implementacji. Jako parametr szablonu przyjmuje typ właściwości, którą steruje moduł, a w konstruktorze – treść fragmentów asemblerowych.

Klasy `Mod_SimulatePre` i `Mod_SimulatePost` są pseudomodułami – nie występują w konfiguracjach emiterów. Generator automatycznie dołącza je odpowiednio na początku i na końcu kodu przetwarzania cząstek. Realizują zadania niezmienne, takie, jak postępowanie kroku symulacji (wraz z unicestwianiem cząstki, która przekracza czas istnienia), czy wykonanie równań kinematyki Newtona (wyliczenie nowej prędkości chwilowej oraz nowej pozycji).

5. Wybrane zagadnienia implementacji

5.1. Uruchamianie procesów potomnych

Jako, że podstawą funkcjonalności zaplecza asemblera x86 silnika jest wywołanie asemblera NASM w celu asemblacji wcześniej automatycznie wygenerowanego kodu źródłowego, istniała potrzeba wywoływania procesów potomnych z możliwością przechwycenia ich standardowego wyjścia oraz odczytu kodu wyjścia procesu. Informacje te są konieczne, by móc stwierdzić sukces lub porażkę asemblacji, i w razie błędów wyemitować odpowiednie komunikaty.

Metody sprawowania takiej kontroli nad uruchomieniem procesu potomnego ściśle zależą od platformy, dlatego też metoda `CodeGeneratorInterface::RunProcess()`, zamykająca w sobie cały kod odpowiedzialny za uruchamianie procesu potomnego, jest rozdzielona dyrektywami preprocesora na części napisane dla platform Windows i POSIX. Ogólny algorytm dla obu jest jednak podobny:

1. Stworzyć potoki zastępcze dla urządzeń, które chcemy przechwycić (`stderr` i `stdout`).
2. Uruchomić proces, zastępując urządzenia końcami zapisującymi potoków.
3. Dopóki proces potomny nie zakończy się lub potoki nie zostaną przerwane:
    1. Odczytać z potoków dane do buforów.
    2. Wrócić do punktu 3.
4. Odczytać kod wyjścia procesu.

Oczywiście różnice pojawiają się w szczegółach. Przykładowo, API systemu Windows wymaga dużej liczby struktur sterujących, w których przekazuje się uchwyty potoków, zasady ich dziedziczenia czy też atrybuty bezpieczeństwa. Natomiast programując w API POSIX należy zduplikować i zachować kopie deskryptorów przekierowywanych urządzeń oraz skorzystać z mechanizmu `fork()`, który tworzy kopię bieżącego procesu, a następnie inaczej sterować deskryptorami przekierowywanych urządzeń zależnie od starszeństwa w drzewie procesów.

5.2. Preprocesory i translatory kodu

W kodzie asemblerowym modułów emiterów konieczne są odwołania do pól struktur danych biblioteki. Struktury te są jednakże opisane w języku C. W asemblerze mamy więc do wyboru albo ręcznie wyliczać adresy, albo przetłumaczyć deklaracje z języka C. Oczywiście bezpieczniejszą (minimalizującą możliwość pomyłki) metodą jest ostatnia; jednakże ręczne tłumaczenie deklaracji stwarza ryzyko zdezaktualizowania się którejś z wersji, jeśli programista zapomni wprowadzić analogiczne zmiany w drugim pliku. Problem rozwiązuje automatyczna translacja odpowiedniego kodu.

Z podobnym zagadnieniem można zetknąć się próbując wbudować fragmenty kodu asemblerowego modułów w plik dynamicznie dołączanej biblioteki. Autorowi bardzo zależało na braku potrzeby dystrybucji z silnikiem dodatkowych plików, toteż postanowił konwertować owe fragmenty na plik nagłówkowy języka C.

Oba te zadania wykonują skrypty w języku Python, wywoływane w procesie budowy biblioteki przed wywołaniem kompilatora.

6. Przykład wykorzystania

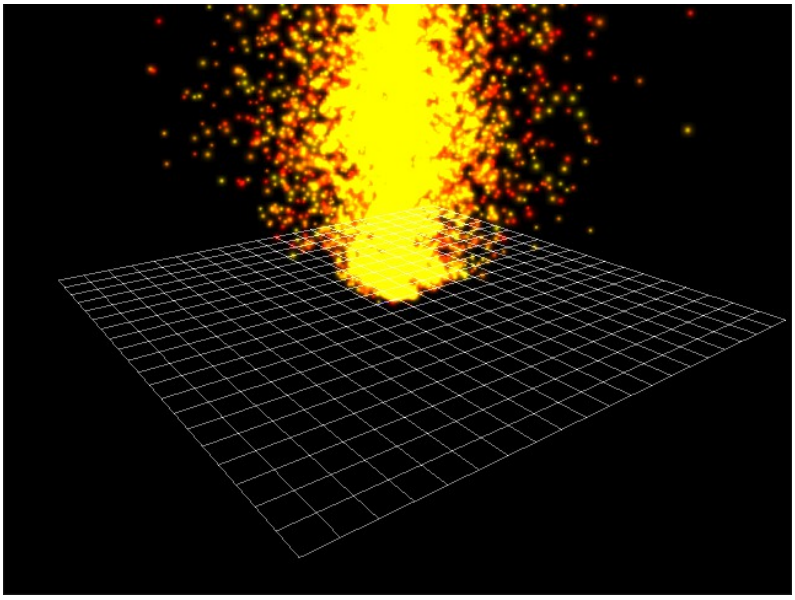
W niniejszym rozdziale przedstawiony jest skomentowany fragment listingu hipotetycznej aplikacji korzystającej z silnika w celu symulacji cząstek. Przykład zakłada program w języku C++ na platformę Linux, z rendererem opartym o OpenGL.

6.1. Integracja z aplikacją kliencką

```
#include <libparticlasm2.h>

// blok zmiennych globalnych
// uchwyt biblioteki
void *g_ptc_handle;
// interfejs biblioteki
ptcAPIExports g_ptcAPI;

// tablica emiterów
ptcEmitter *g_ptc_emitters;
// liczba emiterów
size_t g_ptc_nemitters;
// liczba wyemitowanych wierzchołków
volatile size_t g_ptc_nvertices;
// bufor cząstek
ptcParticle g_ptc_particles[MAX_PARTICLES];
// bufor wierzchołków
ptcVertex g_ptc_vertices[sizeof(g_ptc_particles)
    / sizeof(g_ptc_particles[0]) * 4];

// wybór architektury, na którą kompilowany jest kod
#if defined(_M_AMD64) || defined(amd64) || defined (__amd64__)
    #define LOCAL_TARGET ptcTarget_x86_64
#else
    #define LOCAL_TARGET ptcTarget_x86
#endif

// ...

// inicjalizacja biblioteki
g_ptc_handle = dlopen("libparticlasm2-linux-x86.so", RTLD_NOW);
if (!g_ptc_handle) {
    printf("dlerror: %s\n", dlerror());
    return false;
}

PFNPTCGETAPI ptcGetAPI = (PFNPTCGETAPI)dlsym(g_ptc_handle,
    PTC_ENTRY_POINT);
if (ptcGetAPI && ptcGetAPI(PTC_API_VERSION, &g_ptcAPI)
    && g_ptcAPI.InitializeTarget(LOCAL_TARGET, NULL))
    return true;

printf("dlerror: %s\n", dlerror());
dlclose(g_ptc_handle);
return false;

// ...

// kompilacja emiterów
g_ptc_nemitters = load_emitters(&g_ptc_emitters);
for (size_t i = 0; i < g_ptc_nemitters; ++i) {
    g_ptc_emitters[i].ParticleBuf = g_ptc_particles;
    g_ptc_emitters[i].MaxParticles = sizeof(g_ptc_particles)
```

```
        / sizeof(g_ptc_particles[0]);
    if (!g_ptcAPI.CompileEmitter(&g_ptc_emitters[i]))
    {
        printf("Nie udało się skompilować emitera.\n");
        return false;
    }
}
// powiązanie tablicy wierzchołków
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, sizeof(ptcVertex),
    &g_ptc_vertices[0].Location[0]);
glTexCoordPointer(2, GL_SHORT, sizeof(ptcVertex),
    &g_ptc_vertices[0].TexCoords[0]);
glColorPointer(4, GL_FLOAT, sizeof(ptcVertex),
    &g_ptc_vertices[0].Colour[0]);

// ...

// fragment pętli symulacji i renderowania
// zmienne implikowane:
// float frame_time         - krok symulacji w sekundach
// float camera_cs[3][3]    - macierz obrotu obserwatora
// wyzeruj licznik wierzchołków
g_ptc_nvertices = 0;
// liczba wolnych miejsc w tablicy wierzchołków
size_t space_left = sizeof(g_ptc_vertices) / sizeof(g_ptc_vertices[0]);
// roboczy wskaźnik do bufora wierzchołków
ptcVertex *buffer = g_ptc_vertices;
// iteruj po wszystkich emiterach
for (size_t i = 0; i < g_ptc_nemitters; ++i)
{
    // uruchom program emitera
    const size_t verts_emitted = g_ptcAPI.ProcessEmitter(
        &g_ptc_emitters[i], frame_time, camera_cs, buffer, space_left);
    // uaktualnij liczniki i wskaźnik do bufora
    space_left -= verts_emitted;
    g_ptc_nvertices += verts_emitted;
    buffer += verts_emitted;
}
// renderuj wierzchołki jako prostokąty od pierwszego indeksu
glDrawArrays(GL_QUADS, 0, ptc_nvertices);

// ...

// zwalnianie biblioteki
if (g_ptc_handle) {
    g_ptcAPI.ShutdownTarget(NULL);
    dlclose(g_ptc_handle);
}
```

6.2. Przykładowa konfiguracja emitera



Ilustracja 11: Przykładowa konfiguracja emitera, realizowana przez aplikację hostapp*, stworzoną jako demonstrator projektu*

W ramach prac Autorowi nie wystarczyło czasu na przygotowanie graficznego edytora konfiguracji emiterów – w chwili składania projektu dostępna była jedynie możliwość stworzenia takiej konfiguracji z kodu źródłowego. Jednakże wykorzystywane struktury danych są bardzo proste, a więc napisanie aplikacji edytora w przyszłości nie będzie stanowić większego problemu.

Poniżej zaprezentowana jest funkcja w języku C, tworząca przykładową konfigurację emitera cząstek (efekt jego działania przedstawia Ilustracja 11).

```
size_t get_example_emitters(ptcEmitter **emitters) {
    static ptcMod_InitialVelocity initvel;
    static ptcMod_InitialColour initcol;
    static ptcMod_InitialSize initsize;
    static ptcMod_Acceleration accel;
    static ptcMod_Colour alpha;
    static ptcMod_InitialLocation s_il;
    static ptcMod_InitialVelocity s_iv;
    static ptcMod_InitialColour s_ic;
    static ptcMod_InitialSize s_is;
    static ptcMod_Acceleration s_acc;
    static ptcMod_Colour s_alpha;
    static ptcEmitter emit[1];

    memset(emit, 0, sizeof(emit));

    // fire emitter
    emit[0].Config.SpawnRate = 500.f;
    emit[0].Config.BurstCount = 15;
    emit[0].Config.LifeTimeFixed = 1;
    emit[0].Config.LifeTimeRandom = 1;
    emit[0].InternalPtr1 = NULL;
    emit[0].InternalPtr2 = NULL;
```

```
    emit[0].InternalPtr3 = NULL;
    emit[0].Head = (ptcModule *)&initvel;
    emit[0].ParticleBuf = NULL;
    emit[0].NumParticles = 0;

    initvel.Header.ModuleID = ptcMID_InitialVelocity;
    initvel.Header.Next = (ptcModule *)&initcol;
    initvel.Distr.Uniform.DistrID = ptcDID_Uniform;
    initvel.Distr.Uniform.Ranges[0][0] = -200;
    initvel.Distr.Uniform.Ranges[0][1] = 80;
    initvel.Distr.Uniform.Ranges[0][2] = -200;
    initvel.Distr.Uniform.Ranges[1][0] = 200;
    initvel.Distr.Uniform.Ranges[1][1] = 160;
    initvel.Distr.Uniform.Ranges[1][2] = 200;

    initcol.Header.ModuleID = ptcMID_InitialColour;
    initcol.Header.Next = (ptcModule *)&initsize;
    initcol.Distr.Uniform.DistrID = ptcDID_Uniform;
    initcol.Distr.Uniform.Ranges[0][0] = 0.8;
    initcol.Distr.Uniform.Ranges[0][1] = 0.f;
    initcol.Distr.Uniform.Ranges[0][2] = 0.f;
    initcol.Distr.Uniform.Ranges[0][3] = 1.f;
    initcol.Distr.Uniform.Ranges[1][0] = 1.f;
    initcol.Distr.Uniform.Ranges[1][1] = 0.8;
    initcol.Distr.Uniform.Ranges[1][2] = 0.f;
    initcol.Distr.Uniform.Ranges[1][3] = 1.f;

    initsize.Header.ModuleID = ptcMID_InitialSize;
    initsize.Header.Next = (ptcModule *)&accel;
    initsize.Distr.Uniform.DistrID = ptcDID_Uniform;
    initsize.Distr.Uniform.Range[0] = 3.f;
    initsize.Distr.Uniform.Range[1] = 6.f;

    accel.Header.ModuleID = ptcMID_Acceleration;
    accel.Header.Next = (ptcModule *)&alpha;
    accel.Distr.Uniform.DistrID = ptcDID_Uniform;
    accel.Distr.Uniform.Ranges[0][0] = 0.f;
    accel.Distr.Uniform.Ranges[0][1] = -50.f;
    accel.Distr.Uniform.Ranges[0][2] = 0.f;
    accel.Distr.Uniform.Ranges[1][0] = 0.f;
    accel.Distr.Uniform.Ranges[1][1] = -80.f;
    accel.Distr.Uniform.Ranges[1][2] = 0.f;

    alpha.Header.ModuleID = ptcMID_Colour;
    alpha.Header.Next = NULL;
    alpha.Distr.Constant.DistrID = ptcDID_Constant;
    alpha.Distr.Constant.Val[3] = 1.f;
    alpha.Flags = ptcCF_SetAlpha;

    return sizeof(emit) / sizeof(emit[0]);
}
```

7. Testowanie i eksperymenty

Głównym celem pracy było osiągnięcie dobrych rezultatów wydajnościowych. Autor skupił się więc na eksperymentach mierzących czas symulacji odpowiednich efektów cząsteczkowych.

7.1. Metodyka testowania

Pomiar wydajności został przeprowadzony na dwóch maszynach: tym samym laptopie, na którym Autor testował projekt zaliczeniowy z Języków Asemblerowych[13], jak i na nowocześniejszym komputerze typu desktop z procesorem rodziny x86-64[14]. W obu przypadkach komputery działały pod kontrolą systemu Linux, a na potrzeby pomiarów uruchomiono system w trybie awaryjnym (tj. bez podsystemu graficznego X11 i usług), aby inne procesy nie zaburzały wyników.

Pomiar wykonano aplikacją `headless` będącą częścią projektu, która mierzy czas (z rozdzielczością 1 milisekundy) potrzebny na wykonanie określonej liczby klatek symulacji. W przypadku tego testu – 1800 klatek, czyli pozoracja 30 sekund symulacji z częstotliwością odświeżania 60 Hz. Porównanie ma miejsce między silnikiem z zapleczem asemblera x86 a analogicznym funkcjonalnie, choć interpretującego konfigurację emitera w locie, silnikiem napisanym w języku C++.

Za każdym razem wykorzystano przykładową konfigurację emitera przedstawioną w rozdziale *Przykładowa konfiguracja emitera*, s. 32. Emiter ten, o ile pozwala na to rozmiar bufora, utrzymuje w istnieniu średnio 10 000 cząstek (tj. przy ciągłym działaniu w dowolnym momencie istnieje naraz ok. 10 000 cząstek).

Eksperyment zautomatyzowano za pomocą skryptu powłoki *bash*: dla każdej wspieranej przez bieżący procesor architektury porównano wyniki obu implementacji. Powtórzono go dziesięciokrotnie (w celu uśrednienia wyników) dla rozmiarów bufora cząstek od 2 000 do 30 000, z krokiem 2 000.

13 Konfiguracja sprzętowa: Intel Pentium M 758 (1,5 GHz, 2 MB L2 Cache), 1536 MB RAM. Jądro systemu operacyjnego: Linux 3.2.0-4-i386.

14 Konfiguracja sprzętowa: Intel Core 2 Duo E6750 (2,67 GHz, 4 MB L2 Cache), 4 GB RAM. Jądro systemu operacyjnego: Linux 3.2.0-4-amd64.

7.2. Wyniki eksperymentów

7.2.1. Maszyna desktop, tryb x86 (32-bit)

Tabela 1: Wyniki pomiaru czasu symulacji 1800 klatek implementacją C++ w zależności od rozmiaru bufora cząstek

Rozmiar bufora	Śr. czas symulacji [ms]	Śr. czas/klatka [ms]
2000	863,7	0,4798
4000	2686,3	1,4924
6000	4880,4	2,7113
8000	7854,4	4,3636
10000	11821,5	6,5675
12000	12676,5	7,0425
14000	13446,8	7,4704
16000	15832,2	8,7957
18000	15819,7	8,7887
20000	12689,4	7,0497
22000	14667,8	8,1488
24000	15913,1	8,8406
26000	14758,9	8,1994
28000	15547,4	8,6374
30000	12818,9	7,1216

Tabela 2: Wyniki pomiaru czasu symulacji 1800 klatek implementacją asemblera x86 w zależności od rozmiaru bufora cząstek

Rozmiar bufora	Śr. czas symulacji [ms]	Śr. czas/klatka [ms]
2000	651,9	0,3622
4000	1276,7	0,7093
6000	1964,4	1,0913
8000	2672,2	1,4846
10000	3182,2	1,7679
12000	3966,2	2,2034
14000	3748,2	2,0823
16000	3217,4	1,7874
18000	3251,5	1,8064
20000	3927,2	2,1818
22000	3551,6	1,9731
24000	3334,9	1,8527
26000	3504	1,9467
28000	3350,4	1,8613
30000	4070,1	2,2612

7.2.2. Maszyna desktop, tryb x86-64 (64-bit)

Tabela 3: Wyniki pomiaru czasu symulacji 1800 klatek implementacją C++ w zależności od rozmiaru bufora cząstek

Rozmiar bufora	Śr. czas symulacji [ms]	Śr. czas/klatka [ms]
2000	1675	0,9306
4000	5138,6	2,8548
6000	12475,6	6,9309
8000	17348,6	9,6381
10000	27074,9	15,0416
12000	40374,1	22,4301
14000	39371,8	21,8732
16000	36847	20,4706
18000	41518,3	23,0657
20000	37313,1	20,7295
22000	33112,4	18,3958
24000	41501,3	23,0563
26000	40595,4	22,5530
28000	36301,1	20,1673
30000	36159,7	20,0887

Tabela 4: Wyniki pomiaru czasu symulacji 1800 klatek implementacją asemblera x86-64 w zależności od rozmiaru bufora cząstek

Rozmiar bufora	Śr. czas symulacji [ms]	Śr. czas/klatka [ms]
2000	635	0,3528
4000	1343,2	0,7462
6000	1811,7	1,0065
8000	2824,8	1,5693
10000	3404,8	1,8916
12000	3016,9	1,6761
14000	3009	1,6717
16000	3310,6	1,8392
18000	2975,6	1,6531
20000	3314,2	1,8412
22000	3571,1	1,9839
24000	3017,3	1,6763
26000	3024,5	1,6803
28000	3375,1	1,8751
30000	3390,4	1,8836

7.2.3. Maszyna laptop, tryb x86 (32-bit)

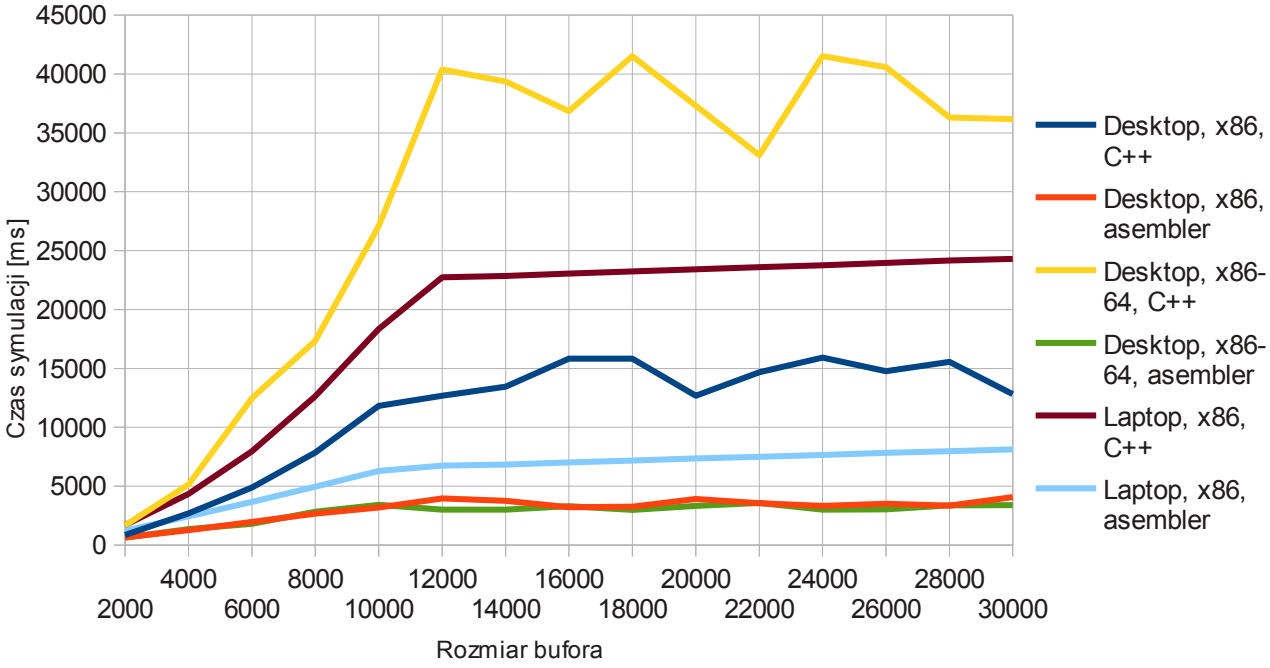
Tabela 5: Wyniki pomiaru czasu symulacji 1800 klatek implementacją C++ w zależności od rozmiaru bufora cząstek

Rozmiar bufora	Śr. czas symulacji [ms]	Śr. czas/klatka [ms]
2000	1687,9	0,9377
4000	4354,9	2,4194
6000	7979,6	4,4331
8000	12623,8	7,0132
10000	18344,4	10,1913
12000	22736	12,6311
14000	22853,5	12,6964
16000	23067,4	12,8152
18000	23235	12,9083
20000	23417,5	13,0097
22000	23602,3	13,1124
24000	23761,8	13,2010
26000	23960,1	13,3112
28000	24157,1	13,4206
30000	24297,9	13,4988

Tabela 6: Wyniki pomiaru czasu symulacji 1800 klatek implementacją asemblera x86 w zależności od rozmiaru bufora cząstek

Rozmiar bufora	Śr. czas symulacji [ms]	Śr. czas/klatka [ms]
2000	1221	0,6783
4000	2445,5	1,3586
6000	3666,3	2,0368
8000	4947,2	2,7484
10000	6297,3	3,4985
12000	6755,2	3,7529
14000	6843	3,8017
16000	7015,9	3,8977
18000	7181,4	3,9897
20000	7346,2	4,0812
22000	7503,4	4,1686
24000	7650,8	4,2504
26000	7829,6	4,3498
28000	7979,9	4,4333
30000	8127,8	4,5154

7.2.4. Wykres



Ilustracja 12: Wykres zależności czasu symulacji 1800 klatek od rozmiaru bufora

7.2.5. Wnioski

Wykres (Ilustracja 12) jasno dowodzi, że cel zasadniczy – uzyskanie wyższej wydajności za pomocą asemblera – został osiągnięty. Można też zauważyć, że implementacja asemblera x86-64 jest marginalnie szybsza, niż ta w trybie 32-bit. Autor przypuszcza, że wynika to ze sprzętowo realizowanego adresowania względem licznika instrukcji (ang. *RIP-relative addressing*), które jest szeroko wykorzystywane w projekcie. W trybie 32-bit musi być ono realizowane programowo. Można zaobserwować ponadto kilka ciekawych faktów.

Przede wszystkim widoczny na wszystkich przebiegach w okolicach rozmiaru bufora o wartości 10 000-12 000 cząstek punkt przegięcia – jest to punkt, w którym bufor zaczyna już być wystarczająco duży, by zawsze pomieścić wszystkie cząstki tworzone przez emiter.

Drugą ciekawostką jest różnica zachowań procesorów laptopa[15] i desktopa[16] po przekroczeniu owego punktu przegięcia – w przypadku starszego procesora czas wykonania rośnie liniowo i stabilnie, natomiast nowsza jednostka w obu trybach (32- i 64-bit) wykazuje chaotyczne fluktuacje.

15 Intel Pentium M 758; premiera w styczniu 2005, mikroarchitektura P6.

16 Intel Core 2 Duo E6750; premiera w lipcu 2007, mikroarchitektura Core.

Pierwsze zjawisko – dalszy wzrost czasu wykonania po punkcie przegięcia – Autor uzasadnia „pustym przebiegiem” po buforze cząstek: algorytm przechodzi nawet po niewykorzystanych elementach tablicy, wykonując sprawdzenie, czy nie są zajęte. Przekłada się to na drobny, proporcjonalny do rozmiaru bufora narzut.

Przyczyny drugiego zjawiska Autorowi nie udało się zgłębić; przypuszcza się, że nowszy procesor posiada lepszy algorytm przewidywania rozgałęzień, co – w połączeniu z losowością rozkładu cząstek w buforze – tłumaczyłoby fluktuacje działające raz na znaczną korzyść, a raz na niekorzyść czasu wykonania.

Zaskakująca jednakże jest wydajność implementacji C++ w trybie 64-bit, dużo gorsza od wariantu 32-bit uruchamianego na tym samym procesorze. Fakt ten dziwi tym bardziej, że kod C++ jest identyczny, jedynie skompilowany dla innego zestawu instrukcji. Przyczyn może być kilka – od narzutu rozmiaru wskaźników (dłuższy adres zakodowany w rozkazie → dłuższy czas dekodowania rozkazu), poprzez efekty pamięci podręcznej procesora po – najbardziej prawdopodobny ze względu na dobry wynik implementacji asemblerowej – gorszą jakości kodu wygenerowanego przez kompilator dla tej platformy. Są to jednak jedynie spekulacje, których Autor nie miał czasu zweryfikować.

8. Przebieg i wyniki prac

8.1. Oczekiwana a otrzymana wydajność

Uśredniając wyniki uzyskane na poszczególnych platformach, silnik uzyskuje **względne** przyrosty prędkości przetwarzania (stosunek czasu implementacji C++ do czasu implementacji asemblerowej) rzędu 283,66% (laptop), 355,15% (desktop, x86) i 987,48% (desktop, x86-64).

Przyrost wydajności w trybie 32-bit jest niestety na obu maszynach mniejszy, niż deklarowane w sprawozdaniu z projektu zaliczeniowego 358% (patrz rozdział *Punkt wyjścia – projekt zaliczeniowy*, s. 12); Autor nie ma pewności, czy wynika to ze zmian w architekturze i kodzie silnika, czy też może z mniej starannie prowadzonych w ramach pracy [6] eksperymentów (metodologia testów nie została tam

udokumentowana, a Autor już jej nie pamięta). Wynik nowej implementacji jest jednak nadal bardzo dobry.

Wynik implementacji asemblera x86-64, jakkolwiek imponujący i obiektywnie dobry (marginalnie szybszy od asemblera w trybie 32-bit), należy niestety w kategoriach względnych potraktować sceptycznie, biorąc pod uwagę słabą wydajność implementacji C++ na tej samej platformie.

Uzyskana wydajność zaplecza asemblerowego jest dla Autora jak najbardziej satysfakcjonująca – na nowszej maszynie czas poświęcony na symulację cząstek nigdy nie przekracza 3 ms, co daje sporo czasu na pozostałe obliczenia aplikacji klienckiej przy działaniu w czasie rzeczywistym.

8.2. Napotkane problemy

8.2.1. Adresowanie w trybie 64-bit

Pierwotnym zamysłem Autora był rozwój jego wcześniejszego projektu zaliczeniowego – poczynił on nawet pewne postępy w tym kierunku, co zostało udokumentowane w historii zmian gałęzi 1.x w repozytorium kodu projektu [9]. Wedle wymagania niefunkcjonalnego NFREQ 3 (s. 15), silnik powinien działać na architekturze x86-64, i od tego aspektu rozpoczęto prace.

Z punktu widzenia asemblera różnice między architekturą 32- a 64-bitową sprowadzają się do następujących punktów:

- rozmiary typów danych i przesunięcia adresów pól w strukturach,
- konwencje wywołań funkcji,
- brak kodowania pewnych wariantów rozkazów dla operandów 64-bitowych.

Pierwsze dwa problemy zdawała się rozwiązywać biblioteka NASMX [10]; mimo to jej zastosowanie wymagało przepisania prawie całego kodu asemblerowego, aby użyć makr i prototypów zapewnianych przez tę bibliotekę. Nierozwiązany pozostawał ostatni punkt listy.

Jak wspomniano w rozdziale *Punkt wyjścia – projekt zaliczeniowy* (s. 12), implementacja pierwotna umieszczała w kodzie maszynowym puste adresy, które następnie były wypełniane bezwzględnymi adresami do danych. Ta technika nie miała prawa zadziałać – przynajmniej bez istotnych modyfikacji – w architekturze

x86-64, gdyż nawet tak podstawowe rozkazy, jak `MOV` – instrukcja przesyłu danej między rejestrami bądź pamięcią – zwyczajnie nie posiada kodowania, pozwalającego na umieszczenie w niej bezwzględnego, 64-bitowego adresu [11]. Oczywiście nie jest to jedyna taka instrukcja.

Wymusiło to stosowanie sztuczek związanych z adresowaniem pośrednim oraz sztuczne rozgałęzianie kodu w niemal każdym miejscu, gdzie następował dostęp do zapisanych w pamięci danych emitera – wszystko w celu obsłużenia przypadków 32 i 64 bitów.

Autor zupełnie zniechęcił się do tej ścieżki rozwoju kiedy okazało się, że kłopoty sprawiać będzie także rozkaz `CALL`, służący do wywoływania funkcji – myśl o tym, że każde wywołanie funkcji ze wskaźnika będzie należało poprzedzić załadowaniem owego wskaźnika do rejestru, i wywoływać będzie można dopiero zawartość tego rejestru, chowając przedtem jego zawartość na stos, przypieczętowała decyzję o rozpoczęciu od nowa i napisaniu generatora kodu źródłowego.

8.2.2. Czerwona strefa

Kolejny problem również wyniknął z własności architektury x86-64, i objawił się przy pierwszych próbach uruchamiania wygenerowanego kodu.

Funkcje, które nie wywołują już żadnych innych funkcji (a co za tym idzie – ich ramka stosu nie może być przykryta ramką innej funkcji) nazywa się **funkcjami-liśćmi**, gdyż można je uznać za liście grafu przepływu kontroli w programie.

W celu unifikacji konwencji wywołań programów emiterów między platformami x86 i x86-64, Autor wprowadził własną, realizowaną za pomocą wstawek asemblerowych w kodzie C++. Wstawki te zawierają instrukcje wywołań funkcji, które jednakże przez swoją formę (tj. kod asemblerowy) nie są „zauważane” przez kompilator. W efekcie mylnie je oznacza jako funkcje-liście.

Konwencja wywołań System V dla procesorów x86-64 [12] wprowadza pojęcie **czerwonej strefy** (ang. *red zone*) – jest to zarezerwowany obszar pamięci o długości 128 bajtów powyżej szczytu stosu, o gwarantowanej nienaruszalności przez procedury obsługi przerwań i sygnałów. Został on przewidziany jako miejsce

na zmienne tymczasowe dla funkcji-liści, z którego można korzystać bez narzutu wynikającego z modyfikacji wskaźnika stosu.

Kompilator C++ skwapliwie korzystał z owej czerwonej strefy, umieszczając w niej wszystkie zmienne lokalne (zadeklarowane w C++) funkcji zawierających wstawki asemblerowe Autora. Ten kod zaś korzysta ze stosu, również czytając i pisząc pod adresami leżącymi w czerwonej strefie. Doprowadza to do uszkodzenia owych zmiennych lokalnych i błędu segmentacji.

Diagnoza problemu była niezwykle kłopotliwa – Autor nie potrafił uzasadnić, dlaczego zupełnie zwyczajne operacje na stosie uszkadzają zmienne sterujące pętli. Trop wskazała analiza zdeasemblowanego prologu funkcji, a zwłaszcza wskaźnika stosu, którego wartość zmieniała się o liczbę mniejszą, niż rozmiar zmiennych lokalnych. Ostatecznie sprawę rozwiązano wyłączając korzystanie z czerwonej strefy przy kompilacji jednostek kodu zawierających wstawki asemblerowe (opcja linii poleceń `-mno-red-zone` w przypadku GCC).

8.2.3. Przewidywanie ponownego wykorzystania

Jedną z optymalizacji stosowanych przez kompilator GCC na trzecim poziomie optymalizacji (opcja linii poleceń `-O3`) jest przewidywanie ponownego wykorzystania (ang. *predictive commoning*). Jest to standardowe usprawnienie polegające na analizie pętli w poszukiwaniu zmiennych nie ulegających zmianom pomiędzy iteracjami, w celu uniknięcia niepotrzebnych transferów danych między rejestrami a pamięcią RAM. Wyczerpujący opis techniki można znaleźć w artykule [13].

Trzeci poziom optymalizacji jest stosowany przez Autora w konstrukcji biblioteki w konfiguracji Release. Sprawiał on kłopot w działaniu biblioteki – cząstki w niektórych klatkach symulacji przestawały być emitowane bez wyraźnego powodu. Autor nie zdążył zgłębić przyczyny problemu, aczkolwiek przypuszcza on, iż jego źródłem są wstawki asemblerowe umieszczone wewnątrz ciał pętli. GCC posiada instrumenty, by wskazać rejestry procesora wykorzystywane przez daną wstawkę, co powinno pozwolić mu na zachowanie ich wartości przed wejściem w kod wstawki, a Autor z tych instrumentów korzysta; możliwe jednak, iż robi to niepoprawnie, lub informacja ta jest dla algorytmu optymalizującego niedostępna lub niewystarczająca.

Diagnoza polegała na ręcznym włączaniu konkretnych optymalizacji poziomu trzeciego, eliminując te, które nie sprawiały kłopotów. Po wyizolowaniu powodu Autor zdecydował się obejść problem wyłączając tę konkretną technikę w konfiguracji Release (opcja linii poleceń `-fno-predictive-commoning`).

8.3. Możliwe kierunki rozwoju

Prace w ramach projektu zaowocowały działającą implementacją przedstawionej koncepcji, natomiast przez ograniczenia czasowe Autor nie był w stanie wprowadzić w życie wszystkich swoich pomysłów.

8.3.1. Optymalizacje

Kod silnika daleki jest od idealnego – wiele miejsc można jeszcze zoptymalizować i usprawnić. Przykładowo, algorytm wyboru miejsca w buforze dla emitowanej cząstki – w tej chwili wylosowanie początkowego indeksu, a następnie zwykłe wyszukiwanie liniowe – można zastąpić którymś z algorytmów znanych z tablic mieszających.

O co najmniej jedną instrukcję przesyłu międzyrejestrowego można byłoby skrócić kod prawie każdego modułu, gdyby generator kodu znał docelowy rejestr dla wartości rozkładu (w tej chwili wynik trafia do rejestru tymczasowego i dopiero stamtąd moduł przesyła go do docelowego). Nie przyniosłoby to znaczącego zysku, aczkolwiek przy bardziej złożonych emiterach mógłby on się uwidocznić i pozwolić uszczknąć kolejnych kilka cennych cykli procesora.

Projekt jest również „obciążony” korzystaniem z jednostki koprocesora matematycznego (x87) do obliczeń na skalarach. Jest ona dostępna w obecnych procesorach, jednakże jej wydajność jest niższa, niż skalarnych instrukcji SSE, jej stosowanie – niezalecane (na rzecz tychże instrukcji SSE) i można się spodziewać, że kiedyś zniknie z chipów, aby zrobić miejsce dla innych obwodów.

8.3.2. Usunięcie zależności od NASM

Zaplecze asemblera x86 w tej chwili zależy od obecności asemblera NASM w systemie klienta. Już po zakończeniu implementacji Autor natknął się na projekt *AsmJit* [14], czyli asemblera techniki „w samą porę” (ang. *Just-In-Time*; JIT) w formie

biblioteki języka C++. Jego funkcjonalność polega na budowie wykonywalnego kodu w czasie wykonania. Korzyści wynikające z jego potencjalnego użycia są więc co najmniej dwojakie:

- usunięcie zależności od zewnętrznego oprogramowania, jakim jest NASM,
- usunięcie poważnego wektora ataku hakerskiego – w obecnej architekturze zaplecza x86 atakujący mógłby podmienić plik wykonywalny NASM i wstrzyknąć w ten sposób w aplikację-klienta złośliwy kod.

8.3.3. Więcej zapleczy

Jest jeszcze wiele innych architektur, które mogą być docelową platformą dla gier komputerowych i innych aplikacji graficznych korzystających z systemów cząstek. Na wsparciu kolejnych zapleczy (platform) projekt może tylko zyskać:

- procesory Power Architecture, napędzające dominujące na rynku konsole do gier takie, jak Xbox 360, PlayStation 3 czy Nintendo Wii U [15],
- procesory ARM, napędzające całą gamę urządzeń mobilnych,
- czy wreszcie wspomniane na początku pracy programy GPGPU.

8.3.4. Graficzny edytor konfiguracji emiterów

Przystępny dla artysty edytor był jednym z pierwszych pomysłów na projekt inżynierski Autora. Obecnie silniki gier komputerowych oferują własne, rozbudowane edytory, dlatego też dobry edytor dla biblioteki narzędziowej (*middleware*) powinien dobrze się z nimi integrować.

8.3.5. Dodatkowa funkcjonalność

Wreszcie silnik mógłby zostać uzupełniony o kilka pomniejszych funkcjonalności, które zwiększyłyby jego atrakcyjność: emisja innych prymitywów geometrycznych, niż kwadratowe billboardy (np. punktów, prostokątów, wstęg), czy też rozdział układu współrzędnych emitera od układu współrzędnych cząstek.

9. Podsumowanie

W ramach prac udało się zrealizować kluczowe założenia projektu – stworzono modularną bibliotekę, której zaplecze asemblera x86 uzyskuje dużo wyższą wydajność, niż analogiczna, napisana w języku C++. Całość prac jest udokumentowana historią zmian w publicznie dostępnym repozytorium kodu źródłowego w serwisie GitHub [16], a sam kod jest wolnym i otwartym oprogramowaniem.

Projekt był dla Autora okazją, aby uzupełnić wiedzę o architekturze procesorów rodziny x86 i optymalizacjach kompilatorów, a także o mechanizmach platformowych w dwóch rodzinach systemów operacyjnych. Nie można go jednak uznać za zamknięte dzieło – pozostało jeszcze wiele niezrealizowanych pomysłów, błędów do wykrycia i poprawienia oraz miejsc, które można usprawnić. Dotknięta problematyka leży w sferze prywatnych zainteresowań Autora i ma on zamiar kontynuować pracę w swoim wolnym czasie.

Silnik w wyniku prac nadaje się do wstępnego użytku w aplikacjach klienckich. Może także zostać użyty do prototypowania efektów cząsteczkowych na potrzeby demosceny – wygenerowany kod asemblerowy można łatwo wykorzystać w innej aplikacji. Wprowadzenie w życie kilku ze wspomnianych w rozdziale *Możliwe kierunki rozwoju* pomysłów (s. 43), zwłaszcza graficznego edytora, może szybko i w znaczący sposób podnieść wartość projektu, umożliwiając szybkie wdrożenie w produkcji gry komputerowej lub innej aplikacji multimedialnej przeznaczonej na komputery PC.

Bibliografia

[1] Wikipedia, the free encyclopedia, "*Particle system*". [Online]. Dostępne pod adresem: http://en.wikipedia.org/wiki/Particle_system#Simulation_stage

[2] Tomasz Dąbrowski, "*Quick Data Transformation*". [Online]. Dostępne pod adresem: http://dabroz.scythe.pl/2010/11/18/quick-data-transformation/

[3] Ingo Berg et al., "*muParser*". [Online]. Dostępne pod adresem: http://muparser.beltoforion.de/

[4] Ingo Berg et al., "*muParserSSE*". [Online]. Dostępne pod adresem: http://beltoforion.de/muparsersse/math_expression_compiler_en.html

[5] NVIDIA Corporation, "*APEX FAQ*". [Online]. Dostępne pod adresem: https://developer.nvidia.com/apex-faq

[6] Leszek Godlewski, "*Particlasm: A modular, data-driven particle system written in x86 assembly*", Politechnika Śląska, Gliwice, 2012. Dostępne pod adresem: https://github.com/inequation/particlasm/tree/1.0

[7] Microsoft Corporation, "*Microsoft Developer Network*", 2012

[8] The Linux man-pages project, "*Linux Programmer's Manual*", 2008

[9] GitHub, "*Commit History · inequation/particlasm*". [Online]. Dostępne pod adresem: https://github.com/inequation/particlasm/commits/1.x

[10] Robert Neff, "*The NASMX Project*". [Online]. Dostępne pod adresem: http://www.asmcommunity.net/projects/nasmx/

[11] Intel Corporation, "*Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z*", 2012

[12] Michael Matz, Jan Hubicka, Andreas Jaeger i Mark Mitchell, "*System V Application Binary Interface. AMD64 Architecture Processor Supplement*", 2010

[13] Arie Tal, "*Second-Order Predictive Commoning*". [Online]. Dostępne pod adresem: http://webdocs.cs.ualberta.ca/~amaral/cascon/CDP04

[14] Petr Kobalicek, "*AsmJit. Complete x86/x64 JIT Assembler for C++ Language*". [Online]. Dostępne pod adresem: http://code.google.com/p/asmjit/

[15] IBM China Development Lab, "*POWER to the people. A history of chipmaking at IBM*". [Online]. Dostępne pod adresem: http://ibm.com/developerworks/power/library/pa-powerppl

[16] Leszek Godlewski, "*Particlasm 2: A modular, data-driven particle system with retargetable back-ends*". [Online]. Dostępne pod adresem: http://github.com/inequation/particlasm

Indeks ilustracji

Ilustracja 1: System cząstek symulujący płomień ognia; Źródło: Wikipedia (http://en.wikipedia.org/wiki/File:Particle_sys_fire.jpg).....6

Ilustracja 2: Billboarding; Normalne cząstek Pn równoległe z osią wzroku obserwatora v.....6

Ilustracja 3: Przykład modularnego systemu cząstek; Na czerwonym tle moduły wykonywane w czasie emisji; Na żółtym – w czasie istnienia; Na zielonym – specjalny moduł symulacji, który wykonuje równania kinematyki Newtona.....8
Ilustracja 4: Czas wykonania prostego efektu cząsteczkowego względem implementacji w C++ (im mniej tym lepiej); Słupki "asm" i "asm, vectorized" to ręcznie zoptymalizowany dla przypadku testowego kod asemblerowy działający odpowiednio na skalarach i wektorach; QDT to omawiane w artykule autorskie rozwiązanie Tomasza Dąbrowskiego [2].....10
Ilustracja 5: Unreal Cascade podczas edycji efektu cząsteczkowego; emitery w prawej górnej części okna są reprezentowane przez łańcuchy modyfikatorów; w prawej dolnej części ekranu znajduje się edytor wykresów, służący do kontrolowania zmian wartości w czasie.....12
Ilustracja 6: Schemat konstrukcji kodu emitera w pierwotnej wersji biblioteki; w pierwszym kroku (strona lewa) w buforze umieszczane są dane (kolorowe kwadraty na górze schematu) i moduły z pustymi adresami; w kroku następnym (strona prawa) adresy w modułach są wypełniane.....13
Ilustracja 7: Ogólny schemat architektury silnika.....26
Ilustracja 8: Diagram współpracy klasy X86Generator.....26
Ilustracja 9: Diagram współpracy klasy X86Launcher.....27
Ilustracja 10: Diagram dziedziczenia klasy X86ModuleInterface.....27
Ilustracja 11: Przykładowa konfiguracja emitera, realizowana przez aplikację hostapp, stworzoną jako demonstrator projektu.....32
Ilustracja 12: Wykres zależności czasu symulacji 1800 klatek od rozmiaru bufora....38

Indeks tabel

Tabela 1: Wyniki pomiaru czasu symulacji 1800 klatek implementacją C++ w zależności od rozmiaru bufora cząstek.....35
Tabela 2: Wyniki pomiaru czasu symulacji 1800 klatek implementacją asemblera x86 w zależności od rozmiaru bufora cząstek.....35
Tabela 3: Wyniki pomiaru czasu symulacji 1800 klatek implementacją C++ w zależności od rozmiaru bufora cząstek.....36
Tabela 4: Wyniki pomiaru czasu symulacji 1800 klatek implementacją asemblera x86-64 w zależności od rozmiaru bufora cząstek.....36
Tabela 5: Wyniki pomiaru czasu symulacji 1800 klatek implementacją C++ w zależności od rozmiaru bufora cząstek.....37
Tabela 6: Wyniki pomiaru czasu symulacji 1800 klatek implementacją asemblera x86 w zależności od rozmiaru bufora cząstek.....37

Załącznik 1. Opis zawartości dołączonej płyty CD

- Kod źródłowy projektu
- Pliki wykonywalne przykładowych aplikacji dla platform Linux i386, Linux amd64 i Windows 32-bit
- Skompilowane dynamicznie łączone biblioteki dla platform Linux i386, Linux amd64 i Windows 32-bit
- Tekst pracy
- Dane źródłowe z wynikami eksperymentów