Dariusz Maciejewski
Nr. Indeksu: 188784

– Camera Fighter –
Gra komputerowa

Promotor: prof. nzw. dr hab. Przemysław Rokita

Pracownia dyplomowa magisterska 1

- raport -

Opis zadania:

Camera Fighter ma być grą umożliwiającą przeprowadzanie wirtualnych pojedynków oraz wspierającą naukę sztuk walki. Zestaw kamer będzie przechwytywał zdjęcia gracza. Po ich analizie, program rozpozna pozycję jaką przyjął gracz. W pozycji takiej zostanie pokazany wirtualny wojownik. Będzie to umożliwiało nowatorskie sterowanie postacią oraz w efekcie przeprowadzanie wirtualnych walk z innymi graczami, bądź naukę poprawnych pozycji i ruchów w wybranych sztukach walki.
Moim zadaniem jest stworzenie logiki, fizyki oraz silnika 3D napędzającego całą grę. Przygotowany przeze mnie program zostanie następnie zintegrowany z przygotowywaną przez kolegę Michała Grędziaka biblioteką, odpowiedzialną za rozpoznawanie pozycji gracza.

Stan pracy na początku trzeciego semestru:

Na początku drugiego semestru posiadałem gotowy szkielet uniwersalnej gry komputerowej. Pozwalał on na utworzenie pustej aplikacji OpenGL oraz wspierał podział kodu na niezależne od siebie logicznie sceny, takie jak menu, tryb rozgrywki, konsola czy konfiguracja.
Ponadto miałem przygotowany system sterowania kamerami z wykorzystaniem matematyki kwaternionów oraz własnego algorytmu obliczania obrotów.
Gotowa była również klasa reprezentująca bazowy obiekt w grze, z którego obecnie wywodzą się różne typy modeli.
Podczas drugiego semestru szkielet programu został poszerzony o klasę zarządzającą wejściem (InputManager), a także o uniwersalnego menadżera uchwytów i wykorzystujące go menadżery tekstur oraz czcionek.
Silnik miał możliwość wyświetlania modeli zaimportowanych z plików .3ds oraz własnego formatu o rozszerzeniu .3dx. Modele były przesyłane do karty graficznej z wykorzystaniem wydajnych funkcji z rodziny gl*Pointer i glDrawElements. Ich dalsze przetwarzanie po stronie karty graficznej można było modyfikować z wykorzystaniem OGL vertex i fragment shaderów.

Zakres prac wykonanych w trzecim semestrze:

1. Animacja szkieletowa modeli

W bieżącym semestrze dodałem możliwość animacji szkieletowej modeli. Wymagało to stworzenia odpowiednich struktur danych oraz zaimplementowania algorytmów działających po stronie vertex shadera oraz po stronie programu. Animowanie postaci po stronie programu jest wymagane przez starsze karty graficzne, system detekcji kolizji, a także do wydajniejszego zaznaczania obiektów po stronie karty graficznej (selection rendering 'nie lubi' shader'ów).

Gra w akcji

2. Edytor szkieletu oraz animacji szkieletowych

Ponieważ animacja szkieletowa nie składa się tylko z samego algorytmu i struktur danych, stworzyłem wbudowany edytor modeli, który umożliwia stworzenie szkieletu, przypisanie mu wierzchołków ('skórowanie modelu') oraz stworzenie zestawu animacji dla tak przygotowanego modelu.

Po lewej: Edycja szkieletu oraz skórowania; Po prawej: Edycja animacji szkieletowej

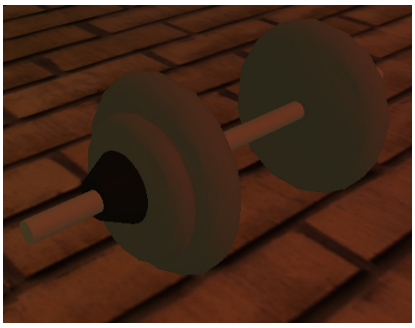
3. Dalsza optymalizacja renderowania za pomocą rozszerzenia VBO

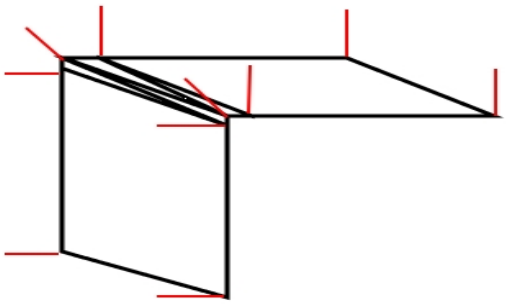
Rozszerzenie Vertex Buffer Object umożliwia znaczne obniżenie ilości danych wysyłanych podczas renderowania każdej klatki do karty graficznej. Dzięki niemu dane trzeba wysyłać tylko wtedy, gdy zmieni się kształt modelu, a jeśli animacja przebiega po stronie karty graficznej (w programie vertex shadera), to nawet wtedy wystarczy wysłać tylko nowe parametry samej animacji.

4. Zaimplementowanie gładkiego i płaskiego cieniowania.

Aby modele były ładniej cieniowane, nie wyglądały jakby składały się z trójkątów, a z obłych elementów, musiałem zaimplementować gładkie cieniowanie. Poprzednio normalne były zdefiniowane dla każdego trójkąta – jest to tak zwane płaskie cieniowanie, cała powierzchnia odbija światło pod tym samym kątem i wygląda płasko. Jeśli normalne będą przekazywane oddzielnie dla każdego wierzchołka, to

uzyskamy gładkie cieniowanie – oświetlenie będzie się różnie odbijało w różnych punktach trójkąta (normalna jest interpolowana), dając złudzenie obłości.
Gładkie cieniowanie jest wspierane przez silnik OpenGL i nie wymaga wielkiego nakładu pracy. Jest to naturalny sposób postrzegania ciał w rzeczywistości, gdyż w naturze praktycznie nie występują ostre krawędzie.
Jeśli chcemy uzyskać płaskie cieniowanie, np. dla pokazania krawędzi kostki składającej się z dwóch trójkątów na ścianę, to musimy użyć płaskiego cieniowania. Ponieważ OpenGL nie pozwala na definiowanie normalnych dla całych trójkątów, najwydajniejszym rozwiązaniem jest skopiowanie wspólnych wierzchołków i nadanie każdemu innej normalnej. Renderowanie każdego trójkąta oddzielnie, zapobiega duplikacji wierzchołków, ale jest mało wydajne, gdyż wymusza wielokrotne wołanie polecenia renderowania.
Bliższym naturze rozwiązaniem byłoby dla np. skrzyni utworzenie dodatkowych, wąskich trójkątów na granicy ścian. Całość byłaby renderowana gładko, a same krawędzie lekko by się zaokrąglały. Stopień ostrości zależny byłby od szerokości trójkątów.





Po lewej: Gładkie cieniowanie; Po prawej: Płaskie cieniowanie w naturze (z wykorzystaniem gładkiego cieniowania)

5. Próba uniezależnienia szkieletu aplikacji od silnika renderującego

Postarałem się o maksymalne rozdzielenie logiki aplikacji od silnika OpenGL. Większość uniwersalnych klas wchodzących w interakcje z OpenGL została zmodyfikowana tak, aby łatwo było je zastąpić wersją dla silnika DirectX. Logika aplikacji widzi uniwersalne interfejsy, nie mając świadomości o tym, jaki silnik jest odpowiedzialny za rendering.

6. Detektory kolizji

Stworzyłem dwa detektory kolizji. Pierwszy wykrywa kolizje między parami trójkątów. Jest on niezbędny do wykrywania kolizji jakie zachodzą między modelami. Algorytm bazuje na rozwiązaniu *Faster-Triangle Intersection Tests* zaproponowanym przez Oliviera Devillers i Philippe Guigue z INRIA. Algorytm został poprzedzony wyborem kandydatów do testu na podstawie analizy drzew Octree każdego modelu.
Zastosowanie techniki Octree przyspiesza detekcję kolizji kilkaset razy. Dla dwóch modeli składających się z kilku tysięcy trójkątów prędkość wzrosła z kilku sekund na klatkę do 50-60 klatek na sekundę, czyli blisko maksimum narzucanego przez proces renderowania (60 FPS).
Drugi detektor kolizji jest detektorem typu trójkąt-promień. Wykrywa on kolizje modelu z zadanym promieniem, np wysłanym przez kursor myszki podczas kliknięcia.

7. Fizyka ciał sztywnych

Posiadając detektor kolizji, mogłem przystąpić do implementacji fizyki. Zacząłem od najprostszego przypadku, czyli od ciał sztywnych. Zderzające się ciała przekazują sobie pęd oraz ewentualnie wpadają w ruch wirowy, gdy zderzenie wystąpiło pod kątem w stosunku do środka masy obiektu.
Algorytm zachowuje się całkiem ładnie przy małych prędkościach. Jednakże przy większych następuje zjawisko przebicia. Modele mogą się minąć nawet nie zauważywszy, iż powinno dojść do kolizji, lub wbiją się w siebie na tyle głęboko, że algorytm nie będzie wiedział jak zareagować na taką sytuację.
Aby uniknąć tych dwóch problemów będę musiał stworzyć dynamiczny detektor kolizji, który na podstawie wiedzy o ruchu ciał, będzie mógł przewidzieć potencjalne zderzenia po drodze i wyliczyć dla nich szczegółowe reakcje. Dodatkowo powinien on potrafić wyliczyć na podstawie ruchu głębokość i kierunek przebicia, by na podstawie tych danych silnik fizyczny mógł odpowiednio zareagować na taką sytuację.

Zderzenia prostych ciał sztywnych (ruch rozpoczyna kostka po prawej)

Plany na kolejny semestr:

Najważniejszymi obecnie zadaniami jest dopracowanie fizyki ciał sztywnych oraz utworzenie fizyki ciał posiadających szkielet. Jeśli starczy czasu, warto by również zaimplementować fizykę ciał elastycznych, takich jak np. lina, trampolina itp.
Wciąż pozostaje do wykonania zintegrowanie gry z biblioteką Motion Capture, przygotowywaną przez Michała Grędziaka, odpowiedzialną za rozpoznawanie pozycji przyjętej przez gracza.
Warto byłoby rozszerzyć silnik graficzny o zaawansowane teksturowanie (mapy normalnych, wypukłości, przeźroczystości, itp). Miłe dla oka byłoby również wykonanie kilku silników cząsteczkowych (płomienie, dym).

Literatura:

Zagadnienia związane z programowaniem gier

'Perełki programowania gier – tom 1' pod redakcją Marca DeLoura, Wydawnictwo Helion 2002

OpenGL

The Red Book
http://fly.srk.fer.hr/~unreal/theredbook/

The Blue Book
http://www.rush3d.com/reference/opengl-bluebook-1.0/index.html

NeHe Productions
http://nehe.gamedev.net/

Jerome Lessons
http://jerome.jouvie.free.fr/OpenGl/Lessons.php

Jerome Tutorials
http://jerome.jouvie.free.fr/OpenGl/Tutorials1-5.php

Tulane.edu tutorial
http://www.eecs.tulane.edu/www/Terry/OpenGL/Introduction.html#Introduction

Dokumentacja rozszerzenia VBO
http://www.spec.org/gpc/opc.static/vbo_whitepaper.html

Dokumentacja rozszerzeń OpenGL
http://www.opengl.org/documentation/specs/man_pages/hardcopy/GL/html/

Animacja szkieletowa

Animacja szkieletowa z wykorzystaniem shaderów wierzchołków
http://grafika.iinf.polsl.gliwice.pl/doc/09-SKE.pdf

Podstawy animacji komputerowej
http://sound.eti.pg.gda.pl/student/sdio/12-Animacja.pdf

Detekcja kolizji

'Faster-Triangle Intersection Tests' autorstwa Oliviera Devillers i Philippe Guigue z INRIA
http://citeseer.ist.psu.edu/687838.html
ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-4488.pdf