Dariusz Maciejewski
Nr. Indeksu: 188784

– Camera Fighter –
Gra komputerowa

Promotor: prof. nzw. dr hab. Przemysław Rokita

Pracownia dyplomowa magisterska 2

- 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 czwartego semestru:

W pierwszym semestrze stworzyłem 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 programów GLSL działających na procesorze karty graficznej.

W trzecim semestrze zaimplementowałem algorytm animacji szkieletowej po stronie CPU jak i GPU. Stworzyłem również wbudowany edytor szkieletu i animacji bazujących na jego ruchu.
W celu poprawienia wydajności renderowania, dodałem do silnika graficznego obsługę rozszerzenia Vertex Buffer Object. Umożliwia ono przechowywanie danych modelu po stronie karty graficznej, dzięki czemu renderowanie przebiega znacznie szybciej.

Aby poprawić jakość wizualną modeli, zaimplementowałem gładkie cieniowanie z wykorzystaniem grup cieniowania trójkątów.
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.
Na koniec trzeciego semestru miałem także gotowe podwaliny silnika fizycznego – efektywny detektor kolizji oraz symulator zachowania ciał sztywnych.

Zakres prac wykonanych w czwartym semestrze:

A. Grafika

1. Poprawiony algorytm wyliczania grup cieniowania

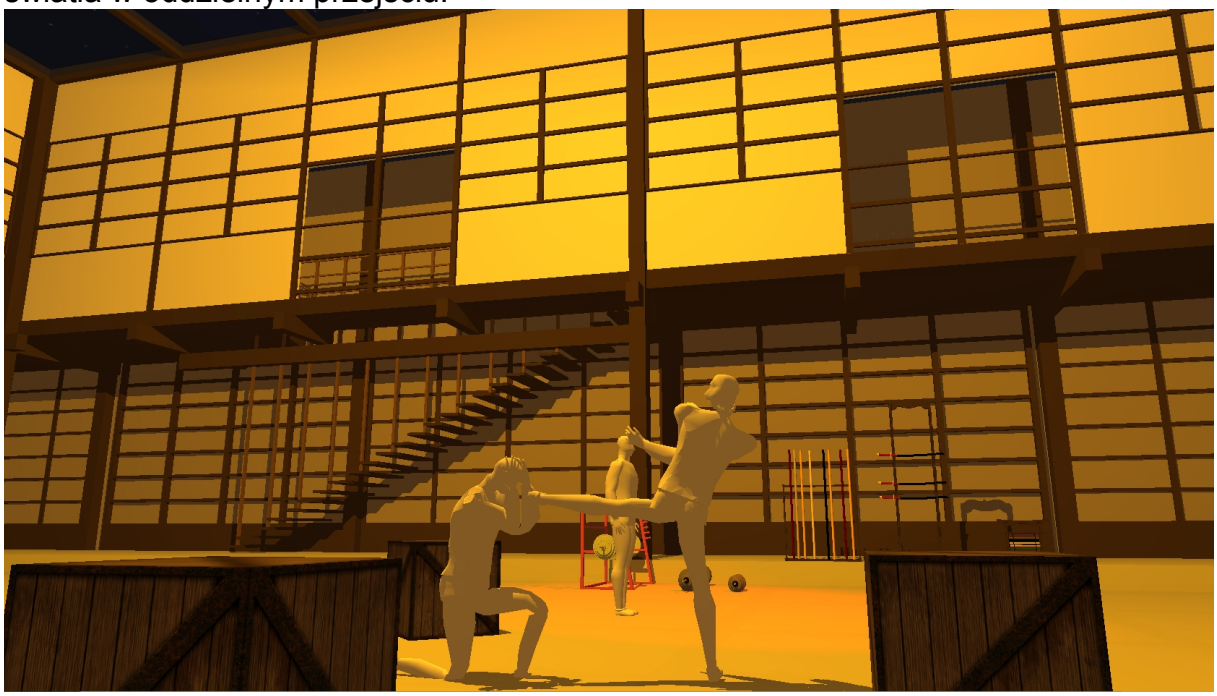
Wprowadziłem korekty poprawiające logikę algorytmu oraz wydajność renderowania. Trójkąty modelu dzielone są na paczki o jednolitych właściwościach (typ materiału oraz grupy cieniowania), dzięki czemu można renderować dużą grupę trójkątów za pomocą pojedynczych poleceń OpenGL.

2. Rozbudowa materiałów

Dodałem możliwość renderowania przeźroczystych oraz dwustronnych trójkątów, a także położyłem podwaliny pod dalszą rozbudowę definiowania materiałów (definiowanie shader'ów dla materiału, itp.)

3.Cieniowanie

Zaimplementowałem dwie techniki cieniowania – z wykorzystaniem mapowania cieni (Shadow Mapping) oraz brył cieni (Shadow Volumes). W obecnej wersji wykorzystuję tę drugą technikę – wolniejszą, ale dającą dużo ładniejsze rezultaty. W celu jej zaimplementowania musiałem również zdecydować się na renderowanie każdego światła w oddzielnym przejściu.



4.Wyspecjalizowane shadery dla różnych typów świateł

W ramach optymalizacji renderowania wieloprzejściowego, rozbiłem uniwersalny program OGSL, na wiele prostszych programów, wyspecjalizowanych do określonego typu światła.

5.Próby optymalizacji renderowania

Wieloprzejściowe renderowanie sceny ma wpływ na spory spadek wydajności. Dlatego dużo czasu poświęciłem na próby zoptymalizowania tego procesu. Niestety efekty wciąż są dalekie od zadowalających. Zastanawiam się nad próbą zaimplementowania 'renderingu odroczonego' (Deffered Rendering), które pozwala na ograniczenie wielokrotnego renderowania tych samych obiektów. Jednakże z tego udało mi się dowiedzieć, większość gier bez problemu radzi sobie używając renderowania wieloprzejściowego, więc mam nadzieję, że uda mi się zadowalająco zoptymalizować bieżące rozwiązanie.

B. Fizyka

1. Wyliczanie głębokości kolizji.

Aby poszerzyć wiedzę silnika fizycznego o kolizji i dzięki temu poprawić reakcję na zderzenia, uwzględniłem w algorytmach fizycznych głębokości przebicia obiektów.

2.Uproszczone modele fizyczne.

Wprowadziłem możliwość definiowania uproszczonych modeli 3d, które nie widoczne podczas rozgrywki, a jedynie wykorzystywane podczas przetwarzania fizyki. Pozwala to na wielokrotne zmniejszenie ilości trójkątów, które musi przetworzyć detektor kolizji, co wpływa na znaczną poprawę wydajności silnika fizycznego.

3. Fizyka szkieletu

Zaimplementowałem algorytm symulacji fizyki dla modeli posiadających szkielet. Bazuje on na silniku całkującym Verlet'a wykorzystanym po raz pierwszy w grze Hitman: Codename 47. Model opisany jest przez zbiór cząsteczek podstawowych (dla modelu ze szkieletem są to stawy) na które nałożono ograniczenia (głównie ograniczenia kątów między sąsiednimi kośćmi). Gdy jakaś cząsteczka (staw) zostanie przemieszczona, pozycja wszystkich cząsteczek jest poprawiana poprzez proces relaksacji, tak by jak najlepiej spełnić wszystkie nałożone ograniczenia.

4. Fizyka ciał sztywnych - przeprojektowanie

Przebudowałem symulator fizyki ciał sztywnych by również bazował na silniku Verlet'a. Dla dowolnego ciała sztywnego wystarczy zdefiniować cztery cząsteczki podstawowe, ustawione w formie czworościanu bądź układu współrzędnych (tę konwencję przyjąłem). Takie rozmieszczenie zapewnia możliwość opisania wszystkich stopni swobody w ruchu modelu.
Podejście Verlet'a bardzo upraszcza opis fizyczny ruchu. Pojedyncza cząsteczka nie posiada objętości, więc nie może się obracać wokół własnej osi – jedyny ruch jaki musimy symulować to ruch po prostej. Pozycja całego modelu definiowana jest przez wzajemne położenie symulowanych cząsteczek.

C. Architektura aplikacji

1. Wczytywanie scen z plików

Zdefiniowałem prosty format tekstowego opisu sceny, dzięki czemu mapy mogą być wczytywane z plików. Dodatkowo utworzona została przykładowa scena – 'dojo'. Na wykończeniu jest również hala do walk sumo.

2. Porządki w kodzie, dostosowanie kodu dla Linuxa

Kod był kilkakrotnie porządkowany, co jakiś czas testuję także kod pod systemem Linux, aby gra dała się kompilować również dla tego systemu.

3. Interfejs wejściowy dla zewnętrznego sterowania postacią

Stworzyłem dwie klasy umożliwiające pobieranie pozycji gracza z zewnętrznych źródeł sterowania, takich jak sieć (dla gry wieloosobowej) oraz biblioteka wykrywania ruchu pisana przez Michała Grędziaka. Michał ma obecnie możliwość testowania swojej pracy poprzez te interfejsy.

Plany na najbliższy czas:

Najważniejszymi obecnie zadaniami jest zaprogramowanie logiki aplikacji, działającej na stworzonych silnikach graficznym i fizycznym. Zderzenia powinny zadawać postaci obrażenia o sile zależnej od punktu oraz siły uderzenia. Wypadało by dodać całości również ładne opakowanie – menu główne, ekran opcji.
Drugą ważną rzeczą jest optymalizacja procesu renderowania, by gra działała zadowalająco na starszych komputerach – sceny nie są bardzo rozbudowane w porównaniu z komercyjnymi grami, więc powinno dać się je znacznie szybciej renderować.
Warto byłoby rozszerzyć silnik graficzny o zaawansowane teksturowanie (mapy normalnych, wypukłości, przeźroczystości, itp). Miłe dla oka byłoby również wykonanie kilku silników cząsteczkowych (płomienie, dym) oraz dodanie fizyki dla ciał elastycznych takich jak liny, trampoliny, materiały lub roślinki (dzięki zastosowaniu silnika Verlet'a jest to niezwykle proste).

Literatura:

Zagadnienia związane z programowaniem gier

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

OpenGL

The Red Book, The Blue Book
http://fly.srk.fer.hr/~unreal/theredbook/
http://www.rush3d.com/reference/opengl-bluebook-1.0/index.html

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

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

Collision Detection - Wikipedia
http://en.wikipedia.org/wiki/Collision_detection

'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

Fizyka

Advanced Character Physics – Verlet Integration
http://www.gamasutra.com/resource_guide/20030121/jacobson_01.shtml

Physically Based Modeling: Principles and Practice
http://www.cs.cmu.edu/~baraff/sigcourse/

Ragdoll Physics – Wikipedia
http://en.wikipedia.org/wiki/Ragdoll_physics