

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

Perełki programowania gier. Vademecum profesjonalisty. Tom 1

Autor: Mark DeLoura

Tłumaczenie: Rafał Jońca

ISBN: 83-7197-704-2

Tytuł oryginału: [Game Programming Gems](#)

Format: B5, stron: 638



W niniejszej książce znajdziesz połączoną wiedzę ponad 40 utalentowanych twórców gier. Współpracując, stworzyli zbiór wskazówek dotyczących programowania gier, dzięki któremu uzupełnisz swoją wiedzę. Jeśli zaimplementujesz zaprezentowane tutaj techniki (wypracowywane przez wiele godzin), wrogowie będą sprytniejsi, bohater płynnie powali przeciwników, a gracze z powodu wysoce realistycznego trójwymiarowego świata będą się bali zgasić światło w nocy.

Niezależnie od tego, czy pytali mnie o nowinki techniczne wprowadzone w nowej konsoli, czy o złożone algorytmy, jedna rzecz stała się dla mnie jasna: ciągle zadajemy pytania. Jako programiści gier często nie wiemy, jak wykonać postawione przed nami zadanie. Może właśnie dlatego tak bardzo lubimy tę pracę! Co masz jednak zrobić, gdy przytrafi Ci się opisana sytuacja? Przeszukasz domową biblioteczkę lub zasoby sieci WWW? A może zajrzysz do archiwalnych numerów fachowych czasopism? Żaden twórca gier nie korzysta z jednego określonego źródła. Czy nie byłoby wspaniale, gdyby jednak istniało takie miejsce, do którego zawsze zajrzysz w pierwszej kolejności? Właśnie w tym celu napisaliśmy książkę, którą trzymasz w ręce.

Rozdziały książki obejmują wiele problemów technicznych, na które możesz się natknąć, pisząc grę. Znajdziesz ogromną liczbę szczegółowo omówionych technik, ale i kilka bardziej ogólnych rozdziałów. Zadaniem książki jest zwiększenie Twojego stopnia zaawansowania niezależnie od aktualnej wiedzy, jaką posiadasz. Na przykład w bardziej ogólnych rozdziałach opisujemy techniki, nie zagłębiając się w szczegóły; na ich omówienie czas przychodzi później. Dobrymi przykładami mogą być rozdziały o kwaternionach oraz część dotycząca algorytmów sztucznej inteligencji.



Spis treści

Podziękowania.....	13
Spis autorów	15
Przedmowa.....	17
O obrazku z okładki	20
Część I Techniki programistyczne	21
Rozdział 1.0 Magia sterowania danymi.....	23
Zasada 1.: Podstawy	23
Zasada 2.: Całkowite minimum.....	23
Zasada 3.: Twórz elastyczne algorytmy	24
Zasada 4.: Do sterowania przebiegiem wykorzystuj skrypty	24
Zasada 5.: Gdy dobre skrypty stają się złymi.....	25
Zasada 6.: Unikaj duplikacji danych	26
Zasada 7.: Kreuj narzędzia, które tworzą dane.....	26
Wnioski	27
Rozdział 1.1 Programowanie obiektowe i techniki projektowania	29
Styl programowania.....	30
Projektowanie klas.....	32
Projektowanie hierarchii klas	33
Wzorce projektowania.....	33
Podsumowanie.....	39
Rozdział 1.2 Szybkie obliczenia matematyczne z wykorzystaniem szablonów....	41
Ciąg Fibonacciego	41
Silnia.....	43
Trygonometria	44
Kompilatory w rzeczywistym świecie.....	45
Jeszcze raz trygonometria.....	45
Szablony i standard C++	46
Macierze	46
Podsumowanie.....	51
Rozdział 1.3 Automatyczne singletony	55
Definicja	55
Zalety.....	55
Problem	56
Tradycyjne rozwiązanie.....	56
Lepszy sposób	56
Jeszcze lepszy sposób.....	57

Rozdział 1.4	Używanie biblioteki STL w programowaniu gier	59
	Rodzaje elementów w STL	59
	Podstawowe pojęcia dotyczące biblioteki STL	60
	Wektory	61
	Listy	63
	Kolejki dwukierunkowe	66
	Mapy	67
	Stosy, kolejki i kolejki priorytetowe	70
	Podsumowanie	71
Rozdział 1.5	Ogólny interfejs dowiązywania funkcji	73
	Wymagania	73
	Platformy sprzętowe i programowe	74
	Pierwsze rozwiązanie	74
	Drugie rozwiązanie	75
	Półowa rozwiązania	77
	Sposoby wywoływania funkcji	77
	Wywoływanie funkcji	79
	Uzupełnianie rozwiązań	80
	Wnioski	83
Rozdział 1.6	Ogólny system zarządzania zasobami oparty na uchwytach	85
	Metoda	86
	Klasa Handle	87
	Klasa HandleMgr	87
	Przykład użycia	89
	Uwagi	89
Rozdział 1.7	Zarządzanie zasobami i pamięcią	97
	Klasa zasobów	97
	Klasa menedżera zasobów	100
	Jak działają uchwyty	102
	Możliwe modyfikacje i rozszerzenia	103
	Wnioski	104
Rozdział 1.8	Sztuczka z szybkim wczytywaniem danych	105
	Wcześniej przetwórz dane	105
	Zapisywanie danych	106
	Prosty sposób wczytywania danych	107
	Bezpieczniejsze wczytywanie danych	107
Rozdział 1.9	Alokacja pamięci oparta na ramkach	109
	Problemy tradycyjnej alokacji pamięci	109
	Wprowadzenie do pamięci opartej na ramkach	109
	Alokacja i zwalnianie pamięci	111
	Przykład	114
	Wnioski	115
Rozdział 1.10	Proste i szybkie tablice bitów	117
	Ogólny opis	117
	Tablica bitów	117
	Pozostałe tablice bitów	118
	Wnioski	119

Rozdział 1.11	Protokół sieciowy w grach internetowych	121
	Definicje	121
	Modyfikacja pakietów	122
	Atak metodą powtarzania pakietów	122
	Dodatkowe zabezpieczenia	124
	Reinżynieria	124
	Implementacja	124
Rozdział 1.12	Maksymalne wykorzystanie makra assert	127
	Podstawy weryfikacji warunków	127
	Pierwsza sztuczka: osadzanie dodatkowych informacji	128
	Druza sztuczka: osadzanie jeszcze większej liczby informacji	129
	Trzecia sztuczka: upraszczanie zapisu	129
	Czwarta sztuczka: napisz własne makro	129
	Piąta sztuczka: dodatkowa opcja prawie bez kosztów	130
	Szosta sztuczka: tylko, gdy jesteś twardy	130
	Siódma sztuczka: kopiowanie i wklejanie, czyli ułatwianie sobie życia	131
Rozdział 1.13	Statystyki i testowanie gier w czasie rzeczywistym	133
	Dlaczego: technologia sterowana potrzebami	133
	Jak: ewolucyjny proces	134
	Co: system oparty na klasach języka C++	134
	Gdzie: zastosowania	137
	Podsumowanie	137
Rozdział 1.14	System profilujący działający w czasie rzeczywistym	139
	Przechodzimy do szczegółów	140
	Czego dowiesz się za pomocą procedury profilującej?	140
	Dodawanie wywołań funkcji kodu profilującego	142
	Implementacja procedury profilującej	142
	Szczegóły dotyczące ProfileBegin	143
	Szczegóły dotyczące ProfileEnd	144
	Przetwarzanie uzyskanych danych	144
	Możliwe udoskonalenia	144
	Łączymy wszystko razem	145
Część II Matematyka		151
Rozdział 2.0	Przewidywalne liczby losowe	153
	Przewidywalne liczby losowe	153
	Alternatywne algorytmy	155
	Algorytmy dla nieskończonych wszechświatów	156
	Wnioski i wskazówki	158
Rozdział 2.1	Metody interpolacji	161
	Zależne od częstotliwości generowania klatek łagodne zakończenie ruchu z wykorzystaniem liczb zmiennoprzecinkowych	161
	Zależne od częstotliwości generowania klatek łagodne zakończenie ruchu z wykorzystaniem liczb całkowitych	162
	Interpolacja liniowa niezależna od częstotliwości generowania klatek	163
	Łagodne rozpoczęcie i zakończenie ruchu niezależne od częstotliwości generowania klatek	164
	Niebezpieczeństwa	165

Rozdział 2.2	Całkowanie równań ruchu ciała sztywnego	169
	Kinematyka — przesunięcie i obrót	169
	Dynamika — siła i moment obrotowy	172
	Dodatkowe właściwości ciała sztywnego	173
	Całkowanie równań ruchu	176
Rozdział 2.3	Przybliżanie funkcji trygonometrycznych wielomianami	179
	Wielomiany	180
	Dziedzina i przeciwdziedzina	181
	Wielomiany parzyste i nieparzyste	184
	Szereg Taylora	185
	Skrócony szereg Taylora	189
	Szereg Lagrange'a	190
	Radzenie sobie z nieciągłościami	193
	Wnioski	194
Rozdział 2.4	Stabilność liczbowa	
	z wykorzystaniem niejawnego całkowania Eulera	195
	Stabilność a problem całkowania początkowej wartości	195
	Metoda jawna Eulera	196
	Metoda niejawna Eulera	197
	Niedokładność	199
	Znajdowanie niejawnych rozwiązań	199
	Wnioski	199
Rozdział 2.5	Wavelet — teoria i kompresja	201
	Zasada działania	201
	Przykład	203
	Zastosowania	204
Rozdział 2.6	Interaktywna symulacja powierzchni wody	205
	Dwuwymiarowe równanie fali	205
	Warunki brzegowe — wyspy i wybrzeża	207
	Kwestie implementacyjne	208
	Interakcja z powierzchnią	209
	Rendering	210
Rozdział 2.7	Kwaterniony w programowaniu gier	213
	Myśl o kwaternionach jako zastępcach macierzy	213
	Dlaczego po prostu nie użyć kątów Eulera?	214
	Co reprezentują X, Y, Z i W?	214
	Jakie jest podłoże matematyczne całego zagadnienia?	215
	Jak kwaterniony reprezentują obroty?	216
Rozdział 2.8	Konwersja macierz-kwaternion	219
	Obrót kwaternionu	219
	Konwersja kwaternionu na macierz	220
	Konwersja z macierzy na kwaternion	221
Rozdział 2.9	Interpolacja kwaternionów	225
	Rachunek kwaternionowy	225
	Interpolacja kwaternionów	226
	Przykładowy kod	228
	Wyprowadzenie 2.9.1: Wzór dla ślerp	228
	Wyprowadzenie 2.9.2: uzyskanie formy potęgowej ślerp	231
	Wyprowadzenie 2.9.3: interpolacja krzywymi sklejanymi	232

Rozdział 2.10	Kwaternion dla najmniejszego kąta	235
	Motywacja	235
	Niestabilność numeryczna	235
	Wyprowadzanie stabilnego wzoru	236
	Warunki, przy których nadal powstaje niestabilność	237
	Przykładowy kod	238
	Wirtualny manipulator kulowy	238
Część III Sztuczna inteligencja		239
Rozdział 3.0	Projektowanie ogólnego i użytecznego mechanizmu sztucznej inteligencji	241
	Sterowanie zdarzeniami kontra odpytywanie obiektów	242
	Koncepcja komunikatu	242
	Automaty stanów	243
	Automat stanów sterowany zdarzeniami w postaci komunikatów	243
	Czas się przyznać	246
	Jeszcze jedno małe przyznanie się	246
	Klocki automatu stanów	247
	Przekazywanie komunikatów do i z automatu stanów	247
	Wysyłanie komunikatów	248
	Wysyłanie opóźnionych komunikatów	249
	Usuwanie obiektu gry	250
	Udoskonalenie — określenie zakresu komunikatu	250
	Udoskonalenie — dziennik wysyłanych komunikatów i zmian stanów	251
	Udoskonalenie — zamiana automatów stanów	252
	Udoskonalenie — kilka automatów stanów	252
	Udoskonalenie — kolejka automatów stanów	252
	Skrypty zachowania poza kodem	253
	Wnioski	253
Rozdział 3.1	Klasa automatu o skończonej liczbie stanów	257
	Klasy FSMclass i FSMstate	259
	Definicja klasy FSMstate	259
	Definicja klasy FSMclass	260
	Tworzenie stanów dla automatu skończonego	262
	Używanie automatu skończonego	262
Rozdział 3.2	Drzewa gier	269
	Odmiana Negamax algorytmu Minimax	270
	Przycinanie alfa-beta	271
	Metody porządkowania ruchów	272
	Udoskonalenia dla alfa-beta	272
Rozdział 3.3	Podstawy A* dotyczące planowania drogi	275
	Problem	275
	Ogólne omówienie rozwiązania	275
	Właściwości A*	277
	Zastosowanie A* do planowania drogi w grach	277
	Słabości algorytmu A*	282
	Inne rozszerzenia	282
Rozdział 3.4	Optymalizacje estetyczne dla A*	283
	Proste ścieżki	283
	Wyprostowane ścieżki w przestrzeni wyszukiwania z wieloboków	284
	Wygładzanie ścieżek	284

	Częściowo obliczone wzory Catmulla-Roma	285
	Poprawa doboru kierunku przy hierarchicznych ścieżkach	286
	Hierarchiczne znajdowanie drogi na otwartej przestrzeni.....	288
	Eliminacja przestojów przy hierarchicznym wyszukiwaniu	288
	Minimalizacja czasu odpowiedzi	288
	Wnioski	289
Rozdział 3.5	Optymalizacja A* pod względem szybkości działania.....	291
	Optymalizacja przestrzeni wyszukiwania	292
	Optymalizacje algorytmu	296
	Wnioski	301
Rozdział 3.6	Uproszczony ruch w trójwymiarowej przestrzeni i znajdowanie drogi przy użyciu siatek nawigacyjnych	305
	W dużym skrócie	305
	Konstrukcja	307
	Ruszmy się	307
	Dostanie się tam to połowa zabawy	310
	Działa, ale nie najlepiej	312
	Wnioski	313
Rozdział 3.7	Algorytm ślada — prosta technika symulacji zbiorowego zachowania	321
	Implementacja	323
	Kod	325
	Ograniczenia i możliwe udoskonalenia.....	327
	Podziękowania i materiały	332
Rozdział 3.8	Logika rozmyta w grach	333
	Jak działa logika rozmyta?	333
	Operacje na logice rozmytej.....	334
	Sterowanie rozmyte	336
	Inne zastosowania logiki rozmytej	341
	Wnioski	342
Rozdział 3.9	Podstawy sieci neuronowych.....	343
	Biologiczna analogia	343
	Zastosowanie w grach	344
	Sieci neuronowe	345
	Czysta logika — Mr. Spock	350
	Klasyfikacja i rozpoznawanie „obrazów”	353
	Algorytm Hebba	356
	Sieć Hopfielda	358
	Wnioski	361
	Część IV Techniki dotyczące wielokątów	363
Rozdział 4.0	Optymalizacja dostarczania wierzchołków w OpenGL	365
	Tryb bezpośredni	365
	Przeplatane dane	366
	Dane krokowe i strumieniowe.....	367
	Skompilowane tablice wierzchołków	368
	Eliminacja kopiowania danych — rozszerzenia producentów	369
	Format danych	369
	Ogólne zalecenia	370
	Wnioski	370

Rozdział 4.1	Ulepszanie wartości głębi rzutowania wierzchołka	373
	Zapoznanie z macierzą rzutowania	373
	Dopracowywanie wartości głębi	374
	Wybór odpowiedniego epsilon	374
	Implementacja	375
	Kod źródłowy	376
Rozdział 4.2	Wektorowa kamera.....	377
	Wprowadzenie do wektorowej kamery	378
	Optymalizacja lokalnej przestrzeni	379
	Wnioski	381
Rozdział 4.3	Techniki sterowania kamerą	383
	Prosta kamera z widokiem z pierwszej osoby.....	383
	Kamera oparta na skryptach.....	385
	Sztuczki z kamerą.....	388
Rozdział 4.4	Szybki test przecięcia walca z obszarem widzenia.....	391
	Obszar widzenia	392
	Liczenie efektywnego promienia	393
	Algorytm	394
	Implementacja	395
Rozdział 4.5	Wykrywanie zderzeń w trójwymiarowej przestrzeni.....	401
	Algorytmy	401
	Wykrywanie zderzeń na podstawie kul otaczających obiekty	402
	Wykrywanie zderzeń trójkątów.....	403
Rozdział 4.6	Wielorozdzielczościowe mapy w interaktywnej detekcji zderzeń... ..	413
	Siatki.....	413
	Problemy z różnorodnością rozmiarów obiektów	413
	Wielorozdzielczościowe mapy	415
	Kod źródłowy	415
Rozdział 4.7	Obliczanie odległości w sektorze	421
	Problem	421
	Opis algorytmu	422
	Zastosowania	424
Rozdział 4.8	Usuwanie niewidocznych obiektów	429
	Usuwanie obiektów poza obszarem widoczności	430
	Usuwanie zakrytych obiektów	431
	Podsumowanie.....	433
Rozdział 4.9	Problemy z doбором szczegółowości geometrii	439
	Wybór poziomu szczegółowości.....	440
	Współczynnik powiększenia	441
	Pętla histerezy.....	441
	Implementacja	442
	Inne problemy.....	443
Rozdział 4.10	Konstrukcje z drzew ółsemkowych.....	445
	Drzewa ółsemkowe	445
	Dane drzewa ółsemkowego	446
	Tworzenie drzewa	447
	Nachodzenie na siebie wielokątów	448

Sąsiedzi	448
Zastosowania	449
Wnioski	449
Rozdział 4.11 Swobodne drzewa ósemkowe	451
Drzewa czwórkowe	452
Bryły otaczające	452
Dzielenie obiektów	453
Tworzenie swobodnego drzewa	455
Porównanie	457
Wnioski	459
Rozdział 4.12 Progresywne siatki niezależne od widoku	461
Progresywne siatki	461
Różne podejścia	462
Funkcje wyboru krawędzi	464
Trudne krawędzie	464
Implementacja	465
Kod źródłowy	469
Rozdział 4.13 Trójwymiarowa animacja za pomocą ujęć kluczowych	471
Interpolacja liniowa	471
Interpolacja wierzchołków i normalnych	473
Interpolacja krzywymi sklejanymi Hermite'a	473
Wierzchołki interpolowane krzywymi sklejanymi	475
Dlaczego krzywe sklepane Hermite'a?	476
Podsumowanie	476
Rozdział 4.14 Szybka i prosta technika deformacji siatki za pomocą kości	477
Dlaczego niewielka liczba trójkątów?	477
Działanie	477
Podsumowanie	478
Rozdział 4.15 Wypełnianie szczelin — zaawansowana animacja z zastosowaniem zszywania i złożonego systemu przypisywania siatce kości	483
Zszywanie	484
Złożone odkształcanie siatki za pomocą kości	486
Zaawansowane tematy	489
Rozdział 4.16 Generowanie realistycznego terenu w czasie rzeczywistym	491
Krajobrazy	491
Budynki	496
Algorytm tworzenia nazw	499
Rozdział 4.17 Fraktalne generowanie terenu — błędne formacje	503
Błędne formacje	503
Zmniejszanie dHeight	504
Generowanie losowych linii	504
Erozja	505
Przykładowy kod	506
Rozdział 4.18 Fraktalne generowanie terenu — przemieszczanie środkowego punktu	507
Przemieszczenie środkowego punktu w jednym wymiarze	507
Przemieszczanie środkowego punktu w dwóch wymiarach — rombowy kwadrat	508
Algorytm romb-kwadrat w polach wysokości	510

Rozdział 4.19	Fraktalne generowanie terenu — osadzanie cząsteczek.....	511
	Modele MBE	511
	Osadzanie cząsteczek	511
	Uzyskanie krateru.....	512
	Przykładowy kod	514
Część V Efekty z wykorzystaniem pikseli		515
Rozdział 5.0	Dwuwymiarowy efekt lens flare	517
	Podejście.....	517
	Implementacja	518
	Kod źródłowy	520
Rozdział 5.1	Użycie sprzętu do grafiki trójwymiarowej dla dwuwymiarowych efektów sprite'ów	521
	Przechodzimy do trzeciego wymiaru	521
	Ustawianie trójwymiarowej sceny	522
	Ustawianie tekstury	522
	Rysowanie trójwymiarowego sprite'a.....	522
	Dodawanie efektów	524
	Wnioski	525
Rozdział 5.2	Statyczne oświetlenie bazujące na określonym motywie	527
	Tradycyjne statyczne oświetlenie.....	527
	Statyczne oświetlenie bazujące na motywach.....	530
	Wnioski	536
Rozdział 5.3	Symulacja oświetlenia w czasie rzeczywistym za pomocą interpolacji kolorów wierzchołków	537
	Metoda oświetlenia.....	538
	Tworzenie grafiki	538
	Interpolacja oświetlenia.....	539
	Wnioski	540
Rozdział 5.4	Mapy zaniku	545
	Omówienie	545
	Porównanie map zaniku z mapami oświetlenia.....	549
	Efekty CSG.....	549
	Mgła bazująca na zasięgu.....	549
	Inne kształty.....	550
	Wnioski	550
Rozdział 5.5	Zaawansowane teksturowanie z wykorzystaniem generacji współrzędnych tekstury	551
	Prosta animacja współrzędnych tekstury.....	552
	Rzutowanie tekstury	552
	Odwzorowywanie odbić.....	554
Rozdział 5.6	Sprzętowe mapowanie nierówności	557
	Jak nałożyć mapę nierówności na obiekt?.....	558
	Dobór przestrzeni dla normalnych	558
	Inne rozwiązanie — używanie mapowania nierówności w przestrzeni stycznej.....	559
	Rozwiązanie — mapowanie nierówności w przestrzeni tekstury	562
	Problemy w przestrzeni tekstury	563
	Wnioski	564

Rozdział 5.7	Cienie na płaskiej podłodze	565
	Matematyka cienia.....	565
	Implementacja	567
	Udoskonalenia	569
Rozdział 5.8	Cienie na złożonych obiektach liczone w czasie rzeczywistym.....	571
	Wprowadzenie.....	571
	Źródło światła, obiekt blokujący i otrzymujący	572
	Cele tego rozdziału	573
	Tworzenie mapy cienia.....	574
	Rzutowanie mapy cienia na obiekcie otrzymującym	580
	Rendering obiektów odbierających	581
	Rozszerzenia i usprawnienia podstawowego algorytmu	582
Rozdział 5.9	Ulepszanie środowiska — odwzorowywanie odbić przy użyciu wcześniejszego filtrowania z połyskiem i czynnika Fresnela	585
	Pierwsze błędne założenie.....	586
	Drugie błędne założenie	588
	Wnioski	588
	Podziękowania.....	588
Rozdział 5.10	Realistycznie wyglądające szkło w grach.....	589
	Wprowadzenie.....	589
	Przezroczyste obiekty	589
	Rasteryzer, bufor ramki, bufor głębi i mieszanie pikseli.....	589
	Obiekty nieprzezroczyste kontra przezroczyste	590
	Rysowanie nieprzezroczystych obiektów.....	591
	Rysowanie przezroczystych obiektów	591
	Odbicia	595
	Kolorowe szkło.....	595
	Łączymy wszystko razem.....	596
	Implementacja	596
Rozdział 5.11	Odwzorowywanie załamań w płynach znajdujących się w pojemnikach.....	597
	Wprowadzenie	597
	Współczynnik załamania.....	598
	Współczynnik odbicia	600
	Czynnik Fresnela	600
	Rendering na sprzęcie.....	600
	Możliwe rozszerzenia techniki	601
	Wnioski	602
Dodatki	603	
Dodatek A	Biblioteka operacji na macierzach	605
Dodatek B	Biblioteka wyświetlania tekstów	607
Dodatek C	Bibliografia.....	609
	Skorowidz.....	619

Rozdział 4.5

Wykrywanie zderzeń w trójwymiarowej przestrzeni

Kevin Kaiser

Mechanizm fizyki działający w czasie rzeczywistym jest najważniejszym elementem, dzięki któremu można tworzyć trójwymiarowe środowiska, na widok których gracz po prostu kręci głową z niedowierzaniem. Mechanizm symulacji fizyki zapewnia realistyczną interakcję obiektów. Wtedy gracz czuje realizm przedstawionego świata, a co za tym idzie, może lepiej się po nim poruszać, ponieważ zachowuje się podobnie do tego, do czego jest przyzwyczajony w realnym świecie. Pierwszym i najważniejszym krokiem przy tworzeniu realistycznej symulacji fizyki jest dokładne wykrywanie zderzeń; gdy już zostanie jakieś wykryte, symulacja może odpowiednio zadziałać. W tym rozdziale ułożymy podwaliny pod tworzenie perfekcyjnej symulacji fizyki, omawiając najważniejszy jej fragment, czyli wykrywanie zderzeń w trójwymiarowej przestrzeni.

Algorytmy

W rozdziale zajmiemy się dwoma algorytmami detekcji zderzeń:

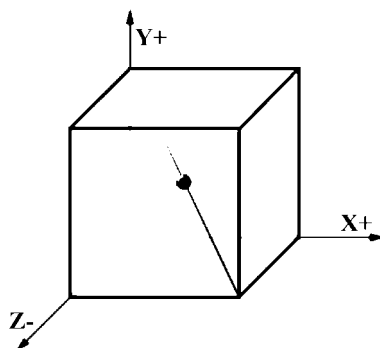
- **Wykrywanie zderzeń na podstawie kul otaczających obiekty.** Używamy ich, ponieważ kod jest stosunkowo prosty, a wyjaśnienie zasad działania nie nastrocza większych problemów. W zasadzie kod sprawdza zderzenia, testując promień jednej kuli z promieniem drugiej.
- **Wykrywanie zderzeń na podstawie przecięć trójkątów.** W tym przypadku, zanim przejdziemy do algorytmu, warto będzie sobie przypomnieć co nieco z matematyki. Ten algorytm używa równań parametrycznych do określenia zderzenia między punktami jednego trójkąta a płaszczyzną innego trójkąta, a następnie określenia, czy te kolidujące punkty znajdują się wewnątrz drugiego trójkąta.

Wykrywanie zderzeń na podstawie kul otaczających obiekty

Wykrywanie zderzeń najlepiej przeprowadzać w hierarchicznych krokach: dla kul otaczających obiekty, następnie dla kul otaczających wieloboki, a na końcu wykorzystać przenikanie trójkątów. Liczenie kul otaczających obiekty jest bardzo proste; musisz znaleźć środek obiektu, następnie policzyć maksymalną odległość między środkiem a wierzchołkiem obiektu. Przechowując promień każdej otaczającej kuli, możesz przeprowadzać test zderzenia, dodając promień i sprawdzając je z odległością środków obiektów. Jeśli suma jest większa niż odległość środków, kule nie przenikają się.

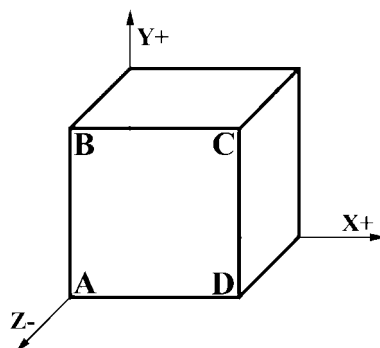
Przejdźmy przez to krok po kroku. Najpierw musimy określić środek siatki. Jedną z metod używa prostopadłościanu otaczającego obiekt i liczy środek przekątnej tej bryły (rysunek 4.5.1). Aby określić prostopadłościan, musimy poznać maksymalne i minimalne wartości x , y i z dla całego obiektu. Można to wykonać, przechodząc przez wszystkie wierzchołki i sprawdzając „aktualne” maksimum i minimum. Po sprawdzeniu wszystkich wierzchołków otrzymamy minimalny prostopadłościan, który otoczy obiekt.

Rysunek 4.5.1.
Znajdowanie środka



Dla prostopadłościanu z ośmioma punktami (ABCDEFGH, patrz rysunek 4.5.2) przypomnijmy sobie ułożenie wierzchołków

Rysunek 4.5.2.
Tworzenie
prostopadłościanu
otaczającego obiekt



A = (minx, miny, minz)
B = (minx, maxy, minz)
C = (maxx, maxy, minz)
D = (maxx, miny, minz)
E = (minx, miny, maxz)

```
F = (minx, maxy, maxz)
G = (maxx, maxy, maxz)
H = (maxx, miny, maxz)
```

Teraz znajdź środek, uśredniając maksymalny i minimalny punkt prostopadłościanu (w naszym przypadku są to punkty **A** i **G**).

```
// Wzór na środek: dane A(x1,y1,z1) i B(x2,y2,z2)
// Środek linii między punktami A i B jest następujący
// [(x1+x2)/2, (y1+y2)/2, (z1+z2)/2]
środek.x = (A.x + G.x)/2;
środek.y = (A.y + G.y)/2;
środek.z = (A.z + G.z)/2;
```

Promień otaczającej obiekt kuli można łatwo obliczyć, przechodząc przez kolejne wierzchołki i znajdując odległość między wierzchołkiem i środkiem obiektu. Jeśli uzyskana odległość jest większa od aktualnego maksimum, zastępujemy maksimum uzyskaną wartością. Po sprawdzeniu wszystkich wierzchołków maksymalna odległość jest promieniem kuli (oczywiście dosyć naturalną optymalizacją jest liczenie pierwiastka kwadratowego dopiero na końcu).

```
// Wzór na odległość:
// odl = sqrt[((x2-x1)^2) + ((y2-y1)^2) + ((z2-z1)^2)]
// odlkw = ((x2-x1)^2) + ((y2-y1)^2) + ((z2-z1)^2)
dla każdego wierzchołka v w obiekcie {
    aktualna_odległość_kw = odlkw(obiekt.środek, v);
    if (aktualna_odległość_kw > max_odległość_kw)
        max_odległość_kw = aktualna_odległość_kw;
}
obiekt.promień = sqrt(max_odległość_kw);
```

Całość powtórzymy na poziomie wielokątów; sprawdzanie przy użyciu otaczających kul jest proste i szybkie, więc użycie tej metody przy tym teście wydaje się logiczne. Po wygenerowaniu otaczających prostopadłościanów i kul dla każdego obiektu i wieloboku możemy zająć się najciekawszą częścią tego rozdziału: testem przecinania się trójkątów! Wyjmij książkę z geometrii — może Ci się przydać.

Wykrywanie zderzeń trójkątów

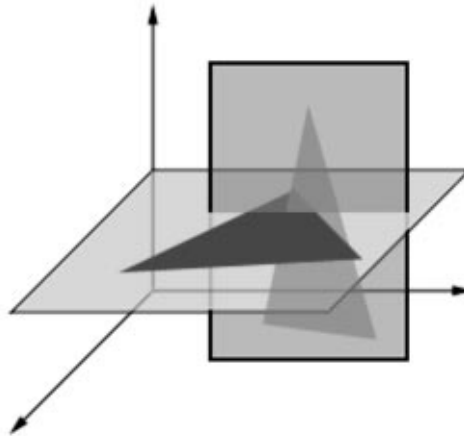
Przedstawiana metoda wykrywania zderzeń dwóch trójkątów nie jest trudna do zrozumienia, ale wykorzystuje pewną matematyczną sztuczkę. Wyobraź sobie, że mamy dwa trójkąty w trójwymiarowej przestrzeni (patrz rysunek 4.5.3). O obydwu musimy zebrać pewną ilość informacji. Dla jednego z nich musimy wyznaczyć równanie płaszczyzny, na której się znajduje. Jeśli masz dobrą pamięć, to zapewne wiesz, że ma ono postać: $Ax + By + Cz + D = 0$. A , B , C i D określimy, licząc iloczyn wektorowy wierzchołków.

```
// dany jest trójkąt tr1l z wierzchołkami a, b i c
// vector3 a, b, c;
vector3 v1, v2, cross_v1xv2;

// tworzymy wektory v1 i v2
// (tr1l.b - tr1l.a i tr1l.c - tr1l.a)
v1 = tr1l.b - tr1l.a;
v2 = tr1l.c - tr1l.a;
```

Rysunek 4.5.3.

Dwa przecinające się
trójkąty



```
// UWAGA: możesz pominąć ten krok i użyć własnych normalnych płaszczyzny, jeśli
// gdzieś je przechowujesz. Uzyskujemy iloczyn wektorowy v1 i v2 (wektor)
cross_v1xv2 = CrossProduct(v1, v2);

// Wstawiamy te wartości do Ax+By+Cz+D=0
tri1.pA = cross_v1xv2.x;
tri1.pB = cross_v1xv2.y;
tri1.pC = cross_v1xv2.z;

// Korzystamy z równania Ax+By+Cz+d=0, by obliczyć D
// Jeśli punkt P(x0,y0,z0) jest punktem na wieloboku
// A = cross_v1xv2.x
// B = cross_v1xv2.y
// C = cross_v1xv2.z
// D = (-A*x0-B*y0-C*z0)
tri1.pD = -DotProduct(cross_v1xv2, P);
```

Przecięcie linia-płaszczyzna

Mamy już równanie płaszczyzny pierwszego trójkąta (tri1) i możemy przejść do następnego etapu: sprawdzania, czy drugi trójkąt (tri2) koliduje z płaszczyzną tri1. Jest to wykonywane w kilku krokach. Główna idea opiera się na tym, że między dwoma wierzchołkami tri2 prowadzimy linię i sprawdzamy, czy przecina ona płaszczyznę tri1. Jeśli punkt kolizji znajduje się między tymi dwoma wierzchołkami, tri2 przecina płaszczyznę tri1; jeśli nie znajduje się między wierzchołkami, przechodzimy do następnych dwóch linii tworzących tri2, aby sprawdzić, czy przypadkiem tam nie występuje kolizja.

Przecięcie linia-płaszczyzna rozwiązujemy, używając równań parametrycznych. Dla danych wierzchołków $\mathbf{a}(x0,y0,z0)$ i $\mathbf{b}(x1,y1,z1)$, tworzymy równanie $\mathbf{a}(x0,y0,z0) * t = \mathbf{b}(x1,y1,z1) * (1 - t)$, gdzie t to współczynnik interpolacji zmieniający się od 0 do 1. Gdy $t=0$ jesteś w punkcie $\mathbf{b}$, gdy jest równe 1 — w $\mathbf{a}$. Jeśli wstawimy te równania parametryczne do równania płaszczyzny, będziemy mogli policzyć t .

$$A*(x0*t + x1*(1 - t)) + B*(y0*t + y1*(1 - t)) + C*(z0*t + z1*(1 - t)) + D = 0$$

Po przekształceniach otrzymamy:

$$t = -(A*x1 + B*y1 + C*z1 + D)/(A*(x0 - x1) + B*(y0 - y1) + C*(z0 - z1))$$

Oto kod obliczający t :

```
// i0 = (A*x0) + (B*y0) + (C*z0)
i0 = (tril->pA*a->x) + (tril->pB*a->y) + (tril->pC*a->z);

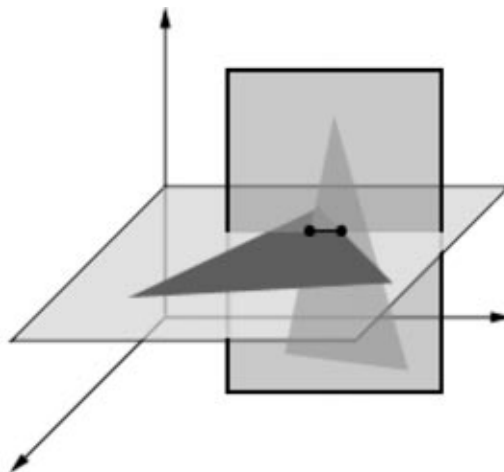
// i1 = (A*x1) + (B*y1) + (C*z1)
i1 = (tril->pA*b->x) + (tril->pB*b->y) + (tril->pC*b->z);

// Tutaj uważaj na możliwe dzielenia przez zero (np. gdy i0 = i1)
final_t = -(i1 + tril->pD) / (i0 - i1);

// Teraz wstaw final_t do funkcji x(), y() i z(), aby otrzymać punkt przecięcia
final_x = (((a->x)*(final_t)) + ((b->x)*(1-final_t)));
final_y = (((a->y)*(final_t)) + ((b->y)*(1-final_t)));
final_z = (((a->z)*(final_t)) + ((b->z)*(1-final_t)));
```

W ten sposób otrzymujemy punkt, w którym linia przecina płaszczyznę (rysunek 4.5.4). Oczywiście wartość t , którą obliczymy, musi się znajdować w przedziale $0 - 1$; w przeciwnym przypadku przecięcie nie znajduje się między wierzchołkami! Szczególny przypadek, na który musisz zwrócić uwagę w tym kroku, to wystąpienie pionowych linii. Najszybszą metodą określenia przecięcia linii jest wstawienie x i z dla obydwu punktów (a i b) do równania płaszczyzny trójkąta i obliczenie y . Wtedy punktem przecięcia będzie punkt ($a.x$, **obliczony**. y , $a.z$).

Rysunek 4.5.4.
Określanie
punktu przecięcia



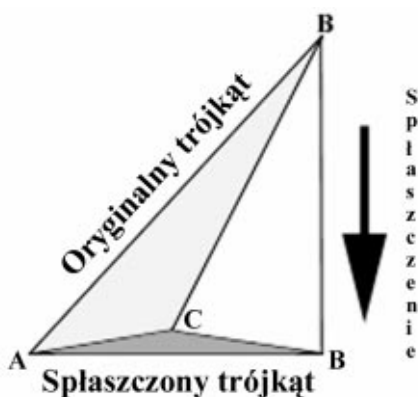
Splaszczanie trójkąta

Będziemy teraz używać praworęcznego systemu współrzędnych. Wyobraź sobie splaszczanie trójkąta względem jednej z płaszczyzn współrzędnych, zależnie od obrotu trójkąta. Może on na przykład utracić współrzędną y i zachować współrzędne x i z . Dokładniej — ta idea nie oznacza wyrzucenia współrzędnej y ; wyrzucamy odpowiednią współrzędną, aby uzyskać efekt splaszczania. Wyboru najłatwiej dokonać, przypatrując się nor-

malnej płaszczyzny. Jeśli ustalisz, który element ma największą wartość bezwzględną, możesz znaleźć płaszczyznę, względem której możesz spłaszczyć, aby trójkąt nie stał się linią prostą (na przykład gdy pionowy trójkąt utraci współrzędną y). Na przykład jeśli element x jest największy, będziesz rzutował na płaszczyznę yz . Niezależnie od obrotu, uzyskany trójkąt będzie płaski (jakby leżał na stole — rysunek 4.5.5). Ta technika jest bardzo przydatna, ponieważ możemy wtedy bardzo łatwo stwierdzić, czy uzyskany punkt przecięcia (po spłaszczeniu w podobny sposób), znajduje się wewnątrz trójkąta. Kod z wydruku 4.5.1 spłaszcza trójkąt względem jednej z płaszczyzn współrzędnych.

Rysunek 4.5.5.

Rzutowanie
wierzchołków



Testowanie, czy punkt jest wewnątrz trójkąta

Teraz, kiedy już spłaszczyliśmy współrzędne, potrzebujemy pewnych obliczeń matematycznych, by określić, czy punkt przecięcia znajduje się wewnątrz spłaszczonego trójkąta, czy też nie. Test można przeprowadzić na kilka sposobów: my zamierzamy skorzystać z równania dla każdej linii spłaszczonego trójkąta. Zauważ, że niezależnie od tego, na jaką płaszczyznę rzutowaliśmy, spłaszczone punkty będziemy określać współrzędnymi x i y . Robimy tak dlatego, że w zasadzie zredukowaliśmy problem do dwóch wymiarów, czyli osi X i Y . Najpierw musimy znaleźć punkt, który *na pewno* znajduje się wewnątrz trójkąta. Najłatwiej zrobić to, obliczając środek trójkąta, liczony jako średnia wierzchołków: $((x0+x1+x2)/2, (y0+y1+y2)/2)$. Skoro znamy już kierunek do wnętrza trójkąta, możemy sprawdzić, czy nasz punkt znajduje się po „wewnętrznej” stronie linii tworzonych przez wierzchołki (rysunek 4.5.6).

Dla danych wierzchołków $v0$ i $v1$ najpierw znajdujemy równanie linii przechodzącej przez te punkty w postaci $y=mx+b$. Zapewne pamiętasz, że wzór na nachylenie (m) to $(y1 - y0)/(x1 - x0)$, więc możemy znaleźć b , używając wcześniej obliczonego nachylenia i punktu, przez który przechodzi prosta. Skoro mamy już równanie linii w postaci z nachyleniem, możemy sprawdzić, czy punkt przecięcia leży wewnątrz spłaszczonego trójkąta. Robimy to, porównując wartości y . Jeśli wstawisz wartość x spłaszczonego punktu przecięcia do wzoru $y=mx+b$, otrzymasz wartość y w punkcie x . Teraz sprawdzisz, czy obliczony środek trójkąta znajduje się nad, czy poniżej linii, testując wartość y . Wiemy, że znajduje się w środku, więc nasz punkt przecięcia musi mieć wartość y w tym samym

Rysunek 4.5.6.

Określanie obszaru
zajmowanego przez
trójkąt



kierunku co środek trójkąta. Jeśli tak rzeczywiście jest, punkt znajduje się wewnątrz trójkąta względem linii **ab**. Musimy to jeszcze powtórzyć dla linii **bc** i **ca**. Jeśli w każdym teście dowiedzieliśmy się, że punkt znajduje się w środku trójkąta, rzeczywiście tak jest. Należy jednak brać pod uwagę szczególny przypadek: rzutowanie pionowej linii. Za pomocą $y=mx+b$ nie możesz narysować takiej linii. Jeśli wystąpi taka sytuacja, sprawdzasz współrzędną x zamiast y ; to znaczy, że najpierw określasz, po której stronie pionowej linii znajduje się wewnętrzny punkt. Następnie sprawdzasz wartość x rzutowanego punktu przecięcia. Jeśli jest po tej samej stronie, co poprzedni punkt, znajduje się wewnątrz trójkąta względem sprawdzanej linii. Implementację znajdziesz na wydruku 4.5.2.

Sprawdzaj wszystkie linie w obydwu trójkątach!

Oczywiście jeśli pierwsza linia nie przecina płaszczyzny, musimy sprawdzić pozostałe. Wystarczy znaleźć jedną kolizję linia-trójkąt, aby wykazać, że trójkąty nachodzą na siebie. Resztę przykładowego kodu znajdziesz na wydruku 4.5.3.

Jeśli nie wykryłeś żadnego zderzenia, musisz odwrócić procedurę, zaczynając tym razem od drugiego trójkąta. Tym samym upewniasz się, że na pewno wykryjesz zderzenie.

Wydruk 4.5.1

```
if (x==FALSE) { // opuszczam współrzędną x
    a1 = tri->a.y;
    b1 = tri->a.z;
    a2 = tri->b.y;
    b2 = tri->b.z;
    a3 = tri->c.y;
    b3 = tri->c.z;
    a4 = vert->y;
    b4 = vert->z;
    inside = 0;
}
else if (y==FALSE) { // opuszczam współrzędną y
    a1 = tri->a.x;
    b1 = tri->a.z;
    a2 = tri->b.x;
    b2 = tri->b.z;
    a3 = tri->c.x;
    b3 = tri->c.z;
    a4 = vert->x;
    b4 = vert->z;
    inside = 0;
}
```

```

    }
    else if (z==FALSE) { // opuszczam współrzędną z
        a1 = tri->a.x;
        b1 = tri->a.y;
        a2 = tri->b.x;
        b2 = tri->b.y;
        a3 = tri->c.x;
        b3 = tri->c.y;
        a4 = vert->x;
        b4 = vert->y;
        inside = 0;
    }

```

Wydruk 4.5.2

```

// Poniższy kod sprawdza pionowe linie w spłaszczonej trójkącie.
// Nie można narysować linii za pomocą y=mx+b, więc musimy sprawdzać,
// czy spłaszczony punkt przecięcia znajduje się między współrzędną x
// dowolnej pionowej linii a środkiem trójkąta, aby przekonać się, czy
// punkt przecięcia znajduje się wewnątrz trójkąta względem pionowej linii.
AB_vert = BC_vert = CA_vert = FALSE;

// y=mx+b dla trzech linii tworzących trójkąt
if ((a2-a1)!=0) {
    m1 = (b2-b1)/(a2-a1); // a->b
    bb1 = (b1)-(m1*a1);    // y/(mx) przy użyciu wierzchołka a
} else if ((a2-a1)==0) {
    AB_vert = TRUE;
}

if ((a3-a2)!=0) {
    m2 = (b3-b2)/(a3-a2); // b->c
    bb2 = (b2)-(m2*a2);    // y/(mx) przy użyciu wierzchołka b
} else if ((a3-a2)==0) {
    BC_vert = TRUE;
}

if ((a1-a3)!=0) {
    m3 = (b1-b3)/(a1-a3); // c->a
    bb3 = (b3)-(m3*a3);    // y/(mx) przy użyciu wierzchołka c
} else if ((a1-a3)==0) {
    CA_vert = TRUE;
}

// znajdź średni punkt trójkąta (na pewno znajduje się on w jego wnętrzu)
center_x = (a1+a2+a3)/3;
center_y = (b1+b2+b3)/3;

// Sprawdź, czy (center_x,center_y) jest powyżej, czy poniżej linii,
// ustaw kierunek na UP, gdy jest wyżej i DOWN, gdy jest poniżej linii

// a->b
if (((m1*center_x)+bb1) >= center_y)
    DIRECTION(direction,UP);
else
    DIRECTION(direction,DOWN);
if (AB_vert==TRUE) {

```

```

    if ((a1<a4)&&(a1<center_x)) // rzutowana pionowo linia
        inside++;
    else if ((a1>a4)&&(a1>center_x)) // rzutowana pionowo linia
        inside++;
} else {
    if (direction==UP) {
        if (b4 <= ((m1*a4)+bb1)) // b4 mniejsze od y, aby był wewnątrz
            inside++;           // (linia powyżej punktu)
    } else if (direction==DOWN) {
        if (b4 >= ((m1*a4)+bb1)) // b4 większa od y, by był wewnątrz
            inside++;           // (linia poniżej punktu)
    }
}

// b->c
if (((m2*center_x)+bb2) >= center_y)
    DIRECTION(direction,UP);
else
    DIRECTION(direction,DOWN);
if (BC_vert==TRUE) {
    if ((a2<a4)&&(a2<center_x)) // rzutowana pionowo linia
        inside++;
    else if ((a2>a4)&&(a2>center_x)) // rzutowana pionowo linia
        inside++;
} else {
    if (direction==UP) {
        if (b4 <= ((m2*a4)+bb2)) // b4 mniejsze od y, aby był wewnątrz
            inside++;           // (linia powyżej punktu)
    } else if (direction==DOWN) {
        if (b4 >= ((m2*a4)+bb2)) // b4 większe od y, aby był wewnątrz
            inside++;           // (linia poniżej punktu)
    }
}

// c->a
if (((m3*center_x)+bb3) >= center_y)
    DIRECTION(direction,UP);
else
    DIRECTION(direction,DOWN);
if (CA_vert==TRUE) {
    if ((a3<a4)&&(a3<center_x)) // rzutowana pionowo linia
        inside++;
    else if ((a3>a4)&&(a3>center_x)) // rzutowana pionowo linia
        inside++;
} else {
    if (direction==UP) {
        if (b4 <= ((m3*a4)+bb3)) // b4 mniejsze od y, aby był wewnątrz
            inside++;           // (linia powyżej punktu)
    } else if (direction==DOWN) {
        if (b4 >= ((m3*a4)+bb3)) // b4 większe od y, aby był wewnątrz
            inside++;           // (linia poniżej punktu)
    }
}
}
if (inside==3) {
    return TRUE;
} else {
    return FALSE;
}

```

Wydruk 4.5.3

```

// Przejdź przez wszystkie trzy linie tworzące trójkąt
// Pierwsza iteracja (a,b)
p=line_plane_collision((vertex_ptr)&tri2.a,(vertex_ptr)&tri2.b,(triangle_ptr)&tri1);

// Określ, na którą oś rzutować
// X jest największe
if ((abs(tri1.pA)>=abs(tri1.pB))&&(abs(tri1.pA)>=abs(tri1.pC)))
    temp = point_inside_triangle((triangle_ptr)&tri1,(vertex_ptr)&p,
                                FALSE,TRUE,TRUE);

// Y jest największe
else if ((abs(tri1.pB)>=abs(tri1.pA))&&(abs(tri1.pB)>=abs(tri1.pC)))
    temp = point_inside_triangle((triangle_ptr)&tri1,(vertex_ptr)&p,
                                TRUE,FALSE,TRUE);

// Z jest największe
else if ((abs(tri1.pC)>=abs(tri1.pA))&&(abs(tri1.pC)>=abs(tri1.pB)))
    temp = point_inside_triangle((triangle_ptr)&tri1,(vertex_ptr)&p,
                                TRUE,TRUE,FALSE);

if (temp==TRUE) {
    // Sprawdzamy punkt, by sprawdzić, czy leży między wierzchołkami
    // Najpierw sprawdzaj szczególny przypadek z pionowymi liniami
    if ((tri2.a.x == tri2.b.x)&&(tri2.a.z == tri2.b.z)) {
        if (((tri2.a.y <= p.y)&&(p.y <= tri2.b.y))||
            ((tri2.b.y <= p.y)&&(p.y <= tri2.a.y)))
            return TRUE;
    }
    // Koniec sprawdzania szczególnego przypadku

    // Teraz sprawdź punkt względem linii
    if (point_inbetween_vertices((vertex_ptr)&tri2.a,(vertex_ptr)&tri2.b,
                                (triangle_ptr)&tri1)==TRUE)
        return TRUE;
    else
        return FALSE;
}

// Druga iteracja (b,c)
p=line_plane_collision((vertex_ptr)&tri2.b,(vertex_ptr)&tri2.c,
                      (triangle_ptr)&tri1);

// Określ, na którą oś rzutować
// X jest największe
if ((abs(tri1.pA)>=abs(tri1.pB))&&(abs(tri1.pA)>=abs(tri1.pC)))
    temp = point_inside_triangle((triangle_ptr)&tri1,(vertex_ptr)&p,
                                FALSE,TRUE,TRUE);

// Y jest największe
else if ((abs(tri1.pB)>=abs(tri1.pA))&&(abs(tri1.pB)>=abs(tri1.pC)))
    temp = point_inside_triangle((triangle_ptr)&tri1,(vertex_ptr)&p,
                                TRUE,FALSE,TRUE);

// Z jest największe
else if ((abs(tri1.pC)>=abs(tri1.pA))&&(abs(tri1.pC)>=abs(tri1.pB)))
    temp = point_inside_triangle((triangle_ptr)&tri1,(vertex_ptr)&p,
                                TRUE,TRUE,FALSE);

if (temp==TRUE) {
    // Sprawdzamy punkt, by sprawdzić, czy leży między wierzchołkami

```

```
// Najpierw sprawdzaj szczególny przypadek z pionowymi liniami
if ((tri2.b.x == tri2.c.x)&&(tri2.b.z == tri2.c.z)) {
    if (((tri2.b.y <= p.y)&&(p.y <= tri2.c.y))||
        ((tri2.c.y <= p.y)&&(p.y <= tri2.b.y)))
        return TRUE;
}

// Teraz sprawdź punkt względem linii
if (point_inbetween_vertices((vertex_ptr)&tri2.b,(vertex_ptr)&tri2.c,
                             (triangle_ptr)&tri1)==TRUE)
    return TRUE;
else
    return FALSE;
}

// Trzecia iteracja (c,a)
p=line_plane_collision((vertex_ptr)&tri2.c,(vertex_ptr)&tri2.a,
                      (triangle_ptr)&tri1);

// Określ, na którą oś rzutować
// X jest największe
if ((abs(tri1.pA)>=abs(tri1.pB))&&(abs(tri1.pA)>=abs(tri1.pC)))
    temp = point_inside_triangle((triangle_ptr)&tri1,(vertex_ptr)&p,
                                FALSE,TRUE,TRUE);

// Y jest największe
else if ((abs(tri1.pB)>=abs(tri1.pA))&&(abs(tri1.pB)>=abs(tri1.pC)))
    temp = point_inside_triangle((triangle_ptr)&tri1,(vertex_ptr)&p,
                                TRUE,FALSE,TRUE);

// Z jest największe
else if ((abs(tri1.pC)>=abs(tri1.pA))&&(abs(tri1.pC)>=abs(tri1.pB)))
    temp = point_inside_triangle((triangle_ptr)&tri1,(vertex_ptr)&p,
                                TRUE,TRUE,FALSE);

if (temp==TRUE) {
    // Sprawdzamy punkt, by sprawdzić, czy leży między wierzchołkami
    // Najpierw sprawdzaj szczególny przypadek z pionowymi liniami
    if ((tri2.c.x == tri2.a.x)&&(tri2.c.z == tri2.a.z)) {
        if (((tri2.c.y <= p.y)&&(p.y <= tri2.a.y))||
            ((tri2.a.y <= p.y)&&(p.y <= tri2.c.y)))
            return TRUE;
    }

    // Teraz sprawdź punkt względem linii
    if (point_inbetween_vertices((vertex_ptr)&tri2.c,(vertex_ptr)&tri2.a,
                                (triangle_ptr)&tri1)==TRUE)
        return TRUE; // Punkt przecięcia znajduje się wewnątrz trójkąta lub na linii
    else
        return FALSE;
}
return FALSE; // Domyślna wartość - brak zderzenia
```