

Cóż tutaj dużo mówić - wydawało by się, że poznaliśmy już wszystkie sztuczki współczesnej grafiki 3D - wiemy jak stworzyć i poruszać wirtualnym światem, jak tworzyć modele i współpracować z grafikami tworzącymi bohaterów i miejsca naszych boju, wiemy jak wycisnąć z karty graficznej ostatnie soki no i poznaliśmy tajemnice najnowszych tworów nieograniczonej ludzkiej wyobraźni, czyli vertex i pixel shaders. Wydawać by się mogło, że czas sięgnąć teraz do zakamarków naszej pamięci, wydobyć z nich dawno zapomniany, zakurzony projekt gry, skołować sobie kumpli w postaci grafików, muzyków, speców od AI no i do dzieła. Tylko powstaje pytanie czy w to, co stworzymy będą ludzie grać? Nie od dzisiaj wiadomo, że już nie wystarczy super grafika naładowana efektami jak ciasto rodzynekami - widzieliśmy już wiele takich gniotów, które oszałamiają grafiką i przyprawiającymi o zawrót głowy wymaganiami sprzętowymi a nie reprezentującymi sobą nic szczególnego. Pierwsza rzecz to oczywiście nowy, oryginalny, super grywalny pomysł. Kiedy to mamy, możemy my, programiści animacji czasu rzeczywistego przystąpić do akcji. Ale wszyscy doskonale wiemy co nadchodzi - dwie najbardziej oczekiwane gry od wielu, wielu lat. Niewątpliwie czeka nas starcie gigantów - ID Software kontra Valve, czyli bardziej przemawiając do wyobraźni Doom3 kontra HL2. Pewnie wszyscy zapaleni gracze już mieli okazję widzieć jeśli nie w tajemniczy sposób pojawiające się w internecie wersje alfa czy beta samych gier to na pewno filmy prezentujące możliwości nowych silników graficznych i nie tylko. Oglądając screeny czy owe filmy czasem mnie samego nadchodzi myśl, że to nasze pisanie i nauka jest zupełnie bez sensu, bo skoro oni takie rzeczy robią to my z naszymi prostymi shaderami możemy po prostu tylko sięść i grać a nie zajmować się pisaniem aplikacji 3D i uzyskiwaniem jako takich efektów graficznych. No ale z drugiej strony sam zakładałem, że stronka nie pożyje dłużej niż rok a tutaj proszę - ma się całkiem nieźle a zanosi się na dalszy rozwój i kto wie, czy nawet nie międzynarodowy :). Ambicja więc to rzecz dobra i doskonale motywująca aby nie ustawać w wysiłkach i aby dogonić najlepszych - a czy to jest możliwe przy takim tempie rozwoju technologii? Jak najbardziej! - a udowodnimy to dzisiejszym i następnymi artykułami :)).

To co szczególnie przyciąga wzrok oczekujący graczy, zwłaszcza w zapowiadanej na kolejną rewolucję w dziedzinie gier Doom-ie 3 to przede wszystkim zmiany w oświetleniu scen. Muszę przyznać, że widziałem już i wersję "do pogrania" HL2 i Doom-a i ten drugi prezentuje się naprawdę znakomicie. Dynamiczne oświetlenie, w tym cienie rzucane przez poruszające się obiekty, niesamowity nastrój panujący we wnętrzach no i co najważniejsze dla nas dzisiaj - poziomy w tej grze to jedna, wielka mapa wypukłości! Koniec więc z płaskimi teksturami, które imitują jednak dosyć nieudolnie jakieś tam właściwości ścian czy innych materiałów - w Doomie jeśli wchodzi się do łazienki to jest to właśnie łazienka a nie jej marna imitacja. Na ścianach i na podłodze widzimy niemal realne kafelki, krata to krata a nie tekstura z dziurami i nawet pordezwiiała beczka wygląda po prostu jak beczka a nie obiekt z 3DStudio ;-).

A wszystko to dzięki nowej technologii, która od pewnego czasu rewolucjonizuje grafikę 3D czasu rzeczywistego w dziedzinie mapowania wypukłości. Dotychczasowe próby zmuszenia trójkątów aby były bardziej wypukłe na ekranie napotykały na mniej lub bardziej stanowczy opór ze strony sprzętu - metody istnieją od dosyć dawna, ale są tak obliczeniowo, że tylko specjalizowane stacje graficzne potrafiły to wyrenderować w jakimś tam sensownym czasie. Jednak postęp technologii nie ominął nas, szaraczków posiadających w swoich komputerach karty graficzne których możliwości przy takich stacjach są śmieszne i służą one tylko i wyłącznie do zabawy. Co śmieszniejsze jeszcze bardziej, rozwój tej technologii pozwala implementować paradoksalnie coraz prostsze operacje w procesorach naszych kart grafiki i osiągnięcie za ich pomocą naprawdę oszałamiających rezultatów. I dzisiaj właśnie będziemy mieli właśnie z taką operacją doczynienia - proszę Państwa czas więc przedstawić dzisiejszego głównego bohatera - operację DOT3.

Sama operacja, jak tysiące jej podobnych matematycznych tworów jest dosyć prosta - ot, po prostu bierzemy kilka liczb, dodajemy do siebie i otrzymujemy wynik. Czegóż zaś w szczególności będzie dotyczyć nasza dzisiejsza operacja? Ponieważ symulować mamy zamiar mapowanie wypukłości, więc zdecydowanie w ruch pójdą kolory a w szczególności ich składowe. Przechodząc do sedna sprawy oto, co wykorzystamy dzisiaj w naszej lekcji:

$$S_{RGB} = Arg1_R \cdot Arg2_R + Arg1_G \cdot Arg2_G + Arg1_B \cdot Arg2_B$$

Arg1 i **Arg2** to oczywiście są kolory biorące udział w naszej operacji. Zostaną one zmieszane ze sobą w pewien sposób, przedstawiony powyżej, dając w połączeniu z innymi zabiegami dosyć zadowalający efekt (mam nadzieję) na ekranie. Wynikiem mieszania dwóch argumentów będzie kolor wynikowy, który wyświetlony na bryle da to, o co nam chodzi. I cóż więcej - o samej operacji to chyba nic więcej powiedzieć nie można, bo cóż tutaj można dodać. No ale sama operacja cudu nam nie sprawi - czas więc dowiedzieć się, co mieszać i jak, żeby mieć rezultat.

W moim przydługawym wstępie napomniałem o tym, że istnieje wiele metod robienia bumpmappingu, czyli po naszymu mapowania wypukłości i swe zdanie podtrzymuje. Wygląda jednak na dzisiaj, że najbardziej obiecującą metodą jest robienie map wypukłości za pomocą dosyć prostej, rzekłbym nawet barbarzyńsko prostej, ale diabelnie skutecznej sztuczki, w której swoje niecne łapska macza oczywiście powyżej przedstawione przeze mnie równanie. Praktycznie we wszystkich metodach w mapowaniu wypukłości biorą udział tekstury, bo bez nich nie idzie się nigdzie obejść. Jednak w przeciwieństwie do dotychczasowych naszych wysiłków dzisiaj tekstury posłużą nam - owszem - do przedstawiania wypukłości na obiekcie, ale w jakże odmienny, zupełnie perfidny sposób. A jeżeli dotrzwacie do końca tego artykułu to z przerażeniem stwierdzicie, jakie to wszystko jest dziecinnie proste ;-P.

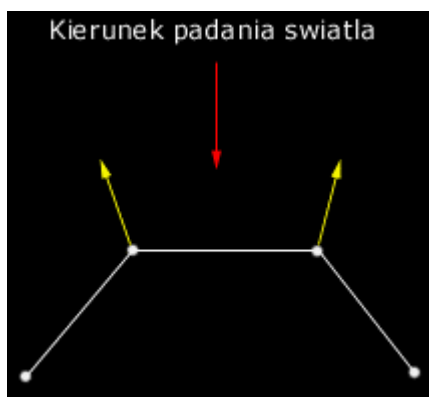
Zanim jednak poznamy szczegóły - trochę teorii na początek, jak zawsze. W rzeczywistości, którą tak namiętnie usiłujemy odwzorować cały czas w naszych przykładach, każdy obiekt - przedmiot jest zbudowany z materiałów, które mają swój kolor, fakturę, połyskliwość, itd... Animując nasze wyimaginowane obiekty na scenie 3D nakładaliśmy w różny sposób tekstury imitujące takie a nie inne właściwości materiałów, oświetlaliśmy je światłami itd... kombinowaliśmy jednym

słowem, żeby jak najwierniej oddać charakter materiału, z którego zbudowany był obiekt. W większości przypadków efekt był oczywiście zadowalający biorąc pod uwagę możliwości sprzętu i przeznaczenie aplikacji - w sumie w grach nikt nie zatrzymuje się i nie ogląda z czego są zrobione ściany mając na karku hordę żadnych krwi, obcych z marsa. Ale zarówno wymagania graczy jak i możliwości sprzętu ciągle rosną więc ciągle też kusi, żeby ten realizm podnosić no i dzisiaj stanęło na dwóch głównych trendach w grafice 3D - czyli dynamicznym oświetleniu i mapowaniu wypukłości.

Mapowanie wypukłości ma na celu dwie rzeczy - po pierwsze oddać fakturę materiału z jakiego wykonany jest dany obiekt a po drugie umożliwić znacznie tańszą w sensie obliczeniowym modyfikację obiektów, aby przypominały one to, co prawdziwe. Wyobraźmy sobie na przykład kulę ziemską - jeśli grafik na przykład chciałby modelować góry, doliny i całą resztę na modelu kulki, żebyśmy potem mogli to potem wyświetlić na ekranie to podejrzewam, że mielibyśmy poważne problemy z ilością wierzchołków, które nawet w sporym uproszczeniu pewnie zawałyby nam kartę grafiki i skutecznie spowolniły wszelkie operacje. Wykorzystując mapowanie wypukłości dzisiaj przedstawioną metodą osiągniemy znacznie lepsze efekty znacznie tańszym kosztem. Faktura materiału zaś i tak przy pewnym powiększeniu jest tak naprawdę problemem uplastycznienia naszej geometrii, więc tak czy siak na jedno wychodzi.

Mapowanie wypukłości robione metodą DOT3 jak wszystko inne w grafice 3D czasu rzeczywistego jest prostą sztuczką, która daje tylko złudzenie, choć bardzo realne i akceptowalne przez większość widzów. Jak doskonale wiemy z kilkunastu naszych lekcji o plastyczności i przestrzenności naszych scen decydują przede wszystkim dwie rzeczy - rzutowanie perspektywiczne dające złudzenie głębi naszych scen na dwuwymiarowym monitorze oraz element nadający scenom realizmu i życia czyli oświetlenie. Rzutowaniem nie będziemy się tutaj zajmować, bo wiele ono do naszej dzisiejszej sprawy nie wnosi, natomiast przyjrzymy się bliżej oświetleniu, bo o nim będzie dzisiaj głównie mowa. Tak naprawdę to wszystko co widzimy w rzeczywistym świecie zawdzięczamy światłu, które odbijając się od obiektów trafia do naszego oka w odpowiednich proporcjach generując obraz. Jeśli ktoś miałby władzę aby modyfikować zachowanie światła w momencie odbicia od przedmiotów to byłby w stanie nam fałszować obraz jaki widzimy - to oczywiste. Można by na przykład zmieniać widziane kolory, zmieniać obiektom fakturę czy je zniekształcać. I tutaj może się może zatrzymamy, bo słowo zniekształcać brzmi nader intrygująco i od razu nasuwa naszym chorym umysłem podświadomą informację - my przecież bardzo lubimy zniekształcać i fałszować ;-).

Spróbujmy zatem przyrzeć się temu bliżej - czy możemy w jakiś sposób wpływać na sposób odbicia światła od obiektów w Direct3D? Mając dotychczasową wiedzę - a i owszem. Poznaliśmy przecież niedawno fajny wynalazek - czyli vertex shader. Obliczając oświetlenie za jego pomocą dokonuje się właśnie operacji DOT3 na wektorach normalnych wierzchołków z wektorami światła dające wypadkową liczbę, która stanowi o charakterystyce światła w danym wierzchołku a aproksymując po całej bryle całkiem niezłe efekty oświetlenia od prostych do wymyślnych i niestandardowych sposobów cieniowania. No i można by w zasadzie się z tego ucieszyć, tyle tylko że to nam nic nie daje - zniekształcić obiekt może i się uda, ale pewnie w niewielkim stopniu bo na pewno będzie on przypominał to co poprzednio. Tę prostą zasadę widzimy na poniższym obrazku. Czerwonym kolorem oznaczyłem kierunek wektora, który oznacza padające promienie światła. Żółte strzałki to wektory imitujące normalne do wierzchołków siatki, białe elementy to oczywiście sama siatka.



Pasowałoby więc mieć możliwość zniekształcania na trochę niższym poziomie... co byście powiedzieli na przykład na piksele? Gdybyśmy mogli w jakiś sposób wpłynąć na renderowane na scenie piksele, powiązać je ze zmianami światła odbijanego od obiektu - to byłoby coś, tak wtedy moglibyśmy naprawdę oszukiwać :-). Ale czy to możliwe? - Na to pytanie mogę odpowiedzieć jedynym słusznym zdaniem w tym przypadku - "są na świecie rzeczy o których się programistom 3D nie śniło" ;-).

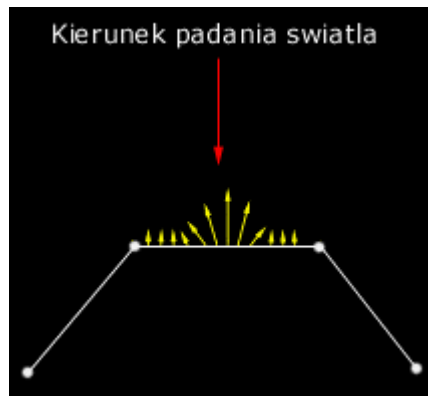
Powstrzymując was od lawiny pytań o sposób podsyć tylko złośliwie waszą ciekawość stwierdzając, że i owszem, ale jednoznacznie zachęcam cały czas do myślenia:

Przede wszystkim - mowa o pikselach a tutaj coś możemy na to poradzić - mamy przecież piksel shader. Tylko w jaki sposób powiązać te małe, wredne kropki na ekranie z wektorami i światłami, które nijak do pikseli sprowadzić się nie dadzą - czyżby niemożliwe?

We wszystkich dotychczasowych przykładach (a było ich naprawdę niewiele), w których mieliśmy do czynienia z oświetleniem wszystko wyglądało bardzo prosto. Wykorzystując wektor normalny do wierzchołka i kierunek lub położenie światła (w zależności od rodzaju światła) obliczaliśmy sobie to oświetlenie, lub robił to za nas wewnętrznie Direct3D (cały czas korzystając z operacji DOT3). Oświetlenie było wyliczane dla każdego wierzchołka i potem wykorzystując cieniowanie

płaskie (lub częściej Gourauda) wyliczone kolory wierzchołków były aproksymowane po całej bryle dają mniej lub bardziej zadowalający efekt, zależny w głównej mierze od ilości wierzchołków - im więcej tym lepiej. No ale im więcej wierzchołków tym więcej normalnych a co za tym idzie więcej obliczeń - a to bardzo, bardzo, bardzo niedobrze. Ten sposób oświetlenia królował do tej pory - było to tzw. oświetlenie liczone metodą "per vertex", czyli dla każdego wierzchołka. Sytuację nieco ratowały głównie w grach tzw. mapy świetlne (ang. lightmaps), które dawały niezłe efekty oświetlenia bez zwiększania komplikacji geometrii a bazowały na sztuczce z wielokrotnym nakładaniem tekstur - popis dały mapy świetlne choćby w legendzie - Quake-u, gdzie zastosowano je jakimś cudem nawet do dynamicznie przesuujących się obiektów. Na długie lata technika ta zakręlowała w grach i do dzisiaj jest na topie. No ale technologia idzie do przodu, wymagania również więc nieuchronnie kiedyś nastąpi jej koniec.

Stosując połączone techniki oświetlania per vertex or lightmaps tworzą dotychczas graficzne hity, które na zawsze zapadną w pamięć graczy w dziedzinie oprawy graficznej gier. Korci pewnie wielu pytanie - skoro dało się obliczać oświetlenie "per vertex", to czy nie da się "per pixel" - odpowiedź jest oczywista - pewnie, że się da! Wystarczy dla każdego piksela wyliczyć normalną w punkcie, w którym ten piksel występuje i potem już po staremu, tak jak w przypadku "per vertex". Kłopot tylko w tym, że ciężko tę normalną policzyć - można wprawdzie kombinować i aproksymować z trójkąta, uśredniać itd... ale ile czasu to zajmie? Brrr, wołę nie myśleć i wam też nie radzę, no na dzisiejszym sprzęcie jest to zupełnie nierealne... Popatrzmy na rysunek poniżej aby sobie przybliżyć ideę. Podobnie jak wyżej, żółte strzałki oznaczają normalne, ale tym razem umieszczone w poszczególnych pikselach na bryle (tutaj na jednej krawędzi trójkąta).



Skoro więc nie ma sensu liczyć... to czy nie można by przygotować jakiegoś zestawu danych dla każdego piksela na bryle, zawierającego potrzebne dane? Tak, to wygląda już o wiele lepiej, tylko znowu powstaje kwestia - jak te dane przechowywać i jak je potem aplikować bryle? Pewnie wielu z was nie raz zastanawiało się nad fundamentalnym pytaniem ile pikseli mieści się na danej bryle? Może co bardziej zapaleni nawet próbowali to liczyć, choć pewnie z marnym skutkiem... Tym, bardziej myśląc operacja wydaje się zupełnie bezsensowna - liczenie pikseli na bryle? Jakaś głupota! I w sumie racja, bo czegoś takiego po prostu sensu liczyć nie ma.

Powinniśmy w tym momencie zwiesić nos na kwintę a marzenia o oświetleniu "per pixel" odłożyć na czas nieokreślony, no bo skoro się na pikselach operować nie da to jak?

No ale my jesteśmy uparci i dociekliwi, więc dalej pytamy - no chwileczkę, a tekstury? Przecież tekstury nakładaliśmy na bryły a one piksele miały. Tutaj mała poprawka od razu - nie myślimy o elementach tekstury jako o pikselach a tzw. teksełach. Powiniennem to rozróżnić od razu w tutorialu o teksturach, ale lepiej późno niż wcale. Piksele kojarzą nam się ewidentnie z kropkami widocznymi na ekranie. Jeśli wyświetlamy teksturę w 2D, czyli tak jak naprawdę wygląda to i owszem - jej teksele pokrywają się pikselami ekranu, jeśli oglądamy ją w powiększeniu 1:1. Jeśli nawiniemy naszą teksturę bryłę i wrzucimy na scenę 3D to już jest inaczej. Bryła może być duża lub mała w stosunku do tekstury i wtedy teksele bryły mogą nam w pewnych przypadkach "wyleźć" brzydko mówiąc, czyli po prostu jeden tekseł tekstury pokryje o wiele za wiele pikseli na naszej bryle. Ideałem byłoby aby jeden tekseł tekstury pokrywał jeden piksel na bryle, ale tak się zrobić oczywiście nie da. Od tego mamy grafików, wymyślne algorytmy filtrowania, aby w efekcie na ekranie wyglądało to jak najbardziej w porządku.

W tym momencie musimy sobie zrobić małe podsumowanie - kombinujemy cały czas z tymi pikselami obiektu. Wiemy, że jak na razie się do nich bezpośrednio dostać nie da. Z drugiej strony są jednak tekstury nakładane na bryły ze swoimi teksełami i pikselami. Gdyby więc udało się stworzyć taką mapę, która w miarę dokładnie pokryłaby ewentualny obiekt swoimi teksełami, aby na ekranie dysproporcja pomiędzy pikselami a teksełami była jak najmniejsza to już byłibyśmy w połowie drogi - bo byłby sposób aby mieć możliwość odnosić się do poszczególnych pikseli obiektu. Tylko pytanie jak teraz połączyć tę mapę pikseli obiektu i normalne do każdego z nich. W tym momencie chyba już wszystko jest oczywiste, no ale aby być do końca porządnym i skrupulatnym jednak wyjaśnimy jeśli ktoś nie skumał.

Pozostała jedna kwestia - jak zapisać wektor złożony z trzech liczb float w jednym pikselu? Ktoś krzyczy - przecież to niemożliwe! Ktoś inny znowu - jeśli dacie odpowiednio duży piksel to zapiszę - nie ma problemu. A prawda jak zawsze jest gdzieś po środku. Wiemy, że dzisiaj do dyspozycji mamy mapy co najwyżej 32-bitowe. Wiemy też, że grafika to do pewnego stopnia uproszczenia i sztuczki, o czym trąbię już od początku dzisiaj. Wyszliśmy na początku od tego, że chcemy mieć możliwość zniekształcania światła w momencie odbicia - skoro będziemy zniekształcać to nie zależy nam przecież na dokładności - zadowolimy się na pewno kompromisem, który przy niewielkim nakładzie da nam oczekiwane rezultaty. I tak też będzie w tym przypadku - nie będziemy potrzebowali pikseli aby pomieścić całe liczby float, bo to zupełnie niepotrzebne

- na ekranie i tak nigdy nie zobaczymy różnic przy takiej dokładności. Wystarczy powiedzieć, że do zupełnie zadowalającego efektu wystarczy nam 8-bitowa dokładność, to znaczy, że jedna współrzędna wektora przyjmie jedną z 256 możliwych wartości. Ponieważ wektor składa się z 3 współrzędnych, więc dostajemy wręcz idealny zestaw - 8 bitów na każdą współrzędną a to przecież nic innego jak 24 bity - co jak dla piksela wydaje się być wartością zupełnie wystarczającą. No a mamy przecież jeszcze podział na kolory podstawowe, więc starczy powiedzieć, że jedna składowa wektora to nic innego jak czerwony, niebieski czy zielony kolor :-). Na koniec wystarczy dodać tylko tyle, że skoro będziemy mieć do czynienia z wektorami normalnymi to oczywiście musimy mieć je znormalizowane - wtedy maksymalna wartość koloru (255) odpowiada wartości 1.0f w wektorze.

I cóż można by rzec - wydawało się niemożliwe? A jednak... poniższy rysunek przedstawia przykładową mapę normalnych z umieszczonymi w niej danymi wektorów normalnych. Każdy piksel takiej mapy reprezentuje 24 bity, z których da się oczywiście złożyć trójwymiarowy wektor w sposób podany przeze mnie powyżej.



Oczywiście jak wszystko tutaj też oczywiście mamy ograniczenia - niestety bez nich się nie da. Im mniejszą mapę wygenerujemy dla obiektu tym efekt gorszy - ale o to niech nas na razie głowa nie boli - my nie będziemy się zajmowali jak na razie tworzeniem map a ich wykorzystywaniem. Załóżmy, że mapę normalnych w jakiś tam tajemniczych okolicznościach udało się nam zdobyć. Wszystko jest tak jak chcieliśmy - mamy do każdego piksela na obiekcie przechowaną normalną w postaci koloru. Teraz wystarczy tylko wziąć ten kolor, wyciągnąć z niego dane wektora normalnego, wykonać na nim operacje w wektorem światła i voila... oświetlenie "per pixel" mamy zrobione. Ale nie radzę się napalać - przed nami jeszcze droga bardzo daleka.

Jeśli nawet uda nam się poprawnie nałożyć teksturę zawierającą normalne na obiekt i wyświetlić to coś, to spróbujmy sobie to zaanimować i zauważyć pewną rzecz. Otóż mamy na scenie jakiś tam obiekt, który następnie obracamy sobie o dowolny kąt. Z tym nie ma oczywiście problemu, bo robimy to jednym palcem. Wyobraźmy sobie teraz, co się dzieje z normalnymi naszego obiektu - oczywiście powinny się one obracać razem z obiektem w odpowiedni sposób. Normalne te mamy zapisane w teksturze w sposób stały, tzn. bez względu na położenie obiektu w świecie dane tekstury nie zmieniają się. Jeśli obiekt posiadający normalne w mapie będzie wariował nam na scenie i obracał się jak szalony to mamy problem. Dlaczego? No bo normalne w mapie, bez względu na położenie obiektu na scenie zawsze pokazują ten sam kierunek, czyż nie? Jeśli na przykład nasza mapa normalnych jest zupełnie "płaska" (oczywiście umownie) - czyli wszystkie normalne wskazują powiedzmy w górę, w stosunku do powierzchni na której leżą. Nasz obiekt niech będzie zwykłym, płaskim kwadratem, który leży sobie na scenie. I teraz robimy tak - odwracamy nasz kwadrat do góry nogami, o 180 stopni. A mapa normalnych co? Ano, nie zmienia się! Jak normalne były skierowane do góry, tak są dalej - bo logicznym oczywiście jest to, że kolory mapy się nie zmieniają a co za tym idzie i kierunki normalnych - przecież one są zapisane w kolorach mapy. Aby wraz z obrotem obiektu obrócić nam się jego normalne z mapy, musielibyśmy oczywiście zmienić tę mapę - wziąć z mapy każdą normalną, zdekodować wartości, obrócić tak jak obiekt i zapisać do mapy z powrotem. Taka czynność oczywiście mija się zupełnie z celem w realtime, bo do niczego dobrego na pewno nas nie doprowadzi. Więc czy to porażka i nie tędy droga? Ech, chyba nie po to tyle brnęliśmy, żeby teraz zawrócić, co?

Ano, myślny dalej. Skoro mapy normalnych modyfikować się nie da przy obrocie obiektu, więc może w drugą stronę - może trzeba pokombinować coś ze światłem, które będzie padało na obiekt i które będzie "reagowało" z naszymi normalnymi zapisanymi w mapie.

W grafice 3D aby jakiegokolwiek przekształcenia i obliczenia na wektorach miały sens wszystkie biorące udział w operacji elementy (przeważnie wektory) muszą się znajdować w tej samej przestrzeni - czyli w skrócie mówiąc ich współrzędne muszą być identyfikowane w jednym, określonym układzie współrzędnych, który jednoznacznie określa ich wzajemne położenie względem siebie. My doskonale znamy takie przestrzenie, jest nią choćby przestrzeń naszego świata, w której mamy zdefiniowane modele, dokonujemy przekształceń obiektów itd.

Na naszej scenie światło będzie przeważnie zdefiniowane także w przestrzeni świata - czyli po prostu podamy żywcem dane o miejscu w którym światło się znajduje i jaki ma kierunek. A teraz zastanówmy się w jakiej przestrzeni są zdefiniowane nasze "pikselowe normalne" zapisane w mapie normalnych zaaplikowanej naszej bryle. Na samym początku można by pomyśleć, że i owszem - w przestrzeni świata, no bo mamy współrzędne niby takie jak powinny być. No ale po obrocie obiektu oczywistym faktem staje się, że tak nie jest, bo współrzędne normalnych się nie zmieniają (jak już pokazaliśmy powyżej) a nasze przekształcenie w świecie i owszem - więc coś jest nie tak, jak powinno być. Wygląda na to, że nasze normalne żyją sobie w jakimś innym światku, niezależnym od naszego świata całej sceny. I tak też jest w istocie - normalne bowiem żyją sobie tylko na teksturze i to jest ich faktyczny świat. I skoro nie możemy sobie przenieść naszych normalnych do świata sceny to przenieśmy scenę, a konkretnie światło z tej sceny do świata normalnych na teksturze - jeśli się nam to uda, to będziemy w domu i będziemy mogli sobie oświetlać obiekty na poziomie pikseli. A czy możliwa jest migracja ze świata sceny do tekstury? A czy muszę o to pytać? ;-)

Wektory normalne, zapisane w naszej mapie są tam umieszczone w taki a nie inny sposób głównie dlatego, że mapa jest nałożona jest na obiekt w jakiś tam specyficzny sposób mapowania, który aplikował grafik. Weźmy dla przykładu jeden trójkąt z naszej bryły, tak dla uproszczenia. Jeśli przypatrzeć się wektorom normalnym z fragmentu mapy nałożonej na ten właśnie trójkąt to możemy stwierdzić, że normalne te niejako są zdefiniowane w układzie wyznaczonym przez ten trójkąt a w szczególności przez jego wierzchołki. Wektor normalny, prostopadły do trójkąta będzie niczym jak odpowiednio skierowanym iloczynem wektorowym dwóch ramion trójkąta - to oczywiste. Pozostałe, trochę mniej prostopadłe będą definiować krzywiznę na tym trójkącie. I w zasadzie moglibyśmy się tym zadowolić, gdyby nie fakt, że mamy do czynienia z teksturami, które jak złośliwe bydło mogą sobie po bryle chodzić tam, gdzie chcą a normalna jest zapisana właśnie w niej. Potrzebujemy więc jakiegoś sposobu, aby przenieść nasz kochany wektor światła do przestrzeni tej tekstury normalnych. Takim sposobem oczywiście zawsze jest odpowiednie przekształcenie - macierz a tą z kolei jesteśmy w stanie wygenerować za pomocą trzech wektorów, które stanowią osie układu współrzędnych naszego świata. A jak te osie wyznaczyć? Ano nic prostszego - jedną na pewno uda nam się wyznaczyć z iloczynu skalarnego dwóch pozostałych, pozostało więc znaleźć tylko dwie ;-).

Wiemy, że każdy trójkąt na bryle może mieć własny sposób mapowania - znaczy się, trzy wierzchołki definiujące ten trójkąt w jakiś tam charakterystyczny sposób wyznaczają rozkład tekstury na bryle. Na jednym tekstuura może być położona w poziomie (jeśli uwzględnimy naturalne położenie tekstury na ekranie w postaci płaskiej) a na innym w pionie a na jeszcze innym całkiem fikuśnie - zależy od tego, jak grafik nam zrobi. Jak wyznaczyć zatem dwie osie układu współrzędnych tekstury, który z kolei będzie stanowił o przekształceniu aplikowanemu wektorowi światła przy przenoszeniu do tej przestrzeni?

Ano nic trudnego. Mówiąc najprościej jak się da - musimy pokazać w jaki sposób zmieniają się współrzędne mapowania w danym wierzchołku w zależności od zmian współrzędnych samych wierzchołków tworzących dany trójkąt. Brzmi koszmarnie? Nie przejmujcie się - kiedy zobaczycie kod to wystarczy tylko zapamiętać regułę - jak szefowi pokażecie efekt końcowy to na pewno nie przyjdzie mu do głowy pytać - jak to działa ;)

Osie układu wyznaczającego przestrzeń tekstury nazywali będziemy w skrócie wektorami bazowymi **U** i **V** - dlaczego, próżno dociekać, w każdym razie jak się gdzieś spotkacie z takim nazewnictwem to żebyście się nie zdziwili. Procedura wyznaczania wektorów bazowych będzie wyglądała następująco:

Dla każdego trójkąta siatki znajdujemy jego trzy wierzchołki. Aby wyznaczyć kolejno rozkłady współrzędnych mapowania względem współrzędnych wierzchołków robimy tak:

Założmy sobie, że wektor w przestrzeni definiuje nam typ złożony z trzech liczb float:

```
struct Vector3f
{
    float x;
    float y;
    float z;
};
```

natomiast na płaszczyźnie analogiczny typ:

```
struct Vector2f
{
    float x;
    float y;
};
```

Założmy, że wierzchołki naszego trójkąta w siatce to **vert1**, **vert2**, **vert3**. Każdy z nich posiada zdefiniowaną oczywiście pozycję w świecie w postaci współrzędnych **x**, **y** i **z** (jako **Vector3f**) a także współrzędne mapowania tekstury - w postaci pary **x** i **y** umieszczonych w typie **Vector2f**.

Ogólnie biorąc jest typu:

```
SVertex
{
    SVector3f pos;
    Svector3f normal;
    Svector2f tex;
    Svector3f U;
    Svector3f V;
    Svector3f UxV;
};
```

Dla składowej **x** wektorów bazowych w przestrzeni tekstury **U** i **V** wykonujemy następującą operację:

```

Vector3f vect1;
Vector3f vect2;
Vector3f vect1x2;

vect1.x = vert2.pos.x - vert1.pos.x;
vect1.y = vert2.tex.x - vert1.tex.x;
vect1.z = vert2.tex.y - vert1.tex.y;

vect2.x = vert3.pos.x - vert1.pos.x;
vect2.y = vert3.tex.x - vert1.tex.x;
vect2.z = vert3.tex.y - vert1.tex.y;

```

Następnie obliczamy iloczyn wektorowy wektorów **vect1** i **vect2**.

```
vect1x2 = vect1 x vect2;
```

Mając składowe wszystkich trzech możemy przystąpić do wyznaczania wektorów bazowych **U** i **V**, które nas w tym momencie najbardziej interesują. A robimy to w sposób następujący:

```

vert1.U.x += -vect1x2.y / vect1x2.x;
vert1.V.x += -vect1x2.z / vect1x2.x;

vert2.U.x += -vect1x2.y / vect1x2.x;
vert2.V.x += -vect1x2.z / vect1x2.x;

vert3.U.x += -vect1x2.y / vect1x2.x;
vert3.V.x += -vect1x2.z / vect1x2.x;

```

Widać, że dla każdego wierzchołka w siatce sumujemy kolejne wartości wektorów bazowych. Da nam to w rezultacie na samym końcu w pewien sposób uśrednione wektory bazowe, co zaowocuje ładnymi krzywiznami na obiekcie a nie jakimiś śmieciami, które zepsują nam cały efekt. Skoro składową **x** mamy już obliczoną, czas na **y** - a jak to będzie przebiegać nie trudno przewidzieć. Ale dla porządku:

```

vect1.x = vert2.pos.y - vert1.pos.y;
vect1.y = vert2.tex.x - vert1.tex.x;
vect1.z = vert2.tex.y - vert1.tex.y;

vect2.x = vert3.pos.y - vert1.pos.y;
vect2.y = vert3.tex.x - vert1.tex.x;
vect2.z = vert3.tex.y - vert1.tex.y;

vect1x2 = vect1 x vect2;

vert1.U.y += -vect1x2.y / vect1x2.x;
vert1.V.y += -vect1x2.z / vect1x2.x;

vert2.U.y += -vect1x2.y / vect1x2.x;
vert2.V.y += -vect1x2.z / vect1x2.x;

vert3.U.y += -vect1x2.y / vect1x2.x;
vert3.V.y += -vect1x2.z / vect1x2.x;

```

Jak widać, po prostu podmieniamy w odpowiednich miejscach współrzędną **y** w obliczeniach, tam gdzie jest wykorzystywana pozycja wierzchołka. I analogicznie dla składowej **z** to już nic trudnego:

```

vect1.x = vert2.pos.z - vert1.pos.z;
vect1.y = vert2.tex.x - vert1.tex.x;
vect1.z = vert2.tex.y - vert1.tex.y;

vect2.x = vert3.pos.z - vert1.pos.z;
vect2.y = vert3.tex.x - vert1.tex.x;
vect2.z = vert3.tex.y - vert1.tex.y;

vect1x2 = vect1 x vect2;

vert1.U.z += -vect1x2.y / vect1x2.x;

```

```

vert1.V.z += -vect1x2.z / vect1x2.x;

vert2.U.z += -vect1x2.y / vect1x2.x;
vert2.V.z += -vect1x2.z / vect1x2.x;

vert3.U.z += -vect1x2.y / vect1x2.x;
vert3.V.z += -vect1x2.z / vect1x2.x;

```

Po przeglądnięciu każdego trójkąta w naszej siatce pozostaje już tylko jedno. Jak widać wektory bazowe **U** i **V** umieściliśmy w strukturze wierzchołka, bo tam też najlepiej je przechowywać (dlaczego, to okaże się w przykładach praktycznych). Mamy dwa wektory bazowe, potrzebny jest nam trzeci, z którym jednak nie ma najmniejszego problemu. Jak powiedzieliśmy wcześniej wystarczy znowu posłużyć się znowu fajną operacją jaką jest iloczyn wektorowy. Zanim jednak do tego przystąpimy musimy znormalizować wszystkie dotychczas obliczone wektory bazowe **U** i **V**:

```

for( int i = 0; i < ilosc_wierzchoлков; i++ )
{
    vert[i].U = VectorNormalize( vert[i].U );
    vert[i].V = VectorNormalize( vert[i].V );

    vert[i].UxV = vert[i].U x vert[i].V;

    if( VectorDot( vert[i].UxV, vert[i].normal ) < 0.0f )
    {
        vert[i].UxV =- vert[i].UxV;
    }
}

```

Po normalizacji oczywiście wykonujemy iloczyn wektorowy aby otrzymać wektor **UxV** i również składujemy go w strukturze wierzchołka. Dodatkowo mamy pewne zabezpieczenie o którym nie wspominałem wcześniej. Otóż może się okazać, że podczas naszych kombinacji z wektorami końcowy wektor bazowy **UxV** wyjdzie niemalże w przeciwnym kierunku niż normalna samego wierzchołka - tak oczywiście być nie może i w tym przypadku badamy w prosty sposób zwrot tego wektora. Wystarczy sprawdzić iloczyn wektorowy wektora bazowego **UxV** i wektora normalnego - jeśli wyjdzie mniejszy od zera to znaczy, że kąt pomiędzy tymi wektorami jest większy niż 90 stopni a to oznacza tak po prawdzie przeciwne zwroty i musimy zmienić zwrot **UxV** aby było poprawnie.

I kiedy mogłoby się wydawać, że to już koniec naszej męki, wyświetlamy sobie piękną bryłę na ekranie i widzimy co? Ano zaczynają nam się na niektórych bryłach robić rzeczy, które nam się absolutnie nie podobają. Wróćmy do naszego przykładu z kulą ziemską, bo to bardzo dobry przykład omawianej sytuacji. Jeśli rysujemy ziemię to przeważnie tekstura reprezentująca kolory jak normalne otacza całą kulę - można sobie to zresztą łatwo wyobrazić. W każdym razie mapowanie wygląda tak, że koniec i początek naszej tekstury spotykają się na kuli wzdłuż któregoś z południków. W takim miejscu spotkania oczywistym jest to, że trzeba tam zduplikować wierzchołki, ponieważ dla początku i końca tekstury nie mogą mieć takich samych współrzędnych mapowania - to oczywiste. Niestety manewr duplikowania wierzchołków niejako burzy cały nasz dotychczasowy wysiłek - w takim miejscu, gdzie stykają się wierzchołki robi się tzw. szew - brzydkie coś, czego na pewno nie chcemy na naszej bryle. Wygląda to dosyć szpetnie i zaczyna bardzo nam przeszkadzać. Cóż więc zrobić w takim przypadku? Trzeba niestety dokonać dodatkowych obliczeń, które pozwolą nam wygenerować poprawne dane. Cały ból z tym wrednym szwem jest taki, że trochę źle są tworzone lokalne przestrzenie - w miejscu styku końców tekstury wektory przestrzeni się nie pokrywają a powinny, zwłaszcza jeśli chodzi o normalne. Grzebiąc w różnych opracowaniach doszukałem się następującego sposobu, który zaleca między innymi i NVidia:

- należy obliczyć normalne wszystkich trójkątów na bryle,
- to samo dla pozostałych wektorów definiujących przestrzeń tekstury,
- dla każdego wierzchołka bryły uśrednić każdy z wektorów definiujących przestrzeń tekstury w taki sposób, że bierzemy wektory z trójkątów, do których dany wierzchołek należy,
- dla każdego wierzchołka uśrednić wektory przestrzeni tekstury dla zduplikowanych wierzchołków, czyli występujących w tym samym miejscu, ale mających np. inne współrzędne mapowania,
- na koniec dokonać pewnej sztuczki, polegającej na tym, że uśrednione wektory przestrzeni tekstury powinny tworzyć osie układu współrzędnych (być do siebie prostopadłe), co po uśrednianiu nie zawsze będzie prawdą. Zabieg ten nosi nazwę ortogonalizacji.

Sądzę, że łatwo sobie wyobrazić cały proces. Szczególną uwagę pragnąłbym zwrócić na punkt przed ostatni, bo jest on bardzo ważny i bardzo przysłuży się naszej kulce. Zabieg uśredniania wektorów przestrzeni tekstury wystąpi dla kulki zmapowanej w podany przykładowy sposób w kilu newralgicznych miejscach - oczywiście na każdym z biegunów, ponieważ tam schodzi się najwięcej wierzchołków mających różne mapowanie no i w każdej parze leżącej na szwie tekstury na południku ziemskim. Po uśrednieniu wektorów w tych miejscach szew na pewno nam z kulki zniknie i będzie ona gładka jak pupcia niemowlaka. Narzędzie takie naprawdę napisać jest bardzo prosto i każdy może się uporać z nim w króciutkim czasie,

zresztą na sieci wala się na pewno tego sporo. Po przepuszczeniu przez takie narzędzie nasza siatka naprawdę będzie przygotowana na najgorsze... Po obliczeniu wszystkich wektorów bazowych umieścimy każdy z nich w odpowiednim wierzchołku. Mając dane trzy wektory będziemy mogli utworzyć z nich układ współrzędnych, który jak powiedziałem wcześniej posłuży do stworzenia w następnej kolejności macierzy przekształcenia z przestrzeni świata do przestrzeni tekstury wektorów światła. Jeśli przeniesiemy wektory światła w czasie działania programu do przestrzeni tekstury to będziemy mogli na nich wykonać działania razem z wektorami zapisanymi w naszej mapie normalnych wykorzystując naszego dzisiejszego bohatera, czyli operację DOT3, który standardowo służy do wyliczania natężenia oświetlenia, tylko w tym przypadku nie w wierzchołkach a na pikselach tekstury. A jaki będzie tego efekt końcowy - zobaczycie w przykładach praktycznych, które już niebawem.

Pozostaje nam jeszcze tylko jedna, niezwykle ważna kwestia - czyli mapy normalnych - skąd je brać i jak generować? Ano tutaj wyjść jest naprawdę wiele. Można oczywiście generować je samemu, algorytmy są dosyć proste i znaleźć na necie nie będzie problemu. Dla leniwych wszyscy liczący się producenci zarówno oprogramowania jak i sprzętu sporządzili jednak gotowe narzędzia, które po prostu wypalają mapę takich normalnych dla obiektów bez żadnego problemu - liderem w tym niewątpliwie jest NVdia, która ostatnio wypuściła kosmiczne wynalazki dla developerów, ułatwiające po prostu życie. Mamy więc narzędzia potrafiące wygenerować mapę normalnych z mapy wysokości (w odcieniach szarości) zarówno w postaci programu pracującego z linii komend, zupełnie bez okien, jest aplikacja okienkowa jakby kto chciał zobaczyć wyniki. Jest plugin do programu Photoshop, który zamienia mapę szarości - wysokości na mapę normalnych i od razu generuje nawet podgląd 3D. Ostatnio pojawiło się narzędzie o nazwie Melody - które pobiera dwa modele tego samego obiektu o różnej ilości wierzchołków ale opisujących ten sam kształt i na ich podstawie dokonuje naprawdę cudów, choć cuda te trwają bardzo długo ;). ATI oczywiście gorsza od konkurencji nie jest i ma podobne wynalazki, choć ich bliżej nie testowałem. Programy do modelowania też nie pozostają w tyle - do pocziwego 3DMaxa jest fajny Texporter - zdaje się, że nawet darmowy plugin do generowania map normalnych, jest także bardziej zaawansowane narzędzi czyli Caldera. Jak w innych, powiem szczerze nie wiem, bo gdybym zaczął sprawdzać to chyba nigdy tego artykułu bym nie dokończył, ale to co jest powinno wam w zupełności wystarczyć. Jak starczy czasu to na pewno napiszemy także artykuł o generowaniu map normalnych aby każdy mógł sobie taką mapę wypalić. Dla potrzeb nauki w zupełności wystarczy nam to, co będziemy w stanie znaleźćw necie.

No i ... no i w zasadzie koniec możnaby powiedzieć. To wszystko jeśli chodzi o teorię oświetlenia per pixel i mapowanie wypukłości, które jak się okazuje jest naprawdę proste. W przykładach praktycznych zobaczymy jak zrealizować to w praktyce za pomocą Direct3D jak i OpenGL (miejmy nadzieję, że kolega Domino się sprawi ;-).