Odnieś sukces na rynku gier!

Profesjonalne tworzenie gier internetowych dla systemu Android

w językach HTML5, CSS3 i JavaScript

Juriy Bura • Paul Coates

Helion

Apress®

Tytuł oryginału: Pro Android Web Game Apps: Using HTML5, CSS3 and JavaScript

Tłumaczenie: Krzysztof Rychlicki-Kicior

ISBN: 978-83-246-7401-5

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/prtwgi
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/prtwgi.zip

Printed in Poland.

Spis treści

Część III Łączenie światów

ROZDZIAŁ 8

3D w przeglądarce

W pierwszej części książki nauczyliśmy się tworzyć gry dwuwymiarowe. Przez wiele lat tego rodzaju gry definiowały maksimum możliwości, jakie można było osiągnąć. Dopiero później pojawienie się coraz bardziej realistycznych gier trójwymiarowych zaczęło zmieniać ten stan rzeczy. Jedynym powodem, dla którego nie możesz uruchomić *Skyrima* na swoim telefonie, są możliwości Twojego urządzenia.

Silniki trójwymiarowe wymagają znacznie więcej mocy obliczeniowej w porównaniu z dwuwymiarowymi. W przypadku silników dwuwymiarowych renderowanie jest całkiem proste — wystarczy wyświetlić na płótnie piksele pobrane z obrazka i ewentualnie uwzględnić jego przezroczystość. Silnik trójwymiarowy wymaga znacznie więcej mocy — związane z nim przekształcenia matematyczne są znacznie bardziej skomplikowane. Jest to też jeden z istotnych powodów, dla których przeglądarki (zarówno stacjonarne, jak i mobilne) do niedawna nie były w stanie natywnie obsługiwać możliwości 3D.

W tym rozdziale skupimy się wyłącznie na tworzeniu trójwymiarowych aplikacji przy użyciu przeglądarki. Obsługa trzech wymiarów w przeglądarce jest możliwa dzięki „standardowej" implementacji, nazywanej WebGL. W świecie mobilnym WebGL stawia dopiero pierwsze kroki. W momencie pisania tej książki jedyną przeglądarką (dostępną dla urządzeń z systemem Android), która obsługuje ten standard, jest Firefox Mobile[1]. Sony Xperia PLAY obsługuje WebGL w domyślnej przeglądarce. Ze względu na dynamiczny rozwój tej technologii w przeglądarkach stacjonarnych możemy się spodziewać, że niebawem będzie ona powszechnie dostępna również na urządzeniach mobilnych.

Im więcej urządzeń będzie obsługiwało grafikę trójwymiarową, tym więcej gier w tej technologii będzie dostępnych w naszych przeglądarkach. Z tego względu programista powinien wiedzieć, jak działają mechanizmy renderowania 3D i co trzeba samodzielnie zaimplementować, aby wszystko działało jak należy. Ten rozdział jest w całości poświęcony podstawom grafiki trójwymiarowej. Jednocześnie stanowi on niejako wprowadzenie do rozdziału 9., w którym poznamy podstawy WebGL.

W tym rozdziale poznasz następujące zagadnienia:

- podstawy renderowania 3D: jak przedstawiać bryły na płaskim ekranie,
- podstawy algebry macierzy: podstawowe przekształcenia matematyczne kryjące się za grafiką trójwymiarową,
- perspektywa i rzuty: jak sprawić, by scena była bardziej naturalna.

Celem tego rozdziału jest utworzenie prostego dema, w którym wyrenderujemy podstawowy model 3D. Rozpoczniemy od najprostszego kształtu — sześcianu.

[1] W momencie tłumaczenia częściowe wsparcie oferuje też Opera Mobile — *przyp. tłum.*

Renderowanie 3D — wprowadzenie

Czym właściwie jest grafika 3D? Pod jakim względem różni się ona od grafiki dwuwymiarowej, którą stosowaliśmy w poprzednich rozdziałach? Grafika 3D jest rzecz jasna iluzją — podobnie jak animacje. Renderując animację, tak naprawdę zmieniamy ramki na tyle szybko, aby użytkownik odniósł wrażenie, że obiekt się porusza. W rzeczywistości nie ma mowy o jakimkolwiek prawdziwym ruchu. Piksele pozostają na swoim miejscu — zmienia się jedynie ich kolor, co nie zmienia faktu, że na ekranie osiągamy efekt ruchu.

Renderowanie sceny trójwymiarowej przebiega podobnie. Wyświetlacz jest płaski, ale scena jest renderowana w taki sposób, aby użytkownik miał wrażenie, że patrzy na trójwymiarowy świat przez okno. Teraz musimy zadać sobie pytanie: czy skoro grafika 3D stanowi tak naprawdę specjalny przypadek grafiki 2D (którą dobrze znamy), to możemy zastosować kontekst płótna 2D do rysowania sceny trójwymiarowej? Jest to w istocie bardzo dobre pytanie. W praktyce utworzenie bardzo prostego silnika 3D jedynie przy użyciu płótna jest możliwe, jednak będziemy musieli sobie poradzić z dwoma podstawowymi problemami. Przede wszystkim będziemy musieli zmagać się ze znacznie większą liczbą obliczeń i przekształceń matematycznych. Ilość pracy, jaką trzeba włożyć od strony matematycznej (także biorąc pod uwagę wszelkiego rodzaju optymalizacje), w przypadku grafiki 3D i 2D różni rząd wielkości. Nawet jeśli poświęciłbyś kilka lat na nauczenie się wszystkiego, co konieczne do stworzenia własnego silnika 3D, szybko napotkałbyś inny problem: brak mocy urządzenia potrzebnej do obsługi zaimplementowanego własnoręcznie renderera 3D.

Gdy gracze, programiści tworzący gry i producenci sprzętu zrozumieli, że grafika trójwymiarowa stanowi niezbędny element gier komputerowych, zauważyli też, że procesor (ang. *Central Processing Unit*, CPU — centralna jednostka przetwarzania) nie jest najlepszym narzędziem do renderowania tego rodzaju grafiki. Podjęli więc decyzję o dodaniu specjalnych kart graficznych, które są odpowiedzialne wyłącznie za tego rodzaju obliczenia. Często karty te są droższe od samych procesorów! W języku JavaScript w przeglądarkach nie możesz bezpośrednio uzyskać dostępu do GPU (ang. *Graphics Processing Unit* — jednostka przetwarzania grafiki) w celu bezpośredniego wykonania renderowania właśnie w niej. Jesteś ograniczony tylko do silnika języka JavaScript, który wszystkie instrukcje wykonuje w ramach zwykłego procesora (CPU). Procesory nie są przystosowane do wykonywania tego rodzaju operacji, dlatego renderowanie zajmie znacznie więcej czasu niż w przypadku GPU.

Zanim w świecie aplikacji internetowych pojawiło się WebGL, utworzenie własnego renderera 3D stanowiło jedyną metodę obsługi grafiki trójwymiarowej w przeglądarce bez użycia Flasha lub Javy. Programiści tworzyli własne silniki; niektóre z nich były naprawdę interesujące. Niestety, koniec końców zawsze czegoś brakowało, ponieważ nie da się stworzyć gry trójwymiarowej o przyzwoitej jakości w ramach kontekstu 2D. WebGL działa inaczej — dzięki niemu jesteś w stanie przekazać najgorszą część obliczeń do GPU.

Większość telefonów dostępnych na rynku nie dysponuje jeszcze odrębną jednostką GPU, niemniej jest to tylko kwestia czasu. Co zatem stanie się, jeśli uruchomisz WebGL na takim urządzeniu? Czy to API w ogóle zadziała? Tak, oczywiście. Problem w tym, że do renderowania sceny zostanie użyty CPU, przez co nie uzyskasz zbyt wielu klatek na sekundę. Gdy scena jest renderowana przez GPU, mamy do czynienia z *renderowaniem sprzętowym* (ang. *hardware rendering*) — oznacza to, że obliczenia są wykonywane w ramach GPU, który jest przystosowany do tego typu pracy. Z drugiej strony renderowanie może być też przeprowadzone *programowo* (ang. *software rendering*). Ma ono miejsce, gdy renderujesz scenę przy użyciu CPU, np. z poziomu języka JavaScript bez użycia WebGL lub na urządzeniu, które w ogóle nie zawiera osobnej jednostki GPU.

■ **Uwaga** Problemy z działaniem WebGL możesz mieć nawet na swoim komputerze stacjonarnym. Istnieje specjalna „czarna lista" kombinacji urządzenie – sterownik – system operacyjny, które powodują problemy z działaniem tego API. Jeśli konfiguracja Twojego komputera zakwalifikuje się na tę czarną listę, kontekst WebGL nie będzie akcelerowany (przyspieszany), a nawet może w ogóle nie działać. Jeśli chcesz zaryzykować, możesz ustawić specjalny parametr, który spowoduje zignorowanie czarnej listy w trakcie działania przeglądarki. W przypadku Chrome musisz uruchomić przeglądarkę z parametrem `ignore-gpu-blacklist`. W Firefoksie musisz przejść na stronę *about:config* i ustawić zmienną `webgl.force-enabled` na `true`.

Jak działa renderowanie 3D?

Ten podrozdział stanowił dla mnie jedno z największych wyzwań podczas pisania tej książki. Renderowanie 3D to temat niezwykle skomplikowany — nietrudno byłoby wypełnić treścią jedną, a nawet dwie książki poświęcone tylko temu zagadnieniu. Spróbuję wyjaśnić tylko najistotniejsze zagadnienia, abyś był w stanie zrozumieć bardziej zaawansowane algorytmy kryjące się za całym procesem. Postaram się również ograniczyć matematyczne opisy do minimum, jednak nie da się ich zupełnie uniknąć.

Przekształcenia matematyczne

Zacznijmy od dobrej informacji — wszystkie operacje wykonywane w przestrzeni dwuwymiarowej będą działać prawidłowo również w trójwymiarze. Wszystko, co robimy w świecie 3D, działa podobnie jak w przypadku dwuwymiarowym, jednak musimy uwzględnić jeszcze jeden wymiar, nazywany zwyczajowo „z". W świecie 2D środek odcinka oblicza się zgodnie z wzorami:

```
x = (x1 + x2)/2, y = (y1 + y2)/2
```

Jeśli dodamy jeszcze jeden wymiar, otrzymamy wzory na środek odcinka trójwymiarowego:

```
x = (x1 + x2)/2, y = (y1 + y2)/2, z=(z1 + z2)/2
```

To samo stwierdzenie jest prawdziwe przy obliczaniu długości odcinka pomiędzy dwoma punktami. W przypadku dwuwymiarowym wzór jest następujący:

```
d = sqrt((x1 - x2)^2 + (y1 - y2)^2)
```

Po dodaniu trzeciego wymiaru otrzymujemy wzór:

```
d = sqrt((x1 - x2)^2 + (y1 - y2)^2 + (z1 - z2)^2)
```

Czemu jest to takie istotne? Niektóre zagadnienia łatwiej zrozumieć w przypadku dwuwymiarowym — wzory i ilustracje są znacznie prostsze. Po zrozumieniu tych zagadnień w 2D zastosowanie ich w świecie trójwymiarowym staje się proste. Jeśli jakakolwiek omawiana kwestia jest dla Ciebie trudna do zrozumienia, spróbuj przeanalizować ją w przypadku dwuwymiarowym. Przeniesienie jej potem na grunt trójwymiarowy nie będzie stanowić dla Ciebie problemu.

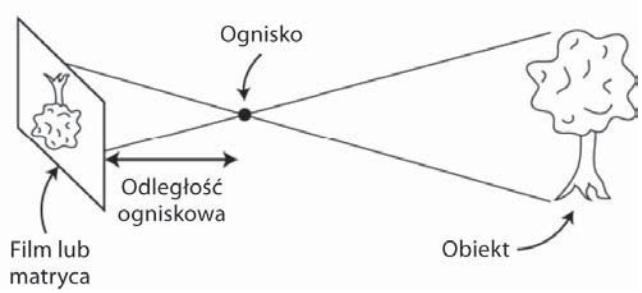
Oczywiście od każdej reguły są pewne wyjątki. W przypadku świata 2D linie albo się przecinają, albo są do siebie równoległe. W trójwymiarze linie mogą nie być równoległe ani się nie przecinać.

Przykład w trójwymiarze

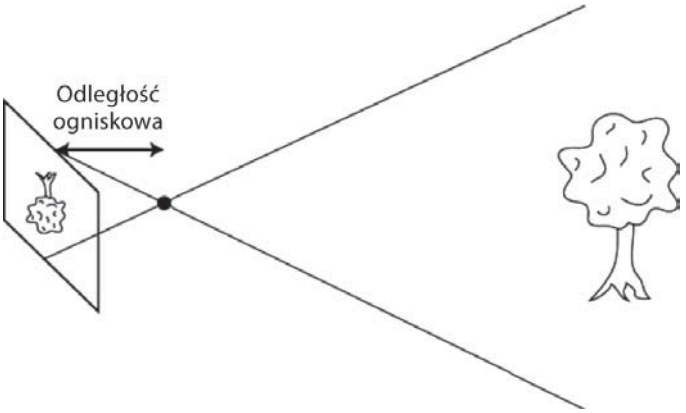
Zacznijmy od przykładu wziętego prosto z życia. Jaki jest najprostszy przykład renderowania świata trójwymiarowego na powierzchni dwuwymiarowej? Oczywiście jest to zdjęcie lub film wideo. Aparat robi zdjęcie sceny trójwymiarowej i umieszcza je na powierzchni 2D. Dawniej powierzchnią był film (taśma) — obecnie jest to cyfrowa matryca. Rysunek 8.1 przedstawia zasadę działania aparatu.

Światło odbijane przez obiekty jest przechwytywane przez soczewki i przekazywane do matrycy. Obraz jest odwrócony, co wynika ze sposobu działania soczewki.

Aparaty cyfrowe udostępniają funkcję przybliżenia (ang. *zoom*). Wciśnięcie przycisku powoduje powiększenie widocznego obiektu. Tak naprawdę wewnątrz aparatu następuje zmiana odległości ogniskowej, co przekłada się na zwiększenie lub zmniejszenie widocznego obszaru (rysunek 8.2). Większy widoczny obszar oznacza możliwość ujęcia na zdjęciu większej liczby obiektów przy jednoczesnym zmniejszeniu ich rozmiarów — innymi słowy, jest to operacja oddalenia (ang. *zoom out*). Odwrotna operacja powoduje zmniejszenie obszaru widoku i powiększenie obiektów, czyli jest to operacja powiększenia (ang. *zoom in*).

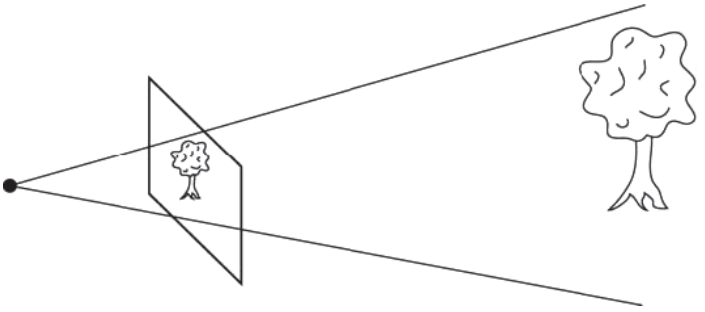


Rysunek 8.1. *Zdjęcie z aparatu stanowi dobry przykład dwuwymiarowej projekcji używanej do przedstawienia fragmentu sceny trójwymiarowej*



Rysunek 8.2. *Zmiana ogniskowej powoduje przybliżanie lub oddalanie: im większy staje się obszar widoku, tym mniejszy jest obiekt na ekranie*

Renderowanie sceny 3D przebiega niemal identycznie, przy czym nie korzystamy rzecz jasna z żadnych soczewek. Nie musimy więc umieszczać „docelowej" powierzchni dwuwymiarowej za ogniskiem. Musimy zachowywać się tak, jakbyśmy stali przed kamerą (rysunek 8.3).



Rysunek 8.3. *W grafice komputerowej nie ma soczewek. Z tego względu płaszczyzna „przechwytująca" scenę musi się znaleźć przed ogniskiem. Mimo różnic zasady przybliżania i oddalania pozostają niezmienione*

Przeanalizujmy teraz różnice pomiędzy renderowaniem sceny 3D a robieniem zdjęcia. W przeciwieństwie do świata rzeczywistego scena 3D jest tworzona z trójkąta (jest to do tej pory

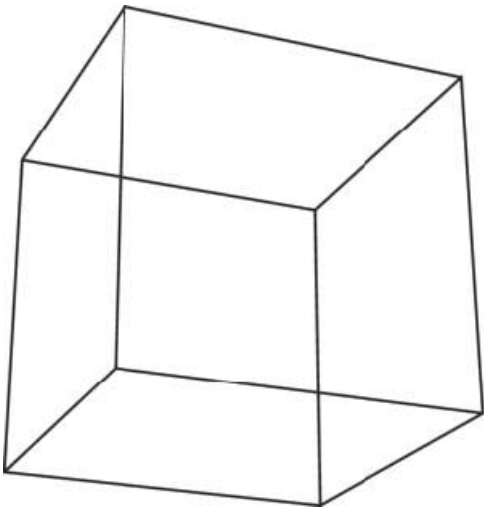
najlepszy sposób przedstawiania trójwymiarowych modeli). Trójkąty nie mają żadnych fizycznych właściwości, w przeciwieństwie do obiektów, z którymi stykasz się każdego dnia. Nie mają wagi, koloru, nie są stworzone z konkretnego materiału itd. Wszystkie właściwości musimy zasymulować, korzystając ze skomplikowanych modeli matematycznych.

Klikając przycisk aparatu w celu zrobienia zdjęcia, masz szczęście, gdyż proste zjawiska fizyczne umożliwiają osiągnięcie pożądanego efektu: światło odbija się w określony sposób od powierzchni, będąc mniej lub bardziej zaburzone w wyniku mgły, wysokiej temperatury lub innych warunków atmosferycznych. Koniec końców dociera ono do aparatu lub siatkówki Twojego oka, tworząc obraz. W świecie trójwymiarowym musimy samodzielnie zaimplementować prawa fizyki i przeprowadzać skomplikowane obliczenia, aby uzyskać pożądany efekt, który będzie widoczny dla użytkownika. To właśnie dlatego silniki trójwymiarowe są takie skomplikowane.

Na szczęście mam dla Ciebie dobrą wiadomość — nie musisz przeprowadzać superrealistycznych symulacji wszystkich praw optyki, aby stworzyć silnik 3D. Jeśli pominiemy symulację oświetlenia i materiałów, silnik staje się w istocie dość prosty — jego zadaniem jest jedynie wyświetlanie wielokątów na ekranie. Programem typu *Witaj, świecie!* dla grafiki trójwymiarowej jest obracający się sześcian. Dojdziemy do tego — i to bez ani jednej instrukcji z WebGL. Dlaczego? Bo w ten sposób znacznie lepiej zrozumiesz zasady działania grafiki trójwymiarowej.

Uproszczony silnik 3D

W tym podrozdziale utworzymy najprostszy możliwy silnik trójwymiarowy, korzystając jedynie ze znanego nam API płótna. Rozpoczniemy od reprezentacji obiektów 3D w języku JavaScript. Następnie przejdziemy do macierzy — matematycznych obiektów pomagających renderować modele na ekranie. Dzięki macierzom i przekształceniom będziemy w stanie obrócić sześcian na ekranie. Na koniec zajmiemy się perspektywami i rzutami (projekcjami), które sprawią, że nasz model będzie bardziej naturalny. Naszym celem jest wyświetlenie obracającego się sześcianu (rysunek 8.4).



Rysunek 8.4. *Ostateczna wersja aplikacji wykonywanej w tym rozdziale — renderujemy sześcian*

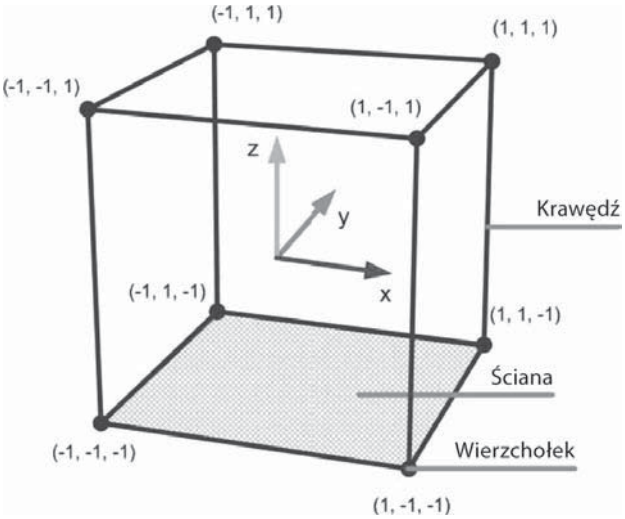
Model i scena

Najpierw musimy określić dane niezbędne do wyrenderowania sceny. Przede wszystkim konieczne jest posiadanie listy wierzchołków tworzących sześcian lub dowolny inny model, który chcemy wyświetlić. Następnie musimy zdefiniować połączenia między punktami, czyli krawędzie. Pierwszym krokiem tworzenia tej aplikacji będzie zdefiniowanie sześcianu w języku JavaScript.

Model

W przestrzeni trójwymiarowej model jest reprezentowany przy użyciu *wierzchołków* (ang. *vertexes*), *krawędzi* (ang. *edges*) i *ścian* (płaszczyzn — ang. *faces*). Wierzchołek to punkt w przestrzeni, krawędź to odcinek łączący dwa wierzchołki, a ściana to powierzchnia ograniczona krawędziami.

Sześcian ma osiem wierzchołków, dwanaście krawędzi i sześć ścian: każda krawędź łączy parę wierzchołków, a każda ściana jest ograniczona czterema krawędziami. Każdy punkt w przestrzeni 3D jest reprezentowany przy użyciu trzech współrzędnych: *x*, *y* i *z*. Zerknijmy na rysunek 8.5, który przedstawia sześcian, kierunki układu współrzędnych i współrzędne wierzchołków sześcianu.



Rysunek 8.5. *Sześcian w przestrzeni 3D*

■ **Uwaga** W przestrzeni 3D nie używa się kierunków: góra, dół, lewo i prawo. Mamy do czynienia z trzema osiami, z których każda wskazuje inny kierunek. O tym, która oś wskazuje „górę", decydujesz Ty — zazwyczaj przyjmuje się jednak, że wysokość jest określana na osi *Z*.

Początek układu współrzędnych znajduje się w środku sześcianu — jest to punkt (0, 0, 0). Dla uproszczenia przyjmujemy, że krawędzi sześcianu mają długość 2, a zatem wszystkie współrzędne wszystkich wierzchołków mają wartość –1 lub 1. Teraz musimy zapisać informacje o wierzchołkach w tablicy. Osiem jej elementów przechowuje wszystkie osiem wierzchołków w tablicy (listing 8.1).

Listing 8.1. *Wierzchołki sześcianu*

```
var vertices = [
    [-1, -1, -1],  [-1, -1,  1],  [-1,  1, -1],  [-1,  1,  1],
    [1, -1, -1], [1, -1,  1], [1,  1, -1], [1,  1,  1]
];
```

Drugą część modelu stanowią krawędzie. Krawędź to odcinek, który łączy dwa wierzchołki. Nie musimy przechowywać współrzędnych wierzchołków krawędzi, ponieważ są one zdefiniowane w tablicy wierzchołków. Wystarczy więc odwoływać się do numerów wierzchołków, posługując się indeksami z pierwszej tablicy. Pierwsza krawędź na listingu 8.2 jest zapisana przy użyciu współrzędnych `[0, 1]`. Oznacza to, że wierzchołek o indeksie 0 łączy się z wierzchołkiem o indeksie 1. Zerknij do tablicy wierzchołków, a przekonasz się, że współrzędne tych punktów to (–1, –1, –1) i (–1, –1, 1). Kod tablicy krawędzi został pokazany na listingu 8.2.

Listing 8.2. *Krawędzie sześcianu — każdy element zawiera indeksy z tablicy wierzchołków*

```
var edges = [
    [0, 1],  [0, 2], [0, 4], [1, 3],
    [1, 5], [2, 3], [2, 6], [3, 7],
    [4, 5], [4, 6], [5, 7], [6, 7]
];
```

Scena

Mamy już sześcian, dlatego teraz skupimy się na możliwości oglądania go pod różnymi kątami. W rzeczywistości nie możemy przemieszczać świata, dlatego zmieniamy położenie aparatu (kamery). W grafice trójwymiarowej zetkniemy się z odwrotną sytuacją — kamera jest nieruchoma, kierunek również nie ulega zmianie. W technologii WebGL kamera jest skierowana w dół, wzdłuż osi *Z*, przez co obiekt o mniejszej wartości współrzędnej *z* wydaje się bardziej oddalony. Aby pokazać użytkownikowi żądany fragment sceny, musimy dopasować kąt i położenie kamery lub zmienić kąt i położenie świata. Zasada działania jest identyczna jak w poprzednim rozdziale: aby użytkownik odniósł wrażenie przemieszczania się po mapie, obszar widoku pozostanie nieruchomy, a będziemy poruszać właśnie mapą (światem).

Obrót to tylko jeden z przykładów przekształceń, które możesz zastosować do swojego modelu. Można także przemieszczać obiekt po scenie lub skalować go względem wybranych osi. Zaimplementowanie tych operacji samodzielnie nie jest trudne. Problemy zaczynają się, gdy zechcesz je grupować. Rozważmy przykład, w którym przesuwasz obiekt o pięć jednostek w lewo, obracasz go o π radianów wokół osi *Z* i skalujesz go trzy razy względem osi *X*. Zaimplementowanie takiego cyklu operacji stanowi dobre ćwiczenie matematyczne, jednak najpierw zajmiemy się innym mechanizmem o sprawdzonej skuteczności.

Renderowanie

Po zdefiniowaniu współrzędnych modelu i zapisaniu ich w tablicy możemy wyrenderować sześcian na ekranie. Jak przekształcić te punkty w szkielet? W tym rozdziale nauczymy się wykonywać bardzo proste renderowanie 3D. Wyrenderowanie bryły trójwymiarowej wymaga obliczenia rzutu wierzchołków na ekranie. Po ich znalezieniu możemy narysować odcinki pomiędzy wszystkimi punktami, które są połączone krawędziami — i gotowe! W ten sposób otrzymujemy szkielet.

Zagadnieniem z zakresu matematyki, które pozwoli nam na znalezienie rzutów (projekcji) i wykonanie renderowania, jest *teoria macierzy*. Rozpoczniemy od najprostszego pojęcia w jej ramach, czyli samej macierzy. Po omówieniu zasad użycia macierzy będziemy mogli zastosować zdobytą wiedzę w praktyce. Wyrenderujemy sześcian, po czym zaczniemy go obracać i stosować odpowiednią perspektywę, dzięki której sześcian wyda się bardziej naturalny.

Macierze

Czym jest macierz (ang. *matrix*)? Nie, nie chodzi nam o równoległą rzeczywistość, w której Neo walczy z agentem Smithem. Macierz to przede wszystkim elementarna struktura danych, niezbędna w każdej aplikacji trójwymiarowej. Rysunek 8.6 przedstawia przykładową macierz.

$$A = \begin{bmatrix} 1 & 3 & 5 & 7 \\ 2 & 8 & 3 & 1 \\ 4 & 5 & 3 & 2 \\ 1 & 0 & 0 & 2 \end{bmatrix}$$

Rysunek 8.6. *Przykład macierzy*

Macierz jest siatką liczb zbudowaną z wierszy i kolumn. Macierze mogą rzecz jasna przyjmować różne rozmiary. Rysunek 8.6 przedstawia macierz o rozmiarze 4 na 4. Naturalnym sposobem reprezentacji macierzy w języku JavaScript jest stosowanie tablic dwuwymiarowych. To, co jest naprawdę ważne w przypadku macierzy o tym rozmiarze, to fakt, że taka macierz może przechować dowolną liczbę przekształceń w dowolnej kolejności, jaką zastosujesz wobec swojej sceny. Jak? Otóż każda macierz reprezentuje pewne przekształcenie: obrót, skalowanie, przesunięcie (translację) lub połączenie wszystkich trzech. Rysunek 8.7 przedstawia kilka przykładów.

$$\begin{bmatrix} 0{,}5 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 5 & 2 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 0{,}7 & -0{,}7 & 0 & 0 \\ 0{,}7 & 0{,}7 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rysunek 8.7. *Macierze reprezentują różne przekształcenia. Od lewej do prawej: zmniejszenie o 50% wzdłuż osi X, przesunięcie o pięć jednostek wzdłuż osi X i o dwie wzdłuż osi Y, obrót wokół osi Z o około 45 stopni*

Jeśli nigdy nie korzystałeś z macierzy, rysunek 8.7 przedstawia dla Ciebie trzy siatki nieistotnych liczb. Całość stanie się bardziej czytelna, jeśli wiesz, jak mnoży się macierze. Zasady nie są intuicyjne — znacznie łatwiej jest opisać je na macierzach o rozmiarze 2 na 2. Załóżmy, że chcesz pomnożyć macierz A przez B.

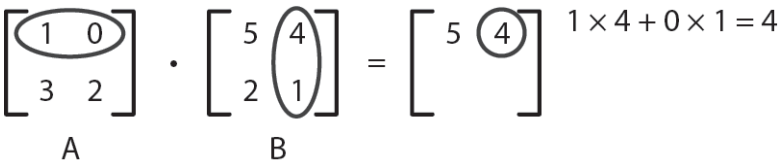
Ogólna zasada mnożenia macierzy jest następująca: aby uzyskać wartość komórki, która znajduje się w *m*-tym wierszu i *n*-tej kolumnie wynikowej macierzy, musisz wziąć wartości z *m*-tego wiersza lewej macierzy i *n*-tej kolumny prawej macierzy, przemnożyć kolejne elementy przez siebie i je zsumować. Aby uzyskać wartość pierwszej komórki (wiersz 1., kolumna 1.), musisz wziąć wartości z pierwszego wiersza lewej macierzy i pierwszej kolumny prawej macierzy, pomnożyć ich kolejne elementy i dodać wszystkie cząstkowe iloczyny. Operacja jest przedstawiona na rysunku 8.8.

$$\begin{bmatrix} 1 & 0 \\ 3 & 2 \end{bmatrix} \cdot \begin{bmatrix} 5 & 4 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 5 & \\ & \end{bmatrix} \qquad 1 \times 5 + 0 \times 2 = 5$$

A B

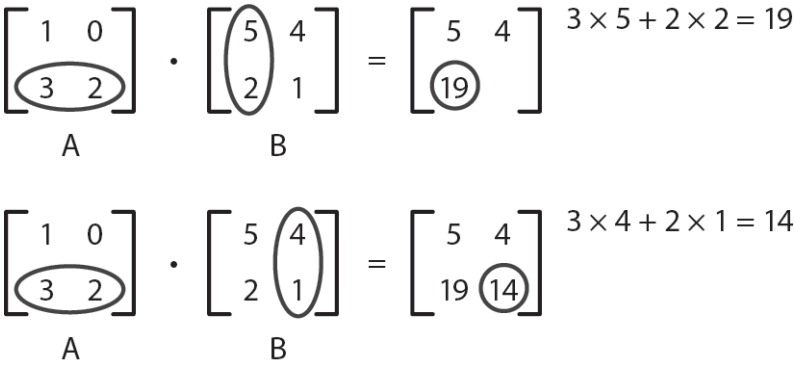
Rysunek 8.8. *Obliczenie pierwszego elementu wynikowej macierzy*

W przypadku elementu znajdującego się w pierwszym wierszu i drugiej kolumnie musisz skorzystać z pierwszego wiersza lewej macierzy i drugiej kolumny prawej macierzy, a następnie wykonać to samo: pomnożyć kolejne elementy przez siebie i dodać cząstkowe iloczyny. Rysunek 8.9 przedstawia przebieg tej operacji.



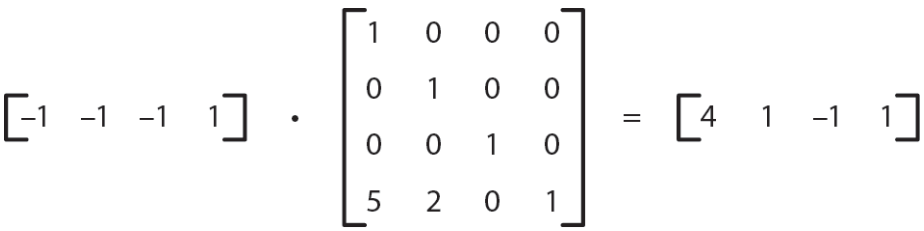
Rysunek 8.9. *Następny krok mnożenia — drugi element w pierwszym wierszu*

Rysunek 8.10 przedstawia obliczenia niezbędne od uzyskania dwóch pozostałych komórek wynikowej macierzy.



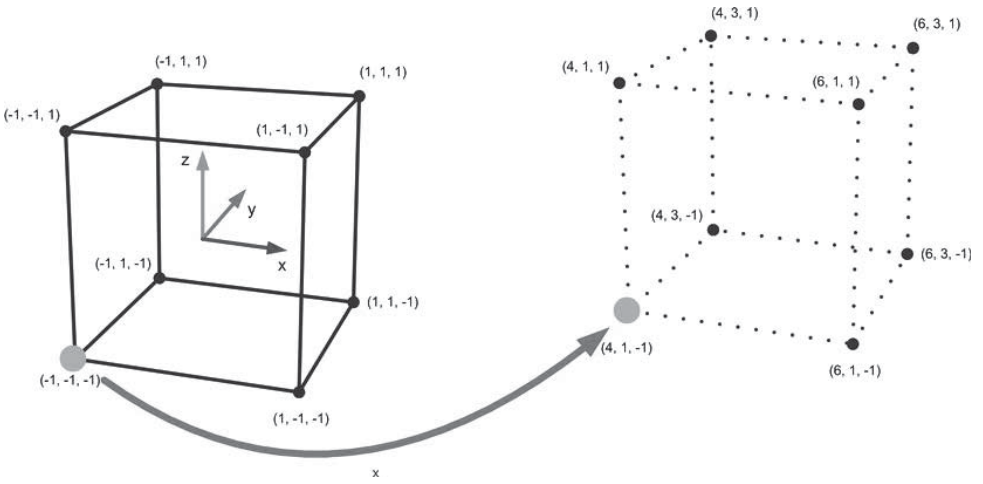
Rysunek 8.10. *Ostateczny wynik mnożenia macierzy*

Wreszcie dochodzimy do długo wyczekiwanej funkcjonalności. Aby zastosować przekształcenie opisane przez macierz, musisz pomnożyć współrzędne punktu przez macierz. Uzyskasz w ten sposób nowe współrzędne punktu — *po* transformacji. Jeśli tę samą transformację zastosujesz wobec wszystkich wierzchołków sześcianu, przekształcisz cały sześcian; w ten sposób możesz obracać, skalować lub przesuwać całe bryły. Teraz możemy omówić sam mechanizm nieco dokładniej. Przede wszystkim musisz zapisać współrzędne punktu w formie macierzy: `[x, y, z, 1]`. Trzeba dodać współrzędną 1, aby było możliwe przeprowadzenie mnożenia macierzy. Zobaczmy, co się stanie, gdy weźmiemy pierwszy wierzchołek sześcianu i zastosujemy do niego drugą z przykładowych macierzy przekształceń. Rysunek 8.11 przedstawia efekt mnożenia.



Rysunek 8.11. *Stosowanie przekształceń do współrzędnych wierzchołków*

Punkt został przesunięty o pięć jednostek na osi *X* i o dwie na osi *Y*. Dokładnie to chcieliśmy osiągnąć! Jeśli powtórzymy tę operację dla wszystkich wierzchołków sześcianu, zostaną one przesunięte w odpowiednie miejsca. Rysunek 8.12 przedstawia zasadę działania mechanizmu. Wykropkowane krawędzie oznaczają położenie sześcianu w momencie stosowania przekształcenia do każdego wierzchołka. Podświetlony punkt oznacza punkt, który został przekształcony.



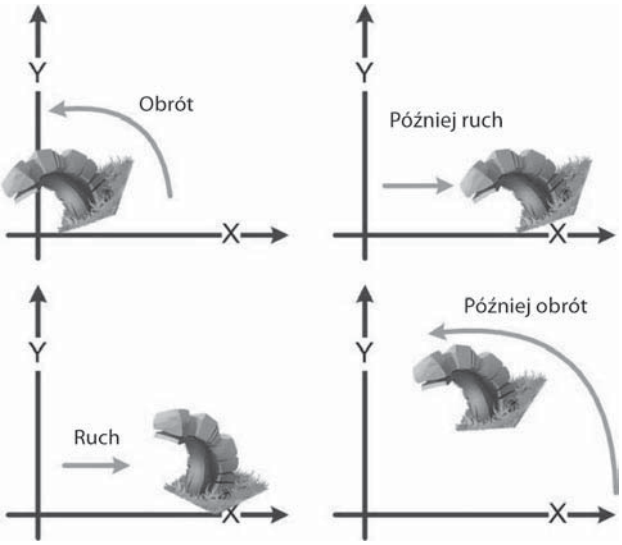
Rysunek 8.12. *Przesunięcie sześcianu. Po przesunięciu wszystkich wierzchołków osiągamy przesunięcie całego sześcianu*

No cóż, nie jest to może porywający efekt, ponieważ moglibyśmy po prostu dodać wartości 5 i 2 do odpowiednich współrzędnych bez zajmowania się macierzami. Prawdziwe możliwości macierzy ujawniają się jednak, gdy musimy wykonać szereg przekształceń na zbiorze punktów. Jeśli musisz wykonać np. dziesięć różnych przekształceń (jedno na drugim), a Twój model ma wiele wierzchołków, musisz albo zastosować przekształcenia dla modelu jedno po drugim, albo utworzyć pojedynczą macierz, która implementuje wszystkie przekształcenia, i zastosować tylko ją. Zgadnij, które podejście jest lepsze.

■ **Uwaga** Ta skromna jedynka, którą dodaliśmy jako czwarty element współrzędnych świata, nie znajduje się tam tylko dla formalności (aby mnożenie było w ogóle możliwe). Silniki 3D reprezentują punkty w nieco innym układzie współrzędnych, zwanym układem współrzędnych jednorodnych (ang. *homogeneous coordinates system*). Czemu są one używane w grafice komputerowej? Ponieważ stosowanie tego rodzaju współrzędnych znacznie ułatwia wykonywanie rzutów. Bardziej szczegółowe informacje na ten temat znajdziesz z łatwością w internecie lub literaturze matematycznej, jednak z pewnością nie będziesz potrzebował niczego więcej poza omówionymi w tej książce mechanizmami. Mimo to zachęcam do poszerzenia wiedzy na ten temat, gdy tylko dobrze zrozumiesz koncepty przedstawione do tej pory.

Utworzenie macierzy, która łączy wiele przekształceń w jedno, jest niezwykle proste — wystarczy pomnożyć macierze, które określają pojedyncze transformacje. Jeśli pomnożysz macierz, która przesuwa wierzchołek w prawo o jedną jednostkę, przez macierz, która przesuwa wierzchołek o jedną jednostkę w górę, a następnie przez macierz, która obraca wierzchołek wokół osi *X*, uzyskasz macierz wykonującą wszystkie trzy operacje: przesuwającą obiekt w górę i w prawo, a także obracającą model.

Operując na macierzach, musimy pamiętać o dwóch kwestiach. Po pierwsze, kolejność wykonywanych działań mnożenia ma znaczenie. Jeśli przeprowadzasz operacje na pojedynczych liczbach, nie ma znaczenia, czy wykonasz operację 5×3, czy 3×5 — w przypadku macierzy tak jednak nie jest. Wpływ kolejności przekształceń na efekt końcowy został przedstawiony na rysunku 8.13.



Rysunek 8.13. *Zmiana kolejności przekształceń ma wpływ na efekt końcowy*

Trzeba również pamiętać o tym, że przekształcenia są stosowane w odwrotnej kolejności, tj. ostatnie wprowadzone przekształcenie zostanie zastosowane wobec modelu jako pierwsze. Na początku jest to nieco mylące, jednak na pewno się do tego przyzwyczaisz.

Drugą kwestią, o której trzeba pamiętać, jest element neutralny. Podobnie jak pomnożenie liczby przez jeden nie zmienia wyniku mnożenia, tak pomnożenie macierzy przez *macierz jednostkową* nie zmieni danej macierzy — niezależnie od tego, czy macierz znajdzie się po lewej, czy po prawej stronie mnożenia. Macierz jednostkowa przedstawia brak jakichkolwiek przekształceń (rysunek 8.14).

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rysunek 8.14. *Macierz jednostkowa opisująca stan bez przekształceń*

Na zakończenie na rysunku 8.15 przedstawiamy krótką listę najczęściej używanych macierzy przekształceń. Nie bój się sinusów i cosinusów pojawiających się w macierzach obrotu — nie musisz zagłębiać się w szczegóły przekształceń punktów.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x & y & z & 1 \end{bmatrix}$$

Translacja

$$\begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Skalowanie

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Obrót wokół osi X

$$\begin{bmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Obrót wokół osi Y

$$\begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Obrót wokół osi Z

Rysunek 8.15. *Różne rodzaje przekształceń przedstawione w formie macierzy*

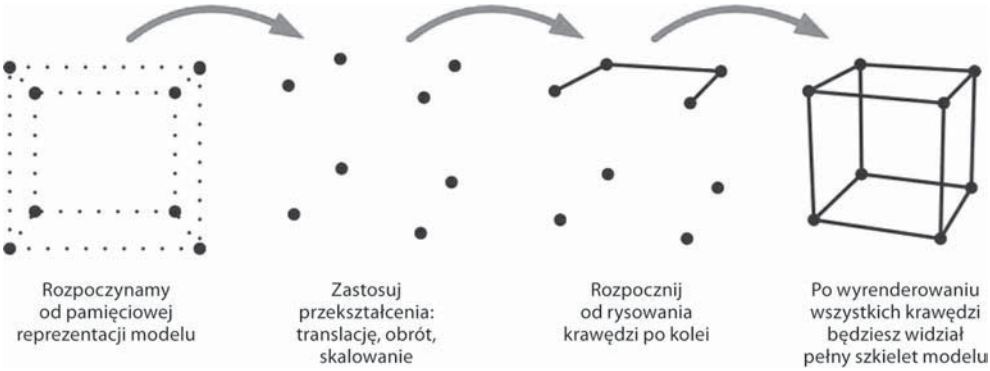
Implementacja przekształceń

Jesteśmy wreszcie gotowi do utworzenia pierwszej wersji naszego „sześciennego" silnika. Na początek dobre informacje. Wszystkie operacje na macierzach wykona za nas biblioteka gl-matrix (*https://github.com/toji/gl-matrix*). Dzięki Brandonowi Jonesowi możemy skorzystać z doskonałego i bardzo szybkiego narzędzia. Pobierz plik *gl-matrix.js* lub *gl-matrix-min.js* i dodaj go do projektu. Zgodnie z nazwą biblioteka jest zazwyczaj używana razem z biblioteką WebGL, jednak możemy ją bez problemów wykorzystać także w innych celach.

Przekształcenie szkieletu sześcianu wymaga wykonania dwóch kroków:

1. Przekształcenia wszystkich wierzchołków (chcemy je obracać).
2. Narysowania linii pomiędzy wszystkimi wierzchołkami, pomiędzy którymi zostały zdefiniowane krawędzie.

Kolejne kroki są przedstawione na rysunku 8.16.



Rysunek 8.16. *Kolejne kroki niezbędne do wyrenderowania szkieletu*

Zaczniemy od pierwszej operacji. Nasz sześcian musimy dwukrotnie przekształcić — obrócić wokół osi *X* i wokół osi *Y* (możesz zastosować także inne przekształcenia). Biblioteka *gl-matrix.js* udostępnia wiele metod, które ukrywają szczegóły implementacji w ramach biblioteki. Kod przekształceń został pokazany na listingu 8.3.

Listing 8.3. *Tworzenie macierzy przekształceń*

```
var modelView = mat4.create();
mat4.identity(modelView);
mat4.rotateX(modelView, xRot);
mat4.rotateY(modelView, yRot);
```

Na początku tworzymy pustą macierz. Jest ona wypełniona zerami, dlatego musimy ją zainicjalizować — w tym celu korzystamy z macierzy jednostkowej; w przeciwnym razie wszystkie przekształcenia dawałyby wynik (0, 0, 0). W trzecim wierszu dodajemy obrót wokół osi *X*. Jest to równoważne utworzeniu macierzy obrotu i pomnożeniu przez nią macierzy jednostkowej. W ostatnim wierszu kończymy przygotowania — w tym momencie jesteśmy gotowi do obrotu sześcianu.

Uwaga Czemu zmienna macierzy nosi nazwę `modelView`? Taka właśnie nazwa jest używana w OpenGL i oznacza ona… widok modelu, czyli wszystkie przekształcenia, które są stosowane wobec wierzchołków modelu. Czemu nie zastosujemy po prostu nazwy „matrix"? Ponieważ w trakcie całego procesu korzystamy również z innych macierzy — w dalszej części rozdziału będziemy się zajmować np. macierzą rzutu (projekcji).

Proces obrotu jest bardzo prosty: stosujemy macierz po kolei do wszystkich wierzchołków. Otrzymujemy w ten sposób przekształcone współrzędne — aby wyświetlić je na ekranie, ignorujemy po prostu współrzędną *z*. Musimy także przeskalować sześcian, aby mieścił się na ekranie. Rozmiar sześcianu to jedynie 2 (od –1 do 1) — jeśli spróbujemy go wyświetlić na ekranie, będzie on niezwykle mały. Na listingu 8.4 znajdziesz kod odpowiedzialny za obliczanie współrzędnych projekcji.

Listing 8.4. *Obliczanie współrzędnych projekcji dla wszystkich punktów*

```
// Dobrze jest utworzyć indeksy, które będą odpowiadać indeksom w tablicy.
var X = 0;
var Y = 1;
var Z = 2;

// Współrzędne ekranowe.
var points = [];

// Współczynnik skalowania — w tym miejscu możesz wpisać wybraną wartość.
var scaleFactor = canvas.width/8;

for (var i = 0; i < vertices.length; i++) {
    // Przekształć punkt do współrzędnych jednorodnych — przygotuj się do mnożenia.
    var point = [vertices[i][X], vertices[i][Y], vertices[i][Z], 1];

    // Zastosuj przekształcenie.
    mat4.multiplyVec4(modelView, point);

    // Dodaj obliczone współrzędne do tablicy.
    points[i] = [
        Math.round(canvas.width/2 + scaleFactor*point[X]),
        Math.round(canvas.height/2 - scaleFactor*point[Y])];
}
```

■ **Uwaga** Rozmiar wynosi dwa? Dwa co? Piksele, cale, metry, melony? W grafice trójwymiarowej nie ma to znaczenia. To od Ciebie zależy, jaką jednostkę przyjmiesz. Musisz jedynie zachować spójność w ramach całego projektu. Jeśli Twoje „dwa" będzie oznaczać „dwa metry", musisz o tym pamiętać — inaczej może się zdarzyć, że utworzysz model królika, który będzie miał wymiary małego samochodu. Nie jest to jednak tak naprawdę problem — zawsze możesz wprowadzić dodatkowe skalowanie. W tym rozdziale we wszystkich odwołaniach do rozmiarów stosuję słowo „jednostki".

Jak widać, obliczenia nie stanowią problemu, gdy tylko przyzwyczaisz się do korzystania z macierzy.

Na zakończenie musimy narysować krawędzie. Dysponujemy tablicą krawędzi i tablicą przekształceń punktów. Łącząc obie tablice, możemy otrzymać obracający się sześcian (nie zapomnij zaktualizować wartości obrotu). Listing 8.5 przedstawia pełny kod dla tego projektu. Spróbuj uruchomić go na swoim telefonie lub w przeglądarce stacjonarnej.

Listing 8.5. *Pełny kod źródłowy dla dema obracającego się sześcianu*

```
<script src="js/utils.js"></script>
<script src="js/gl-matrix-min.js"></script>

<script>

    var canvas = null;
    var ctx = null;

    var X = 0;
    var Y = 1;
    var Z = 2;

    var vertices = [
        [-1, -1, -1],  [-1, -1,  1],  [-1,  1, -1],  [-1,  1,  1],
        [1, -1, -1], [1, -1,  1], [1,  1, -1], [1,  1,  1]
    ];

    var edges = [
        [0, 1],  [0, 2], [0, 4], [1, 3],
        [1, 5], [2, 3], [2, 6], [3, 7],
        [4, 5], [4, 6], [5, 7], [6, 7]
    ];

    var xRot = 0;
    var yRot = 0;

    function init() {
        canvas = initFullScreenCanvas("mainCanvas");
        ctx = canvas.getContext("2d");
        animate(0);
    }

    function animate(t) {
        ctx.clearRect(0, 0, canvas.width, canvas.height);
        renderCube();
        requestAnimationFrame(arguments.callee);
    }

    function renderCube() {

        var modelView = mat4.create();
```

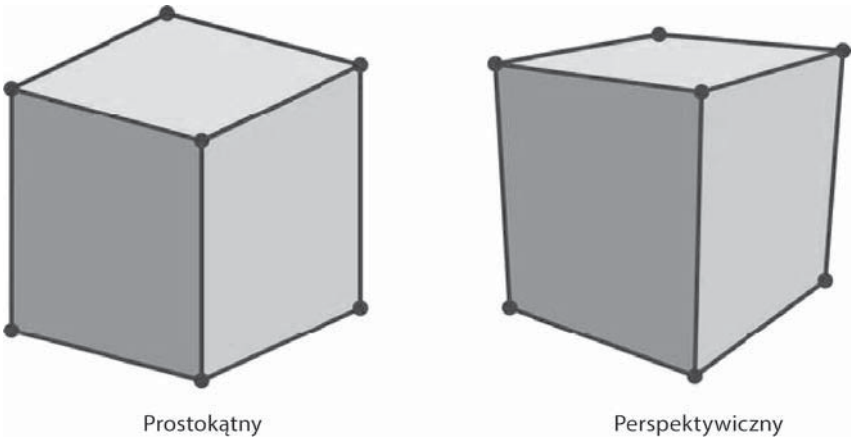
```
        mat4.identity(modelView); // Ustaw macierz jednostkową
        mat4.rotateX(modelView, xRot);
        mat4.rotateY(modelView, yRot);

        var points = [];
        var scaleFactor = canvas.width/8;

        for (var i = 0; i < vertices.length; i++) {
            var point = [vertices[i][X], vertices[i][Y], vertices[i][Z], 1];
            mat4.multiplyVec4(modelView, point);

            points[i] = [
                Math.round(canvas.width/2 + scaleFactor*point[X]),
                Math.round(canvas.height/2 + scaleFactor*point[Y])];
        }

        ctx.strokeStyle = "black";

        // Narysuj szkielet
        ctx.beginPath();
        for (i = 0; i < edges.length; i++ ) {
            ctx.moveTo(points[ edges[i][0] ][X], points[ edges[i][0] ][Y]);
            ctx.lineTo(points[ edges[i][1] ][X], points[ edges[i][1] ][Y]);
        }

        ctx.stroke();
        ctx.closePath();
        xRot += 0.01;
        yRot += 0.01;
    }

    function initFullScreenCanvas(canvasId) {
        var canvas = document.getElementById(canvasId);
        resizeCanvas(canvas);
        window.addEventListener("resize", function() {
            resizeCanvas(canvas);
        });
        return canvas;
    }

    function resizeCanvas(canvas) {
        canvas.width  = document.width || document.body.clientWidth;
        canvas.height = document.height || document.body.clientHeight;
    }
</script>
```

Po uruchomieniu dema przekonasz się, że sześcian nie wygląda naturalnie: tak naprawdę w ogóle nie wygląda on jak prawdziwy sześcian. Problem w tym, że skorzystaliśmy z rzutu prostokątnego — w prawdziwym świecie sytuacja wygląda inaczej. Pełny kod źródłowy z demem do tego rozdziału jest dostępny w pliku *01.spinning_cube.html* wraz z innymi zasobami.

Rzuty

W prawdziwym świecie obiekty, które znajdują się dalej, są mniejsze od obiektów znajdujących się bliżej. W naszym programie nie zastosowaliśmy się do tej zasady — to właśnie dlatego sześcian nie wygląda dobrze. Jak mogłeś się przekonać w kodzie, pominęliśmy po prostu wartość z, przez co nie uczestniczy ona wcale w obliczaniu rzutu.

W naszym demie korzystamy z *rzutu prostokątnego*, w którym podczas renderowania modeli 3D nie stosujemy żadnej perspektywy. W przypadku takiego rzutu samochód znajdujący się pięć metrów od aparatu ma taki sam rozmiar jak samochód znajdujący się dwadzieścia metrów dalej. Nie jest to naturalny widok. *Rzut perspektywiczny* funkcjonuje bardziej naturalnie: obiekty znajdujące się dalej są mniejsze niż te znajdujące się bliżej kamery. Rysunek 8.17 przedstawia różnice pomiędzy obydwoma rzutami.



Rysunek 8.17. *Obraz po lewej stronie przedstawia rzut prostokątny, po prawej — rzut perspektywiczny*

Jak widać, obraz po prawej stronie wygląda znacznie bardziej realistycznie niż ten po lewej. Jak możemy poprawić nasz silnik, aby perspektywa również była brana pod uwagę? Oczywiście musimy wprowadzić kolejną macierz.

Nie będę się zagłębiał w szczegóły matematyczne implementacji macierzy projekcji. Aby tworzyć świetne gry 3D, musisz o niej wiedzieć tylko dwie rzeczy. Po pierwsze, jeśli pomnożysz wynik przekształcenia `modelView` przez tę macierz, uzyskasz przekształcenie rzutu, które weźmie projekcję pod uwagę. Po drugie, Brandon to naprawdę fajny gość, który dołączył do swojej biblioteki mechanizm tworzenia macierzy perspektywy po podaniu parametrów przyjaznych dla użytkownika (listing 8.6).

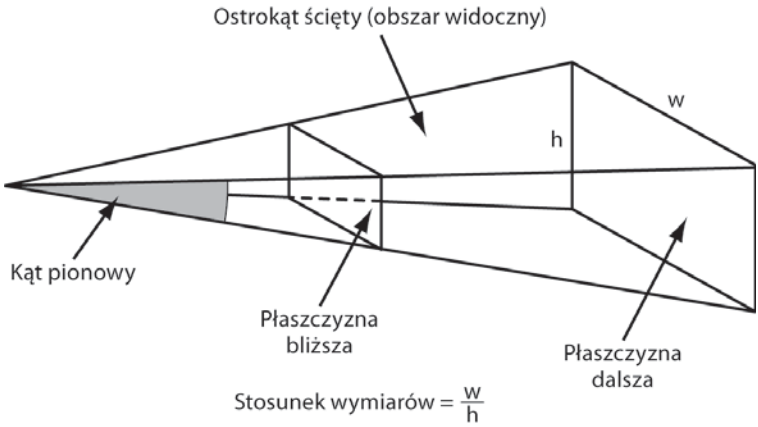
Listing 8.6. *Tworzenie macierzy projekcji*

```
var persp = mat4.create();
mat4.perspective(45, canvas.width/canvas.height, 0.1, 100, persp);
```

Wspomnianymi parametrami są pionowy kąt widoku, stosunek wymiarów (ang. *aspect ratio*) oraz odległość od bliższej i dalszej płaszczyzny. Płaszczyzny bliższa i dalsza określają granice sceny. Dzięki nim możesz stwierdzić, że chcesz widzieć tylko obiekty znajdujące się w odległości od 0,1 do 100 jednostek od kamery. Ostatni parametr określa macierz, która jest inicjalizowana przy użyciu tych wartości (listing 8.7). Rysunek 8.18 przedstawia wpływ parametrów perspektywy na końcowy efekt.

Listing 8.7. *Stosowanie macierzy projekcji*

```
for (var i = 0; i < vertices.length; i++) {
    var point = [vertices[i][X], vertices[i][Y], vertices[i][Z], 1];
    mat4.multiplyVec4(modelView, point);
    mat4.multiplyVec4(persp, point);

    ...
}
```



Rysunek 8.18. *Określenie granic sceny przy użyciu perspektywy*

Współrzędne otrzymane w wyniku stosowania macierzy perspektywy muszą zostać znormalizowane przed narysowaniem wierzchołków na scenie. Mówiąc wprost, przekształcamy współrzędne jednorodne (te zawierające jeden dodatkowy parametr) na zwykłe współrzędne ekranowe zawierające dwa elementy: *x* i *y* (listing 8.8).

Listing 8.8. *Normalizowanie współrzędnych*

```
var ndcPoint = [
    point[X]/point[W],
    point[Y]/point[W]
];
```

Powstałe współrzędne *x* i *y* mieszczą się w zakresie od –1 do 1. „NDC" w zmiennej `ndcPoint` oznacza znormalizowane współrzędne urządzenia (ang. *Normalized Device Coordinates*). Teraz musimy jedynie „rozciągnąć" je na całe płótno, co pokazano na listingu 8.9. Pełny kod dema z rzutem perspektywicznym jest dostępny w pliku *02.perspective.html* dołączonym do kodów źródłowych do tego rozdziału.

Listing 8.9. *Translacja współrzędnych ekranowych*

```
points[i] = [
Math.round(canvas.width/2*(1 + ndcPoint[0])),
    Math.round(canvas.height/2*(1 - ndcPoint[1]))];
```

Listing 8.10 przedstawia pełny kod funkcji `renderCube()`, która obsługuje perspektywę (nowy kod jest pogrubiony).

Listing 8.10. *Ostateczna wersja funkcji renderCube()*

```
function renderCube() {

    var modelView = mat4.create();
    mat4.identity(modelView); // Macierz jednostkowa
    mat4.translate(modelView, [0, 0, -10]);
    mat4.rotateX(modelView, xRot);
    mat4.rotateY(modelView, yRot);

    var persp = mat4.create();
    mat4.perspective(45, canvas.width/canvas.height, 0.1, 100, persp);
```

```
    var points = [];

    for (var i = 0; i < vertices.length; i++) {
        var point = [vertices[i][X], vertices[i][Y], vertices[i][Z], 1];
        mat4.multiplyVec4(modelView, point);
        mat4.multiplyVec4(persp, point);

        var ndcPoint = [
            point[X]/point[W], // W = 3, położenie czwartej współrzędnej
            point[Y]/point[W]  // w tablicy
        ];

        points[i] = [
            Math.round(canvas.width/2*(1 + ndcPoint[0])),
            Math.round(canvas.height/2*(1 - ndcPoint[1]))];
    }

    ctx.strokeStyle = "black";

    // Narysuj szkielet
    ctx.beginPath();
    for (i = 0; i < edges.length; i++ ) {
        ctx.moveTo(points[ edges[i][0] ][X], points[ edges[i][0] ][Y]);
        ctx.lineTo(points[ edges[i][1] ][X], points[ edges[i][1] ][Y]);
    }

    ctx.stroke();
    ctx.closePath();
    xRot += 0.01;
    yRot += 0.01;
}
```

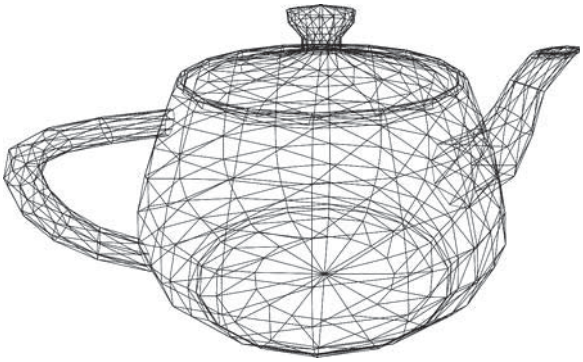
Świetnie! Utworzyłeś bardzo prosty silnik 3D! To naprawdę duże osiągnięcie. I wiesz co? Właśnie zduplikowaliśmy fragment mechanizmu renderującego biblioteki OpenGL — serce tego API. Przeanalizujmy, co właściwie udało nam się osiągnąć.

1. Skorzystaliśmy ze współrzędnych wierzchołków i krawędzi jako danych wejściowych.
2. Zastosowaliśmy przekształcenia wobec wierzchołków, obracając model wokół dwóch osi.
3. Zastosowaliśmy macierz perspektywy na wyniku poprzedniej operacji.
4. Znormalizowaliśmy współrzędne.
5. Przekształciliśmy współrzędne źródłowych wierzchołków na współrzędne płótna 2D.
6. Narysowaliśmy krawędzie obracającego się sześcianu.

Oczywiście cały proces w bibliotece WebGL jest bardziej skomplikowany, jednak jego główne części — przekształcenie wierzchołków, zastosowanie tablic, perspektywy i inne omówione w tym rozdziale zagadnienia — pozostają identyczne. Teraz możemy powiedzieć, że rozumiemy matematyczne podstawy kryjące się za operacjami wykonywanymi w WebGL.

Na koniec pozostaje nam jedno pytanie. Czy jesteśmy w stanie rysować cokolwiek poza prostymi sześcianami, prostokątami i kołami? Oczywiście. Rysunek 8.19 przedstawia bardzo złożony model, który udało mi się znaleźć w internecie. Nie będę wklejał teraz współrzędnych kilkuset wierzchołków, jednak cały kod możesz znaleźć w źródłach dołączonych do tego rozdziału. Rysunek 8.19 przedstawia dzbanek.

Pełny kod dzbanka jest dostępny w pliku *03.teapot.html*.



Rysunek 8.19. *Dzbanek do herbaty wyrenderowany przy użyciu tego samego mechanizmu co sześcian*

Podsumowanie

W tym rozdziale skupiliśmy się całkowicie na zrozumieniu podstaw teoretycznych odpowiedzialnych za działanie silników 3D. Obliczenie położenia i zastosowanie perspektywy stanowią jedynie pierwszy krok w tworzeniu prawdziwego silnika, który renderuje kolejne ramki wewnątrz gry. Następne kroki są bardziej skomplikowane: usuwanie wielokątów, które nie są widoczne, aby zaoszczędzić czas, zastosowanie tekstur do ścian modeli, obliczanie efektów oświetleniowych itd. Silniki 3D zazwyczaj wykorzystują znacznie bardziej skomplikowane wzory i zależności matematyczne w porównaniu do silników 2D, a do tego silniki trójwymiarowe cały czas się zmieniają. Programiści i matematycy pracują wspólnie w celu uzyskania jeszcze lepszych algorytmów, które pozwolą na renderowanie coraz bardziej rzeczywistych obrazów.

Nawet najbardziej skomplikowana matematyka trójwymiarowa zaczyna się od współrzędnych wierzchołków i obliczeń przekształceń modelu. Jeśli nie pracowałeś nigdy z silnikiem 3D, to teraz możesz się pochwalić, że jesteś jednym z nielicznych programistów, którzy rozumieją, co się dzieje w ramach tego typu silników.

W tym rozdziale utworzyliśmy najprostsze możliwe demo 3D — obracający się sześcian. Podczas tego kursu nauczyliśmy się, jak:

- reprezentować model 3D w pamięci,
- korzystać z macierzy do przemieszczania, obracania i skalowania modeli,
- renderować szkielet,
- implementować rzut perspektywiczny w ramach sceny.

Teraz jesteśmy gotowi, aby zająć się bardziej zaawansowanym tematem — WebGL API.

Skorowidz

E

F

G

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

X

Z

Ź

Ż

Potencjał HTML5, CSS3 i JavaScriptu pozwala na tworzenie niesamowitych gier i aplikacji. Dzięki tym technologiom powstało oprogramowanie działające w mobilnej przeglądarce internetowej. Jeżeli dodamy do tego moc współczesnych telefonów i tabletów działających z systemem Android, może się okazać, że to trio stanowi niezastąpiony zestaw narzędzi.

Dzięki tej książce rozwiniesz swoje umiejętności programistyczne. W trakcie lektury dowiesz się, jak najszybciej rozpocząć przygodę z grami dla platformy Android. W kolejnych rozdziałach nauczysz się korzystać z grafiki i animacji w przeglądarce, obsługiwać zdarzenia i operacje wykonywane przez użytkownika oraz stosować różne sposoby renderowania świata gry. Zbudujesz też własny silnik izometryczny oraz wykorzystasz WebGL do stworzenia zaawansowanej grafiki 3D. Dodatkowo zdobędziesz wiedzę na temat programowania sztucznej inteligencji oraz zapewniania komunikacji aplikacji z serwerem. W tej chwili tylko krok dzieli Cię od tworzenia gier w trybie multiplayer! Książka ta jest doskonałym i kompletnym źródłem informacji dla wszystkich osób chcących wykorzystać platformę Android i przeglądarkę internetową do pisania zaawansowanych i atrakcyjnych gier.

Stwórz grę:

- **działającą na platformie Android**
- **wykorzystującą mistrzowskie trio: HTML5, CSS3 i JavaScript**
- **zawierającą zaawansowane elementy graficzne 3D**
- **dla wielu graczy**
- **i odnieś sukces!**

Niezastąpiony podręcznik dla każdego programisty tworzącego gry!

ISBN 978-83-246-7401-5
9 788324 674015
Cena: 79,00 zł