

O'REILLY®

Aplikacje 3D

Przewodnik po HTML5, WebGL i CSS3

TWÓJ PRZEWODNIK PO GRAFICE 3D W HTML!



HELION

Tony Parisi

Spis treści

Przedmowa	9
Część I. Podstawy	15
1. Wprowadzenie	17
HTML5 jako nowe medium wizualne	19
Przeglądarka jako platforma	20
Przeglądarkowa rzeczywistość	21
Grafika trójwymiarowa	22
Co to jest trójwymiarowość?	22
Trójwymiarowe układy współrzędnych	23
Siatki, wielokąty i wierzchołki	24
Materiały, tekstury i oświetlenie	24
Przekształcenia i macierze	25
Kamery, perspektywa, obszary widoku oraz projekcje	26
Programy cieniąjące	27
2. Renderowanie grafiki trójwymiarowej na bieżąco przy użyciu biblioteki WebGL	31
Podstawy WebGL	32
API WebGL	33
Anatomia aplikacji WebGL	34
Prosty przykład użycia WebGL	34
Kanwa i kontekst rysunkowy WebGL	35
Obszar widoku	36
Bufory, bufory tablicowe i tablice typowane	36
Macierze	37
Shader	38
Rysowanie obiektów podstawowych	40

Tworzenie brył	41
Animacja	45
Mapy tekstur	46
Podsumowanie	51
3. Three.js — mechanizm do programowania grafiki trójwymiarowej w JavaScriptie	53
Najbardziej znane projekty zbudowane przy użyciu Three.js	53
Wprowadzenie do Three.js	56
Przygotowanie do pracy z Three.js	58
Struktura projektu Three.js	58
Prosty program Three.js	59
Tworzenie renderera	61
Tworzenie sceny	61
Implementacja pętli wykonawczej	62
Oświetlenie sceny	64
Podsumowanie	65
4. Grafika i renderowanie w Three.js	67
Geometria i siatki	67
Gotowe typy geometryczne	67
Ścieżki, kształty i ekstrusze	68
Bazowa klasa geometrii	69
Geometria buforowana do optymalizacji renderowania siatki	73
Importowanie siatek z programów do modelowania	73
Graf sceny i hierarchia przekształceń	75
Zarządzanie sceną za pomocą grafu sceny	75
Grafy sceny w Three.js	75
Reprezentowanie przesunięcia, obrotu i skali	78
Materiały	79
Standardowe materiały siatki	79
Dodawanie realizmu poprzez zastosowanie kilku tekstur	81
Oświetlenie	84
Cienie	87
Shadery	91
Klasa ShaderMaterial: zrób to sam	91
Stosowanie kodu GLSL z biblioteką Three.js	93
Renderowanie	95
Przetwarzanie końcowe i renderowanie wieloprzebiegowe	96
Renderowanie opóźnione	97
Podsumowanie	98

5. Animacje trójwymiarowe	99
Sterowanie animacją za pomocą funkcji requestAnimationFrame()	100
Używanie funkcji requestAnimationFrame() we własnych aplikacjach	102
Funkcja requestAnimationFrame() a wydajność	103
Animacje klatkowe a animacje czasowe	103
Animowanie przy użyciu programowego aktualizowania właściwości	104
Animowanie przejść przy użyciu międzyklatek	106
Interpolacja	106
Biblioteka Tween.js	107
Funkcja prędkości animacji	109
Tworzenie skomplikowanych animacji przy użyciu klatek kluczowych	110
Animacje obiektów połączonych z użyciem klatek kluczowych	113
Tworzenie wrażenia płynnego ruchu przy użyciu krzywych i śledzenia ścieżki	115
Animacja postaci i twarzy przy użyciu morfingu	118
Animowanie postaci przy użyciu animacji szkieletowej	121
Animowanie przy użyciu shaderów	124
Podsumowanie	129
6. Tworzenie zaawansowanych efektów na stronach przy użyciu CSS3	131
Przekształcenia CSS	133
Przekształcenia trójwymiarowe w praktyce	134
Perspektywa	136
Tworzenie hierarchii przekształceń	138
Kontrolowanie renderowania tylnej ściany obiektów	140
Zestawienie właściwości przekształcaniowych CSS	143
Przejścia CSS	143
Animacje CSS	147
Zaawansowane funkcje CSS	151
Renderowanie obiektów trójwymiarowych	151
Renderowanie środowisk trójwymiarowych	152
Tworzenie zaawansowanych efektów przy użyciu filtrów CSS	153
Renderowanie trójwymiarowe w CSS przy użyciu Three.js	154
Podsumowanie	155
7. Kanwa dwuwymiarowa	157
Kanwa — podstawowe wiadomości	158
Element kanwy i dwuwymiarowy kontekst rysunkowy	158
Właściwości API Canvas	160
Renderowanie obiektów trójwymiarowych przy użyciu API Canvas	164

Trójwymiarowe biblioteki oparte na kanwie	167
K3D	168
Renderer biblioteki Three.js rysujący na kanwie	169
Podsumowanie	174
Część II. Techniki tworzenia aplikacji	175
8. Proces powstawania treści trójwymiarowej	177
Proces tworzenia grafiki trójwymiarowej	177
Modelowanie	178
Teksturowanie	178
Animowanie	179
Sztuka techniczna	180
Narzędzia do tworzenia trójwymiarowych modeli i animacji	181
Klasyczne programy komputerowe	181
Przeglądarkowe środowiska zintegrowane	185
Repozytoria 3D i darmowe zdjęcia	188
Trójwymiarowe formaty plików	190
Formaty modelowe	190
Formaty animacyjne	192
Formaty do zapisywania całych scen	193
Wczytywanie treści do aplikacji WebGL	201
Format JSON biblioteki Three.js	202
Format binarny biblioteki Three.js	207
Wczytywanie sceny w formacie COLLADA przy użyciu biblioteki Three.js	208
Ładowanie sceny glTF przy użyciu biblioteki Three.js	211
Podsumowanie	212
9. Trójwymiarowe silniki i systemy szkieletowe	213
Koncepty szkieletów trójwymiarowych	214
Czym jest system szkieletowy?	214
Wymagania dotyczące systemów szkieletowych dla WebGL	215
Przegląd systemów szkieletowych dla WebGL	217
Silniki gier	217
Prezentacyjne systemy szkieletowe	220
Vizi — komponentowy system do tworzenia wizualnych aplikacji sieciowych	223
Tło i metody projektowania	223
Architektura systemu Vizi	224
Podstawy obsługi systemu Vizi	226
Prosta aplikacja Vizi	226
Podsumowanie	232

10. Budowa prostej aplikacji trójwymiarowej	233
Projektowanie aplikacji	234
Tworzenie trójwymiarowej treści	235
Eksportowanie sceny Maya do formatu COLLADA	236
Konwertowanie pliku COLLADA na glTF	237
Podglądanie i testowanie treści trójwymiarowej	238
Narzędzie do podglądu na bazie systemu Vizi	239
Klasa Vizi.Viewer	240
Klasa wczytująca Vizi	241
Integrowanie treści trójwymiarowej z aplikacją	244
Trójwymiarowe zachowania i interakcje	247
Metody API grafu sceny Vizi: findNode() i map()	247
Animowanie przezroczystości za pomocą klasy Vizi.FadeBehavior	249
Automatyczne obracanie modelu za pomocą klasy Vizi.RotateBehavior	251
Wyświetlanie informacji o częściach za pomocą klasy Vizi.Picker	251
Sterowanie animacjami w interfejsie użytkownika	252
Zmienianie kolorów przy użyciu wybieraka	254
Podsumowanie	255
11. Tworzenie trójwymiarowego środowiska	257
Tworzenie warstwy wizualnej	259
Podglądanie i testowanie środowiska	260
Podglądanie sceny w trybie pierwszoosobowym	261
Przeglądanie grafu sceny	261
Przeglądanie właściwości obiektów	265
Wyświetlanie ramek obiektów	266
Oglądanie wielu obiektów	269
Wyszukiwanie za pomocą przeglądarki innych problemów ze sceną	270
Tworzenie trójwymiarowego tła przy użyciu pudła nieba	272
Trójwymiarowe pudło nieba	272
Obiekt Skybox systemu Vizi	272
Dodawanie do aplikacji trójwymiarowej treści	275
Ładowanie i inicjowanie środowiska	275
Ładowanie i inicjowanie modelu samochodu	277
Implementowanie nawigacji pierwszoosobowej	279
Kontrolery kamery	281
Kontroler pierwszoosobowy — obliczenia	281
Wybieranie kierunku patrzenia za pomocą myszy	283
Proste wykrywanie kolizji	283
Posługiwianie się wieloma kamerami	284
Tworzenie animowanych i czasowych przejść	286

Implementacja zachowań obiektów	288
Implementowanie własnych składników na bazie klasy Vizi.Script	288
Kontroler samochodu	288
Dodawanie dźwięków do środowiska	294
Renderowanie dynamicznych tekstur	296
Podsumowanie	300
12. Tworzenie aplikacji dla urządzeń przenośnych	301
Przenośne platformy trójwymiarowe	302
Tworzenie aplikacji dla mobilnych wersji przeglądarek internetowych	303
Dodawanie obsługi interfejsu dotykowego	304
Debugowanie mobilnej funkcjonalności w stacjonarnej wersji przeglądarki Chrome	309
Tworzenie aplikacji sieciowych	311
Tworzenie aplikacji sieciowych i narzędzia do ich testowania	311
Pakowanie aplikacji sieciowych do dystrybucji	312
Tworzenie aplikacji hybrydowych	313
CocoonJS jako technologia tworzenia gier i aplikacji HTML dla urządzeń mobilnych	314
Składanie aplikacji przy użyciu biblioteki CocoonJS	316
Tworzenie hybrydowych aplikacji WebGL — konkluzja	322
Wydajność mobilnych aplikacji trójwymiarowych	322
Podsumowanie	324
A. Źródła informacji	327
Skorowidz	339

Przedmowa

Dzieje grafiki 3D w internecie mają około 20 lat i są bardzo zawiłe. W 1994 r. uwaga całej branży internetowej skupiona była na najnowszym wynalazku o nazwie *VRML*, który — niestety — po pierwszym szale internetowym zmienił się w niechciane dziecko programowania sieciowego. Około 2000 r. pojawił się kolejny hit o nazwie *Shockwave 3D*, który miał zdemokratyzować tworzenie gier. W 2004 r. technologia ta również odeszła do lamusa. W 2007 r. branżę technologii multimedialnych zaskoczył system światów wirtualnych o nazwie *Second Life*, który pojawił się na okładce magazynu „Business Week”. Miał on zawiadnić całym obszarem grafiki trójwymiarowej — i to dosłownie, bo ludzie grupowo wynajmowali wyspy *Second Life*, próbując kolonizować cyberprzestrzeń, która nigdy się nie zmaterializowała. W 2010 r. wirtualne światy były już przeszłością, ponieważ konsumenci przerzucili się na gry społecznościowe i mobilne. Z jednej strony, opisane zdarzenia są kroniką porażek, z drugiej jednak można je postrzegać jak szereg ciężkich doświadczeń.

Dobre pomysły mogą długo czekać na realizację, ale nigdy nie umierają. Dotyczy to także grafiki trójwymiarowej w internecie. Jeśli przyjrzeć się tym wszystkim naiwnym, choć pełnym dobrych chęci, wczesnym próbom, można dowiedzieć się czegoś, co wszyscy i tak już wiemy: grafika 3D jest tylko jednym z wielu typów mediów. Nieważne, czy wykorzystuje się ją do budowy internetowej gry dla wielu graczy, interaktywnej lekcji chemii, czy też w jakimkolwiek innym celu, grafika trójwymiarowa to tylko kolejny sposób na poruszanie pikselami po ekranie ku uciecie użytkownika. Na szczęście, programiści przeglądarek internetowych zaczęli dostrzegać te nowe trendy i powoli, lecz skutecznie zamieniają swoje produkty w multimedialne platformy programistyczne z obsługą wspomaganej sprzętowo obróbki grafiki oraz ze zintegrowaną architekturą kompozytową. Mówiąc prościej: grafika trójwymiarowa już istnieje i lepiej się do niej przyzwyczać.

Celem tej książki jest dostarczenie informacji niezbędnych do tworzenia wysokiej jakości trójwymiarowych aplikacji przeznaczonych do uruchamiania w środowisku przeglądarek komputerów stacjonarnych i przenośnych przy użyciu takich nowoczesnych technologii jak WebGL, kanwa oraz CSS3. W książce poruszone też są pokrewne tematy dotyczące wydajności JavaScriptu, programowania dla urządzeń mobilnych oraz projektowania wydajnych aplikacji sieciowych. Ponadto omówione zostały przyspieszające pracę narzędzia i biblioteki, takie jak Three.js, Tween.js, nowe systemy szkieletowe do budowy aplikacji oraz wiele możliwości tworzenia trójwymiarowej treści.

Czytelnicy mojej pierwszej książki, zatytułowanej *WebGL Up and Running*, zauważą, że część materiału się pokrywa. Nie da się tego uniknąć, ponieważ znaczna część materiału w początkowych rozdziałach ma charakter przeglądowy i wprowadzający, a więc nie można jej pominąć, odsyłając czytelnika do innej publikacji. Niemniej jednak, wyłączając pozorne podobieństwa widoczne w początkowych rozdziałach, czytelnicy pierwszej książki znajdą tu mnóstwo nowych wiadomości. Nawet początkowe rozdziały są o wiele bardziej szczegółowe niż w poprzedniej książce, która miała nieco inne przeznaczenie. Poza tym treść wszystkich rozdziałów od 4. włącznie jest praktycznie całkowicie nowa. Celem książki *WebGL Up and Running* było wprowadzenie w przystępny sposób do nowego i przytłaczającego tematu. To, czego brakowało tamtej książce pod względem technicznym, zostało nadrobione entuzjazmem. Jeśli ktoś po jej przeczytaniu nabrał ochoty, aby dowiedzieć się więcej, znaczy to, że osiągnąłem zamierzony cel. Natomiast w tej książce oferuję czytelnikowi praktyczne i teoretyczne podstawy, aby mógł rozpocząć budowanie trójwymiarowych aplikacji nadających się do ogólnego użytku.

Adresaci książki

Książka przeznaczona jest dla doświadczonych programistów sieciowych interesujących się programowaniem grafiki trójwymiarowej. Zakładam, że czytelnik jest średnio zaawansowanym programistą posiadającym solidne podstawy w posługiwaniu się technologiami HTML, CSS oraz JavaScript i przynajmniej trochę zna jQuery. Nie oczekuję natomiast wiedzy z zakresu grafiki trójwymiarowej ani animacji, chociaż z pewnością będzie pomocna. Książka zawiera krótkie wprowadzenie do technologii trójwymiarowych na początku oraz dodatkowe objaśnienia w różnych miejscach, w których są potrzebne.

Organizacja książki

Książka podzielona jest na cztery części.

Część I — „Podstawy” — zawiera opis API HTML5 i innych technologii związanych z programowaniem grafiki trójwymiarowej w przeglądarkach internetowych, takich jak WebGL, Canvas czy CSS3.

- Rozdział 1. to wprowadzenie do programowania aplikacji trójwymiarowych oraz objaśnienie najważniejszych pojęć dotyczących grafiki trójwymiarowej.
- Rozdziały od 2. do 5. zawierają opis podstaw programowania przy użyciu biblioteki WebGL, włącznie z API głównym i dwiema popularnymi, otwartymi bibliotekami do tworzenia grafiki i animacji, takimi jak Three.js i Tween.js.
- Rozdział 6. stanowi opis nowych narzędzi w CSS3 przeznaczonych do tworzenia trójwymiarowych efektów na stronach internetowych i w interfejsach użytkownika.
- Rozdział 7. jest omówieniem dwuwymiarowego API Canvas oraz sposobów emulacji przy jego użyciu efektów trójwymiarowych na platformach o ograniczonych zasobach.

W części II — „Techniki programowania aplikacji” — poruszone zostały tematy dotyczące programowania sieciowego w praktyce, w tym proces tworzenia grafiki trójwymiarowej, programowanie przy użyciu systemów szkieletowych oraz wdrażanie aplikacji na mobilnych platformach obsługujących HTML5.

- Rozdział 8. zawiera opis całego procesu kreowania treści, czyli narzędzi i formatów plików używanych przez artystów do tworzenia trójwymiarowych modeli i animacji.
- Rozdział 9. to przedstawienie metody wykorzystania systemów szkieletowych do szybkiego tworzenia aplikacji trójwymiarowych oraz wprowadzenie do Vizi, czyli otwartego systemu szkieletowego do budowania nadających się do wielokrotnego użycia elementów trójwymiarowych.
- Rozdziały 10. i 11. poświęcone są tworzeniu specyficznych rodzajów aplikacji trójwymiarowych: prostych programów do prezentowania pojedynczych interaktywnych obiektów w postaci animacji z możliwością interakcji. W rozdziale tym będzie można nauczyć się także używania skomplikowanych środowisk trójwymiarowych z wyszukaną nawigacją i wieloma oddziałującymi na siebie obiekttami.
- Rozdział 12. to opis kwestii związanych z programowaniem aplikacji trójwymiarowych dla nowej generacji urządzeń przenośnych i systemów operacyjnych obsługujących HTML5.

Przyjęte konwencje

W książce zastosowano następujące konwencje typograficzne.

Pogrubienie

Oznaczono nim nowe, ważne pojęcia.

Kursywa

Oznaczono nią adresy URL i e-mail, nazwy oraz rozszerzenia plików, a także opcje i nazwy okien programów.

Czcionka o stałej szerokości znaków

Oznaczono nią listingi kodu źródłowego programów oraz użyto jej do oznaczania fragmentów programów w akapitach, np. nazw zmiennych i funkcji, baz danych, typów danych, zmiennych środowiskowych, instrukcji oraz słów kluczowych.

Pogrubiona czcionka o stałej szerokości znaków

Oznaczono nią polecenia i inne teksty, które użytkownik powinien gdzieś wpisać, nic w nich nie zmieniając.

Czcionka o stałej szerokości znaków i kursywa

Oznaczono nią tekst do wpisania, który użytkownik powinien zastąpić własną lub wynikającą z kontekstu wartością.



W ten sposób oznaczone są ogólne uwagi.

Pliki z kodem źródłowym

Wszystkie pliki z kodem źródłowym przykładów opisanych w tej książce można pobrać pod adresem <ftp://ftp.helion.pl/przyklady/apli3d.zip>.

Większość przykładów należy wczytać z serwera sieciowego, a nie uruchamiać lokalnie w komputerze przy użyciu adresów `file://`. Wynika to z tego, że kod JavaScript ładuje dodatkowe zasoby, takie jak obrazy w formatach JPEG i PNG, a wbudowane w model WebGL zabezpieczenia dotyczące pochodzenia zasobów nakazują, aby pliki te były dostarczane z serwera sieciowego przy użyciu protokołu HTTP.

W swoim komputerze MacBook zainstalowałem standardowy zestaw LAMP, ale potrzebujesz tylko składnika oznaczonego literą A, czyli serwera sieciowego, np. Apache. A jeśli masz zainstalowany interpreter języka Python, możesz użyć modułu `SimpleHTTPServer`, który uruchomisz, wykonując w głównym katalogu przykładów poniższe polecenie:

```
python -m SimpleHTTPServer
```

Później wystarczy wpisać w przeglądarce internetowej adres `http://localhost:8000/`. Przydatne wskazówki można znaleźć w portalu Linux Journal pod adresem <http://bit.ly/linuxjournal-http-python>.

W przykładowych plikach znajdują się pełne wersje opisywanych w książce aplikacji, które można uruchomić. W niektórych przypadkach konieczne będzie pobranie dodatkowych plików, np. trójwymiarowych modeli, z internetu. Szczegółowe informacje na ten temat znajdują się w pliku `README`, umieszczonem w katalogu głównym.



Wiele używanych w tej książce zasobów jest chronionych prawami autorskimi. Właściciele tych praw udzielili mi zezwolenia na ich dystrybucję na użytkę tej książki *wyłącznie* w celu obsługi zamieszczonych w niej przykładowych programów. Jeśli chcesz ich użyć w jakimkolwiek innym celu, szczególnie zaś do budowy własnych aplikacji, musisz pobrać te zasoby jeszcze raz, co może wiązać się z koniecznością zakupu licencji.

Zasady wykorzystania przykładów

Książka ta ma pomóc Ci w pracy. Ogólnie rzecz biorąc, kodu znajdującego się w tej książce można używać we własnych programach i dokumentacjach bez proszenia kogokolwiek o zgodę, chyba że wykorzystasz duże fragmenty. Jeśli np. w pisany programie użyjesz kilku fragmentów kodu z tej książki, nie musisz pytać o pozwolenie. Aby sprzedawać i rozpowszechniać płyty CD-ROM z przykładami z książek wydawnictwa Helion, trzeba mieć zezwolenie. Aby odpowiedzieć komuś na pytanie, cytując fragment tej książki wraz z kodem źródłowym, nie trzeba mieć zezwolenia. Aby wykorzystać dużą ilość kodu źródłowego z tej książki w dokumentacji własnego produktu, trzeba mieć pozwolenie.

Informacje o źródle użytych fragmentów są mile widziane, ale niewymagane. Notka powinna zawierać tytuł publikacji, nazwisko autora, nazwę wydawcy oraz datę i miejsce publikacji, np. *Programming 3D Applications with HTML and WebGL*, Tony Parisi, Helion, Gliwice 2014.

Jeśli chcesz się upewnić, że wykorzystujesz przykłady kodu, nie łamiąc naszych zasad, wyślij pytanie na adres helion@helion.com.

Podziękowania

Książka jest owocem współpracy wielu osób; doprowadzenie do jej publikacji w pojedynkę byłoby niemożliwe. Dziękuję zespołowi wydawnictwa O'Reilly. Współpracująca ze mną redaktor Mary Treseler jest świetną liderką, dzięki której sprostałem wygórowanym wymaganiom, jakie przede mną postawiono. Pisanie tej książki zajęło prawie rok, co w kategoriach internetowych jest wiecznością; wielokrotnie musiałem ją modyfikować, aby nadążyć za zmianami technologii i potrzeb odbiorców. Mary była bardzo cierpliwa i pomocna przez cały ten czas. Redaktor Brian Anderson szybko i rzeczowo skomentował strukturę oraz układ rozdziałów, a asystentka redakcji Meghan Connolly wykazała się niezwykłymi umiejętnościami, przepuszczając moje surowe pliki Worda przez cały proces wydawniczy.

Na moją głęboką wdzięczność za doskonale recenzje zasłużyli Ray Camden, Raffaele Cecco, Mike Korcynski oraz Daniel Smith. Ich szczegółowe uwagi pomogły wyjaśnić wiele rzeczy oraz ulepszyć przykłady kodu źródłowego. Ponadto pozytywne reakcje utwierdziły mnie w przekonaniu, że znajduję się na właściwej drodze.

Napisanie książki o programowaniu grafiki wymaga utworzenia dużej ilości trójwymiarowej treści. Dlatego też jestem dozgólnie wdzięczny TC Changowi za ścisłą współpracę nad koncepcyjnym samochodem Futurgo, który został opisany w rozdziałach od 10. do 12. Jest to niewątpliwie perelka, której nie da się już poprawić. Chciałbym także podziękować artystom, którzy zezwolili na redystrybucję ich dzieł wraz z przykładami do tej książki. Szczegółowe informacje na ich temat znajdują się w pliku *README* oraz w plikach HTML i JavaScript poszczególnych przykładów. Specjalne podziękowania kieruję pod adresem Christella Gause'a, dyrektora TurboSquid, za pomoc w uzyskaniu pozwoleń od artystów z TurboSquid.

Mamy to szczęście, że społeczność programistów grafiki trójwymiarowej działa bardzo prężnie. Dziękuję zespołowi zajmującemu się biblioteką Three.js, a zwłaszcza jej twórcy Ricardo Cabelli (Mr.doob), za tę pionierską pracę. Wśród programistów budujących światowej klasy implementacje WebGL są Ken Russell i Brandon Jones z Google, którzy mimo dużej ilości zajęć zawsze znajdą czas, aby odpowiedzieć na pytania oraz podzielić się opinią na temat, w jakim kierunku zmierza rozwój technologii. Programowanie grafiki to jednak nie tylko WebGL, ale również aktywnie rozwijający się świat grafiki trójwymiarowej w CSS oraz grafika dwuwymiarowa na kanwie. Pionierzy tych dziedzin, czyli Dacid DeSandro, Keith Clark oraz Kevin Roast, których dokonania są przełomowe, uprzejmie pozwolili mi na odwołanie się do ich prac. Dodatkowo nie mogę też zapomnieć o wielkich podziękowaniach dla mojego przyjaciela Dona Olmsteada, którego sesje przeprowadzone ze mną kilka lat temu zaowocowały nowym systemem szkieletowym do programowania grafiki trójwymiarowej o nazwie Vizi. Piszę o nim dużo w różnych miejscach tej książki.

Na końcu chciałbym podziękować rodzinie. Wykazali wręcz anielską cierpliwość, gdy pisalem tę książkę, zajmując się jednocześnie jeszcze kilkoma innymi rzeczami. Marino i Lucianie, jestem wam winny wakacje i to trzy razy.

CZĘŚĆ I

Podstawy

Wprowadzenie

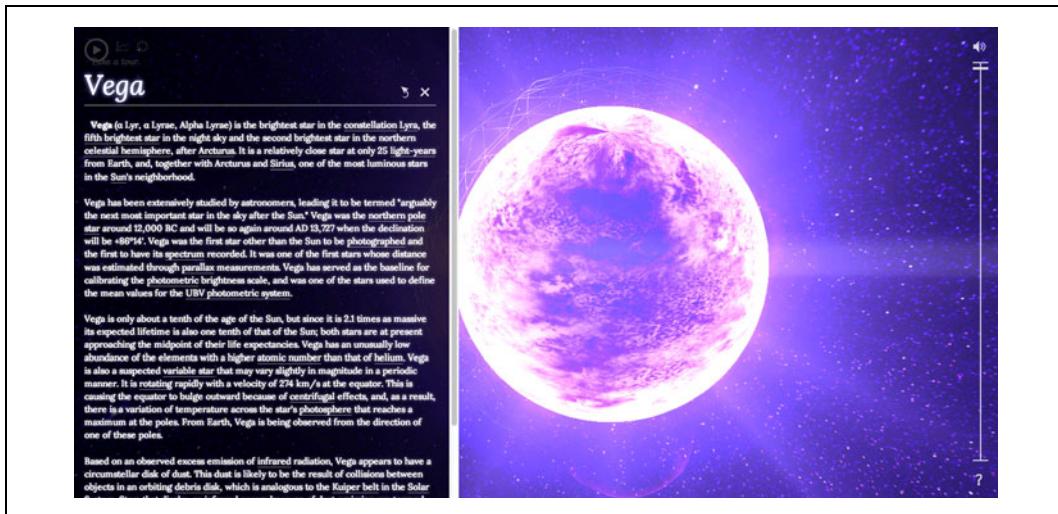
Żyjemy w świecie trójwymiarowym. Ludzie poruszają się, myślą i ogólnie funkcjonują w trzech wymiarach.

Także znacząca część mediów jest trójwymiarowa, mimo że przedstawia się je na płaskim ekranie. Filmy animowane tworzy się z generowanych komputerowo trójwymiarowych obrazów. Internetowe mapy umożliwiają przyjrzenie się miejscu, w które chcemy pojechać. Większość gier wideo, czy to na konsole, czy telefony komórkowe, również jest trójwymiarowa. Nawet wiadomości są w trzech wymiarach: niezgrabnie poruszający się wśród wirtualnych plansz analityk CNN dziś już nikogo nie dziwi, bo różne kanały konkurują o widza 24 godziny na dobę.

Grafika trójwymiarowa istnieje prawie tak długo, jak same komputery, gdyż jej korzenie sięgają lat 60. ubiegłego wieku. Wykorzystuje się ją w takich dziedzinach jak inżynieria, edukacja, szkolenia, architektura, finanse, handel i marketing, gry oraz rozrywka. Kiedyś do uruchomienia aplikacji trójwymiarowych trzeba było mieć najwyższej klasy komputer i drogie oprogramowanie, ale to już przeszłość. Sprzęt do przetwarzania grafiki trójwymiarowej znajduje się obecnie w każdym komputerze i urządzeniu mobilnym, a dzisiejsze smartfony mają większe możliwości przetwarzania grafiki niż profesjonalne stacje robocze sprzed piętnastu lat. Co ważniejsze, oprogramowanie potrzebne do renderowania grafiki trójwymiarowej jest powszechnie dostępne i darmowe. Są nim przeglądarki internetowe.

Na rysunku 1.1 przedstawiono fragment projektu 100 000 Stars, przeglądarkowego trójwymiarowego symulatora lotu przez kosmos w obrębie Drogi Mlecznej. Przy użyciu myszy można obracać płaszczyznę galaktyki oraz przybliżać konkretne gwiazdy. Gwiazdy są reprezentowane za pomocą punktów o cechach zbliżonych do ich rzeczywistej skali i koloru. Każda gwiazda jest podpisana. Gdy najedzie się kursem na etykietę, zostaje ona podświetlona. Kliknięcie powoduje pojawienie się nakładki zawierającej artykuły z Wikipedii na temat danej gwiazdy. Kliknięcia łącza w tej nakładce powoduje otwarcie nowej strony na nowej karcie. 100 000 Stars to zapierające dech w piersiach widowisko, pełne pięknych grafik, pulsujących animacji i podniosłej muzyki połączone z dwuwymiarowym interfejsem użytkownika.

Projekt 100 000 Stars jest owocem eksperymentu zespołu Data Arts firmy Google, którego celem było zademonstrowanie bogatych możliwości przeglądarki Chrome. Aplikacja jest eksperymentalna, ale zastosowane do jej budowy technologie już nie. Została zbudowana przy użyciu składników HTML5, które obecnie obsługiwane są już przez większość przeglądarek internetowych. Galaktyka i gwiazdy są generowane na bieżąco za pomocą WebGL, nowego



Rysunek 1.1. Projekt Google 100 000 Stars (<http://workshop.chromeexperiments.com/stars/>); obraz opublikowany dzięki uprzejmości firmy Google Inc.

standardu generowania grafiki trójwymiarowej z przyspieszeniem sprzętowym. Etykiety zostały umieszczone w pobliżu gwiazd z wykorzystaniem trójwymiarowych przekształceń CSS3, a nakiadki łączą się niepostrzeżenie z trójwymiarową treścią, dlatego że przeglądarki składają wszystkie elementy strony w jednolitą prezentację.

Jeszcze kilka lat temu efekty, jakie obserwujemy w aplikacji 100 000 Stars, można było osiągnąć tylko w macierzystej aplikacji o dużych rozmiarach, którą trzeba było pobrać i zainstalować w komputerze. Tworzący ją programiści musieliby używać skomplikowanych narzędzi oraz poświęcić na nią mnóstwo czasu, co generowałoby znaczące koszty. Dziś taki program można zbudować przy użyciu przeglądarki internetowej, darmowych i otwartych narzędzi oraz standardowych technologii internetowych. Co więcej, wystarczy odświeżyć stronę, aby szybko pobrać aktualizację, można też załadować informacje z dowolnego miejsca w sieci przy użyciu adresu URL lub kliknąć hiperłącze i dowiedzieć się więcej na wybrany temat.

Tematem tej książki są techniki pozwalające na wykorzystanie niesamowitej mocy drzemiącej w nowoczesnych przeglądarkach internetowych do tworzenia nowego typu połączonych aplikacji wizualnych. Niektóre z nich przypominają swoich przodków, są zasadniczo przeróbkami tradycyjnych trójwymiarowych produktów przystosowanymi do nowego środowiska, aby dotrzeć do nowych odbiorców i zredukować koszty. Jednak znacznie ciekawsze są nowatorskie zastosowania tych technik w takich dziedzinach jak reklama, marketing, obsługa klienta, edukacja, szkolenia, turystyka, gry czy rozrywka. Trzeci wymiar wnosi powiew świeżości do aplikacji interaktywnych, a w połączeniu z technologiami sieciowymi jest już dostępny dla każdego na tej planecie.



100 000 Stars stanowi pokaz możliwości interaktywnych mediów. Jeden z twórców aplikacji, Michael Chang, napisał świetne studium przypadku na temat tego projektu, które zostało opublikowane na stronie <http://www.html5rocks.com/en/tutorials/casestudies/10000stars/>.

HTML5 jako nowe medium wizualne

Język HTML znacznie ewoluował od czasów, gdy w internecie istniały tylko statyczne strony internetowe, formularze oraz przycisk zatwierdzania. Na początku 2000 r. w przeglądarkach pojawiły się pierwsze zwiastuny nadchodzącej interaktywnej ery w postaci technologii Ajax umożliwiającej dynamiczne odświeżanie tylko wybranych części stron. Jednak zastosowanie technik ajaksowych było ograniczone przez niewielkie możliwości graficzne technologii HTML i CSS. Jeśli ktoś potrzebował bardziej zaawansowanych efektów, musiał korzystać z multimedialnych wtyczek, takich jak Flash i QuickTime.

Tak mniej więcej wyglądała sytuacja w pierwszych latach XXI wieku, ale od tamtej pory wiele się zmieniło. Niektóre nowoczesne technologie, które w tamtych czasach były dopiero w fazie rozwojowej, wcielono do nowego standardu HTML5. Przeglądarka obsługująca HTML5 jest w istocie platformą, na której można uruchamiać wyszukane aplikacje o takich wydajnościach i funkcjonalnościach, jakie mogą rywalizować z programami macierzystymi. Standard HTML5 wnosi do języka HTML wiele nowości i zmian, takich jak dopracowana składnia, nowe funkcje i interfejsy API dla JavaScriptu, funkcje do obsługi urządzeń przenośnych oraz przełomowe narzędzia dla multimedialnych. Bardzo ważnym aspektem platformy HTML5 jest zestaw zaawansowanych technologii graficznych, którym poświęcona jest ta książka. Oto one.

- **WebGL** to wspierana sprzętowo technologia renderowania grafiki trójwymiarowej przy użyciu JavaScriptu. Standard ten oparty na znany i cenionym API OpenGL jest już obsługiwany przez praktycznie wszystkie najważniejsze przeglądarki na komputery stacjonarne oraz coraz większą liczbę przeglądarek na urządzeniach przenośnych.
- **Trójwymiarowe przekształcenia, przejścia i filtry CSS3** pozwalają na stosowanie na stronach internetowych zaawansowanych efektów wizualnych. Język CSS przeszedł szereg zmian w ciągu kilku ostatnich lat, dzięki czemu obecnie umożliwia korzystanie ze wspomaganych sprzętowo funkcji renderingu trójwymiarowego i tworzenia animacji.
- **Element kanwy** i jego API do dwuwymiarowego kontekstu rysunkowego są obsługiwane przez wszystkie przeglądarki. Przy użyciu tego interfejsu JavaScript można rysować dowolne grafiki na powierzchni elementu DOM. Mimo że kanwa jest powierzchnią dwuwymiarową, można na niej renderować także obrazy trójwymiarowe, jeśli użyje się odpowiednich bibliotek języka JavaScript. Stanowi ona zatem alternatywę dla WebGL i trójwymiarowych składników CSS3.

Każda z wymienionych technologii ma zalety i wady oraz techniczne ograniczenia, ale każda z nich ma też swoje miejsce w interaktywnym, wizualnym świecie trójwymiarowych aplikacji. Wybór jednej z nich jest podykowany kilkoma czynnikami. Należy rozważyć, co chce się osiągnąć, na jakich platformach ma działać program, trzeba wziąć pod uwagę wymagania dotyczące wydajności itd. Powiedzmy np., że chcemy utworzyć strzelankę FPS odznaczającą się najwyższą jakością grafiką. Bez dostępu do sprzętu renderującego, który to dostęp umożliwia biblioteka WebGL, zrealizowanie tego planu byłoby trudne. Z drugiej strony, gdybyśmy tworzyli przypominający radio interfejs odtwarzacza filmów na stronie internetowej, zawierający animowane miniatury, pokrętła, rozmyte przejścia między kolejnymi filmami itd., doskonalem wyborem mogłyby być techniki CSS3.



Jeden standard zamiast wielu...

To, co większość programistów sieciowych określa mianem HTML5, w rzeczywistości jest zbiorem technologii i standardów. Niektóre zostały już w pełni zatwierdzone przez W3C (ang. *World Wide Web Consortium*) i zaimplementowane w przeglądarkach. Nad innymi wciąż się pracuje, mimo że też już są całkiem dobrze obsługiwane. Istnieje też grupa stabilnych i dokończonych standardów, na które W3C nie ma wpływu, np. WebGL.

Przeglądarka jako platforma

Technologia HTML5 umożliwiła wykorzystanie bogatych grafik w internecie, chociaż niewiele by z tego wynikło, gdyby nie wprowadzono pewnych innych ważnych udoskonaleń w przeglądarkach internetowych. Oto lista technik, które utorowały drogę do tworzenia bogatych interaktywnych aplikacji przy użyciu HTML5.

Wydajność maszyny wirtualnej JavaScript

WebGL i Canvas 2D to API JavaScript, więc wydajność animacji i elementów interaktywnych jest zależna od szybkości działania kodu JavaScript, na którym bazują. Jeszcze kilka lat temu wydajność maszyny wirtualnej była zbyt mała, aby można było myśleć o praktycznym wykorzystaniu technik programowania grafiki trójwymiarowej. Jednak dzisiejsze maszyny wirtualne to prawdziwe demony prędkości.

Przyspieszone składanie

Przeglądarka bardzo szybko składa różne elementy strony w jedną całość i nie powoduje przy tym powstawania żadnych artefaktów. Od czasu gdy zaczęto tworzyć bardziej dynamiczną treść, możliwości przeglądarek w zakresie składania stron, włącznie z renderowaniem elementów dwu- i trójwymiarowych, znacznie wzrosły.

Wspomaganie animacji

Wprowadzono funkcję `requestAnimationFrame()`, której należy używać do sterowania animacjami zamiast funkcji `setInterval()` i `setTimeout()`. Ta nowa metoda pozwala znacznie zwiększyć wydajność oraz eliminuje powstawanie artefaktów, ponieważ treść elementów na kanwie może być rysowana w tym samym przebiegu, co rysowanie elementów strony.

Ponadto przeglądarki obsługujące HTML5 obsługują wielowątkowość (**Web Workers**), pełnodupleksową komunikację TCP/IP (**WebSockets**), lokalne przechowywanie danych oraz wiele innych technologii umożliwiających tworzenie najwyższej klasy aplikacji sieciowych. Wszystko to w połączeniu z WebGL, CSS3 3D oraz kanwą stanowi nową rewolucyjną **platformę** do uruchamiania połączonych aplikacji wizualnych działających na każdym komputerze.

Na rysunku 1.2 można zobaczyć demonstracyjną wersję aplikacji Epic Citadel studia Epic Games działającą w przeglądarce Firefox. Wykorzystano w niej WebGL do renderowania grafiki, ale prawdziwą nowością jest niespotykana do tej pory wydajność silnika gier. Użyto silnika **Unreal**, który przerobiono z macierzystego kodu C++ na implementację przeglądarkową za pomocą kompilatora **Emscripten** (<https://github.com/kripken/emscripten/wiki>) oraz asm.js, czyli nowego zoptymalizowanego niskopoziomowego podzbioru JavaScript. Każdy, kto ma przeglądarkę internetową, może wpisać odpowiedni adres URL, aby cieszyć oko pełnoekranową grą konsolową działającą z szybkością 60 klatek na sekundę. A przy tym czas pobierania jest bardzo krótki i nic nie trzeba instalować.



Rysunek 1.2. Demonstracja Epic Citadel działająca w Firefoksie. Jest to przeglądarkowa gra oparta na WebGL i asm.js działająca z prędkością 60 fps (<http://www.unrealengine.com/html5>); obraz opublikowany dzięki uprzejmości firmy Epic Games

Przeglądarkowa rzeczywistość

W czasie pisania tej książki narzędzia do programowania grafiki trójwymiarowej nie były w pełni obsługiwane przez wszystkie przeglądarki internetowe. Poza tym każda przeglądarka obsługuje inny zestaw składników. Szczegółowe informacje podaję w poszczególnych rozdziałach, a poniżej przedstawiam tylko ogólne spostrzeżenia.

- WebGL obsługują wszystkie przeglądarki na komputery stacjonarne. Firma Microsoft wprowadziła obsługę tej technologii w Internet Explorerze 11 pod koniec 2013 r. Implementacja ta na razie nie dorównuje implementacjom w innych przeglądarkach, ale należy się spodziewać, że Microsoft szybko nadrobi zaległości.
- WebGL obsługują prawie wszystkie przeglądarki dla urządzeń przenośnych: Chrome (Android), Firefox (Android i Firefox OS), Amazon Silk (Kindle Fire HDX), nowy system operacyjny Tizen firmy Intel oraz BlackBerry 10. Ponadto w ograniczonym stopniu bibliotekę tę obsługuje przenośna wersja przeglądarki Safari (tylko w systemie iAds).
- Trójwymiarowe przekształcenia CSS są obsługiwane przez wszystkie przeglądarki i platformy przenośne. Filtry CSS są obsługiwane tylko eksperymentalnie przez Chrome, Safari, przenośną wersję Safari oraz BlackBerry 10, ani Firefox, ani IE nie obsługują tych filtrów.

Niewątpliwie nie jest to komfortowa sytuacja, ale tak to już jest z programowaniem aplikacji sieciowych. Niespójna obsługa różnych technologii przez przeglądarki internetowe od zawsze sprawia problemy programistom, a dynamiczny rozwój nowych technologii HTML5 i pojawianie

się jak grzyby po deszczu nowych urządzeń tylko pogarszają sytuację. Jedyną pociechą jest to, że alternatywa jest jeszcze gorsza. Pisanie, testowanie, wdrażanie i przenoszenie macierzystych aplikacji jest jeszcze trudniejsze. No cóż, takie jest życie programisty w XXI wieku.



Przy tak dużej liczbie różnych standardów powinniśmy zbliżyć się do idealnej sytuacji, w której wystarczy napisać kod tylko raz. Jednak — niestety — codziennie boleśnie przekonujemy się, że powtarzane jak mantra słowa: „Napisz raz, uruchamiaj wszędzie” na razie należy zastąpić skargą: „Napisz raz, debuguj wszędzie”.

Grafika trójwymiarowa

W tej części znajduje się objaśnienie podstawowych pojęć z dziedziny grafiki trójwymiarowej. Programiści, którzy do tej pory posługiwali się dwuwymiarową kanwą, mogą znaleźć tu sporo nowych wiadomości. Warto je zapamiętać, ponieważ wiedza ta będzie potrzebna do zrozumienia treści książki. Czytelnicy obeznani z tematyką programowania grafiki trójwymiarowej i biblioteką OpenGL mogą przejść do następnego rozdziału.

Co to jest trójwymiarowość?

Skoro wziąłeś tę książkę do ręki, zakładam, że przynajmniej z grubsza wiesz, co oznacza pojęcie **grafika trójwymiarowa**. Żeby nie pozostawiać jakichkolwiek wątpliwości, zdefiniuję to pojęcie formalnie i przeanalizuję jego definicję. Poniżej znajduje się fragment definicji z angielskiej Wikipedii (http://en.wikipedia.org/wiki/3D_computer_graphics):

Komputerowa grafika trójwymiarowa (w odróżnieniu od dwuwymiarowej) to grafika wykorzystująca trzy wymiary do reprezentacji danych geometrycznych (często kartezjańskich), które są przechowywane w komputerze w celu wykonywania obliczeń i renderowania obrazów dwuwymiarowych. Obrazy takie można przechowywać w celu wyświetlenia w odpowiednim momencie albo wyświetlać na bieżąco.

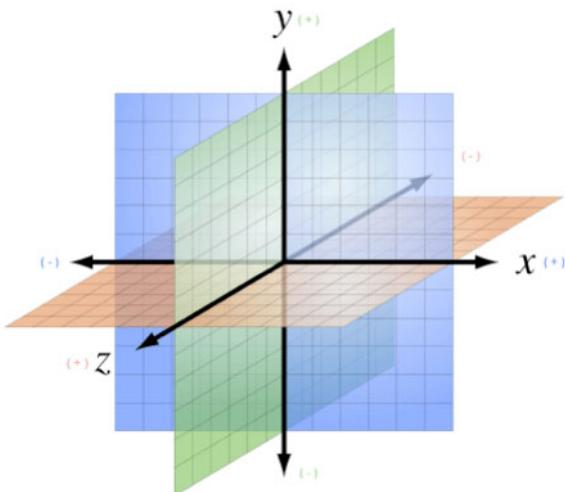
Podzielę tę definicję na kilka części: 1) dane są reprezentowane w trójwymiarowych układzie współrzędnych; 2) ostatecznie grafika jest rysowana (**renderowana**) jako obraz dwuwymiarowy (np. na ekranie monitora); 3) grafikę tę można wyświetlać na bieżąco: gdy dane są zmieniane w ramach animacji albo z powodu działań użytkownika, następuje ich aktualizacja bez widocznego opóźnienia. Kluczowa dla tworzenia interaktywnych aplikacji jest trzecia część przytoczonej definicji. Jest ona tak ważna, że na niej opiera się warta miliardy dolarów branża zajmująca się tworzeniem specjalnych urządzeń wspomagających renderowanie grafiki trójwymiarowej na bieżąco, w której liderami są z pewnością dobrze znane firmy, takie jak NVIDIA, ATI i Qualcomm.

Równie ważne jest to, czego w definicji tej *nie uwzględniono*: grafika trójwymiarowa nie wymaga używania specjalnych urządzeń wejściowych typu trackballe czy dżojstiki, chociaż mogą one znacznie poprawić komfort użytkownika. Nie trzeba też używać żadnego specjalnego sprzętu wizualnego, żadnych okularów ani biletów do trójwymiarowego kina. Grafikę trójwymiarową najczęściej renderuje się na płaskim dwuwymiarowym ekranie. Oczywiście, nie znaczy to, że *nie można jej wyświetlić* w technologii stereo i oglądać w specjalnych okularach albo na specjalnym telewizorze. Chodzi tylko o to, że nie ma takiego przymusu.

Programowanie grafiki trójwymiarowej wymaga nauczenia się nowych rzeczy i nabycia umiejętności wykraczających poza standardową wiedzę programisty sieciowego. Jednak dysponując podstawowymi wiedomościami i odpowiednimi narzędziami, można szybko wszystkiego się nauczyć. Do końca tego rozdziału opisuję podstawowe pojęcia dotyczące grafiki trójwymiarowej, którymi będę się posługiwał dalej w książce. Oczywiście, przedstawiam tylko podstawowe informacje (szczegółowe potraktowanie tematu zajęłoby całą książkę), dzięki którym będziesz mógł rozpocząć pracę. Czytelnicy znający już programowanie grafiki trójwymiarowej mogą od razu przejść do rozdziału 2.

Trójwymiarowe układy współrzędnych

Jeśli nie są Ci obce kartezjańskie układy współrzędnych, np. okien albo dokumentów HTML, wiesz, czym są współrzędne x i y . Za ich pomocą określa się położenie elementów `<div>` na stronach internetowych oraz miejsce rysowania piórem albo pędzlem na kanwie HTML. Podobnie jest z grafiką trójwymiarową, która rysuje się w trójwymiarowym układzie współrzędnych zawierającym dodatkową współrzędną z oznaczającą głębię (tzn. jak daleko w głąb lub na zewnątrz ekranu rysowany jest obiekt). Układy współrzędnych używane w tej książce przedstawiają się tak, jak pokazano na rysunku 1.3. Oś x biegnie w poziomie (od lewej), y jest pionowa, a dodatnia oś z kieruje się na zewnątrz ekranu. Osoby obeznane z dwuwymiarowym układem współrzędnych nie powinny mieć problemu z przyzwyczajeniem się do trójwymiarowego.



Rysunek 1.3. Trójwymiarowy układ współrzędnych (<http://bit.ly/wikimedia-3d-coordinate>)

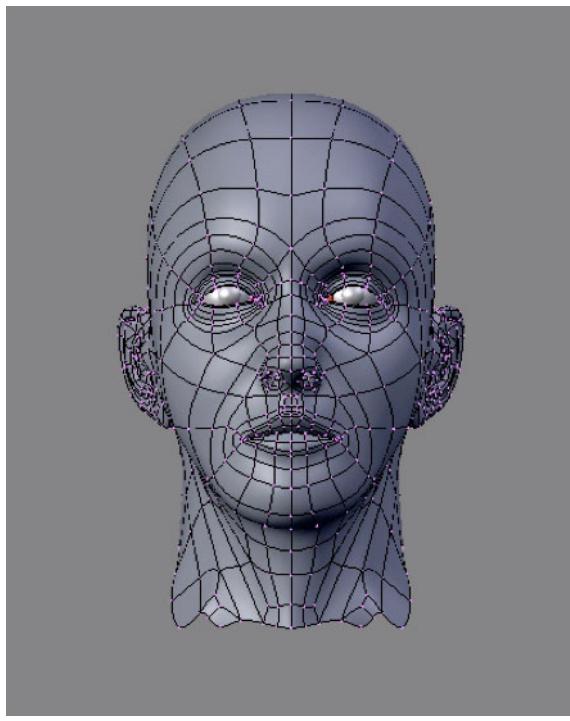


W WebGL dodatnia część osi y biegnie od dołu okna, natomiast w dwuwymiarowym API Canvas i przekształceniach CSS dodatnia część osi y biegnie w dół. Jest to niekorzystna sytuacja, ale odzwierciedla pochodzenie każdej z wymienionych technologii: WebGL opiera się na starych standardach graficznych, a kanwa i CSS bazują na konwencjach języka HTML, który sam wykorzystuje stare konwencje systemu okien. Jeśli dwie różniące się pod tym względem technologie są używane w jednym projekcie, trzeba pamiętać o tej różnicy. Mogło być jeszcze gorzej — oś z też mogła być zdefiniowana na różne sposoby, ale — na szczęście — nie jest.

Siatki, wielokąty i wierzchołki

Istnieje kilka sposobów rysowania grafiki trójwymiarowej, ale najczęściej używa się tzw. **siatki** (ang. *mesh*). Siatka to obiekt złożony z jednego wielokąta lub większej ich liczby; tworzone są one przy użyciu **wierzchołków** (trójkę współrzędnych x , y i z) określających miejsca w trójwymiarowym układzie współrzędnych. Najczęściej używanymi kształtami są trójkąty (grupy trzech wierzchołków) i czworokąty (grupy czterech wierzchołków). Trójwymiarową siatkę często nazywa się **modelem**.

Na rysunku 1.4 widoczna jest trójwymiarowa siatka. Ciemne linie to boki składających się na nią czworokątów tworzących kształt twarzy (w wyrenderowanym obrazie linie te są niewidoczne, tu zostały pokazane w celach edukacyjnych). Współrzędne x , y i z wierzchołków siatki definiują *tylko* kształty. Natomiast właściwości siatki, takie jak kolor i cieniowanie, definiuje się przy użyciu osobnych atrybutów, o których jest mowa dalej.



Rysunek 1.4. Trójwymiarowa siatka (<http://bit.ly/1dnAjAG>). Licencja Uznanie autorstwa-Na tych samych warunkach 3.0 Unported

Materiały, tekstury i oświetlenie

Właściwości powierzchni siatki definiuje się przy użyciu dodatkowych atrybutów. Mogą one być proste, np. jednolity kolor, lub skomplikowane i składające się z kilku rodzajów informacji określających sposób odbijania światła od obiektów lub intensywność połysku przedmiotów. Ponadto informacje dotyczące powierzchni siatki mogą mieć postać mapy bitowej, czyli tzw.

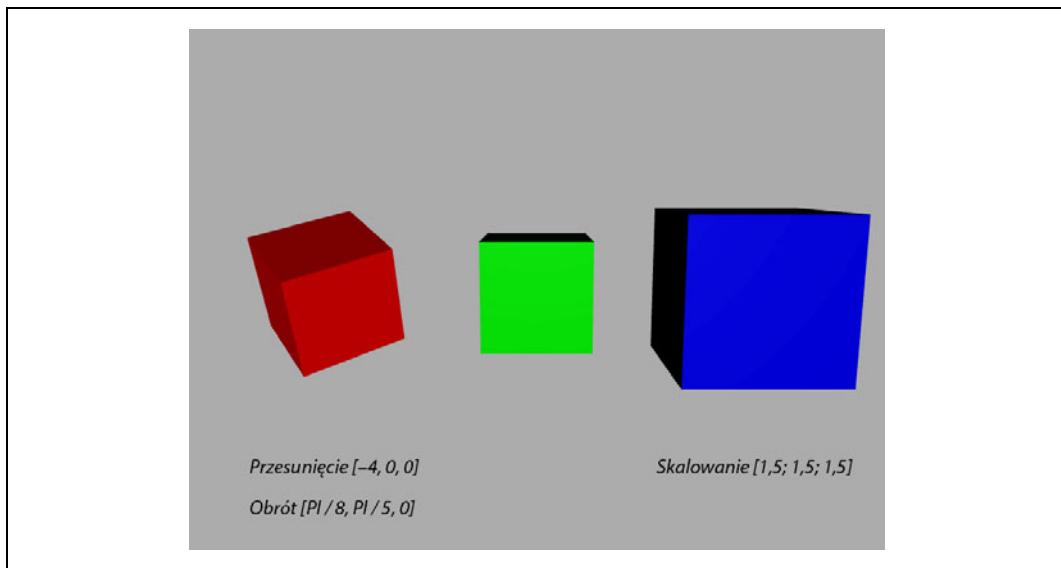
mapy teksturowej (albo krótko **tekstury**). Za pomocą tekstur można dokładnie określić wygląd powierzchni (np. obraz wydrukowany na koszulce) albo można je łączyć z innymi teksturami, aby uzyskać wyszukane efekty, takie jak pofałdowanie podłoża czy też opalizacja. W większości systemów graficznych właściwości powierzchni siatki ogólnie nazywa się **materiałami**. Ich prezentacja z reguły zależy od obecności źródeł **oswietlenia**, które określają oświetlenie sceny.

Przedstawiona na rysunku 1.4 głowa jest powleczena materiałem w fioletowym kolorze oraz ma cieniowanie zdefiniowane przez światło padające z lewej strony. Zwróć uwagę na znajdujące się po prawej cienie.

Przekształcenia i macierze

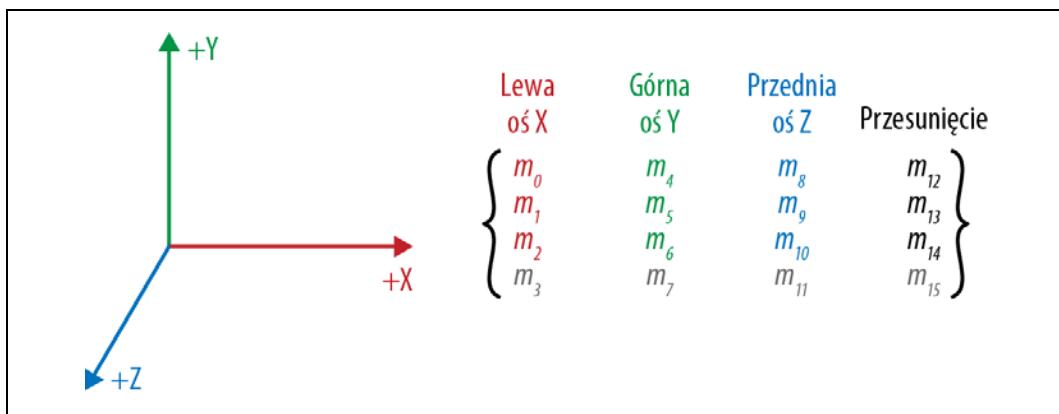
Kształt trójwymiarowej siatki jest zdefiniowany przez odpowiednio rozmieszczone wierzchołki. Gdyby jednak przy każdym przesuwaniu siatki w inne miejsce na widoku trzeba było zmieniać pozycję każdego wierzchołka, zwłaszcza w animacji, praca byłaby bardzo żmudna. Dlatego w większości trójwymiarowych systemów dostępne są tzw. **przekształcenia** (ang. *transforms*), czyli operacje umożliwiające względne przesuwanie siatki na określoną odległość bez zmieniać za pomocą pętli wszystkich wierzchołków. Przy użyciu przekształceń można skalować, obracać i przesuwać siatkę, nie zmieniając ani jednej wartości jej wierzchołków.

Na rysunku 1.5 pokazano kilka rodzajów przekształceń trójwymiarowych. Na scenie znajdują się trzy sześciany. Każdy z nich jest sześcienną siatką zdefiniowaną przez wierzchołki o takich samych wartościach. Aby przesunąć, obrócić i przeskalać siatkę, nie trzeba zmieniać wartości jej wierzchołków. Wystarczy tylko zastosować przekształcenie. Znajdujący się po lewej stronie czerwony sześcian został przesunięty o cztery jednostki w lewo (-4 na osi x) oraz obrócony wokół osi x i y (wartości obrotu określa się w **radianach**, o których więcej piszę w rozdziale 4.). Sześcian po prawej został przesunięty o cztery jednostki w prawo oraz przeskalowany o współczynnik $1,5$ we wszystkich trzech wymiarach. Zielony sześcian w środku nie był przekształcany.



Rysunek 1.5. Przekształcenia trójwymiarowe: przesunięcie, obrót i skalowanie

Przekształcenia trójwymiarowe najczęściej przedstawia się w postaci **macierzy przekształceń**, czyli matematycznego obiektu reprezentującego zestaw wartości służących do obliczenia pozycji wierzchołków po przekształceniu. W WebGL większość przekształceń wykonuje się przy użyciu **macierzy 4×4** , czyli tablicy 16 liczb rozmieszczonych w czterech rzędach po cztery kolumny. Na rysunku 1.6 pokazany jest układ takiej macierzy. Wartości dotyczące przesunięcia znajdują się w elementach m_{12} , m_{13} oraz m_{14} , które odpowiadają wartościom przesunięcia x , y i z . Wartości x , y i z skalowania znajdują się w elementach m_0 , m_5 oraz m_{10} (na **przekątnej** macierzy). Wartości obrotu reprezentują elementy m_1 i m_2 (oś x), m_4 i m_6 (oś y) oraz m_8 i m_9 (oś z). Mnożąc trójwymiarowy wektor przez tę macierz, otrzymuje się przekształconą wartość.



Rysunek 1.6. Macierz przekształceń trójwymiarowych (http://www.songho.ca/opengl/gl_transform.html); adaptacja za pozwoleniem

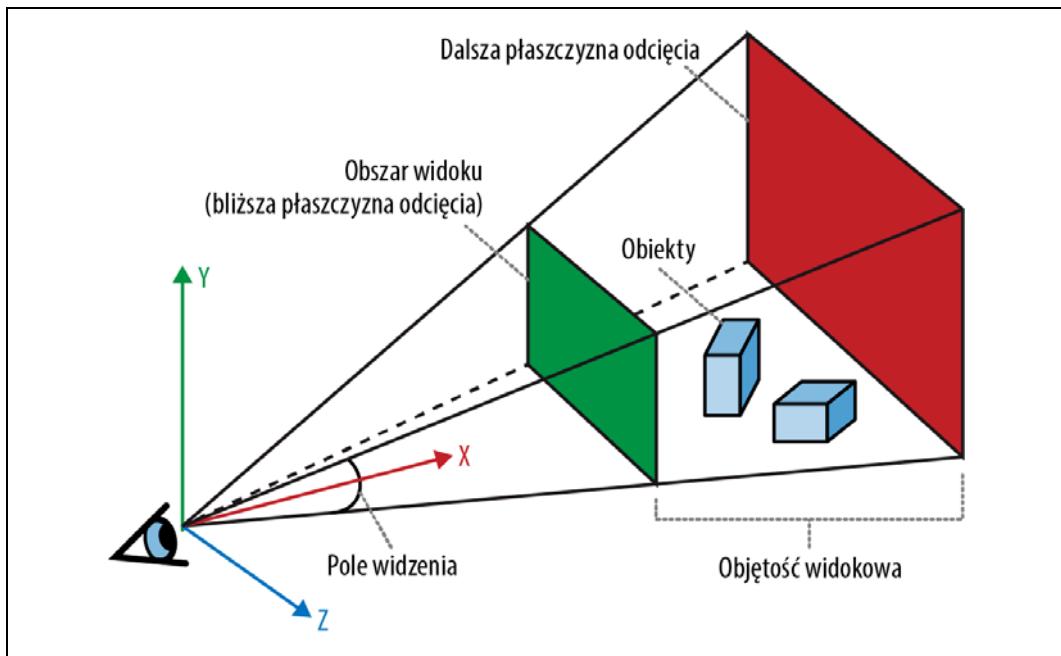
Jeśli lubisz algebrę, tak jak ja, powyższa macierz nie jest niczym niezwykłym. A jeśli nie, nie martw się. Dzięki narzędziom użytym do budowy przykładów przedstawionych w tej książce macierze takie można traktować jak czarne skrzynki — wystarczy wydać polecenie skalowania, obrotu i przesunięcia... i wszystko zostaje wykonane.

Kamery, perspektywa, obszary widoku oraz projekcje

Każda renderowana scena musi mieć określony punkt patrzenia, z którego użytkownik będzie ją oglądał. W systemach trójwymiarowych najczęściej używa się pojęcia **kamery**, czyli obiektu określającego pozycję i orientację (względem sceny) użytkownika, oraz jej właściwości, takich jak np. rozmiar **pola widzenia**, które definiuje **perspektywę** (tzn. obiekty znajdujące się dalej wydają się mniejsze). Właściwości kamery wspólnie dostarczają wyrenderowany ostateczny obraz sceny trójwymiarowej do dwuwymiarowego **obszaru widoku** znajdującego się w oknie lub na kanwie.

Kamery prawie zawsze reprezentują dwie macierze. Pierwsza z nich określa pozycję i orientację kamery, podobnie jak macierz używana do przekształceń. Natomiast druga macierz reprezentuje translację trójwymiarowych współrzędnych kamery na dwuwymiarową powierzchnię rysunkową obszaru widoku. Nazywa się ją **macierzą projekcji** (ang. *projection matrix*). Wiem, wiem: znów matematyka. Jednak wewnętrzne mechanizmy działania kamery są ukryte w większości narzędzi, więc programista musi tylko celować, strzelać i renderować.

Na rysunku 1.7 przedstawione są podstawowe pojęcia dotyczące kamery, obszaru widoku i projekcji. Na dole po lewej znajduje się ikona oka reprezentująca lokalizację kamery. Czerwony wektor w prawo (oznaczony jako oś x) reprezentuje kierunek, w którym zwrócona jest kamera. Niebieskie sześciiany to obiekty znajdujące się na trójwymiarowej scenie. Prostokąty zielony i czerwony to odpowiednio **bliższa i dalsza płaszczyzna odcięcia**. Płaszczyzny te stanowią granice podzbioru przestrzeni trójwymiarowej zwanej **objętością widokową** (ang. *view volume*) lub **ostrosłupem ściętym** (ang. *frustum*). Na ekranie renderowane są tylko obiekty znajdujące się w tym obszarze. Bliższa płaszczyzna odcięcia pokrywa się z obszarem widoku, w którym widoczny jest ostatecznie wyrenderowany obraz.



Rysunek 1.7. Kamera, obszar widoku oraz projekcja (<http://bit.ly/obviam-perspective>); adaptacja za pozwoleniem

Kamera jest bardzo ważna, ponieważ określa relację użytkownika ze sceną trójwymiarową oraz odpowiada za realizm. Ponadto kamery są jednym z narzędzi animacji: poruszając nimi dynamicznie, można tworzyć efekty kinematyczne oraz sterować narracją.

Programy cieniujące

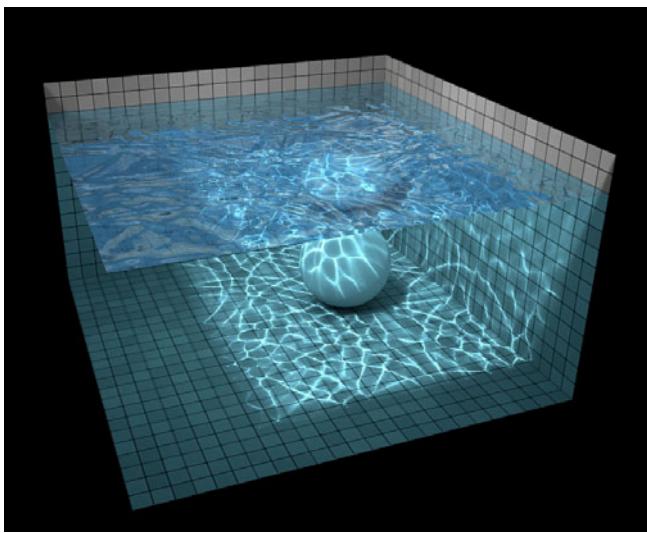
Aby wyrenderować obraz siatki, programista musi dokładnie określić sposób interakcji wierzchołków, przekształceń, materiałów, światła i kamery. Służą do tego programy cieniujące (tzw. **shadery**). **Shader** to program zawierający implementacje algorytmów odpowiadających za przesunięcie pikseli tworzących siatkę na ekran. Sprzęt graficzny rozpoznaje wierzchołki, tekstury i niewiele więcej. Obce są mu takie pojęcia jak materiał, oświetlenie, przekształcenie czy kamera. Są to konstrukcje wysokopoziomowe podlegające interpretacji przez program cieniujący. Shadery najczęściej pisze się w wysokopoziomowym języku, podobnym do C, a następnie kompliluje się je po to, by mogły być używane przez procesor graficzny (GPU).



Wszystkie nowoczesne komputery i urządzenia zawierają procesor graficzny, czyli osobną jednostkę obok CPU, której zadaniem jest tylko renderowanie grafiki trójwymiarowej. W większości technik programowania grafiki trójwymiarowej opisanych w tej książce przyjęte jest założenie, że komputer zawiera GPU.

Shadery dają programiście niezwykłe możliwości. Dzięki nim zyskuje się pełną kontrolę nad każdym pikselem za każdym razem, gdy renderowany jest obraz. To przy ich użyciu otrzymuje się oszałamiające efekty specjalne w hollywoodzkich produkcjach, filmach animowanych i grach wideo. Mając je do dyspozycji w przeglądarkach internetowych, można w aplikacjach WebGL stosować takie same techniki produkcyjne, jakie wykorzystuje się w najlepszych grach wideo. A dodatkowo, jeśli trzeba, można posłużyć się technikami CSS, żeby dostosować prezentację i animację elementów na stronach.

Na rysunku 1.8 można zobaczyć symulator wody napisany przy użyciu WebGL i renderowany przez shader. Zmarszczki wodne i płaszące światła są niezwykle realistyczne, a użytkownik może „bawić się wodą” w trakcie trwania symulacji. Pamiętaj, że to wszystko odbywa się w oknie przeglądarki internetowej!



Rysunek 1.8. Symulacja wody w WebGL wyrenderowana przy użyciu shaderów. Autor Evan Wallace (<http://madebyevan.com/webgl-water/>); przedruk za pozwoleniem

Efekty uzyskiwane za pomocą shaderów nie muszą się ograniczać do WebGL. Można je też stosować do obróbki elementów DOM przy użyciu eksperymentalnej technologii o nazwie **CSS Custom Filters** (własne filtry CSS). Temat ten rozwinąłem w rozdziale 6.

Oto kilka uwag na temat shaderów w odniesieniu do technologii, które są opisane w tej książce.

- WebGL i własne filtry CSS wykorzystują shadery, które definiuje się w języku o nazwie OpenGL ES Shader Language (w skrócie GLSL ES). Shadery dla WebGL nieco różnią się od tych dla CSS, ale podstawy języków są identyczne.

- Programista używający WebGL *musi* dostarczyć shadery, aby jego obiekty zostały wyrenderowane. Jeśli ich brakuje albo wystąpią błędy podczas ich komplikacji lub ładowania, na ekranie nic nie zostanie wyrenderowane.
- Dla filtrów CSS3 shadery *nie są obowiązkowe*. Gdy z filtrem CSS3 używany jest shader, filtr taki nazywa się **filtrem własnym**.
- Dwuwymiarowe API kanwy *nie obsługuje* shaderów. Jeśli chcesz wykorzystać kanwę jako wyjście awaryjne dla WebGL, musisz wziąć to pod uwagę.Więcej na ten temat piszę w rozdziale 7.

Nauka posługiwania się shaderami zajmuje trochę czasu, bo trzeba nauczyć się nowych pojęć i języka programowania, oraz wymaga cierpliwości. Jeśli Cię to przytłoczy, nie martw się. Istnieje wiele otwartych bibliotek narzędzi, dzięki którym można ukryć wewnętrzne mechanizmy działania shaderów. Pracując jako programista grafiki trójwymiarowej, możesz nigdy nie napisać nawet jednego wiersza kodu GLSL; jednak zalecam przynajmniej podjęcie próby, choćby po to, bo potem można było powiedzieć, że się to robiło.

Znasz już podstawowe pojęcia dotyczące programowania grafiki trójwymiarowej. Każda z opisanych w tej książce technologii różni się od pozostałych pewnymi szczegółami, ale ogólne koncepcje są uniwersalne. W kilku następnych rozdziałach będziesz mógł dokładniej poznać tworzenie animowanej treści przy użyciu WebGL, CSS3 oraz Canvas 2D.

Renderowanie grafiki trójwymiarowej na bieżąco przy użyciu biblioteki WebGL

WebGL to standardowe API grafiki trójwymiarowej dla internetu. Przy jego użyciu i zastosowaniu języka JavaScript można w pełni wykorzystać moc sprzętu renderującego grafikę w przeglądarce internetowej. Przed powstaniem tej biblioteki programista, który chciał zaprezentować na stronie internetowej trójwymiarową grafikę renderowaną z wykorzystaniem akceleratora, musiał używać dodatkowych wtyczek lub pisać aplikacje macierzyste oraz prosić użytkowników o pobranie i zainstalowanie dodatkowych programów.

Biblioteka WebGL nie należy do oficjalnej specyfikacji HTML5, ale jest obsługiwana przez większość przeglądarek stosujących tę technologię. WebGL, podobnie jak Web Workers, WebSockets i inne technologie nieobjęte oficjalnymi rekommendacjami W3C, należy do ogólnego pakietu. Programiści z takich firm jak Google, Apple, Mozilla, Microsoft, Amazon, Opera, Intel i BlackBerry uważają, że obsługa grafiki trójwymiarowej jest podstawowym warunkiem do tego, by przeglądarki internetowe stały się poważną platformą dla aplikacji.

WebGL działa w większości przeglądarek komputerów stacjonarnych i w prawie wszystkich przeglądarkach w urządzeniach przenośnych¹. Na świecie istnieją miliony gotowych na obsługę WebGL komputerów, wśród których najprawdopodobniej znajduje się też Twoja maszyna. Ponadto ciągle tworzy się nowe witryny internetowe zawierające gry, wizualizacje, projekty komputerowe itp.

WebGL to niskopoziomowe API graficzne. Przekazuje się do niego tablice danych i shader, aby coś narysować. Programiści przyzwyczajeni do API graficznych w rodzaju Canvas 2D od czują brak wysokopoziomowych konstrukcji, ale powstało kilka otwartych zestawów narzędzi JavaScript dających dostęp do API na wyższym poziomie, dzięki czemu można z nim pracować w sposób zbliżony do tego, w jaki pracuje się ze zwykłymi bibliotekami graficznymi. Jednak nawet wtedy, gdy używa się pomocnych narzędzi, programowanie grafiki to i tak trudna praca, chociaż dzięki nim zawsze łatwiej wdrożyć się niedoświadczonym programistom. A doświadczeni programiści, korzystając z takich dodatków, potrafią zaoszczędzić mnóstwo czasu.

¹ W czasie pisania tej książki jedynym wyjątkiem w obsłudze WebGL w przeglądarkach na urządzenia przenośne była przeglądarka Mobile Safari dla systemu iOS. To duży problem, ale — na szczęście — istnieją dodatki, za pomocą których można go obejść. Więcej na ten temat piszę w rozdziale 12.

W tym rozdziale poznasz podstawy działania biblioteki WebGL. W większości przykładów opisywanych w tej książce używane będą dodatkowe narzędzia ukrywające większość szczegółów tego API, ale należy wiedzieć, na jakiej bazie działają. Czas na poznanie podstawowych pojęć i API.



Podobnie jak wielu innych nowości HTML5, Twój komputer może nie obsługiwać WebGL. Biblioteka ta jest obsługiwana przez większość najważniejszych przeglądarek na komputery stacjonarne, ale w przypadku niektórych dotyczy to tylko najnowszych wersji (np. Internet Explorer 11). Ponadto niektóre stare komputery nie zawierają odpowiedniego procesora graficznego do przetwarzania grafiki trójwymiarowej. Przeglądarki w nich działające automatycznie wyłączają WebGL. Jeżeli chcesz się dowiedzieć, czy Twoje docelowe komputery, urządzenia i przeglądarki obsługują WebGL, wejdź na stronę <http://caniuse.com/> i w polu wyszukiwania wpisz frazę **WebGL** albo przejdź bezpośrednio do testu WebGL na stronie <http://caniuse.com/#feat=webgl>.

Podstawy WebGL

Biblioteka WebGL powstała jako eksperymentalny projekt w 2006 r. pod okiem inżyniera z Mozilla, Vladimira Vukićevića. Vukićević chciał utworzyć API do rysowania grafiki trójwymiarowej dla elementu kanwy, który miałby być odpowiednikiem API Canvas 2D. Projekt swój, o nazwie **Canvas 3D**, słusznie oparł na bibliotece OpenGL ES, czyli standardowym API zyskującym coraz większą popularność wśród twórców grafiki dla urządzeń przenośnych. Do 2007 r. powstały niezależne implementacje Canvas 3D w przeglądarkach Mozilla i Opera.

W 2009 r. Vukićević podjął współpracę z ochronikami z Opery, Apple i Google, aby powołać grupę roboczą ds. WebGL w organizacji Khronos Group. Organizacja ta to ciało standaryzacyjne zajmujące się także takimi standardami jak OpenGL, COLLADA i jeszcze paroma innymi. Khronos zarządza projektem WebGL do dziś. Vukićević był pierwszym kierownikiem grupy roboczej do 2010 r., w którym rolę tę przejął Kenneth Russell z Google.

Oto oficjalny opis WebGL ze strony internetowej organizacji Khronos.

WebGL to darmowe, wieloplatformowe API umożliwiające wykorzystanie OpenGL ES 2.0 na stronach internetowych jako kontekstu rysunkowego dla grafiki trójwymiarowej w kodzie HTML, dostępne w postaci niskopoziomowych interfejsów DOM. Wykorzystuje język cieniowania OpenGL GLSL ES i może być używane w połączeniu z innymi rodzajami treści internetowej, którą można umieścić na warstwie grafiki trójwymiarowej lub pod tą warstwą. Najlepiej nadaje się do tworzenia dynamicznych aplikacji trójwymiarowych w języku programowania JavaScript oraz zostanie w pełni zintegrowane z najważniejszymi przeglądarkami internetowymi.

W definicji tej znajduje się kilka ważnych stwierdzeń.

- **WebGL to API** — dostęp do WebGL można uzyskać wyłącznie za pomocą zestawu interfejsów programistycznych języka JavaScript. Nie używa się żadnych dodatkowych znaczników HTML. Renderowanie grafiki trójwymiarowej z wykorzystaniem WebGL, podobnie jak rysowanie grafiki dwuwymiarowej na kanwie, odbywa się poprzez wywołania API JavaScript. W istocie dostęp do WebGL odbywa się poprzez istniejący element kanwy i specjalny kontekst rysunkowy.

- **WebGL bazuje na OpenGL ES 2.0** — OpenGL ES to adaptacja stabilnego standardu renderowania grafiki trójwymiarowej o nazwie OpenGL. Przyrostek **ES** to skrót od słów *embedded systems* (układy wbudowane) oznaczający, że jest to biblioteka przeznaczona do użytku w niewielkich urządzeniach komputerowych, głównie telefonach i tabletach. API OpenGL ES jest wykorzystywane do obsługi grafiki trójwymiarowej w iPhone'ach, iPadaach oraz Androidzie. Projektanci WebGL uznali, że niewielkie zapotrzebowanie na pamięć biblioteki OpenGL ES ułatwi im utworzenie spójnego, wieloplatformowego API działającego we wszystkich przeglądarkach internetowych.
- **WebGL można łączyć z innymi rodzajami treści internetowej** — WebGL działa na pozostałej treści strony internetowej lub pod nią. Kanwa trójwymiarowa może zająć tylko fragment strony albo cały dokument. Może być umieszczona w elementach `<div>` ułożonych w stos. Oznacza to, że grafikę trójwymiarową tworzy się przy użyciu WebGL, ale wszystkie pozostałe elementy strony buduje się w taki sam sposób, jak zawsze przy użyciu języka HTML. Przeglądarki składają wszystkie znajdujące się na stronie grafiki w jedną całość.
- **WebGL służy do budowy dynamicznych aplikacji sieciowych** — biblioteka ta powstała z myślą o internecie. Pochodzi od OpenGL ES, ale ma specjalne właściwości, które dobrze integrują się z przeglądarkami internetowymi, współpracują z JavaScriptem oraz ułatwiają przesyłanie treści przez internet.
- **WebGL jest wieloplatformowa** — biblioteka WebGL może działać w każdym systemie operacyjnym, we wszystkich urządzeniach od telefonów, przez tablety, po komputery stacjonarne.
- **WebGL jest darmowa** — podobnie jak w przypadku wszystkich innych otwartych specyfikacji internetowych, korzystanie z biblioteki WebGL jest bezpłatne. Za posługiwanie się nią nie trzeba płacić żadnych tantiem.

Twórcy przeglądarek Chrome, Firefox, Safari oraz Opera włożyli wiele wysiłku we wsparcie rozwoju biblioteki WebGL, a programiści z ich zespołów pełnią kluczowe role w grupie robocej ds. tej specyfikacji. Proces rozwijania specyfikacji WebGL jest otwarty dla wszystkich członków grupy Khronos. Ponadto istnieją listy mailingowe dostępne dla wszystkich zainteresowanych. W dodatku A znajduje się wykaz takich list oraz innych źródeł wiadomości na temat specyfikacji.

API WebGL

WebGL bazuje na stabilnym API do programowania grafiki o nazwie OpenGL. Biblioteka OpenGL powstała pod koniec lat 80. ubiegłego wieku i na wiele lat stała się powszechnie obowiązującym standardem. Wytrzymała konkurencję z silnym rywalem, jakim jest biblioteka DirectX Microsoftu, i stała się niekwestionowanym liderem w dziedzinie programowania grafiki trójwymiarowej.

Jednak nie wszystkie biblioteki OpenGL są takie same. Różnice między platformami — takimi jak komputery stacjonarne, dekodery telewizyjne, smartfony i tablety — są tak duże, że konieczne było utworzenie kilku wersji OpenGL. Biblioteka OpenGL ES to wersja przeznaczona do użytku w niewielkich urządzeniach, takich jak dekodery i smartfony. Twórcy tej wersji biblioteki pewnie się nie spodziewali, że ich produkt będzie się idealnie nadawał jako podstawa do budowy biblioteki WebGL. Biblioteka ta jest niewielka i zwięzła, dzięki czemu jest

względnie łatwa do zaimplementowania w przeglądarce oraz można się spodziewać, że programiści różnych przeglądarek zaimplementują ją w podobny sposób w swoich produktach, a to z kolei sprawi, że raz napisaną aplikację będzie można uruchomić bez modyfikowania w każdej przeglądarce.

Zwięzłość biblioteki WebGL sprawia, że więcej pracy mają twórcy aplikacji. Scena trójwymiarowa nie ma reprezentacji w DOM, nie istnieją specjalne formaty plików do ładowania geometrii i animacji oraz (nie licząc kilku niskopoziomowych zdarzeń) nie ma wbudowanego modelu zdarzeń do raportowania tego, co się dzieje na kanwie (np. brak zdarzeń kliknięcia myszą informujących, który obiekt został kliknięty). Dla przeciętnego programisty aplikacji nauka programowania przy użyciu WebGL jest dość trudna i wymaga zapoznania się z wieloma dziwnymi pojęciami.

Dobra wiadomość jest taka, że istnieje kilka otwartych bibliotek ułatwiających używanie biblioteki WebGL. Można je traktować jako coś podobnego do jQuery albo Prototype.js, chociaż może nie jest to najlepsze porównanie. O tych bibliotekach również będzie mowa w kolejnych rozdziałach. Na razie jednak omówię podstawy działania WebGL. Jeśli nawet nigdy nie będziesz pisać niskopoziomowego kodu w swoich projektach, warto wiedzieć, co naprawdę dzieje się w naszych programach.

Anatomia aplikacji WebGL

WebGL to po prostu biblioteka graficzna — kolejny rodzaj kanwy, podobny do kanwy dwuwymiarowej obsługiwanej przez wszystkie przeglądarki pracujące z HTML5. W istocie WebGL wykorzystuje kanwę HTML5 do prezentowania trójwymiarowej grafiki na stronie internetowej.

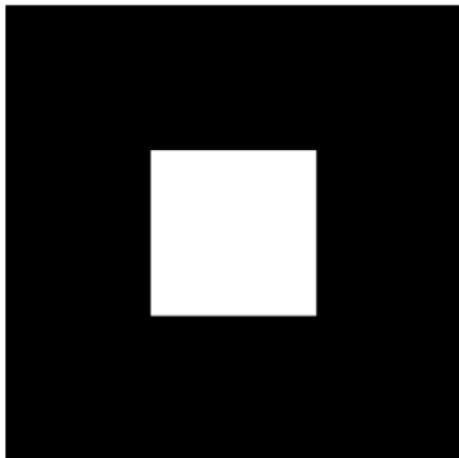
Aby wyrenderować treść WebGL na stronie, aplikacja musi wykonać przynajmniej takie czynności jak:

1. utworzenie elementu kanwy,
2. uzyskanie dostępu do kontekstu rysunkowego dla kanwy,
3. inicjacja obszaru widoku,
4. utworzenie przynajmniej jednego bufora na dane do wyrenderowania (najczęściej wierzchołki),
5. utworzenie przynajmniej jednej macierzy definiującej rzutowanie zawartości buforów wierzchołków na przestrzeń ekranową,
6. utworzenie przynajmniej jednego shadera implementującego algorytm rysujący,
7. inicjacja shaderów z parametrami,
8. rysowanie.

Przeanalizuję ten proces na kilku przykładach.

Prosty przykład użycia WebGL

Podstawy działania API WebGL przedstawię na przykładzie prostego programu rysującego na kanwie biały prostokąt. Cały kod znajduje się w pliku *r02/example2-1.html*, a efekt jego działania widać na rysunku 2.1.



Rysunek 2.1. Prostokąt narysowany przy użyciu biblioteki WebGL



Przedstawione w tej części rozdziału przykłady zostały zainspirowane lekcjami z kursu Learning WebGL (<http://learningwebgl.com>), doskonałego serwisu internetowego, którego oryginalnym twórcą jest Giles Thomas (<http://gilesthomas.com/>). Kurs ten jest fantastycznym sposobem na naukę obsługi API WebGL. Ponadto w serwisie można znaleźć tygodniowe zestawienie aplikacji WebGL, a więc warto do niego zaglądać, by na bieżąco śledzić nowości w branży.

Kanwa i kontekst rysunkowy WebGL

Renderowanie treści WebGL zawsze odbywa się w określonym **kontekście**, obiekcie DOM przeglądarki dostarczającym kompletne API WebGL. Struktura ta jest odpowiednikiem dwuwymiarowego kontekstu rysunkowego dostarczanego w kanwie HTML5. Aby umieścić treść WebGL na stronie internetowej, należy utworzyć element <canvas>, uzyskać dostęp do reprezentującego go obiektu DOM (np. za pomocą funkcji `document.getElementById()`), a następnie uzyskać dostęp do jego kontekstu WebGL.

Na listingu 2.1 znajduje się przykład ilustrujący sposób uzyskiwania dostępu do kontekstu WebGL z elementu kanwy DOM. Metoda `getContext()` przyjmuje jeden z następujących identyfikatorów określających kontekst: `2d` (kontekst dwuwymiarowy opisany w rozdziale 7.), `webgl` (kontekst WebGL) oraz `experimental-webgl` (kontekst WebGL dla starszych przeglądarek). Kontekst `experimental-webgl` jest obsługiwany także przez najnowsze przeglądarki, które pracują już z `webgl`, a więc będziemy go używać, aby nasze programy działały w jak największej liczbie przeglądarek.

Listing 2.1. Uzyskiwanie dostępu do kontekstu WebGL w kanwie

```
function initWebGL(canvas) {  
  
    var gl = null;  
    var msg = "Twoja przeglądarka nie obsługuje WebGL, " +  
        "albo obsługa WebGL jest w niej wyłączona.";
```

```

try
{
    gl = canvas.getContext("experimental-webgl");
}
catch (e)
{
    msg = "Błąd tworzenia kontekstu WebGL!: " + e.toString();
}

if (!gl)
{
    alert(msg);
    throw new Error(msg);
}

return gl;
}

```



Trzeba tu zwrócić uwagę na blok try-catch. Jest bardzo ważny, ponieważ niektóre przeglądarki wciąż nie obsługują WebGL, a nawet jeśli obsługują, nie każdy użytkownik ma zainstalowaną najnowszą wersję programu. Ponadto nawet jeśli przeglądarka jest zaktualizowana, przestarzały może być sprzęt, przez co utworzenie poprawnego kontekstu WebGL również będzie niemożliwe. Dlatego kod sprawdzający obsługę biblioteki umożliwia dostarczenie awaryjnego rozwiązania, np. kanwy dwuwymiarowej, albo przynajmniej wyświetlenie informacji i eleganckie zamknięcie programu.

Obszar widoku

Po uzyskaniu poprawnego kontekstu rysunkowego WebGL z kanwy należy określić prostokątny obszar, w którym odbywa się będzie rysowanie. W WebGL obszar taki nazywa się **obszarem widoku** (ang. *viewport*). Jego utworzenie jest łatwe, wystarczy wywołać metodę `viewport()` kontekstu, co pokazano na listingu 2.2.

Listing 2.2. Ustawianie obszaru widoku WebGL

```

function initViewport(gl, canvas)
{
    gl.viewport(0, 0, canvas.width, canvas.height);
}

```

Przypomnę że użyty tu obiekt `gl` został utworzony przez naszą funkcję pomocniczą `initWebGL()`. W tym przypadku zainicjalizujemy obszar widoku WebGL obejmujący cały obszar kanwy.

Bufory, bufory tablicowe i tablice typowane

Mamy już gotowy kontekst do rysowania i na tym kończą się podobieństwa do kanwy dwuwymiarowej.

W WebGL do rysowania używa się **obiektów podstawowych** (ang. *primitives*) — różnych typów obiektów, które można rysować. Zaliczają się do nich trójkąty, punkty i linie. Najczęściej korzysta się z trójkątów, które są dostępne w dwóch formach: jako zestawy trójkątów (tablice trójkątów) i jako pasy trójkątów (ich opis znajduje się nieco dalej). Obiekty podstawowe korzystają z tablic danych zwanych **buforami**, które określają położenie wierzchołków do narysowania.

Na listingu 2.3 pokazano, jak utworzyć bufor wierzchołków dla prostokąta o wymiarach 1×1 . Wynik jest zwracany w obiekcie JavaScript zawierającym dane bufora wierzchołków, rozmiar struktury wierzchołków (w tym przypadku trzy liczby zmiennoprzecinkowe odpowiadające współrzednym x , y i z), liczbę wierzchołków do narysowania oraz typ obiektów podstawowych, które zostaną użyte do narysowania prostokąta — w tym przypadku jest to pas trójkątów. Pas trójkątów to podstawowy obiekt renderingu definiujący szereg trójkątów, z których pierwszy wyznacza trzy pierwsze wierzchołki, a pozostałe są tworzone przez jeden nowy wierzchołek i dwa poprzednie.

Listing 2.3. Tworzenie danych bufora wierzchołków

```
// Tworzy dane wierzchołków dla prostokąta, który ma zostać narysowany.
function createSquare(gl) {
    var vertexBuffer;
    vertexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
    var verts = [
        .5, .5, 0.0,
        -.5, .5, 0.0,
        .5, -.5, 0.0,
        -.5, -.5, 0.0
    ];
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(verts), gl.STATIC_DRAW);
    var square = {buffer:vertexBuffer, vertSize:3, nVerts:4,
        primtype:gl.TRIANGLE_STRIP};
    return square;
}
```

Warto zwrócić uwagę na użyty w tym kodzie typ `Float32Array`. Jest to nowy typ danych wprowadzony do przeglądarek internetowych dla WebGL. Typ `Float32Array` jest typem **bufora tablicowego** (ang. *ArrayBuffer*) zwanego również **tablicą typowaną** (ang. *typed array*). Jest to typ JavaScript służący do przechowywania kompaktowych danych binarnych. Tablic typowanych w tym języku można używać tak samo jak zwykłych tablic, ale są one znacznie szybsze i zajmują mniej pamięci. Dlatego doskonale nadają się do pracy z danymi binarnymi, gdy wydajność ma kluczowe znaczenie. Tablice typowane można wykorzystać do ogólnych zastosowań, ale zostały wprowadzone do przeglądarek ze względu na bibliotekę WebGL. Ich najnowszą specyfikację można znaleźć w witrynie Khronos (<http://www.khronos.org/registry/typedarray/specs/latest/>).

Macierze

Aby narysować prostokąt, należy utworzyć dwie macierze. Pierwsza będzie określać położenie figury w trójwymiarowym układzie współrzędnych względem kamery. Nazywa się ona **macierz model-widok** (ang. *ModelView matrix*), ponieważ tworzy kombinację przekształceń modelu (siatki trójwymiarowej) i kamery. W naszym przykładzie przekształcamy prostokąt, przesuwając go wzduż ujemnej części osi z (tj. przesuwamy go od kamery o -3,333 jednostki). Druga macierz to **macierz rzutowania** (ang. *projection matrix*) potrzebna shaderowi do przekształcenia trójwymiarowych współrzędnych modelu w przestrzeni kamery na współrzędne dwuwymiarowe rysowane w przestrzeni obszaru widoku. W tym przykładzie macierz rzutowania definiuje kamerę z perspektywą 45-stopniowego pola widzenia (odświeżenie wiadomości na temat rzutowania perspektywy znajduje się w rozdziale 1.).

W WebGL macierze są reprezentowane w postaci typowanych tablic liczb. Przykładowo macierz o wymiarach 4×4 jest reprezentowana jako obiekt typu `Float32Array` zawierający 16 elementów.

Do tworzenia macierzy i pracy z nimi używa się świetnej otwartej biblioteki o nazwie glMatrix (<https://github.com/toji/gl-matrix>) autorstwa Brandona Jonesa, aktualnie programisty Google. Kod tworzący macierz znajduje się na listingu 2.4. Macierze glMatrix są typu mat4 i tworzy się je za pomocą funkcji fabrycznej mat4.create(). Funkcja initMatrices() buduje macierze model-widok i rzutowania oraz zapisuje je w globalnych zmiennych o nazwach odpowiednio modelViewMatrix i projectionMatrix.

Listing 2.4. Tworzenie macierzy model-widok i rzutowania

```
var projectionMatrix, modelViewMatrix;

function initMatrices(canvas)
{
    // Tworzy macierz model-widok z kamerą w punkcie 0, 0, -3.333.
    modelViewMatrix = mat4.create();
    mat4.translate(modelViewMatrix, modelViewMatrix, [0, 0, -3.333]);

    // Tworzy macierz rzutowania z 45-stopniowym polem widzenia.
    projectionMatrix = mat4.create();
    mat4.perspective(projectionMatrix, Math.PI / 4,
                      canvas.width / canvas.height, 1, 10000);
}
```

Shader

Jesteśmy już prawie gotowi do wyrenderowania sceny, ale pozostał jeszcze jeden ważny składnik do wykonania; jest nim shader. Jak napisałem wcześniej, shadery to niewielkie programy w języku GLSL (wysokopoziomowym języku podobnym do C), definiujące sposób rysowania na ekranie pikseli składających się na trójwymiarowe obiekty. W WebGL każdy rysowany obiekt musi mieć przypisany shader. Jeden program cieniujący (shader) można wykorzystać do narysowania wielu obiektów, więc często dla całej aplikacji wystarczy napisać tylko jeden taki program oraz wykorzystać go wielokrotnie z różnymi ustawieniami geometrii oraz parametrami.

Typowy shader składa się z dwóch części: **shadera wierzchołków** i **shadera fragmentów** (zwanego również **shaderem pikseli**). Shader wierzchołków służy do przekształcania współrzędnych obiektu na przestrzeń dwuwymiarową. Natomiast shader fragmentów jest używany do generowania ostatecznego koloru każdego piksela dla przekształconych wierzchołków. W pracy wykorzystuje takie dane wejściowe jak kolor, tekstura, oświetlenie i materiał. W naszym prostym przykładzie shader wierzchołków wykorzystuje wartości vertexPos, modelViewMatrix oraz projectionMatrix, a shader fragmentów zwraca tylko ustawiony na sztywno kolor biały.

Tworzenie shadera w WebGL składa się z kilku etapów, takich jak kompilacja poszczególnych fragmentów kodu źródłowego GLSL, a następnie ich połączenie. Na listingu 2.5 przedstawiony jest kod źródłowy shadera. Przeanalizuję go szczegółowo. Najpierw zdefiniowano funkcję pomocniczą createShader(), która za pomocą metod WebGL kompiluje shadery wierzchołków i fragmentów z kodu źródłowego.

Listing 2.5. Kod shadera

```
function createShader(gl, str, type) {
    var shader;
    if (type == "fragment") {
        shader = gl.createShader(gl.FRAGMENT_SHADER);
    } else if (type == "vertex") {
        shader = gl.createShader(gl.VERTEX_SHADER);
```

```

    } else {
        return null;
    }

    gl.shaderSource(shader, str);
    gl.compileShader(shader);

    if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
        alert(gl.getShaderInfoLog(shader));
        return null;
    }

    return shader;
}

```

Kod źródłowy GLSL jest dostarczany w postaci łańcuchów JavaScript, które definiuje się jako globalne zmienne vertexShaderSource i fragmentShaderSource.

```

var vertexShaderSource =
    " attribute vec3 vertexPos;\n" +
    " uniform mat4 modelViewMatrix;\n" +
    " uniform mat4 projectionMatrix;\n" +
    " void main(void) {\n" +
    " //Zwraca przekształconą i rzutowaną wartość wierzchołka\n" +
    " gl_Position = projectionMatrix * modelViewMatrix * \n" +
    " vec4(vertexPos, 1.0);\n" +
    " }\n";
}

var fragmentShaderSource =
    " void main(void) {\n" +
    " //Zwraca kolor piksela: zawsze biały\n" +
    " gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);\n" +
    " }\n";

```



Kod źródłowy GLSL jest dostarczany w postaci łańcuchów JavaScript przechowywanych w zmiennych. Nie jest to najlegantniejsze rozwiązanie, bo trzeba łączyć łańcuchy podzielone na wiele wierszy. Innym rozwiązaniem jest zdefiniowanie shadera w zewnętrznych plikach tekstowych i ładowanie ich za pomocą Ajaksa. Można też utworzyć ukryte elementy DOM i wstawić kod źródłowy do ich atrybutów `textContent`. Rozwiązanie zastosowane w tym przykładzie wybrałem ze względu na prostotę. We własnym kodzie możesz wybrać jedno z elegantszych rozwiązań.

Skompilowane części shadera należy połączyć w jeden program za pomocą metod `gl.createProgram()`, `gl.attachShader()` oraz `gl.linkProgram()`. Później pozostaje do zrobienia jeszcze jedna rzecz: pozyskanie uchwytu do każdej zmiennej zdefiniowanej w kodzie GLSL, aby można było ją zainicjować wartościami z kodu JavaScript. Do tego służą metody WebGL `gl.getAttributeLocation()` i `gl.getUniformLocation()`. Poniżej znajduje się definicja funkcji `initShader()`.

```

var shaderProgram, shaderVertexPositionAttribute,
    shaderProjectionMatrixUniform,
    shaderModelViewMatrixUniform;

function initShader(gl) {
    //Zaladowanie i komplikacja shaderów fragmentów i wierzchołków.
    var fragmentShader = createShader(gl, fragmentShaderSource,
        "fragment");
    var vertexShader = createShader(gl, vertexShaderSource,
        "vertex");

```

```

// Połączenie ich w jeden program.
shaderProgram = gl.createProgram();
gl.attachShader(shaderProgram, vertexShader);
gl.attachShader(shaderProgram, fragmentShader);
gl.linkProgram(shaderProgram);

// Pobranie wskaźników do parametrów shadera.
shaderVertexPositionAttribute =
    gl.getAttribLocation(shaderProgram, "vertexPos");
gl.enableVertexAttribArray(shaderVertexPositionAttribute);

shaderProjectionMatrixUniform =
    gl.getUniformLocation(shaderProgram, "projectionMatrix");
shaderModelViewMatrixUniform =
    gl.getUniformLocation(shaderProgram, "modelViewMatrix");

if (!gl.getProgramParameter(shaderProgram,
    gl.LINK_STATUS)) {
    alert("Nie można zainicjować shaderów");
}
}

```

Rysowanie obiektów podstawowych

Teraz możemy narysować nasz prostokąt. Kontekst został utworzony, obszar widoku ustawiony, bufor wierzchołków, macierze i shadery zostały utworzone i zainicjowane. Definiujemy więc funkcję `draw()`, która pobiera kontekst WebGL i utworzony wcześniej obiekt prostokąta. Przeanalizuj teraz jej budowę.

Funkcja `draw()` zaczyna od oczyszczania kanwy i nałożenia na nią czarnego tła. Metoda `gl.clearColor()` ustawia kolor na czarny. Przyjmuje ona wartość koloru w formacie RGBA (czerwony, zielony, niebieski, alfa). Pamiętaj, że wartości RGBA WebGL muszą być liczbami zmiennoprzecinkowymi z przedziału od 0.0 do 1.0 (w odróżnieniu od całkowitoliczbowego zakresu od 0 do 255 używanego np. w CSS). Następnie funkcja `gl.clear()` przy użyciu zdefiniowanego koloru czyści zawartość **bufora kolorów** WebGL, czyli obszaru w pamięci GPU, który jest używany do renderowania bitów na ekranie. (W WebGL można skorzystać z kilku typów **buforów** do rysowania, wśród których znajdują się bufor kolorów oraz **bufor głębi** do sprawdzania głębi — więcej o nim piszę w następnym podrozdziale).

Następnie funkcja `draw()` ustawia (**wiąże** — bind) bufor wierzchołków dla prostokąta, który ma zostać narysowany, ustawia (**używa** — use) shader, który ma zostać wykonany w celu narysowania obiektu, oraz łączy bufor wierzchołków i macierze z shaderem, przekazując je do niego na wejściu. Na koniec wywoływana jest metoda WebGL `drawArrays()` w celu narysowania prostokąta. Funkcji tej podawany jest tylko typ obiektu do narysowania oraz liczba jego wierzchołków. Wszystkie pozostałe informacje WebGL już ma, bo zostały wcześniej ustawione (wierzchołki, macierze, shadery) jako stan w kontekście. Cały kod jest pokazany na listingu 2.6.

Listing 2.6. Kod rysujący

```

function draw(gl, obj) {

    // Wyczyszczenie tła (na czarno).
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    gl.clear(gl.COLOR_BUFFER_BIT);

    // Ustawienie bufora wierzchołków do narysowania.
    gl.bindBuffer(gl.ARRAY_BUFFER, obj.buffer);
}

```

```

// Ustawienie shadera, który ma zostać użyty.
gl.useProgram(shaderProgram);

// Doliczenie parametrów shadera: macierze pozycji wierzchołków oraz rzutowania/modelu.
gl.vertexAttribPointer(shaderVertexPositionAttribute,
    obj.vertSize, gl.FLOAT, false, 0, 0);
gl.uniformMatrix4fv(shaderProjectionMatrixUniform, false,
    projectionMatrix);
gl.uniformMatrix4fv(shaderModelViewMatrixUniform, false,
    modelViewMatrix);

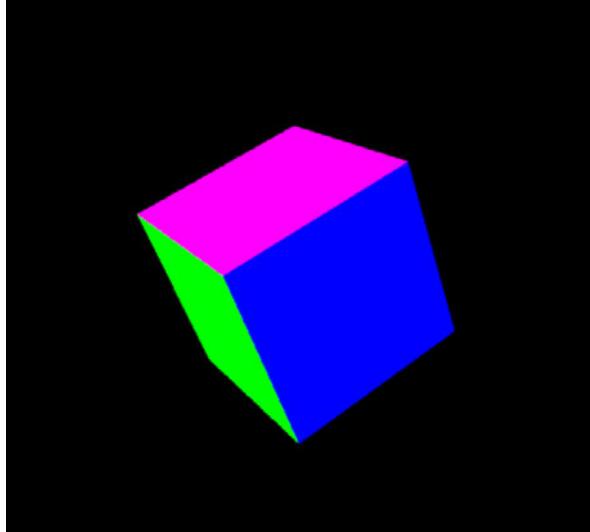
// Rysowanie obiektu.
gl.drawArrays(obj.primtype, 0, obj.nVerts);
}

```

To wszystko. W efekcie otrzymamy biały prostokąt na czarnym tle widoczny na rysunku 2.1.

Tworzenie brył

Prostokąt to najprostszy możliwy przykład wykorzystania biblioteki WebGL. Oczywiście, nie jest on porywający, nawet nietrójwymiarowy, ale do jego napisania trzeba było aż 200 wierszy kodu. Analogiczny przykład rysujący na kanwie dwuwymiarowej składałby się co najwyżej z około 30 wierszy kodu. Nie ma wątpliwości, że biblioteka WebGL nie jest idealna do rysowania obiektów w dwóch wymiarach i są lepsze od niej rozwiązania w tej dziedzinie. Teraz przejdziemy do czegoś ciekawszego. Narysujemy prawdziwy trójwymiarowy obiekt. Będziemy potrzebować trochę dodatkowego kodu, aby utworzyć geometrię trójwymiarowej kolorowej kostki, oraz zmodyfikujemy nieco shader i funkcję rysującą. Dodatkowo dorzucimy jeszcze prostą animację, aby oglądać sześcian ze wszystkich stron. Na rysunku 2.2 można zobaczyć efekt, jaki staramy się uzyskać.



Rysunek 2.2. Kolorowy sześcian

Aby utworzyć i wyrenderować trójwymiarową kostkę, musimy w kilku miejscach zmienić poprzedni program. Najpierw zmienimy kod tworzący bufory w taki sposób, by zamiast geometrii prostokąta tworzył geometrię sześcianu. Potem w kodzie rysującym użyjemy innej metody rysowania WebGL. Cały opisywany kod umieszczony został w pliku *r02/example2-2.html*.

Na listingu 2.7 znajduje się kod konfiguracji bufora. Jest on nieco bardziej skomplikowany od poprzedniego przykładu i ma to związek nie tylko z większą liczbą wierzchołków, ale również z tym, że każdy bok naszej bryły ma mieć inny kolor. Zaczniemy od utworzenia danych bufora wierzchołków, które zapiszemy w zmiennej *vertexBuffer*.

Listing 2.7. Konfiguracja buforów geometrii, kolorów oraz indeksów

```
// Tworzy dane wierzchołków, kolorów oraz indeksów dla kolorowej kostki.
function createCube(gl) {

    // Dane wierzchołków.
    var vertexBuffer;
    vertexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
    var verts = [
        // Przód.
        -1.0, -1.0, 1.0,
        1.0, -1.0, 1.0,
        1.0, 1.0, 1.0,
        -1.0, 1.0, 1.0,

        // Tył.
        -1.0, -1.0, -1.0,
        -1.0, 1.0, -1.0,
        1.0, 1.0, -1.0,
        1.0, -1.0, -1.0,

        // Góra.
        -1.0, 1.0, -1.0,
        -1.0, 1.0, 1.0,
        1.0, 1.0, 1.0,
        1.0, 1.0, -1.0,

        // Dół.
        -1.0, -1.0, -1.0,
        1.0, -1.0, -1.0,
        1.0, -1.0, 1.0,
        -1.0, -1.0, 1.0,

        // Prawa.
        1.0, -1.0, -1.0,
        1.0, 1.0, -1.0,
        1.0, 1.0, 1.0,
        1.0, -1.0, 1.0,

        // Lewa.
        -1.0, -1.0, -1.0,
        -1.0, -1.0, 1.0,
        -1.0, 1.0, 1.0,
        -1.0, 1.0, -1.0
    ];
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(verts), gl.STATIC_DRAW);
}
```

Następnie tworzymy dane kolorów, po jednej czteroskładnikowej wartości dla każdego wierzchołka, które zapiszemy w zmiennej colorBuffer. Wartości kolorów przechowywane w tablicy faceColors są w formacie RGBA.

```
// Dane kolorów.  
var colorBuffer = gl.createBuffer();  
gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);  
var faceColors = [  
    [1.0, 0.0, 0.0, 1.0], //Przód.  
    [0.0, 1.0, 0.0, 1.0], //Tyl.  
    [0.0, 0.0, 1.0, 1.0], //Góra.  
    [1.0, 1.0, 0.0, 1.0], //Dół.  
    [1.0, 0.0, 1.0, 1.0], //Prawa.  
    [0.0, 1.0, 1.0, 1.0] //Lewa.  
];  
var vertexColors = [];  
for (var i in faceColors) {  
    var color = faceColors[i];  
    for (var j=0; j < 4; j++) {  
        vertexColors = vertexColors.concat(color);  
    }  
}  
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertexColors),  
    gl.STATIC_DRAW);
```

I na koniec utworzymy nowy rodzaj bufora, tzw. **bufor indeksów**, do przechowywania indeksów do danych znajdujących się w buforze wierzchołków. Zapiszemy go pod nazwą zmiennej cubeIndexBuffer. Robimy to, ponieważ metoda rysowania, której użyjemy w naszej nowej funkcji draw(), wymaga do zdefiniowania trójkątów indeksów do zbioru wierzchołków, a nie samych wierzchołków. Dlaczego? Geometria trójwymiarowa często składa się z ciągłych zamkniętych obszarów, w których wiele trójkątów wykorzystuje te same wierzchołki, i bufore indeksów pozwalają na bardziej kompaktowe przechowywanie danych, bo eliminują powtórzenia.

```
// Dane indeksów (definiują trójkąty, które mają zostać narysowane).  
var cubeIndexBuffer = gl.createBuffer();  
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, cubeIndexBuffer);  
var cubeIndices = [  
    0, 1, 2, 0, 2, 3,      //Przód.  
    4, 5, 6, 4, 6, 7,      //Tyl.  
    8, 9, 10, 8, 10, 11,   //Góra.  
    12, 13, 14, 12, 14, 15, //Dół.  
    16, 17, 18, 16, 18, 19, //Prawa.  
    20, 21, 22, 20, 22, 23 //Lewa.  
];  
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(cubeIndices),  
    gl.STATIC_DRAW);  
  
var cube = {buffer:vertexBuffer, colorBuffer:colorBuffer,  
    indices:cubeIndexBuffer,  
    vertSize:3, nVerts:24, colorSize:4, nColors: 24, nIndices:36,  
    primtype:gl.TRIANGLES};  
return cube;
```

Aby kolory kostki zostały narysowane, należy przekazać je do shadera. Na listingu 2.8 znajduje się zmodyfikowany kod shadera kolorów. Trzeba zwrócić uwagę na pogrubione wiersze, w których deklarowany jest nowy atrybut wierzchołków reprezentujący kolor. Ponadto musimy zadeklarować zmienną GLSL varying vColor, służącą do przekazywania kolorów wierzchołków z shadera wierzchołków do shadera fragmentów. W odróżnieniu od typów uniform, takich jak

wcześniej opisane macierze, typy varying reprezentują informacje, dla których shader dla każdego wierzchołka może zwrócić inną wartość. W tym przypadku będziemy pobierać dane kolorów z bufora zapisanego w atrybutie vertexColor. Shader fragmentów używa zmiennej vColor bez zmian do wysyłania ostatecznych wartości kolorów pikseli.

Listing 2.8. Kod shadera renderującego kostkę z kolorami

```
var vertexShaderSource =  
"    attribute vec3 vertexPos;\n" +  
"    attribute vec4 vertexColor;\n" +  
"    uniform mat4 modelViewMatrix;\n" +  
"    uniform mat4 projectionMatrix;\n" +  
"    varying vec4 vColor;\n" +  
"    void main(void) {\n" +  
"        // Zwraca przekształconą i rzutowaną wartość wierzchołka\n" +  
"        gl_Position = projectionMatrix * modelViewMatrix * \n" +  
"                      vec4(vertexPos, 1.0);\n" +  
"        // Wysyła vertexColor do vColor\n" +  
"        vColor = vertexColor;\n" +  
"    }\n";  
  
var fragmentShaderSource =  
"    precision mediump float;\n" +  
"    varying vec4 vColor;\n" +  
"    void main(void) {\n" +  
"        // Zwraca kolor piksela: zawsze biały\n" +  
"        gl_FragColor = vColor;\n" +  
"    }\n";
```



Ten kod, służący do ustawienia jednej wartości koloru, może wydawać się dość skomplikowany. Jednak w bardziej zaawansowanym shaderze — np. implementującym model oświetlenia albo animującym proceduralną teksturę trawy lub wody, albo jeszcze jakieś inne efekty — na wartości vColor wykonywanych byłoby jeszcze wiele innych obliczeń. Shadery dają programiście bardzo duże możliwości, ale — jak słusznie zauważał Ben Parker — wielka siła to także wielka odpowiedzialność.

Przechodzimy do przedstawionego na listingu 2.9 kodu rysującego. Musimy zmienić kilka rzeczy w porównaniu z poprzednim kodem, ponieważ teraz zajmujemy się bardziej skomplikowaną geometrią kostki. Zmiany są zaznaczone pogrubieniem. Najpierw informujemy WebGL o tym, że chcemy narysować trójwymiarowe obiekty posegregowane według głębi i w tym celu włączamy testowanie głębi. Gdybyśmy tego nie zrobili, nie byłoby gwarancji, że WebGL narysuje boki mające znajdować się „z przodu” w taki sposób, aby zasłaniały boki znajdujące się „z tyłu”. (Jeśli chcesz zobaczyć, co się stanie bez testowania głębi, wyłącz ten wiersz kodu za pomocą komentarza. Niektóre boki sześcianu nadal będą widoczne, ale nie wszystkie).

Listing 2.9. Zmieniony kod rysujący sześcian

```
function draw(gl, obj) {  
  
    // Wyczyszczenie tła na czarno.  
    gl.clearColor(0.0, 0.0, 0.0, 1.0);  
    gl.enable(gl.DEPTH_TEST);  
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);  
  
    // Ustawienie shadera.  
    gl.useProgram(shaderProgram);
```

```

// Podłączenie parametrów shadera: położenie wierzchołków, kolory oraz macierze rzutowania i modelu.
// Ustawienie buforów.
gl.bindBuffer(gl.ARRAY_BUFFER, obj.buffer);
gl.vertexAttribPointer(shaderVertexPositionAttribute,
    obj.vertSize, gl.FLOAT, false, 0, 0);
gl.bindBuffer(gl.ARRAY_BUFFER, obj.colorBuffer);
gl.vertexAttribPointer(shaderVertexColorAttribute,
    obj.colorSize, gl.FLOAT, false, 0, 0);
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, obj.indices);

gl.uniformMatrix4fv(shaderProjectionMatrixUniform, false,
    projectionMatrix);
gl.uniformMatrix4fv(shaderModelViewMatrixUniform, false,
    modelViewMatrix);

// Rysuje obiekt.
gl.drawElements(obj.primtype, obj.nIndices, gl.UNSIGNED_SHORT, 0);
}

```

Następnie musimy wykonać powiązanie utworzonych wcześniej w funkcji `createCube()` buforów kolorów i indeksów. Na koniec zamiast `gl.drawArray()` użyjemy metody WebGL o nazwie `gl.drawElements()`, która rysuje zbiór obiektów podstawowych przy użyciu informacji z bufora indeksów.

Animacja

Jeśli chcemy, aby nasza kostka wyglądała jak trójwymiarowy obiekt, a nie jak statyczny dwuwymiarowy rysunek, musimy ją animować. Na początek zastosujemy bardzo prostą technikę animacji polegającą na obracaniu obiektu wokół jednej osi. Odpowiedni kod jest pokazany na listingu 2.10. Funkcja `animate()` obraca kostkę wokół wcześniej zdefiniowanej osi `rotationAxis` w czasie pięciu sekund.

Listing 2.10. Animacja kostki

```

var duration = 5000; //ms
var currentTime = Date.now();
function animate() {
    var now = Date.now();
    var deltat = now - currentTime;
    currentTime = now;
    var fract = deltat / duration;
    var angle = Math.PI * 2 * fract;
    mat4.rotate(modelViewMatrix, modelViewMatrix, angle, rotationAxis);
}

function run(gl, cube) {
    requestAnimationFrame(function() { run(gl, cube); });
    draw(gl, cube);
    animate();
}

```

Funkcja `animate()` jest wywoływana bez przerwy przez funkcję `run()`, która steruje ciągłą animacją sceny przy użyciu nowej funkcji przeglądarkowej o nazwie `requestAnimationFrame()`. Funkcja ta prosi przeglądarkę o wywołanie funkcji zwrotnej, gdy przychodzi czas na ponowne narysowanie zawartości strony. (W kolejnych rozdziałach znajduje się szczegółowy opis funkcji `requestAnimationFrame()` i różnych technik animacji). Po każdym wywołaniu funkcja

`animate()` zapisuje różnicę między bieżącym czasem a poprzednim czasem jej wywołania w zmiennej `deltaT` i na podstawie tego wyniku oblicza kąt obrotu `modelViewMatrix`. Efektem tych działań jest pełny obrót wokół osi `rotationAxis` w czasie pięciu sekund.

Mapy tekstur

Na końcu tego rozdziału wyjaśnię jeszcze, na czym polega w WebGL mapowanie tekstur. **Mapa tekstury** albo po prostu **tekstura** to mapa bitowa wyświetlna na powierzchni obiektu geometrycznego. Dane graficzne tekstury tworzy się przy użyciu elementu DOM `Image`, co oznacza, że można używać standardowych formatów grafiki stosowanych w internecie, takich jak JPEG i PNG, poprzez ustawienie właściwości `src` elementu `Image`.



Tekstury WebGL nie muszą być tworzone z plików graficznych. Można też budować je przy użyciu dwuwymiarowej kanwy, a więc programista może rysować na powierzchni obiektów, korzystając z dwuwymiarowego API kanwy. Można nawet używać elementów wideo, aby odtwarzać filmy na powierzchni obiektów. Więcej na temat zaawansowanych możliwości teksturowania piszę w rozdziale 11.

Zmieniłem kod obracającej się kostki tak, aby zamiast jednolitego koloru na bokach wyświetlić teksturę. Efekt tych działań pokazano na rysunku 2.3.



Rysunek 2.3. Kostka z tekstem

Cały kod źródłowy omawianego przykładu umieszczony został w pliku `r2/example2-3.html`. Na lisingu 2.11 widać kod ładujący teksturę. Najpierw wywołujemy funkcję `gl.createTexture()` w celu utworzenia nowego obiektu tekstury WebGL. Następnie ustawiamy właściwość `image` tej tekstury na nowo utworzony obiekt `Image`. Na końcu ustawiamy właściwość `src` obrazu na plik JPEG — w tym przypadku jest to 156-pikselowa kwadratowa wersja oficjalnego logo WebGL — ale wcześniej rejestrujemy procedurę obsługi zdarzeń dla zdarzenia `onload` obrazu. Robimy to, ponieważ po załadowaniu obrazu chcemy jeszcze coś zrobić z obiektem tekstury.



Krótkie wyjaśnienie na temat tego przykładu dla tych, którzy próbowali go uruchomić, otwierając plik bezpośrednio w eksploratorze plików w systemie operacyjnym. Przykład ten musi być uruchamiany na serwerze sieciowym, ponieważ ładujemy w nim teksturę z pliku JPEG, a operacja taka — zgodnie z zabezpieczeniami WebGL dotyczącymi ograniczenia ładowania zasobów z różnych domen — musi być wykonywana na serwerze. Ogólnie większość przykładów opisanych w tej książce powinno się uruchamiać na serwerze.

W moim komputerze MacBook mam zainstalowany standardowy zestaw LAMP, ale wystarczy tylko serwer sieciowy, np. Apache. A jeśli masz zainstalowany interpreter języka Python, możesz użyć modułu SimpleHTTPServer, który powinieneś uruchomić, przechodząc do katalogu głównego *przykładów* i wykonując poniższe polecenie:

```
python -m SimpleHTTPServer
```

Następnie w przeglądarce wpisz adres **http://localhost:8000/**. Świeży artykuł na ten temat znajduje się też na stronie <http://bit.ly/linuxjournal-http-python>.

Listing 2.11. Tworzenie tekstury z obrazu graficznego

```
var okToRun = false;

function handleTextureLoaded(gl, texture) {
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
        texture.image);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
    gl.bindTexture(gl.TEXTURE_2D, null);
    okToRun = true;
}

var webGLTexture;

function initTexture(gl) {
    webGLTexture = gl.createTexture();
    webGLTexture.image = new Image();
    webGLTexture.image.onload = function () {
        handleTextureLoaded(gl, webGLTexture)
    }
}

webGLTexture.image.src = "../images/webgl-logo-256.jpg";
}
```

W metodzie zwrotnej `handleTextureLoaded()` wykonujemy kilka czynności. Najpierw za pomocą funkcji `gl.bindTexture()` informujemy WebGL, której tekstury będziemy używać w kolejnych wywołaniach API. Wszystkie wywołania API dotyczące tekstur będą odnosiły się do tej wybranej tekstury, dopóki ponownie nie wywołamy funkcji `gl.bindTexture()` — zrobimy to na końcu funkcji w celu ustalenia tekstury na `null`, aby później nie zmienić przypadkowo jej bitów.

Następnie wywołujemy funkcję `gl.pixelStorei()`, aby odwrócić wartości `y` wszystkich pikseli w teksturze, ponieważ w WebGL współrzędne teksturowe na osi `y` rosną do góry, a w sieciowych formatach graficznych wartości `y` pikseli rosną w dół.



Litera `i` w `gl.pixelStorei()` oznacza `integer` (liczba całkowita). Nazwy metod w WebGL zostały utworzone zgodnie z zasadami obowiązującymi w OpenGL, tzn. często dodaje się do nich przyrostek w postaci litery oznaczającej typ danych parametrów funkcji. Dane obrazów są przechowywane w postaci tablic wartości całkowitoliczbowych (kolory RGB lub RGBA) i stąd litera `i`.

Teraz możemy skopiować bity z załadowanego obrazu do obiektu tekstury WebGL. Posłuży do tego metoda `texImage2D()`, która ma kilka wersji sygnatury (szczegółowe informacje na temat sposobu jej użycia można znaleźć w specyfikacji WebGL). W tym przypadku tworzymy teksturę dwuwymiarową na poziomie zerowym — poziomy tekstur tworzy się w technice zwanej **mipmapowaniem**, której opis znajduje się dalej w tej książce — z formatem koloru RGBA i danymi źródłowymi w postaci tablicy bajtów bez znaku.

Ponadto musimy ustawić opcje filtrowania tekstury, które są parametrami decydującymi o sposobie obliczania przez WebGL kolorów pikseli w teksturze podczas zmieniania jej rozmiaru, gdy obraz jest przybliżany i oddalany. W naszym przykładzie użyliśmy najprostszej i najłatwiejszej do obliczenia opcji filtrowania o nazwie `gl.NEAREST`, która powoduje, że kolory pikseli są obliczane na podstawie skalowania oryginalnego obrazu. Przy tym ustawieniu tekstury wyglądają dobrze, dopóki nie zostaną zbyt mocno przeskalowane. Nadmierne przybliżenie (powiększenie) ujawnia bloki, a oddalenie (pomniejszenie) sprawia, że obraz przestaje być gładki. Ponadto w WebGL dostępne są jeszcze dwa inne filtry: `gl.LINEAR`, który liniowo interpoluje piksele w celu zapewnienia gładkości powiększonych tekstur, oraz `gl.LINEAR_MIPMAP_NEAREST`, który dodaje filtrowanie mipmapowe w celu wygładzenia tekstur pomniejszonych.

Aby na własne oczy zobaczyć wady filtrowania `gl.NEAREST`, pokombinuj z umiejscowieniem kostki. W wierszu 47. kodu źródłowego pliku `r2/example2-3.html` zmień współrzędną `z` położenia kostki, czyli `-8`, na inną wartość, aby przybliżyć obiekt do ekranu lub go od niego oddalić.

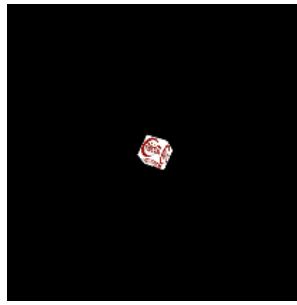
```
mat4.translate(modelViewMatrix, modelViewMatrix, [0, 0, -8]);
```

Sprawdź, co się stanie, gdy zamienisz `-4` na `-8`. Gdy kostka będzie bliżej, staną się widoczne poszczególne piksele (rysunek 2.4.).



Rysunek 2.4. Filtrowanie `gl.NEAREST`: w dużym przybliżeniu widać poszczególne piksele tekstury

A teraz zmień wartość `-8` na `-32`, aby znacznie oddalić kostkę od ekranu. Teraz krawędzie staną się poszarpane (rysunek 2.5.).



Rysunek 2.5. Filtrowanie gl.NEAREST: w dużym oddaleniu tekstury robią się niewyraźne

Po ustawieniu opcji tekstury odłączamy ją poprzez ustawienie null w funkcji gl.bindTexture(). Na koniec ustawiamy globalną zmienną okToRun na true, aby zasygnalizować funkcji run(), że mamy gotową teksturę i można wywołać kod rysujący.

Ponadto — jak zwykle — musimy zmienić jeszcze kilka innych fragmentów kodu: ten dotyczący tworzenia bufora, shader oraz część kodu rysującego napełniającą wartości shadera. Zaczniemy od zamiany kodu tworzącego bufor danych kolorów na kod, który będzie budował bufor **współrzędnych teksturowych**. Współrzędne te są parami liczb zmiennoprzecinkowych o wartościach zawierających się zazwyczaj w przedziale od 0 do 1. Reprezentują one miejsca na osiach x i y w danych mapy bitowej. Shader dane te wykorzysta do pobrania informacji o pikselach z mapy bitowej, co wkrótce zobaczymy, gdy zajmiemy się shaderem. Wartości współrzędnych teksturowych naszej kostki są nieskomplikowane: na każdy bok nakładana jest cała tekstura, więc wartości rogów wszystkich boków kostki pokrywają się z rogami tekstuury, np. [0, 0], [0, 1], [1, 0] lub [1, 1]. Pamiętaj, że kolejność tych wartości musi być taka sama jak kolejność wierzchołków w buforze wierzchołków. Na listingu 2.12 przedstawiony został kod tworzący bufor współrzędnych teksturowych.

Listing 2.12. Kod tworzący bufor dla kostki pokrytej tekstem

```
var texCoordBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, texCoordBuffer);

var textureCoords = [
    // Pród.
    0.0, 0.0,
    1.0, 0.0,
    1.0, 1.0,
    0.0, 1.0,
    // Tyl.
    1.0, 0.0,
    1.0, 1.0,
    0.0, 1.0,
    0.0, 0.0,
    // Góra.
    0.0, 1.0,
    0.0, 0.0,
    1.0, 0.0,
    1.0, 1.0,
```

```

// Dół.
1.0, 1.0,
0.0, 1.0,
0.0, 0.0,
1.0, 0.0,

// Prawa.
1.0, 0.0,
1.0, 1.0,
0.0, 1.0,
0.0, 0.0,

// Lewa.
0.0, 0.0,
1.0, 0.0,
1.0, 1.0,
0.0, 1.0,
];

gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(textureCoords),
gl.STATIC_DRAW);

```

Musimy zmodyfikować kod shadera, aby zamiast danych kolorów wykorzystywał dane tekstury. Shader wierzchołków zawiera definicję atrybutu wierzchołka `texCoord`, który jest przekazywany z danymi wierzchołka, oraz definicję zmiennej `varying vTexCoord`, która będzie wysyłana do shadera fragmentów dla każdego wierzchołka. Shader fragmentów wykorzystuje tę współrzędną teksturową jako indeks do danych tekstury, które są przekazywane jako typ `uniform` do shadera fragmentów w zmiennej `uSampler`. Dane pikseli pobieramy z tekstury za pomocą funkcji GLSL o nazwie `texture2D()`, która pobiera sampler oraz dwuwymiarowy wektor zawierający współrzędne *x* i *y*. Zmieniony kod shadera jest pokazany na listingu 2.13.

Listing 2.13. Kod shadera dla kostki pokrytej teksturą

```

var vertexShaderSource =
    "    attribute vec3 vertexPos;\n" +
    "    attribute vec2 texCoord;\n" +
    "    uniform mat4 modelViewMatrix;\n" +
    "    uniform mat4 projectionMatrix;\n" +
    "    varying vec2 vTexCoord;\n" +
    "    void main(void) {\n" +
    "        // Zwraca przekształconą i rzutowaną wartość wierzchołka\n" +
    "        gl_Position = projectionMatrix * modelViewMatrix * \n" +
    "                      vec4(vertexPos, 1.0);\n" +
    "        // Wysyła współrzędną tekstury do vTexCoord\n" +
    "        vTexCoord = texCoord;\n" +
    "    }\n";

var fragmentShaderSource =
    "    precision mediump float;\n" +
    "    varying vec2 vTexCoord;\n" +
    "    uniform sampler2D uSampler;\n" +
    "    void main(void) {\n" +
    "        // Zwraca kolor piksela: zawsze biały\n" +
    "        gl_FragColor = texture2D(uSampler, vec2(vTexCoord.s, vTexCoord.t));\n" +
    "    }\n";

```

Ostatnią czynnością w procesie pokrycia naszej kostki teksturą jest nieznaczna modyfikacja funkcji rysującej, którą pokazano na listingu 2.14. Kod tworzenia bufora kolorów zastępujemy kodem tworzącym bufor współrzędnych teksturowych. Ponadto ustawiamy teksturę, która ma

zostać użyta, oraz łączymy ją z danymi wejściowymi shadera. (Podobnie jak w przypadku shaderów i innych stanów w API WebGL, istnieje też pojęcie bieżącej, czyli **aktywnej**, tekstury). Teraz można narysować naszą kostkę za pomocą funkcji `gl.drawElements()`.

Listing 2.14. Tworzenie danych tekstury do narysowania

```
gl.vertexAttribPointer(shaderTexCoordAttribute, obj.texCoordSize, gl.FLOAT, false, 0, 0);
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, obj.indices);

gl.uniformMatrix4fv(shaderProjectionMatrixUniform, false, projectionMatrix);
gl.uniformMatrix4fv(shaderModelViewMatrixUniform, false, modelViewMatrix);

gl.activeTexture(gl.TEXTURE0);
gl.bindTexture(gl.TEXTURE_2D, webGLTexture);
gl.uniform1i(shaderSamplerUniform, 0);
```

Podsumowanie

W tym rozdziale dowiedziałeś się, jak renderować grafikę przy użyciu API WebGL. Wiesz, jak utworzyć podstawową aplikację WebGL, czyli umiesz utworzyć kontekst, obszar widoku, bufory, macierze, shadery oraz podstawowe obiekty rysunkowe. Nauczyłeś się tworzyć dwu- i trójwymiarowe obiekty geometryczne oraz pokrywać je jednolitym kolorem i tekstem. Skorzystałeś nawet z pomocy otwartych bibliotek `glMatrix` i `RequestAnimationFrame.js` — dwóch filarów programowania dla WebGL.

Powinno być już oczywiste, że programowanie przy użyciu WebGL, na najniższym poziomie, jest bardzo pracochłonne. Zdolaliśmy przedstawić dość skomplikowaną animowaną geometrię z kolorami i teksturami na stronie, ale musielibyśmy w tym celu napisać kilkaset wierszy kodu. W technologii tej drzemie wielka moc, bo z każdym wierzchołkiem i pikselem można zrobić praktycznie wszystko, co się chce, i na dodatek ma się do dyspozycji fantastycznie szybki sprzęt. Jednak trzeba się nieźle napracować. Projektanci standardu świadomie poświęcili zwięzłość na rzecz wydajności. Samo API jest niewielkie i proste, ale przez to programista musi sam napisać więcej kodu.

Jeśli jesteś doświadczonym programistą grafiki lub gier i chcesz mieć pełną kontrolę nad wydajnością oraz funkcjonalnością swoich aplikacji, dobrym wyborem może być praca bezpośrednio z API WebGL. A jeśli budujesz aplikację o specyficznych wymaganiach dotyczących renderowania — np. program do przetwarzania grafiki albo narzędzie do modelowania trójwymiarowego — powinieneś trzymać się blisko narzędzi tej biblioteki. Zapewne wcześniej czy później i tak będziesz musiał skorzystać z jakiejś abstrakcji — przecież nie ma sensu przepisywanie w nieskończoność tych samych 40 wierszy kodu odpowiadających np. za tworzenie kostki — ale będzie to Twoja własna warstwa oprogramowania, w której będziesz mieć pełną kontrolę nad każdym znakiem.

Jeśli natomiast jesteś zwykłym śmiertelnikiem, jak większość z nas, powinieneś programować na nieco wyższym poziomie niż WebGL, najlepiej używając gotowych narzędzi. Najlepsze jest to, że narzędzia te są już gotowe do użytku, istnieje kilka świetnych otwartych bibliotek bazujących na WebGL. Ich opis znajduje się w kolejnych rozdziałach. Bierzemy się do pracy.

Three.js

— mechanizm do programowania grafiki trójwymiarowej w JavaScriptie

W poprzednim rozdziale pokazałem wielkie możliwości i poziomy komplikacji programowania przy użyciu biblioteki WebGL. Biblioteka ta pozwala w pełni wykorzystać moc procesora graficznego do tworzenia na bieżąco pięknych trójwymiarowych obrazów i animacji na stronach internetowych. Aby jednak zrobić cokolwiek innego niż narysowanie najprostszej geometrii przy użyciu tego API, trzeba włożyć mnóstwo wysiłku i napisać dosłownie setki wierszy kodu źródłowego. Nie jest ono zatem najlepszym wyborem do szybkiego tworzenia aplikacji sieciowych. Większość programistów ma dwie możliwości do wyboru, w zależności od rodzaju projektu: zbudować własną bibliotekę pomocniczą ułatwiającą dalszą pracę lub skorzystać z jednej z gotowych takich bibliotek.

Podczas gdy pracę z WebGL można zacząć na wiele sposobów, najczęściej wybieraną w tym celu biblioteką pomocniczą jest Three.js (<http://threejs.org/>). Oferuje ona łatwy w użyciu i intuicyjny zestaw typowych obiektów stosowanych do programowania grafiki trójwymiarowej. Ponadto jest szybka i do jej budowy wykorzystano wiele najlepszych praktyk programistycznych. Dodatkowo biblioteka Three.js jest projektem typu open source, jest dostępna w serwisie GitHub i dobrze prowadzona oraz rozwijana przez kilku programistów.

Zasadniczo Three.js to praktyczne standard dla programowania przy użyciu WebGL. Jest wykorzystywana w większości świetlnych, dostępnych w internecie projektów, m.in. w 100 000 Stars Google (rozdział 1.) i wielu innych nowatorskich produktach.

Najbardziej znane projekty zbudowane przy użyciu Three.js

Najszerzej znanym projektem WebGL zbudowanym przy użyciu Three.js jest chyba RO.ME 3 *Dreams of Black* (<http://www.ro.me/>), czyli interaktywny film utworzony w 2011 r. przez filmowca Chrisa Milka z pomocą specjalistów z firmy Google. Film ten jest dodatkiem do piosenki *Black* z ROMĘ, czyli muzycznego projektu Danger Mouse i Daniela Luppi wykonanego we współpracy z Jackiem Whitem i Norą Jones (rysunek 3.1).



Rysunek 3.1. RO.ME, 3 Dreams of Black: interaktywny film inspirowany piosenką Black z albumu ROMЕ (<http://www.ro.me/>)

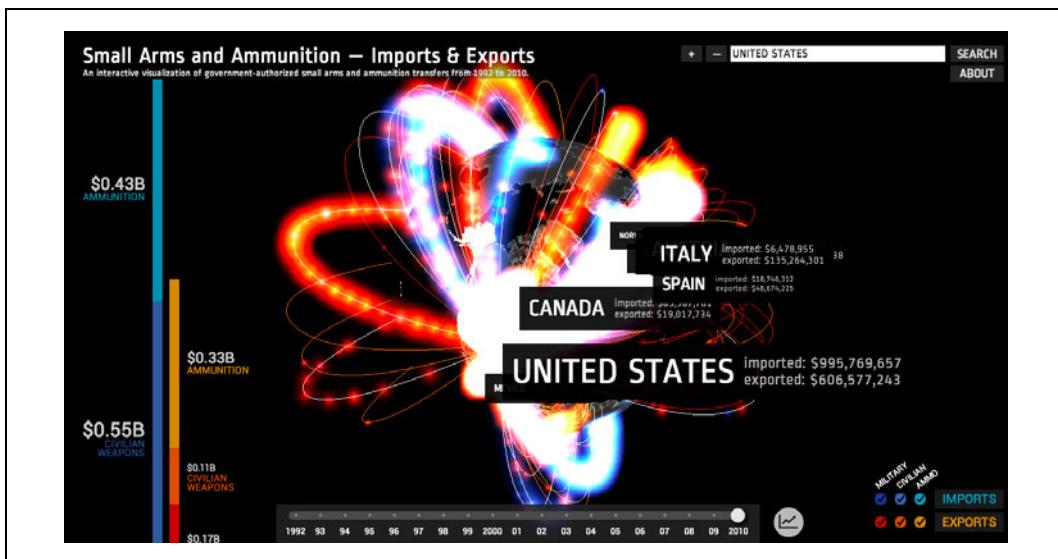
RO.ME to stworzony z rozmachem wirtualny świat, w którym widz może interaktywnie sterować kamerą, dodawać różne przedmioty oraz widzieć elementy dodane przez innych użytkowników. Do produkcji użyto biblioteki Three.js i nowatorskich jak na tamte czasy efektów WebGL, takich jak np. shader głębi pola sprawiający, że bliskie obiekty są wyraźne, a oddalone — zaćmione. Użyto także shadera kreskówkowego w celu uzyskania stylu przypominającego film animowany. Ponadto zastosowano grupowanie zachowań obiektów (ang. *flocking behaviors*) i do renderowania geometrii skorzystano z chmur punktów. Więcej informacji na temat technologii użytych do produkcji RO.ME można znaleźć na stronie <http://www.ro.me/tech/>.

Równie imponujące zastosowania WebGL i Three.js można znaleźć także w bardziej przyziemnych dziedzinach niż film, np. wizualizacji produktów. Na rysunku 3.2 widać zrzut ze znanego i obsypanego nagrodami konfiguratora samochodów utworzonego przez niemiecki zespół o nazwie Plus 360 Degrees. Użytkownik może obracać scenę, wybierać spośród kilku bardzo szczegółowo odtworzonych modeli samochodów, zmieniać kolor lakieru oraz opony, aby poskładać auto marzeń. Podobne aplikacje konfiguracyjne są znane od lat i niektóre z nich działają nawet w przeglądarkach za pomocą wtyczki Flash. Jednak wartość produkcyjna opisywanej aplikacji przewyższa wszystko, co było znane do tej pory. Szczegółowe odzwierciedlanie samochodów przy użyciu dużej liczby wielokątów, mapy środowiskowe symulujące refleksy oraz zastosowanie światła i cieni sprawiają, że efekt jest bardzo realistyczny i nie można od niego oderwać wzroku.

Za pomocą Three.js można renderować nie tylko rzeczywiste przedmioty, ale i również abstrakcyjne obiekty. Niesamowity przykład można zobaczyć na rysunku 3.3. Jest to utworzona w ramach eksperymentu Google wizualizacja globalnego obrotu bronią strzelecką i amunicją. Uwzględniono ponad milion punktów danych dotyczących eksportu i importu w celu zobrazowania przepływu broni strzeleckiej i ręcznej oraz amunicji w 250 krajach i terytoriach na całym świecie w latach od 1992 do 2010. Użyto w tym celu kolorów, linii oraz efektów blasku nałożonych na wirtualną kulę ziemską.



Rysunek 3.2. Konfigurator i wizualizator samochodów firmy Plus 360 Degrees (<http://carvisualizer.plus360degrees.com/threejs/>)



Rysunek 3.3. Small Arms Imports/Exports: eksperyment Google wykonany przez Google Ideas (<http://www.chromeexperiments.com/detail/arms-globe/>)

Biblioteka Three.js nie jest typowym mechanizmem do tworzenia gier (myśl tą rozwinię nieco dalej), ale można jej używać jako bazy do budowy mechanizmu gier oraz całkiem dobrych gier. Thibaut Despoulain w hołdzie dla Wipeout i serii gier F-Zero utworzył HexGL, czyli futurystyczną grę wyścigową, której akcja dzieje się w kosmosie. HexGL ma wiele cech produkcji najwyższej jakości, takich jak efekty poświaty, systemy cząsteczek, realistyczne renderingi budynków i statków, wizualny postprocessing oraz piękny wyświetlacz ekranowy. Na rysunku 3.4 przedstawiono zrzut ekranu z gry.



Rysunek 3.4. HexGL to futurystyczna, szybka gra zbudowana przez Thibauta Despoulaina przy użyciu HTML5, JavaScriptu oraz WebGL (<http://hexgl.bkcore.com/>)

Wprowadzenie do Three.js

Three.js to produkt mieszkającego w Barcelonie Ricarda Cabella Miguela, bardziej znanego pod pseudonimem *Mr.doob* (nigdy nie miałem śmiałości zapytać, skąd ten przydomek). Biblioteka ta jest owocem ponad 10 lat pracy Mr.dooba przy tworzeniu trójwymiarowych prezentacji na jazdy ludzi związanych ze sztuką komputerową. Mając dość dostępnych narzędzi i silników, Mr.doob zaczął budować własne, które początkowo pisał w języku ActionScript dla Adobe Flash. Gdy jednak kilka lat później pojawiły się przeglądarka Google Chrome, szybki JavaScript oraz HTML5, programista przerzucił się na nową platformę, dzięki czemu w 2010 r. powstała biblioteka Three.js. Pierwsza wersja renderowała do SVG i kanwy, ale po kilku miesiącach, gdy pojawiła się biblioteka WebGL, Three.js została zmodyfikowana i dostosowana do tej nowej technologii, co wg Mr.dooba „było łatwe w implementacji”, pewnie dlatego, że wcześniej zbudował już dwa inne mechanizmy renderowania przy użyciu tej samej technologii. Od tego czasu biblioteka Three.js rośnie w siłę i staje się coraz bardziej zaawansowana; jest też najpopularniejszym produktem do tworzenia trójwymiarowych aplikacji przy użyciu WebGL.

Wybór Three.js jako podstawy przykładów prezentowanych w tej książce nie ma ścisłego związku z popularnością tej biblioteki, chociaż muszę przyznać, że to również miało znaczenie. Przede wszystkim jednak używam tej biblioteki we własnych projektach, bo po prostu ją lubię. Prócz tego uważam, że jest to biblioteka najbardziej kompletna, jeśli chodzi o obsługę funkcji WebGL. Dodatkowo podoba mi się, że zajmuje się nią kilku programistów, którzy pracują w kontekście realnych zastosowań. I w końcu łatwo rozpocząć pracę z tą biblioteką. Już samo to jest wystarczającym powodem, by jej użyć. Należy jednak podkreślić, że Three.js to tylko jedna z wielu możliwości do wyboru, wśród których jest też miejsce dla własnego projektu dostosowanego do indywidualnych potrzeb (i własnego temperamentu). W kolejnych rozdziałach tej książki bardzo dobrze poznasz bibliotekę Three.js, a na razie przedstawiam zestawienie jej najważniejszych właściwości.

- **Three.js ukrywa niskopoziomowe szczegóły renderowania WebGL.** W bibliotece tej scena trójwymiarowa jest reprezentowana jako zbiór siatek, materiałów i światel (czyli typów obiektów, z którymi na co dzień pracują programiści grafiki).
- **Three.js daje bardzo duże możliwości.** Jest to coś więcej niż prosta nakładka na WebGL. Zawiera wiele gotowych obiektów przydanych podczas tworzenia gier, animacji, prezentacji wizualizacji, aplikacji do modelowania oraz efektów specjalnych. Istnieje też wiele gotowych do użycia dodatków.
- **Biblioteka Three.js jest łatwa w użyciu.** API Three.js jest wygodne i łatwo się go nauczyć. Do biblioteki dołączono wiele przykładów, od których można rozpocząć własną pracę.
- **Biblioteka Three.js jest szybka.** Przy budowie Three.js zastosowano najlepsze techniki programowania, dzięki czemu nie trzeba było poświęcać walorów użytkowych na rzecz prędkości.
- **Biblioteka Three.js jest niezawodna.** Mechanizmy sprawdzania błędów, wyjątki i ostrzeżenia konsolowe pozwalają programistom dowiedzieć się o każdym błędzie.
- **Biblioteka Three.js obsługuje interakcje.** W WebGL brakuje jakichkolwiek mechanizmów pozwalających sprawdzić, kiedy kurSOR znajduje się nad obiektem. Three.js pomaga w tym i ułatwia dodanie interaktywnych elementów do aplikacji.
- **Biblioteka Three.js wykonuje obliczenia.** Three.js zawiera przydatne i łatwe w użyciu obiekty do wykonywania obliczeń związanych z grafiką trójwymiarową, a więc obliczeń na macierzach, na wektorach oraz dotyczących rzutowania.
- **Biblioteka Three.js obsługuje wbudowany format plików.** Można ładować pliki w formatach tekstowych eksportowanych przez popularne pakiety do modelowania trójwymiarowego. Ponadto dostępne są formaty JSON i binarne, specyficzne dla Three.js.
- **Biblioteka Three.js jest obiektowa.** Programista może korzystać z normalnych obiektów JavaScript, a nie tylko wykonywać wywołania funkcji JavaScript.
- **Biblioteka Three.js jest rozszerzalna.** Dodawanie funkcji i dostosowywanie Three.js jest w miarę łatwe. Jeśli brakuje jakiegoś typu danych, można go napisać i dodać.
- **Biblioteka Three.js renderuje też na dwuwymiarowej kanwie oraz w SVG i CSS.** Mimo wielkiej popularności, biblioteka WebGL nie jest jeszcze obsługiwana wszędzie i nie zawsze jest najlepszym wyborem. Natomiast Three.js większość treści może też renderować na kanwie dwuwymiarowej i elemencie SVG. Możliwości te są szczególnie przydatne, gdy nie ma dostępnego kontekstu kanwy trójwymiarowej, bo wówczas program może elegancko przejść na alternatywne rozwiązanie. Ponadto Three.js można używać do renderowania i przekształcania elementów CSS, o czym jest mowa w rozdziale 6.

Należy też wiedzieć, czego z pomocą biblioteki Three.js *nie da się zrobić*. Nie jest ona mechanizmem do tworzenia gier i brakuje w niej typowych elementów, jakich szukamy w takich produktach, np. bilbordów, awatarów, skończonych maszyn stanów oraz fizyki. Three.js nie ma też wbudowanej obsługi sieci, która byłaby przydatna przy tworzeniu gry dla wielu graczy. Jeśli potrzebujesz takich funkcji, musisz je dodać samodzielnie albo użyć jakiejś specjalistycznej biblioteki. Three.js nie jest też systemem szkieletowym do budowy aplikacji, więc nie udostępnia funkcji do tworzenia, niszczenia, obsługi zdarzeń oraz uruchamiania pętli. W kolejnych rozdziałach pokazuję, jak oszczędzać czas i unikać wielokrotnego pisania tego samego kodu przy użyciu szkieletów. W końcu Three.js nie jest środowiskiem programistycznym. Nie ma w niej zestawu narzędzi do budowy trójwymiarowej aplikacji od początku do końca.

Nie bacząc na ograniczenia, bibliotekę Three.js należy lubić za to, czym jest, a jest wydajnym, dojrzalym funkcjonalnie, łatwym w użyciu mechanizmem do renderowania grafiki trójwymiarowej w przeglądarkach internetowych. To bardzo dużo. A teraz zobaczymy, jak działa.

Przygotowanie do pracy z Three.js

Aby korzystać z biblioteki Three.js, należy ją najpierw pobrać z serwisu GitHub. Aktualnie repozytorium znajduje się pod adresem <https://github.com/mrdoob/three.js/>. Sklonuj je i używaj niezminimalizowanej wersji kodu JavaScript znajdującej się w pliku *build/three.js*. (W pliku *build/three.min.js* znajduje się też zminimalizowana wersja biblioteki, którą można stosować w gotowych projektach; ale do pracy z przykładami zalecam używanie pełnej wersji, aby łatwiej znajdować błędy). Warto też mieć pod ręką pełny kod źródłowy umieszczony w folderze *src*. Na stronie w GitHub znajduje się odnośnik do dokumentacji, ale nie jest ona zbyt rozbudowana, więc źródło może się przydać, aby coś sprawdzić.



W książce tej używana jest wersja 58. biblioteki Three.js (r58). Mr.doob i spółka często zmieniają wersje, więc jeśli pobierzesz najnowszy produkt, może się on nieco różnić od używanego w przykładach z tej książki. Dlatego wszystkie przedstawione w tej książce przykłady są samodzielne, a w folderze *libs/three.js.r58/* znajduje się wersja 58. biblioteki.

Struktura projektu Three.js

Poświęć chwilę czasu na zapoznanie się ze strukturą kodu źródłowego i przykładami, aby oswoić się z biblioteką. Jest sporo do zrobienia i pewnie nie możesz doczekać się, aż zaczniesz pisać kod. Mimo to, oddaj sobie przysługę i zrób to, o czym piszę. Przejrzyj przynajmniej folder z przykładami *examples*. Gwarantuję, że nie pożałujesz.

Oto krótka charakterystyka najważniejszych folderów projektu.

build/

Katalog wyjściowy zminimalizowanej i niezminimalizowanej wersji biblioteki. Three.js jest budowana przy użyciu kompilatora Google Closure: jeden plik wyjściowy komplikacji zawiera całą bibliotekę Three.js skompilowaną z kilku plików źródłowych. Jeśli nie znasz Closure, a chcesz poznać, wejdź na stronę <http://code.google.com/closure/compiler/>. Nie trzeba samodzielnie kompilować biblioteki ze źródła, więc jeśli nie chcesz tego robić, możesz po prostu użyć gotowego pliku *three.js* lub *three.min.js*.

docs/

Folder zawierający dokumentację API w formacie HTML. Dokumentacja nie jest szczegółowa, ale zawiera przynajmniej niezłe wprowadzenie, pozwalające na poznanie podstaw biblioteki.

editor/

Twórcy Three.js rozpoczęli prace nad systemem edycji do tworzenia scen trójwymiarowych. Podczas pisania tej książki system ten znajdował się dopiero w początkowej fazie rozwoju i nie był zbyt przydatny w produkcji. Trzeba jednak przyznać, że Mr.doob żadnej pracy się nie boi, jeśli może użyć przeglądarki internetowej i edytora tekstu.

examples/

Folder zawierający setki przykładów zastosowania wielu funkcji i efektów renderowanych różnymi metodami, tzn. przy użyciu kanwy, CSS oraz WebGL. Niektóre z tych przykładów to zwykłe „demówki techniczne” pokazujące sposób działania wybranych funkcji, ale są też oszałamiające dzieła sztuki stworzone przy użyciu kilku funkcji. Obejrzyj dokładnie każdy przykład i nie zapomnij zajrzeć do kodu źródłowego. Jest to najlepszy sposób na poznanie wielkich możliwości biblioteki Three.js.

src/

Pliki źródłowe biblioteki. Tworzą skomplikowane drzewo, które można z grubsza podzielić na dwie części, **główną i dodatki**. W części głównej znajdują się narzędzia podstawowe, więc jest czymś w rodzaju **minimalnego zbioru funkcji**. Bez niej nie dałoby się renderować scen. Natomiast dodatki zawierają wiele dodatkowych przydatnych funkcji, takich jak wbudowane figury geometryczne, np. sześciany, sfery i cylindry, narzędzia do animacji oraz klasy do ładowania obrazów. Wszystkie te funkcje można zbudować samodzielnie na bazie Three.js, ale nie każdemu się chce. Mimo że klasy te zaliczane są do *dodatków*, wszystkie są dostępne w komplikacji.

utils/

Folder zawierający różne narzędzia, takie jak skrypty Google Closure do tworzenia zminimalizowanej i niezminimalizowanej komplikacji, konwertery plików z różnych formatów trójwymiarowych na format JSON Three.js i formaty binarne (więcej o nich piszę nieco dalej) oraz eksportery plików z popularnych pakietów do modelowania, takich jak Blender i Maya.

Prosty program Three.js

Znasz już podstawy budowy biblioteki Three.js, czas więc napisać jakiś program. Na podstawie pierwszego przykładu zorientujesz się, jak wielkim ułatwieniem jest ta biblioteka w porównaniu z bezpośredniem używaniem API WebGL.

Przypomnij sobie pokrytą teksturą kostkę z poprzedniego rozdziału. Utworzmy ją jeszcze raz, ale tym razem przy użyciu Three.js. Kod dotyczący biblioteki znajduje się na *listingu 3.1*, a całość można znaleźć w pliku *r3/threejscube.html*.

Listing 3.1. Tworzenie pokrytej teksturową kostką przy użyciu biblioteki Three.js

```
<script type="text/javascript">

    var renderer = null,
        scene = null,
        camera = null,
        cube = null;

    var duration = 5000; //ms
    var currentTime = Date.now();
    function animate() {

        var now = Date.now();
        var deltat = now - currentTime;
        currentTime = now;
        var fract = deltat / duration;
        var angle = Math.PI * 2 * fract;
        cube.rotation.y += angle;
    }

</script>
```

```

function run() {
    requestAnimationFrame(function() { run(); });

    // Renderuje scenę.
    renderer.render( scene, camera );

    // Obraca kostkę w następnej klatce.
    animate();

}

$(document).ready(
    function() {
        var canvas = document.getElementById("webglcanvas");

        // Tworzy renderer Three.js i wiąże go z kanwą.
        renderer = new THREE.WebGLRenderer(
            { canvas: canvas, antialias: true } );

        // Ustawia rozmiar obszaru widoku.
        renderer.setSize(canvas.width, canvas.height);

        // Tworzy nową scenę Three.js.
        scene = new THREE.Scene();

        // Dodaje kamerę, aby można było zobaczyć scenę.
        camera = new THREE.PerspectiveCamera( 45,
            canvas.width / canvas.height, 1, 4000 );
        scene.add(camera);

        // Tworzy pokrytą tekstonią kostkę i dodaje ją do sceny.
        // Najpierw tworzy się tekstonię.
        var mapUrl = "../images/webgl-logo-256.jpg";
        var map = THREE.ImageUtils.loadTexture(mapUrl);

        // Następnie tworzy się podstawowy materiał.
        var material = new THREE.MeshBasicMaterial({ map: map });

        // Utworzenie geometrii kostki.
        var geometry = new THREE.CubeGeometry(2, 2, 2);

        // Wstawienie geometrii i materiału do siatki.
        cube = new THREE.Mesh(geometry, material);

        // Odsunięcie siatki od kamery i pochylenie jej w kierunku użytkownika.
        cube.position.z = -8;
        cube.rotation.x = Math.PI / 5;
        cube.rotation.y = Math.PI / 5;

        // Dodanie siatki do sceny.
        scene.add( cube );

        // Uruchomienie pętli.
        run();
    }
);
</script>

```

Funkcje animacji i uruchamiania pętli są podobne do użytych w rozdziale 2. Drobne różnice wyjaśnię nieco dalej. Najważniejszy w tym przypadku jest kod tworzący scenę: w surowej wersji WebGL zajął 300 wierszy kodu, a w tej tylko 40. Nasza funkcja zwrotna ready() jQuery

mieści się na jednej stronie. To o wiele lepszy wynik. Przedstawiony przykład jest prosty, ale przynajmniej wiesz już, jak tworzyć pełne aplikacje podobne do opisanych na początku tego rozdziału. Teraz przeanalizujemy ten przykład szczegółowo.

Tworzenie renderera

Najpierw należy zbudować renderer. W bibliotece Three.js mechanizm renderowania jest zrealizowany za pomocą wtyczki, czyli jedną scenę można wyrenderować przy użyciu różnych API, np. WebGL albo Canvas 2D. W tym przypadku utworzony został obiekt THREE.WebGLRenderer z dwoma parametrami inicjalizacyjnymi; są to canvas, oznaczający element <canvas> utworzony w pliku HTML, oraz antialias włączający sprzętowy **antialiasing** wielopróbkowy (ang. *multisample antialiasing* — MSAA). Antialiasing eliminuje brzydkie artefakty powodujące, że niektóre linie są poszarpane. Biblioteka Three.js wykorzystuje te parametry do utworzenia kontekstu rysunkowego WebGL powiązanego z obiektem renderowania.

Po utworzeniu renderera ustawiamy jego rozmiar na całą szerokość i wysokość kanwy. Jest to równoważne z wywołaniem `gl.viewport()` w celu ustawienia rozmiaru obszaru widoku, które zastosowaliśmy w rozdziale 2. Cała konfiguracja renderera zajmuje zaledwie dwa wiersze kodu.

```
// Tworzy renderer Three.js i wiąże go z kanwą.  
renderer = new THREE.WebGLRenderer(  
{ canvas: canvas, antialias: true } );  
  
// Ustawia rozmiar obszaru widoku.  
renderer.setSize(canvas.width, canvas.height);
```

Tworzenie sceny

Następnie tworzymy **scenę**, czyli nowy obiekt THREE.Scene, który jest podstawą całej hierarchii obiektów w bibliotece Three.js. Zawiera on wszystkie inne obiekty graficzne (w Three.js obiekty występują w hierarchii rodzic-dziecko, o czym szerzej napiszę już wkrótce).

Do posiadanej sceny dodajemy dwa obiekty — **kamerę** i **siatkę**. Kamera określa, z którego miejsca oglądamy scenę (w tym przykładzie pozostawiamy ją w domyślnym położeniu, na początku). Nasza kamera jest typu THREE.PerspectiveCamera. Zainicjalowaliśmy ją z 45-stopniowym polem widzenia, rozmiarami obszaru widoku oraz wartościami przedniej i tylnej płaszczyzny odcienia. Wewnętrznie biblioteka Three.js wykorzysta te wartości do utworzenia macierzy rzutowania perspektywy służącej do renderowania trójwymiarowej sceny na dwuwymiarowej powierzchni. (Jeśli zapomniałeś, czym są kamery, obszary widoku i rzutowanie, zatrzyj do opisu podstaw grafiki trójwymiarowej w rozdziale 1.).

Kod tworzący scenę i dodający kamerę jest bardzo zwięzły.

```
// Tworzy nową scenę Three.js.  
scene = new THREE.Scene();  
  
// Dodaje kamerę, aby można było zobaczyć scenę.  
camera = new THREE.PerspectiveCamera( 45,  
    canvas.width / canvas.height, 1, 4000 );  
scene.add(camera);
```

Czas na dodanie siatki do sceny. W Three.js siatka składa się z obiektu geometrycznego i **materiału**. Obiektem geometrycznym jest kostka o boku 2, którą utworzyliśmy przy użyciu wbudowanego obiektu biblioteki CubeGeometry. Natomiast materiał określa, jak ma wyglądać powierzchnia

tego obiektu geometrycznego. W tym przykładzie użyliśmy materiału typu `MeshBasicMaterial`, a więc prostego materiału pozabawionego efektów świetlnych. Chcemy też umieścić na kostce logo WebGL jako **teksturę**. Tekstury, zwane też **mapami teksturowymi**, to mapy bitowe reprezentujące atrybuty powierzchni trójwymiarowej siatki. Można używać ich w najprostszy sposób do definiowania koloru powierzchni oraz łączyć w celu tworzenia skomplikowanych efektów, takich jak wznowienia i podświetlenia.

W WebGL dostępnych jest kilka funkcji służących do pracy z teksturami, a ponadto w standardzie istnieją pewne zabezpieczenia, takie jak brak możliwości użycia tekstur z innych domen. Na szczęście, biblioteka Three.js zawiera narzędzia znacznie ułatwiające ładowanie tekstur i wiązanie ich z materiałami. Wywołujemy funkcję `THREE.ImageUtils.loadTexture()`, aby załadować teksturę z pliku graficznego, a następnie wiążemy otrzymaną teksturę z materiałem, ustawiając parametr `map` konstruktora materiału.

```
// Tworzy pokrytą teksturą kostkę i dodaje ją do sceny.  
// Najpierw tworzy się teksturę.  
var mapUrl = "../images/webgl-logo-256.jpg";  
var map = THREE.ImageUtils.loadTexture(mapUrl);  
  
// Następnie tworzy się podstawowy materiał.  
var material = new THREE.MeshBasicMaterial({ map: map });
```

Powyższy kod zmusza bibliotekę Three.js do ciężkiej pracy. Mapuje ona bity obrazu JPEG na odpowiednie części każdego boku kostki. Obraz nie jest rozciągany, aby owinąć całą kostkę ani żeby pokryć w całości któryś z boków. Może się wydawać, że to nic takiego, ale w poprzednim rozdziale widzieliśmy, że jest inaczej. Używając bezpośrednio biblioteki WebGL, trzeba samodzielnie pamiętać o wielu rzeczach, a dzięki Three.js wystarczy napisać tylko kilka wierszy kodu.

Na końcu budujemy siatkę. Utworzyliśmy już geometrię, materiał i teksturę. Teraz wszystko wstawiamy do obiektu `THREE.Mesh`, który zapisujemy w zmiennej o nazwie `cube`. Zanim dodamy go do sceny, odsuwamy kostkę o osiem jednostek od kamery, podobnie jak w rozdziale 2., ale tym razem nie musimy „bawić się” z obliczeniami macierzowymi. Po prostu ustawiamy własność `position.z` kostki. Za pomocą ustawienia własności `position.x` przechylamy też kostkę w kierunku użytkownika, aby było widać jej górny bok. Później dodajemy ją do sceny i gotowe, możemy renderować.

```
// Odsunięcie siatki od kamery i pochylenie jej w kierunku użytkownika.  
cube.position.z = -8;  
cube.rotation.x = Math.PI / 5;  
cube.rotation.y = Math.PI / 5;  
  
// Dodanie siatki do sceny.  
scene.add( cube );
```

Implementacja pętli wykonawczej

Podobnie jak w przykładzie opisany w poprzednim rozdziale, musimy przy użyciu funkcji `requestAnimationFrame()` zaimplementować pętlę wykonawczą. Szczegóły tej implementacji są nieco inne. W poprzedniej wersji funkcja `draw()` musiała tworzyć bufore, ustawiać stany renderowania, czyścić obszary widoku, konfigurować shadery i tekstury itd. W Three.js wystarczy jeden wiersz kodu:

```
renderer.render( scene, camera );
```

Biblioteka sama zrobi wszystko, co trzeba. Moim zdaniem, już samo to jest wystarczającym powodem, aby zainteresować się tym produktem.

Ostatnią czynnością jest obracanie kostki, aby w pełni ukazać jej trójwymiarowość. W bibliotece Three.js to również jest bardzo łatwe. Wystarczy ustawić własność `rotation.y` na nowy kąt, a biblioteka sama wykona odpowiednie obliczenia macierzowe. W następnym przebiegu pętli wykonawczej funkcja `render()` użyje nowej wartości obrotu wokół osi `y`, co spowoduje, że kostka się obróci. Poniżej jeszcze raz przedstawiam kod funkcji `animate()` i `render()`.

```
var duration = 5000; //ms
var currentTime = Date.now();
function animate() {

    var now = Date.now();
    var deltat = now - currentTime;
    currentTime = now;
    var fract = deltat / duration;
    var angle = Math.PI * 2 * fract;
    cube.rotation.y += angle;
}

function run() {
    requestAnimationFrame(function() { run(); });

    // Renderuje scenę.
    renderer.render( scene, camera );

    // Obraca kostką w następnej klatce.
    animate();
}
```

Pokazany na rysunku 3.5 efekt naszej pracy powinien wyglądać znajomo.



Rysunek 3.5. Pokryta teksturową kostka wyrenderowana przy użyciu biblioteki Three.js

Oświetlenie sceny

Program przedstawiony na listingu 3.1 ilustruje sposób utworzenia przy użyciu biblioteki Three.js najprostszej z możliwych scen trójwymiarowych. Może jednak zauważysz, że nasza kostka niby trójwymiarowa wydaje się jakaś taka płaska. Oczywiście, gdy się obraca, widać z grubsza jej kształt, głównie dzięki nałożonej na nią teksturze. Czegoś tu jednak brakuje i jest to cieniowanie. Jedną z najciekawszych rzeczy przy renderowaniu na bieżąco grafiki trójwymiarowej jest możliwość tworzenia imitacji cieniowania za pomocą światła. Spójrz na rysunek 3.6, na którym krawędzie kostki są wyraźnie zaznaczone, podobnie jak w realnych przedmiotach. Efekt ten został osiągnięty po dodaniu do sceny oświetlenia.



Rysunek 3.6. Kostka z oświetleniem i cieniowaniem Phong wyrenderowana przy użyciu biblioteki Three.js

Miałem zamiar dodać to światło już do przykładu w rozdziale 2., ale uznałem, że praca potrzebna do tego, by odpowiednio dostosować dane wierzchołków w buforze oraz przepisać shadery wierzchołków i fragmentów, nie jest tego warta. Mam nadzieję, że wystarczająco dobrze wyjaśniłem, iż można spędzić pół życia na konstruowaniu nawet najprostszych aplikacji, kiedy korzysta się z surowego kodu WebGL. Biblioteka Three.js znacznie ułatwia wykonywanie takich zadań. Wystarczy tylko kilka dodatkowych wierszy kodu, co widać na listingu 3.2. Pełny kod źródłowy tego przykładu znajduje się w pliku *r3/threejscube1it.html*.

Listing 3.2. Oświetlenie kostki przy użyciu narzędzi z biblioteki Three.js

```
// Dodanie światła w celu ukazania obiektu.  
var light = new THREE.DirectionalLight( 0xfffffff, 1.5 );  
  
// Ustawienie źródła światła tak, by było skierowane na początek sceny.  
light.position.set(0, 0, 1);  
scene.add( light );  
  
// Utworzenie cieniowanej pokrytej tekstoną kostki i dodanie jej do sceny.  
// Najpierw tworzymy tekstonę.  
var mapUrl = "../images/webgl-logo-256.jpg";
```

```
var map = THREE.ImageUtils.loadTexture(mapUrl);  
  
// Następnie tworzymy materiał Phong, aby uwidocznić cieniowanie; przekazujemy teksturę.  
var material = new THREE.MeshPhongMaterial({ map: map });
```

Najważniejsze są pogrubione fragmenty. Najpierw dodajemy do sceny światła. Są one rodzajem obiektów, a więc po utworzeniu dodaje się je do sceny, a ich wartości są wykorzystywane do renderowania innych obiektów. W tym przykładzie użyliśmy **światła kierunkowego** (ang. *directional light*), czyli świecącego równoległymi promieniami w określonym kierunku. Uważam, że stosowana w bibliotece Three.js składnia do tworzenia takich światel jest nieintuicyjna, bo trzeba określić ich pozycję oraz miejsce docelowe (domyślnie na początku sceny, więc tu opuszczone). Mając te informacje, biblioteka oblicza kierunek: odejmuje pozycję docelową od pozycji światła. W naszym przykładzie oznacza to, że światło jest skierowane w ekran i biegnie od punktu (0, 0, 1) do (0, 0, 0), a więc wprost na kostkę, która jest umieszczona na początku sceny.

Aby efekt dodania światła był widoczny, trzeba spełnić jeszcze jeden warunek: zamiast podstawowego materiału, jaki został użyty w poprzednim przykładzie, należy zastosować materiał **Phong**. W bibliotece Three.js oświetlenie obiektów zależy nie tylko od dodanych do sceny światel, ale i od typu materiałów. Typ Phong implementuje prosty, ale dość realistyczny model cieniowania, o nazwie **cieniowanie Phonga**, i jest bardzo efektywny. Po jego zastosowaniu widzimy krawędzie kostki: boki zwrócone do źródła światła są lepiej oświetlone niż pozostałe i dobrze widać krawędzie na ich złączeniach. To tylko podstawowe wiadomości dotyczące oświetlenia i należy zaznaczyć, że w rzeczywistości temat ten jest nieco bardziej skomplikowany. Zajmiemy się nim szczegółowo w następnym rozdziale. A na razie cieszymy się realistycznym, trójwymiarowym obiektem utworzonym przy użyciu zaledwie jednej strony kodu JavaScript.



Cieniowania Phonga zostało opracowane na University of Utah przez Bui Tuong Phonga. Algorytmy tego badacza, początkowo uważane za radykalne, obecnie stanowią podstawę metod cieniowania w wielu aplikacjach renderujących, zwłaszcza renderujących w bieżąco, ponieważ są bardzo wydajne i umożliwiają uzyskanie realistycznego efektu. Więcej informacji na temat cieniowania Phonga znajduje się w Wikipedii (http://pl.wikipedia.org/wiki/Cieniowanie_Phonga).

Podsumowanie

W rozdziale tym poznaleś bibliotekę Three.js, która jest najpopularniejszym zestawem narzędzi do tworzenia trójwymiarowych aplikacji sieciowych przy użyciu WebGL. Dowiedziałeś się o kilku fantastycznych, budowanych przy jej użyciu projektach z tak różnych dziedzin jak film i wizualizacje. Pobrałeś najnowszą wersję kodu z serwisu GitHub i przejrzałeś z grubsza strukturę plików. Na końcu przeanalizowałeś budowę kilku prostych przykładowych programów, aby przekonać się, jak bardzo Three.js ułatwia pracę programisty. Program, którego napisanie przy użyciu surowego kodu WebGL wymaga setek wierszy kodu, można za pomocą Three.js skrócić do kilkudziesięciu linijek. Ponadto biblioteka pomocnicza umożliwia korzystanie w pracy z ugruntowanych pojęć grafiki trójwymiarowej i programowania obiektowego.

W tym rozdziale pokazałem, jak szybko można tworzyć przy użyciu biblioteki Three.js. W kilku kolejnych pokażę, co z jej pomocą można osiągnąć.

Grafika i renderowanie w Three.js

W tym rozdziale dowiesz się, jakie narzędzia do rysowania grafiki i renderowania scen dostępne są w bibliotece Three.js. Jeśli jesteś początkującym programistą grafiki trójwymiarowej, nie staraj się jednocześnie zrozumieć wszystkich poruszanych tematów. Lepiej wybieraj po jednym i analizuj przykłady. W ten sposób szybko nauczysz się tworzyć wspaniałe strony z trójwymiarowymi efektami.

Biblioteka Three.js ma bogaty system graficzny, inspirowany wieloma wcześniejszymi bibliotekami trójwymiarowymi, który powstał z wykorzystaniem doświadczenia ich twórców. Ma wszystko, co powinna mieć biblioteka do tworzenia grafiki trójwymiarowej, a więc dwu- i trójwymiarową geometrię budowaną z siatek wielokątów, grafy scen z hierarchicznymi obiektami i przekształceniemi, materiały, tekstury i światła, generowane na bieżąco cienie, programowalne shadery oraz elastyczny system renderingu umożliwiający stosowanie technik wielopowtarzeniowych i opóźnień w celu uzyskania zaawansowanych efektów specjalnych.

Geometria i siatki

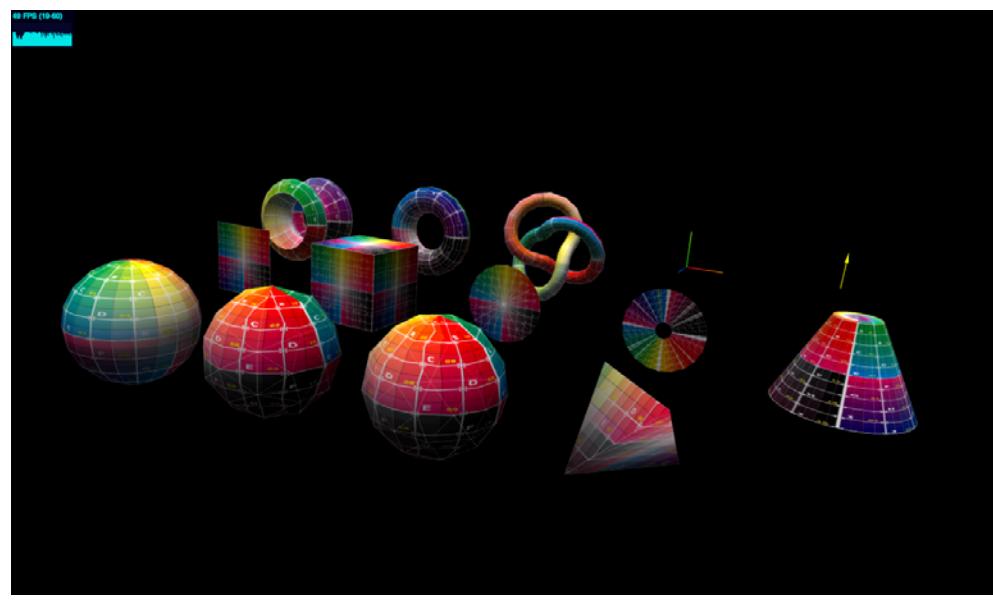
Jedną z największych zalet posługiwania się biblioteką Three.js w porównaniu z używaniem wprost API WebGL jest oszczędność pracy potrzebnej do tego, by utworzyć i narysować figury geometryczne. Przypomnij sobie całe strony kodu z rozdziału 2., napisane w celu utworzenia danych kształtów i tekstuur dla prostej kostki oraz przeniesienia tego wszystkiego do pamięci WebGL, aby ostatecznie narysować to na ekranie. Biblioteka Three.js oszczędza wielu kłopotów, udostępniając kilka gotowych obiektów geometrycznych, wśród których znajdują się kostki i cylindry, kształty ścieżkowe, wytłaczana geometria dwuwymiarowa oraz klasa bazowa do rozszerzania, aby użytkownik mógł tworzyć własne kształty. Przyjrzyjmy się tym udogodnieniom.

Gotowe typy geometryczne

Biblioteka Three.js zawiera wiele gotowych typów geometrycznych reprezentujących najczęściej używane kształty. Znajdują się wśród nich proste jednolite figury, takie jak kostki, sfery i cylindry, oraz bardziej skomplikowane kształty parametryczne, jak ekstruzje i kształty ścieżkowe, torusy czy węzły, płaskie dwuwymiarowe kształty renderowane w przestrzeni trójwymiarowej, takie jak koła, kwadraty i pierścienie, a nawet trójwymiarowy wyciskany (ang. *extruded*)

tekst generowany z łańcuchów tekstowych. Ponadto biblioteka Three.js ułatwia rysowanie punktów i linii trójwymiarowych. Większość z tych obiektów można z łatwością utworzyć przy użyciu jednowierszowego konstruktora, ale niektóre wymagają podania złożonych parametrów i napisania nieco większej ilości kodu.

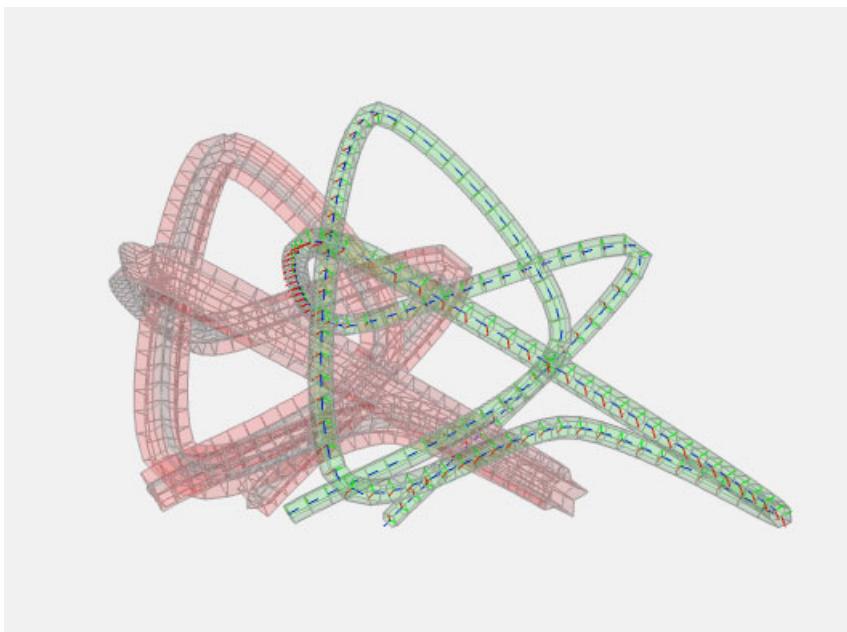
Aby zobaczyć na żywo, jak wyglądają gotowe obiekty geometryczne biblioteki Three.js, otwórz przykładowy plik projektu Three.js znajdujący się w folderze *examples/webgl_geometries.html* (rysunek 4.1). Każdy obiekt siatki zawiera inny typ geometrii, a tekstuра ukazuje sposób generowania współrzędnych teksturowych. Tekstury zostały udostępnione przez PixelCG Tips and Tricks, fantastyczny portal z poradami na temat grafiki komputerowej (<http://www.pixelcg.com/blog/>). Scena jest oświetlona światłem kierunkowym, aby ukazać cieniowanie każdego z obiektów.



Rysunek 4.1. Przykłady obiektów geometrycznych biblioteki Three.js. Od lewej i od przodu: sfera, dwudziestościan, ośmiościan, czworościan; płaszczyzna, kostka, koło, pierścień, cylinder; „tokarka”, torus i węzeł z torusa; osie x, y i z

Ścieżki, kształty i ekstruzje

Klasy Path, Shape i ExtrudeGeometry umożliwiają tworzenie obiektów geometrycznych na wiele sposobów, np. wyciskanie obiektów z krzywych. Na rysunku 4.2 przedstawiona jest ekstruzja wygenerowana przy użyciu krzywej składanej (ang. *spline curve*). Aby zobaczyć ją w swoim komputerze, otwórz plik *examples/webgl_geometry_extrude_shapes.html*, natomiast w pliku *examples/webgl_geometry_extrude_splines.html* można wybierać algorytmy generowania krzywej składanej, a nawet poruszać się po niej za pomocą animowanej kamery. Połączenie krzywej składanej z ekstruzją to doskonaly sposób na generowanie naturalnie wyglądających kształtów. Szczegółowy opis krzywych składanych znajduje się w rozdziale 5.



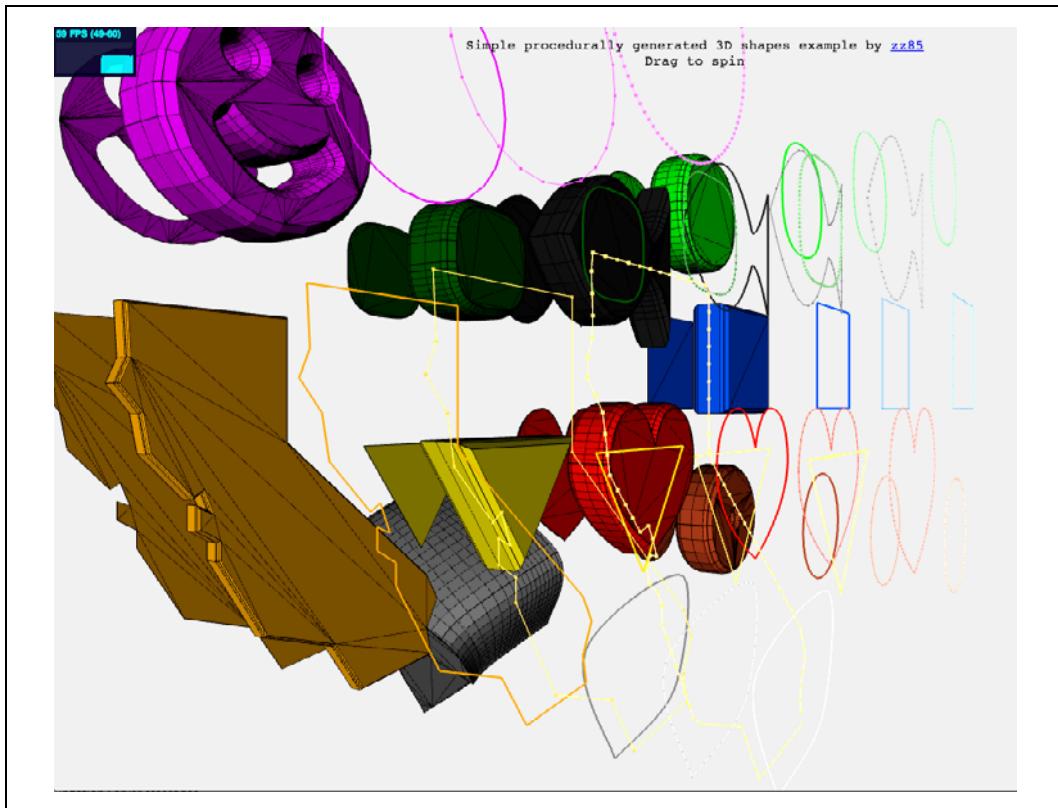
Rysunek 4.2. Ekstruze utworzone przy użyciu krzywej składanej z biblioteki Three.js

Klasy typu Shape można też stosować do tworzenia płaskich figur dwuwymiarowych oraz ich trójwymiarowych ekstruzji. Powiedzmy, że mamy bibliotekę danych dwuwymiarowych wielokątów (np. granic geopolitycznych albo grafiki wektorowej). Dane te można w miarę łatwo zaimportować do Three.js za pomocą klasy Path zawierającej metody do generowania ścieżek, takie jak `moveTo()` i `lineTo()`, które powinny być znane każdemu, kto zajmuje się rysowaniem grafiki dwuwymiarowej. (W istocie jest to dwuwymiarowe API rysunkowe osadzone w bibliotece grafiki trójwymiarowej). Po co to robić? Dwuwymiarowy kształt można wykorzystać do utworzenia płaskiej siatki istniejącej w przestrzeni trójwymiarowej, którą można przekształcać, tak jak każdy inny obiekt trójwymiarowy (przesuwać, obracać i skalować). Można ją pokrywać materiałami, oświetlać oraz cieniować wraz z pozostałymi przedmiotami na scenie albo ekstrudować w celu utworzenia prawdziwych trójwymiarowych kształtów z dwuwymiarowego zarysu.

Doskonałą ilustrację tych możliwości przedstawiono na rysunku 4.3, będącym zrzutem ekranu z pliku `examples/webgl_geometry_shapes.html`. Widać na nim zarys Kalifornii, kilka prostych wielokątów oraz serca i uśmiechnięte buźki, wyrenderowane w różnych formach, takich jak dwuwymiarowe płaskie siatki, ekstrudowane i ścięte trójwymiarowe siatki oraz linie, a wszystko wygenerowane z danych ścieżkowych.

Bazowa klasa geometrii

Wszystkie gotowe typy geometryczne biblioteki Three.js pochodzą od klasy bazowej THREE.Geometry (`src/core/Geometry.js`), której można też używać do tworzenia własnych kształtów geometrycznych. Aby dowiedzieć się, jak to robić, zajrzyj do kodu źródłowego gotowych typów, który



Rysunek 4.3. Ekstrudowane kształty utworzone na bazie ścieżek przy użyciu biblioteki Three.js

znajduje się w folderze `src/extras/geometries` projektu biblioteki. Na listingu 4.1 znajduje się kod jednego z najprostszego obiektów o nazwie `THREE.CircleGeometry`. Jak widać, nie jest zbyt obszerny, bo zmieścił się na jednej stronie.

Listing 4.1. Kod geometrii koła z biblioteki Three.js

```
/*
 * @author hughes
 */

THREE.CircleGeometry = function ( radius, segments, thetaStart, thetaLength ) {

    THREE.Geometry.call( this );
    radius = radius || 50;

    thetaStart = thetaStart !== undefined ? thetaStart : 0;
    thetaLength = thetaLength !== undefined ? thetaLength : Math.PI * 2;
    segments = segments !== undefined ? Math.max( 3, segments ) : 8;

    var i, uvs = [],
        center = new THREE.Vector3(), centerUV = new THREE.Vector2( 0.5, 0.5 );

    this.vertices.push( center );
    uvs.push( centerUV );
```

```

for ( i = 0; i <= segments; i ++ ) {

    var vertex = new THREE.Vector3();
    var segment = thetaStart + i / segments * thetaLength;

    vertex.x = radius * Math.cos( segment );
    vertex.y = radius * Math.sin( segment );

    this.vertices.push( vertex );
    uvs.push( new THREE.Vector2( ( vertex.x / radius + 1 ) / 2,
        ( vertex.y / radius + 1 ) / 2 ) );
}

var n = new THREE.Vector3( 0, 0, 1 );

for ( i = 1; i <= segments; i ++ ) {

    var v1 = i;
    var v2 = i + 1 ;
    var v3 = 0;

    this.faces.push( new THREE.Face3( v1, v2, v3, [ n, n, n ] ) );
    this.faceVertexUvs[ 0 ].push( [ uvs[ i ], uvs[ i + 1 ], centerUV ] );
}

this.computeCentroids();
this.computeFaceNormals();

this.boundingSphere = new THREE.Sphere( new THREE.Vector3(), radius );
};

THREE.CircleGeometry.prototype = Object.create( THREE.Geometry.prototype );

```

Konstruktor klasy `THREE.CircleGeometry` generuje płaski okrągły kształt na płaszczyźnie XY, tzn. z wszystkimi wartościami z ustawionymi na zero. Sercem tego algorytmu jest kod generujący dane wierzchołków kształtu znajdujący się w pierwszej pętli `for`.

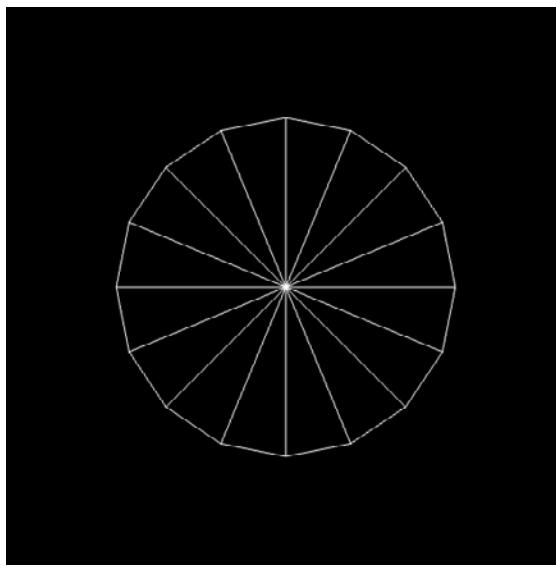
```

vertex.x = radius * Math.cos( segment );
vertex.y = radius * Math.sin( segment );

```

W rzeczywistości trójwymiarowe koło jest rozetą trójkątów stykających się wierzchołkami w jej środku. Przy użyciu odpowiedniej liczby takich trójkątów można uzyskać gładką krawędź obwodu, co pokazano na rysunku 4.4.

Pierwsza pętla oblicza tylko współrzędne x i y wierzchołków na obwodzie koła. Trzeba jeszcze utworzyć **bok** (wielokąt) reprezentujący każdy z tych trójkątów zbudowanych przez trzy wierzchołki: środek i dwa znajdujące się na obwodzie. Robi to druga pętla `for`, która tworzy dane i wstawia je do tablicy `this.faces`. Każdy bok zawiera indeksy trzech wierzchołków z tablicy `this.vertices` oznaczonych jako $v1$, $v2$ i $v3$. Wierzchołek $v3$ ma zawsze wartość zero, ponieważ odpowiada środkowi. (Może pamiętaś z rozdziału 2., że w WebGL używa się funkcji `gl.drawElements()` do renderowania trójkątów przy użyciu tablicy indeksowanej. To samo dzieje się tutaj, tylko wewnętrz mechanizmów biblioteki Three.js).



Rysunek 4.4. Trójkąty składające się na obiekt THREE.CircleGeometry

W każdej z pętli pomineliśmy jeden szczegół: generowanie współrzędnych teksturowych. Biblioteka WebGL „nie wie”, jak mapować piksele tekstuury na rysowane trójkąty, więc trzeba jej to podpowiedzieć. Pętle `for` generują współrzędne teksturowe, zwane również **współrzędnymi UV**, i zapisują je w tablicy `this.faceVertexUvs` w podobny sposób, jak generowały wartości wierzchołków.

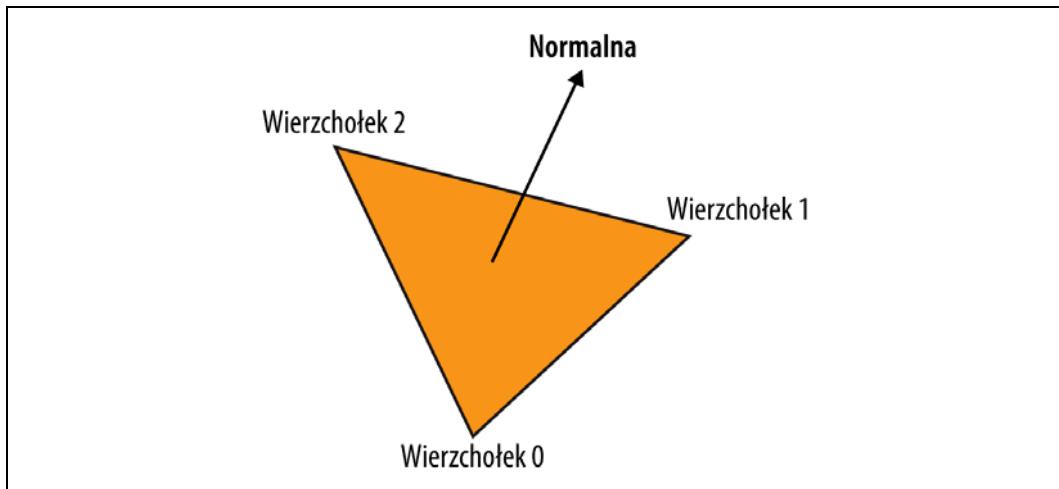
Przypomnę, że współrzędne teksturowe to zdefiniowane dla każdego wierzchołka pary liczb zmiennoprzecinkowych o wartościach najczęściej mieszczących się w przedziale od 0 do 1. Reprezentują one miejsca w danych mapy bitowej, a shader używa ich do pobierania z tej mapy informacji o pikselach. Współrzędne teksturowe dla pierwszych dwóch wierzchołków w każdym trójkącie obliczamy, podobnie jak dane tych wierzchołków, tzn. przy użyciu cosinusa kąta dla wartości x i sinusa dla y , a wartości mieszczące się w przedziale [0..1] uzyskujemy przez podzielenie wartości wierzchołków przez promień koła. Współrzędna teksturowa trzeciego wierzchołka każdego trójkąta, odpowiadająca środkowi, to po prostu dwuwymiarowy środek obrazu (0.5, 0.5).



Co oznaczają litery *UV*? Są to oznaczenia poziomej i pionowej osi dwuwymiarowej tekstuury, używane, aby odróżniały się od współrzędnych *X*, *Y* i *Z* oznaczających osie trójwymiarowe obiektu. Dokładniejszy opis tych współrzędnych znajduje się w Wikipedii (http://pl.wikipedia.org/wiki/UV_mapping).

Po wygenerowaniu danych wierzchołków i UV biblioteka Three.js ma wszystko, co jest potrzebne do renderowania geometrii. Ostatnie wiersze kodu konstruktora `THREE.Circle` zawierają już tylko wywołania funkcji pomocniczych z bazowej klasy geometrii. Funkcja `computeCentroids()` określa geometryczny środek obiektu poprzez przejrzenie za pomocą pętli wszystkich jego wierzchołków i uśrednienie pozycji.

Bardzo ważna jest funkcja `computeFaceNormals()`, ponieważ wektory normalne obiektu, albo w skrócie **normalne**, decydują o sposobie jego cieniowania. W przypadku płaskiego koła normalne każdego boku są prostopadłe do jego powierzchni. Funkcja `computeFaceNormals()` określa je, obliczając wektor prostopadły do płaszczyzny zdefiniowanej przez trzy wierzchołki wyznaczające każdy trójkąt koła. Na rysunku 4.5 przedstawiona jest normalna płaskiego cieniowanego trójkąta.



Rysunek 4.5. Normalna płaskiego cieniowanego trójkąta

Na koniec konstruktor inicjuje bryłę brzegową (ang. *bounding volume*) dla obiektu, w tym przypadku sferę, która jest przydatna do wybierania, usuwania niewidocznych powierzchni i wprowadzania różnych optymalizacji.

Geometria buforowana do optymalizacji renderowania siatki

W bibliotece Three.js jakiś czas temu wprowadzono zoptymalizowaną wersję geometrii o nazwie `THREE.BufferGeometry`. Służy ona do przechowywania danych w tablicach typowanych, co pozwala uniknąć narzutu związanego z używaniem tablic liczb JavaScript. Ponadto klasa ta jest przydatna do przechowywania statycznej geometrii, np. tel i przedmiotów scen, gdy wiadomo, że wartości wierzchołków się nie zmieniają, a obiekty nie są animowane, więc nie zmienią położenia na scenie. Jeśli ma się takie informacje, można utworzyć obiekt klasy `THREE.BufferGeometry`, a biblioteka Three.js wprowadzi szereg optymalizacji znacznie przyspieszających renderowanie tych obiektów.

Importowanie siatek z programów do modelowania

Do tej pory uczyłeś się technik tworzenia geometrii za pomocą kodu źródłowego, ale w większości aplikacji stosuje się inne rozwiązanie, które polega na wczytywaniu trójwymiarowych modeli utworzonych w profesjonalnych programach do modelowania, takich jak 3ds Max, Maya czy Blender.

Biblioteka Three.js oferuje kilka narzędzi do konwersji i wczytywania plików modelowych. Prześledzimy przykład ładowania siatki — zarówno geometrii, jak i materiałów. Otwórz plik `examples/webgl_loader_obj_mtl.html` z projektu Three.js, aby zobaczyć model widoczny na rysunku 4.6.



Rysunek 4.6. Siatka załadowana z pliku w formacie Wavefront OBJ

Widoczny na tym rysunku mężczyzna został zaimportowany z pliku w formacie Wavefront OBJ (o rozszerzeniu `.obj`). Jest to popularny format tekstowy używany przez wiele programów do modelowania. Pliki te są proste, ale mogą zawierać wyłącznie dane geometryczne: wierzchołki, normalne i współrzędne teksturowe. Dlatego firma Wavefront opracowała dodatkowy format plików dla materiałów, MTL, którego można używać do wiązania materiałów z obiektami w plikach OBJ.

Kod źródłowy mechanizmu Three.js wczytującego pliki w formacie OBJ (z materiałami) znajduje się w pliku `examples/js/loaders/OBJMTLLoader.js`. Jeśli przeanalizujesz sposób jego działania, zauważysz, że tworzy on obiekty `THREE.Geometry`, podobnie jak gotowe klasy geometrii i kształtów. Parser MTL tłumaczy opcje tekstowe znajdujące się w pliku MTL na materiały zrozumiałe dla Three.js. Następnie tworzony jest jeden obiekt `THREE.Mesh`, który można dodać do sceny.

Biblioteka Three.js zawiera przykładowe mechanizmy wczytyujące dla wielu formatów plików. Większość formatów umożliwia definiowanie obiektów przy użyciu geometrii i materiałów, ale niektóre dają większe możliwości i umożliwiają np. reprezentowanie całych scen, kamer, światel oraz animacji. Szczegółowy opis tych formatów (i narzędzi do tworzenia plików) znajduje się w rozdziale 8., poświęconym procesowi tworzenia treści.



Większość kodu dotyczącego wczytywania plików znajduje się poza rdzeniem biblioteki Three.js, w przykładach. Dlatego każdy mechanizm wczytujący należy dodać do projektu osobno, kiedy jest potrzebny. Jeśli nie napisano inaczej, narzędzia wczytujące podlegają takiej samej licencji, co biblioteka, więc można ich używać bez ograniczeń we własnych programach.

Graf sceny i hierarchia przekształceń

W WebGL nie ma standardowej notacji do określania struktury sceny trójwymiarowej. Biblioteka ta to po prostu API do rysowania na kanwie i nic więcej, a struktura sceny jest sprawą konkretnej aplikacji. W Three.js istnieje model do tworzenia struktury sceny oparty na ugruntowanej koncepcji **grafu sceny** (ang. *scene graph*). Graf sceny to zbiór trójwymiarowych obiektów przechowywanych w hierarchii obiekt nadzędny-obiekt podrzędny, której podstawę stanowi **korzeń**. Aplikacja renderuje graf sceny poprzez wyrenderowanie korzenia, a następnie rekurencyjnie dodaje jego obiekty podrzędne.

Zarządzanie sceną za pomocą grafu sceny

Grafy sceny są najbardziej przydatne do reprezentowania złożonych obiektów w hierarchiach. Wyobraź sobie np. robota, pojazd albo układ słoneczny. Każdy z tych obiektów składa się z pewnej liczby części — kończyn, nóg, satelitów — z których każda zachowuje się we właściwy sobie sposób. Graf sceny umożliwia, w zależności od potrzeb, traktowanie tych wszystkich obiektów jak indywidualnych części albo jak stanowiących całość grup. Nie jest to jedynie udogodnienie organizacyjne, lecz także technika umożliwiająca skorzystanie z tzw. **hierarchii przekształceń** (ang. *transform hierarchy*), w której obiekt podrzędny dziedziczy informacje dotyczące przekształceń trójwymiarowych (np. przesunięcia, obrotu, skalowania) po obiekcie nadzędznym. Powiedzmy przykładowo, że tworzymy animację samochodu poruszającego się po określonej ścieżce. Bryła samochodu przesuwa się w wyznaczonym kierunku, ale jego koła kręczą się w sposób niezależny od niej. Kiedy zdefiniujemy koła jako **obiekty podrzędne** względem bryły samochodu, sprawimy, że będą one poruszać się przez trójwymiarową przestrzeń wraz z całym samochodem. W ten sposób unikniemy konieczności animowania ruchu kół i wystarczy, że zdefiniujemy ich rotację.

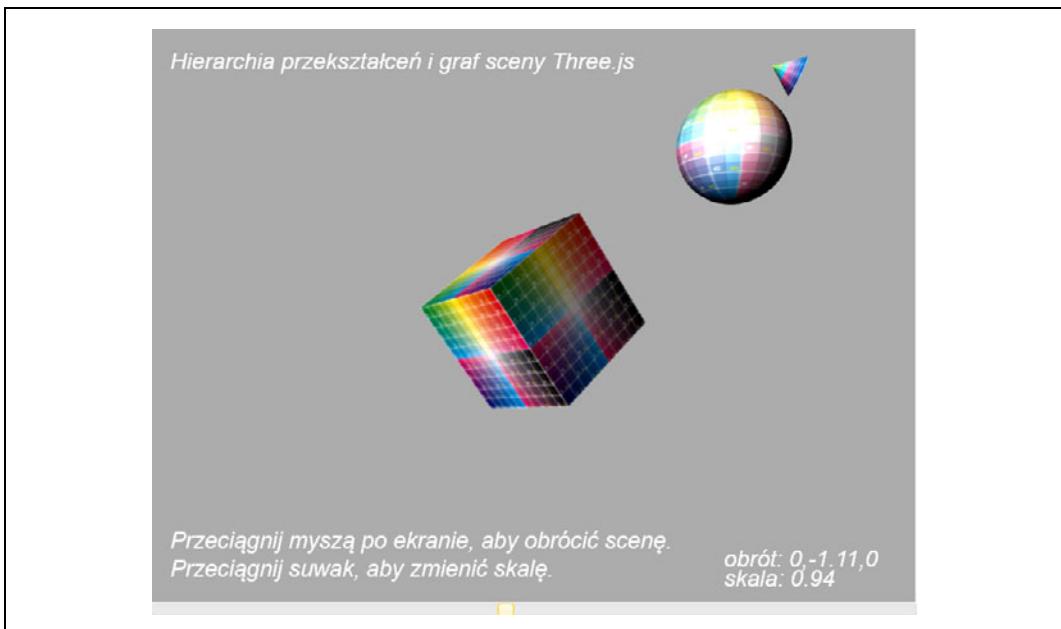


Określenie **graf** sceny w Three.js nie jest najlepsze. W technikach renderowania grafiki trójwymiarowej graf to pojęcie odnoszące się do **skierowanego grafu acyklicznego** (DAG — ang. *directed acyclic graph*), matematycznego bytu oznaczającego zbiór węzłów w relacji rodzic-dziecko, w której każde dziecko może mieć wielu rodziców. W grafie sceny biblioteki Three.js obiekty mogą mieć tylko jednego rodzica (obiekt nadzędny). Podczas gdy zasadniczo nie jest błędem nazywanie tej hierarchii grafem, lepsze byłoby określenie jej jako **drzewo**. Więcej informacji na temat grafów w matematyce znajduje się w Wikipedii (http://en.wikipedia.org/wiki/Directed_acyclic_graph).

Grafy sceny w Three.js

Podstawowy obiekt grafu sceny w bibliotece Three.js jest typu THREE.Object3D (plik *src/core/Object3D.js* w źródłach projektu Three.js). Klasa ta stanowi bazę dla typów wizualnych, takich jak siatki, linie i systemy cząsteczek, również jest używana do grupowania innych obiektów w hierarchię grafu sceny.

Każdy obiekt klasy `Object3D` we właściwościach `position` (przesunięcie), `rotation` oraz `scale` zawiera informacje dotyczące przekształceń. Ustawiając ich wartości, można dany obiekt przesunąć, obrócić i przeskalać. Jeżeli obiekt ma potomków (dzieci i ich dzieci), dziedziczą one jego ustawienia przekształceń. Jeśli właściwości przekształceń obiektów podległych zostaną zmienione, połączą się z ustawieniami obiektu nadrzędnego i ma to zastosowanie do samego dołu hierarchii. Oto przykład. Na rysunku 4.7 widać bardzo prostą hierarchię przekształceń. Kostka (`cube`) jest bezpośrednim potomkiem grupy kostek (`cubeGroup`). Grupa sfer (`sphereGroup`) również jest bezpośrednim potomkiem grupy kostek (a więc obiektem równorzędnym z kostką). Natomiast sfera (`sphere`) i stożek (`cone`) to potomkowie grupy sfer.



Rysunek 4.7. Graf sceny i hierarchia przekształceń `Three.js`

Przykład przedstawiony na powyższym rysunku znajduje się w pliku `r4/threejsscene.html`. Wyświetla on w oknie przeglądarki kostkę, sferę i stożek; każdy z tych obiektów obraca się w miejscu. Sceną można poruszać, klikając na niej i przeciągając ją myszą, oraz można zmieniać jej skalę za pomocą znajdującego się na dole suwaka.

Na listingu 4.2 przedstawiony jest kod źródłowy dotyczący tworzenia oraz obsługi tego grafu sceny i jego hierarchii przekształceń. Najważniejsze fragmenty zostały pogrubione. Aby zbudować scenę, utworzono obiekt klasy `Object3D` o nazwie `cubeGroup`, który służy jako korzeń całego grafu. Następnie bezpośrednio do niego dodano siatkę kostki oraz kolejny obiekt klasy `Object3D`, o nazwie `sphereGroup`. Do tego nowego obiektu dodano stożek i sfery. Ponadto przesunięto nieco stożek do góry i oddalono go od sfery za pomocą odpowiedniego ustawienia właściwości `position`.

Listing 4.2. Scena z hierarchią przekształceń

```
function animate() {
    var now = Date.now();
    var deltat = now - currentTime;
```

```

currentTime = now;
var fract = deltat / duration;
var angle = Math.PI * 2 * fract;

// Obraca kostkę wokół osi Y.
cube.rotation.y += angle;

// Obraca grupę sfery wokół osi Y.
sphereGroup.rotation.y -= angle / 2;

// Obraca stożek wokół osi X (do przodu).
cone.rotation.x += angle;
}

function createScene(canvas) {

// Tworzy renderer Three.js i wiąże go z kanwą.
renderer = new THREE.WebGLRenderer( { canvas: canvas, antialias: true } );

// Ustawia rozmiar obszaru widoku.
renderer.setSize(canvas.width, canvas.height);

// Tworzy nową scenę Three.js.
scene = new THREE.Scene();

// Dodaje kamerę, aby można było oglądać scenę.
camera = new THREE.PerspectiveCamera( 45, canvas.width / canvas.height,
    1, 4000 );
camera.position.z = 10;
scene.add(camera);

// Tworzy grupę do przechowywania wszystkich obiektów.
cubeGroup = new THREE.Object3D();

// Dodaje światło kierunkowe, aby pokazać obiekty.
var light = new THREE.DirectionalLight( 0xffffff, 1.5 );
// Pozycjonuje światło od sceny, aby wskazywało na jej początek.
light.position.set(.5, .2, 1);
cubeGroup.add(light);

// Tworzy teksturowany materiał typu Phong dla kostki.
// Najpierw tworzy tekstury.
var mapUrl = "../images/ash_uvgrid01.jpg";
var map = THREE.ImageUtils.loadTexture(mapUrl);
var material = new THREE.MeshPhongMaterial({ map: map });

// Tworzy geometrię kostki.
var geometry = new THREE.CubeGeometry(2, 2, 2);

// Wstawia geometrię kostki i materiał do siatki.
cube = new THREE.Mesh(geometry, material);

// Pochyla siatkę w kierunku użytkownika.
cube.rotation.x = Math.PI / 5;
cube.rotation.y = Math.PI / 5;

// Dodaje siatkę kostki do grupy.
cubeGroup.add( cube );

// Tworzy grupę dla sfery.
sphereGroup = new THREE.Object3D();
cubeGroup.add(sphereGroup);
}

```

```

// Przesuwa grupę sfery do góry i tyłu względem kostki.
sphereGroup.position.set(0, 3, -4);

// Tworzy geometrię sfery.
geometry = new THREE.SphereGeometry(1, 20, 20);

// Wstawia geometrię sfery i materiał do siatki.
sphere = new THREE.Mesh(geometry, material);

// Dodaje siatkę sfery do grupy.
sphereGroup.add( sphere );

// Tworzy geometrię stożka.
geometry = new THREE.CylinderGeometry(0, .333, .444, 20, 5);

// Wstawia geometrię stożka i materiał do siatki.
cone = new THREE.Mesh(geometry, material);

// Przesuwa stożek do góry i oddala go nieco od sfery.
cone.position.set(1, 1, -.667);

// Dodaje siatkę stożka do grupy.
sphereGroup.add( cone );

// Dodaje grupę do sceny.
scene.add( cubeGroup );
}

function rotateScene(deltax)
{
    cubeGroup.rotation.y += deltax / 100;
    $("#rotation").html("obrót: 0," + cubeGroup.rotation.y.toFixed(2) + ",0");
}

function scaleScene(scale)
{
    cubeGroup.scale.set(scale, scale, scale);
    $("#scale").html("skala: " + scale);
}

```

Kolejna animacja. W funkcji `animate()` widać, że gdy obraca się obiekt `sphereGroup`, obraca się sfera oraz stożek krąży w przestrzeni wokół niej. Zwróć uwagę, że nie ma kodu obracającego siatkę sfery ani poruszającego stożkiem. Obiekty te automatycznie odziedziczyły swoje przekształcenia po `sphereGroup`. Także implementacja interakcji ze sceną w celu jej obrócenia i skalowania jest banalnie prosta. Po prostu ustawiliśmy właściwości `rotation` i `scale` obiektu `cubeGroup`, a zmiany te zostały przez bibliotekę automatycznie przekazane do obiektów podrzędnych.

Reprezentowanie przesunięcia, obrotu i skali

W bibliotece Three.js przekształcenia wykonuje się przy użyciu obliczeń arytmetycznych na trójwymiarowych macierzach, więc nic dziwnego, że składnikami przekształceń obiektów klasy `Object3D` są trójwymiarowe wektory: `position`, `rotation` oraz `scale`. Znaczenie wartości wektora `position` jest oczywiste: są to składniki x , y i z określające jego przesunięcie względem początku obiektu. Wektor `scale` też jest prosty: wartości x , y i z wykorzystuje się do pomnożenia skali macierzy przekształcenia w każdym z trzech wymiarów.

Natomiast składniki wektora `rotation` wymagają nieco szerszego objaśnienia. Każda z wartości x , y i z definiuje obrót wokół odpowiedniej osi. Przykładowo wartość $(0, \text{Math.PI}/2, 0)$ oznacza obrót o 90 stopni wokół osi y (zwróć uwagę, że stopnie są wyrażone w radianach, a więc

$2^*\pi$ wynosi 360 stopni). Ten rodzaj obrotu — złożenie obrotów wokół osi x , y i z — nazywa się **kątem Eulera**. Podejrzewam, że Mr.doob wybrał właśnie tę technikę jako podstawową reprezentację, ponieważ jest intuicyjna i łatwa w zastosowaniu. Jednak wiążą się z nią pewne matematyczne problemy. Dlatego w bibliotece Three.js do określania kątów można również używać **kwaternionów**, które są pozbawione problemów kątów Eulera, ale za to wymagają więcej pracy programistycznej. Kwaterniony są precyzyjne, ale trudniejsze w użyciu.

Wewnętrznie biblioteka Three.js tworzy macierz z własności przekształceń każdego obiektu klasy `Object3D`. Macierze obiektów mających wielu przodków są pomnożone przez macierze tych przodków w sposób rekurencyjny, tzn. Three.js przechodzi w dół do każdego liścia drzewa grafu sceny i oblicza macierz przekształceń dla każdego obiektu na scenie w każdym przebiegu renderowania. W przypadku głębokich i skomplikowanych grafów obliczeń do wykonania może być bardzo dużo i dlatego dla obiektów klasy `Object3D` zdefiniowano własność `matrixAutoUpdate`, którą można ustawić na `false`, aby uniknąć narzutu. Niestety, korzystanie z tego udogodnienia może powodować subtelne błędy („Dlaczego moja animacja się nie aktualizuje?”), więc należy z niego korzystać bardzo ostrożnie.

Materiały

Kształty, które oglądamy w aplikacjach WebGL, mają pewne właściwości powierzchni, takie jak kolor, cieniowanie i tekstura (mapa bitowa). Tworzenie tych właściwości przy użyciu niskopoziomowych wywołań API WebGL wymaga pisania shaderów w języku GLSL oraz posiadania zaawansowanych umiejętności programistycznych nawet wtedy, kiedy tylko chce się zrobić coś prostego. Na szczęście, biblioteka Three.js zawiera gotowy do użytku kod GLSL w obiektach zwanych **materiałami** (ang. *materials*).

Standardowe materiały siatki

Przypomnę, że posługujący się biblioteką WebGL programista, który chce narysować jakikolwiek obiekt, musi dostarczyć shader. Z pewnością zauważyleś też, że do tej pory w tym rozdziale nie pokazałem jeszcze ani jednego wiersza kodu shadera. To nie jest niedopatrzenie. Biblioteka Three.js tworzy shadery automatycznie, ponieważ zawiera zbiór gotowych fragmentów kodu GLSL przeznaczonych do różnych zastosowań.

Tradycyjne biblioteki oparte na grafach scen i popularne programy do modelowania reprezentują shadery przy użyciu tzw. **materiałów**. Materiał to obiekt definiujący właściwości powierzchni trójwymiarowej siatki, punktu lub linii, takie jak kolor, przezroczystość oraz polaskliwość. Materiały mogą też zawierać tekstury, czyli mapy bitowe nawinięte na powierzchnię obiektów. Właściwości materiałów łączą się z danymi wierzchołków siatki, informacjami dotyczącymi oświetlenia na scenie oraz pozycją kamery i innymi globalnymi właściwościami, w efekcie czego powstaje ostateczna postać każdego obiektu.

Biblioteka Three.js obsługuje najczęściej używane typy materiałów w klasach `MeshBasicMaterial`, `MeshPhongMaterial` oraz `MeshLambertMaterial`. (Przedrostek `Mesh` oznacza, że obiekty tych typów powinny być używane w połączeniu z obiektami siatki, a więc nie z liniami czy częsteczkami; istnieją też specjalne typy materiałów przeznaczone do użytku z innymi typami obiektów. Kompletny i najbardziej aktualny zestaw obiektów znajduje się w kodzie źródłowym w folderze `src/materials`). Te trzy typy materiałów implementują odpowiednio trzy poniższe, dobrze znane techniki materiałowe.

Brak oświetlenia (lub oświetlenie gotowe)

W tym typie materiału do renderowania powierzchni obiektu używane są tylko tekstury, kolory oraz poziom przezroczystości. Nie jest brane pod uwagę oświetlenie sceny. Jest to doskonały rodzaj materiału do renderowania płaskich obiektów i rysowania prostych obiektów geometrycznych bez cieniowania. Ponadto można go używać, gdy oświetlenie obiektów jest wliczane w tekstury przed uruchomieniem programu (np. przez narzędzie do modelowania trójwymiarowego), a więc nie musi być obliczane przez renderer.

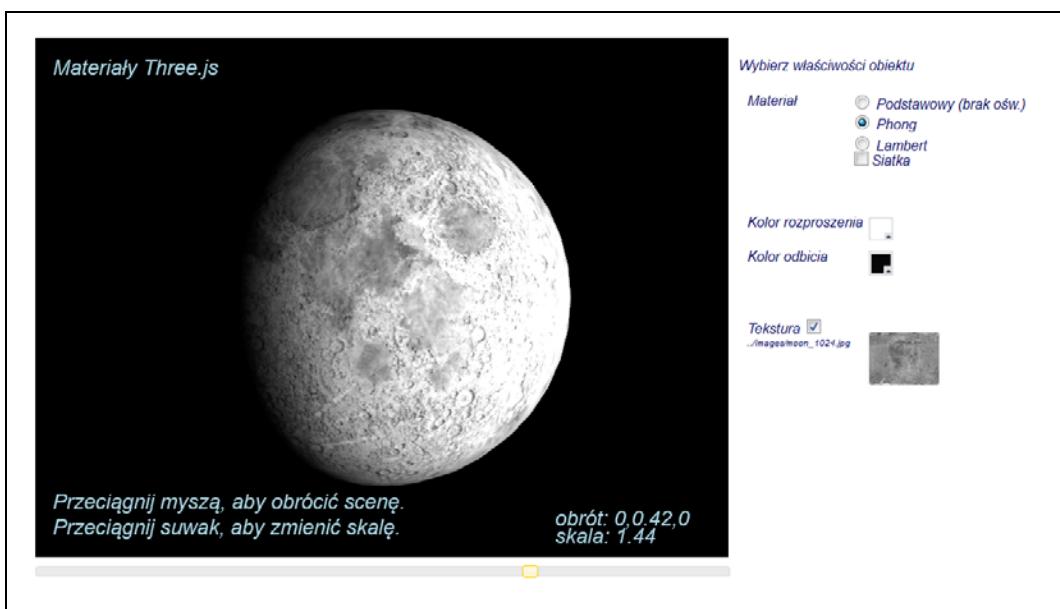
Cieniowanie Phonga

Ten typ materiału implementuje prosty, ale dość realistyczny model cieniowania i jest bardzo wydajny. Jest najczęściej wybierany, gdy trzeba szybko i bez nadmiaru pracy uzyskać klasyczny cieniowany wygląd. Wykorzystuje się go w wielu grach i aplikacjach. Obiekty cieniowane tą techniką mają jasno oświetlone obszary (**refleksy**) w miejscach, na które bezpośrednio pada światło, są dobrze oświetlone na krawędziach zwróconych w kierunku światła oraz mają ciemne cienie na krawędziach, które są odwrócione od światła.

Cieniowanie Lambert'a

W cieniowaniu Lambert'a jasność powierzchni dla obserwatora jest taka sama, niezależnie od kąta patrzenia. Bardzo dobrze sprawdza się w przypadku chmur, które rozpraszają dochodzące do nich światło, oraz satelitów, takich jak księżyc, które mają wysokie **albedo** (odbijają dużo światła od powierzchni).

Aby zobaczyć, jak wyglądają różne rodzaje materiałów, otwórz plik *r4/threematerials.html*. Na rysunku 4.8 pokazano jasno oświetloną sferę z nałożoną tekturem imitującą powierzchnię księżyca. Książyc jest bardzo dobrym obiektem do przedstawienia różnic między różnymi typami materiałów. Za pomocą przełączników można wybierać rodzaj materiału (np. *Phong* albo *Lambert*), aby sprawdzić, który jest w tym przypadku lepszy. Wybierając ustawienie *Podstawowy*, można zobaczyć samą teksturę, bez oświetlenia.



Rysunek 4.8. Standardowe materiały siatki biblioteki Three.js: Podstawowy (brak oświetlenia), Phong i Lambert

Zmień kolory rozproszenia i odbicia, aby zobaczyć, co się stanie. **Kolor rozproszenia** materiału określa, jak bardzo obiekt odbija źródła światła rzucające promienie w określonym kierunku — tzn. kierunkowe, punktowe i reflektorowe (opis rodzajów oświetlenia znajduje się dalej w tym rozdziale). **Kolor odbicia** łączy się ze światłami sceny w celu utworzenia refleksów odbitych od wierzchołków obiektu skierowanych ku źródłom światła. (Refleksy są widoczne tylko na materiałach typu Phong, w innych typach materiałów nie są obsługiwane). Ponadto wyłącz teksturę, usuwając zaznaczenie pola wyboru *Tekstura*, aby zobaczyć, jaki wpływ mają materiały na samą geometrię bryły. Na koniec zobacz też, jaki wpływ mają różne ustawienia na samą siatkę.

Dodawanie realizmu poprzez zastosowanie kilku tekstur

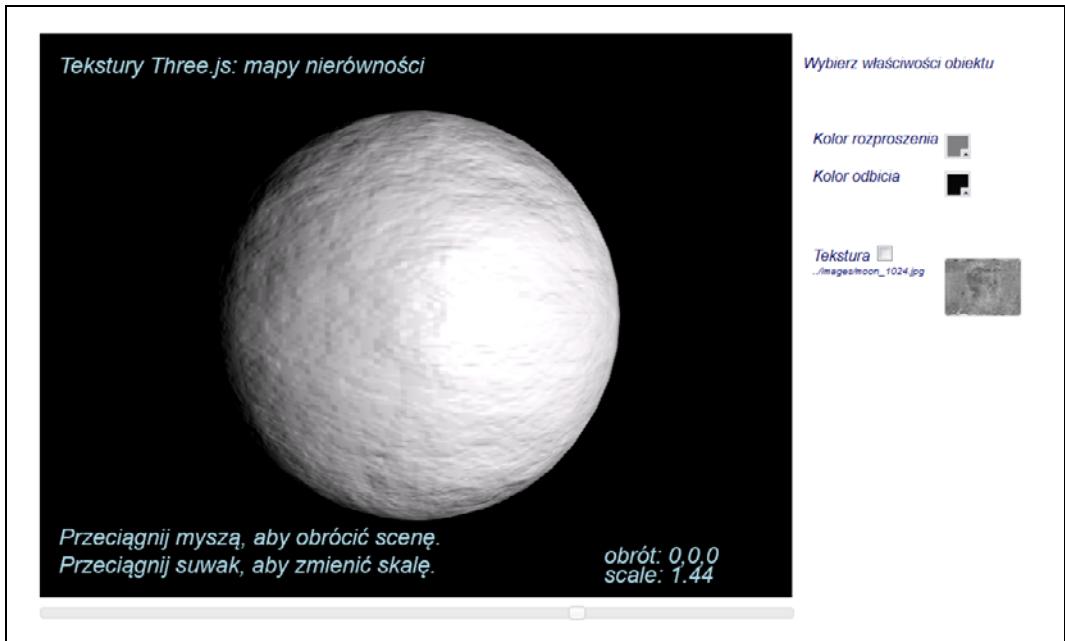
W poprzednim przykładzie pokazałem, jak zdefiniować wygląd powierzchni obiektu za pomocą tekstury. Większość typów materiałów w bibliotece Three.js umożliwia stosowanie wielu teksturow, co pozwala osiągnąć bardziej realistyczny efekt. W technice tej, zwanej **multiteksturowaniem**, chodzi o zwiększenie realizmu bez wykonywania nadmiernej ilości dodatkowych obliczeń. Alternatywą jest użycie większej liczby wielokątów lub wyprowadzenie obiektu w kilku przebiegach. Oto kilka przykładów ilustrujących najczęściej stosowane techniki multiteksturowania obsługiwane przez Three.js.

Mapy nierówności to mapy bitowe służące do przemieszczania wektorów normalnych powierzchni siatki w celu, jak sama nazwa wskazuje, utworzenia imitacji nierównej nawierzchni. Wartości pikseli mapy bitowej są traktowane nie jako wartości kolorów, lecz jako wysokości. Przykładowo wartość zero oznacza brak przemieszczenia względem powierzchni, a wartości różne od zera mogą oznaczać odsunięcie od powierzchni. Najczęściej ze względu na wydajność używa się jednokanałowych czarnych i białych map bitowych, chociaż można też wykorzystać mapy RGB, aby dostarczyć więcej szczegółów w większej liczbie wartości. Używa się map bitowych zamiast trójwymiarowych wektorów, ponieważ są bardziej kompaktowe i pozwalają na szybkie obliczanie przemieszczenia normalnych w kodzie shadera. Jeśli chcesz zobaczyć efekt działania mapy nierówności, otwórz plik *r4/threejsbumpmap.html* (rysunek 4.9). Włącz i wyłącz główną tekture księżyca oraz pozmieniaj wartości kolorów rozproszenia i odbicia. Zauważysz, że efekty wprawdzie są ciekawe, ale mogą powstawać nieprzyjemne artefakty. Mimo to, mapy nierówności są dobrym sposobem na zwiększenie realizmu obrazu.

Używanie map nierówności w bibliotece Three.js jest bardzo łatwe. Wystarczy przekazać teksturę we właściwości *bumpMap* obiektu parametrów przekazywanego do konstruktora klasy THREE.MeshPhongMaterial.

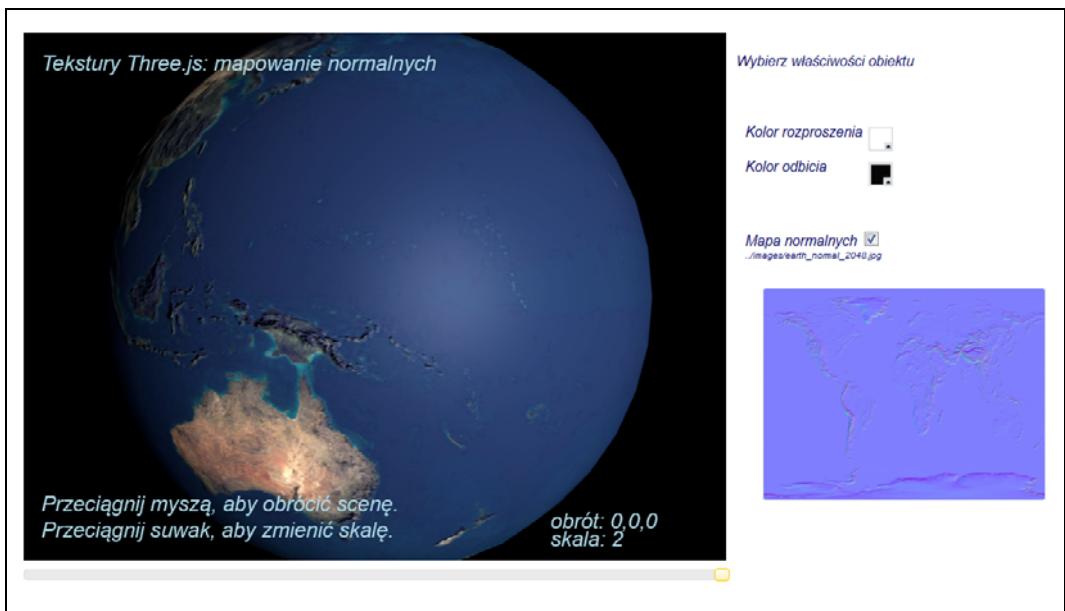
```
material = new THREE.MeshPhongMaterial({map: map,
  bumpMap: bumpMap});
```

Mapy normalnych to technika umożliwiająca przekazanie jeszcze większej ilości szczegółów dotyczących powierzchni niż mapy nierówności i również nie wymaga dodawania wielokątów. Mapy normalnych są zazwyczaj większe i wymagają większej mocy przetwarzania niż mapy nierówności, ale dodatkowe uzyskiwane dzięki nim szczegóły mogą być tego warte. W mapach takich koduje się wartości wektorów normalnych w mapach bitowych jako dane RGB, zazwyczaj stosując znacznie większą rozdzielcość niż w danych wierzchołków siatki. Shader wprowadza te informacje normalne do swoich obliczeń oświetlenia (wraz z bieżącymi wartościami kamery i źródła światła) w celu otrzymania szczegółowej powierzchni. Efekt działania mapy normalnych można obejrzeć, otwierając plik *r4/threensnormalmap.html*. Użyta mapa normalnych



Rysunek 4.9. Mapowanie nierówności

jest widoczna na dole po prawej (rysunek 4.10). Zwróć uwagę na zarysy wzgórz Ziemi. Włącz i wyłącz mapę normalnych, aby zobaczyć, jak wiele szczegółów dzięki niej zostaje dodanych do obrazu. To zadziwiające, jak bardzo mapa bitowa może zmienić taki prosty obiekt jak sfera.



Rysunek 4.10. Ziemia z mapą normalnych

W bibliotece Three.js mapy normalnych są łatwe w użyciu. Wystarczy przekazać teksturę we właściwości `normalMap` obiektu parametrów przekazywanego do konstruktora klasy `THREE.MeshPhongMaterial`.

```
Material = new THREE.MeshPhongMaterial({ map: map,
    normalMap: normalMap});
```

Mapowanie środowiskowe to kolejna technika umożliwiająca zastosowanie dodatkowych tekstur w celu zwiększenia realizmu obrazu. W odróżnieniu od map nierówności i normalnych, w których dodaje się szczegóły powierzchni przez pozorne zmiany w geometrii, w mapach środowiskowych symuluje się refleksy od obiektów w otaczającym środowisku.

Przykład zastosowania mapowania środowiskowego można obejrzeć, otwierając plik `r4/threejsenvmap.html`. Przeciągnij myszą w obszarze treści, aby obrócić scenę, oraz pokręć kółkiem myszy, aby ją zmniejszyć lub powiększyć. Zwróć uwagę, jak obraz znajdujący się na powierzchni sfery sprawia wrażenie, jakby odbijał otaczające niebo (rysunek 4.11). W istocie nic takiego nie ma miejsca. Po prostu na sferze wyrenderowano piksele z tej samej tekstury, która jest nałożona wewnętrz kostki użytej jako tło sceny. Sztuczka polega na tym, że na materiale sfery użyto **tekstury sześciennnej**, czyli utworzonej z sześciu odrębnych map bitowych połączonych w jeden obraz wewnętrz sześcianu. W tym przykładzie utworzono w ten sposób tło ilustrujące niebo. Poszczególne pliki składające się na ten produkt znajdują się w folderze `images/cubemap/skybox`. Ten rodzaj mapowania środowiskowego nazywa się **sześciennym mapowaniem środowiskowym**, ponieważ używa się w nim tekstur sześciennych.



Rysunek 4.11. Sześcienne mapy środowiskowe umożliwiają uzyskanie realistycznych tła scen i efektów odbicia

Używanie tekstur sześciennych w Three.js nie jest tak łatwe jak map nierówności i normalnych. Najpierw należy utworzyć teksturę sześcienną przy użyciu funkcji `ImageUtils.loadTextureCube()`, której przekazuje się adresy URL sześciu obrazów. Następnie ustawia się ją jako wartość parametru `envMap` obiektu `MeshPhongMaterial` przy wywoływaniu konstruktora. Ponadto określa się wartość `reflectivity` definiującą, jaka ilość tekstury sześciennej ma zostać „odbita” na materiale.

W tym przypadku podana została nieco większa wartość niż domyślna 1, aby mapa była dobrze widoczna.

```
var path = "../images/cubemap/skybox/";

var urls = [ path + "px.jpg", path + "nx.jpg",
            path + "py.jpg", path + "ny.jpg",
            path + "pz.jpg", path + "nz.jpg" ];

envMap = THREE.ImageUtils.loadTextureCube( urls );
materials["phong-envmapped"] = new THREE.MeshBasicMaterial(
    { color: 0xffffffff,
      envMap : envMap,
      reflectivity:1.3} );
```

Jest jeszcze jedna rzecz do zrobienia. Aby efekt był realistyczny, odbijana mapa bitowa musi zgadzać się z otaczającym ją środowiskiem. Żeby tak było, tworzymy **pudło nieba** (ang. *skybox*), czyli duży sześcian wyłożony od środka tekstrurą z tych samych obrazów reprezentujących panoramę nieba. Zrobienie tego normalnie wymagałoby bardzo dużo pracy, ale — na szczęście — biblioteka Three.js zawiera wbudowaną funkcję pomocniczą, która nas wyręczy. Oprócz standardowych materiałów Basic, Phong i Lambert, biblioteka Three.js zawiera w *THREE.ShaderLib* zbiór shaderów pomocniczych. Wystarczy utworzyć siatkę z geometrii sześcianu i jako materiału użyć shadera *cube*. Shader ten automatycznie zajmie się renderowaniem wewnętrz kostki przy użyciu tej samej tekstury, którą wykorzystaliśmy do utworzenia mapy środowiskowej.

```
// Tworzy pudło nieba.
var shader = THREE.ShaderLib[ "cube" ];
shader.uniforms[ "tCube" ].value = envMap;

var material = new THREE.ShaderMaterial( {

    fragmentShader: shader.fragmentShader,
    vertexShader: shader.vertexShader,
    uniforms: shader.uniforms,
    side: THREE.BackSide
} ),

mesh = new THREE.Mesh(new THREE.CubeGeometry( 500, 500, 500 ), material);
scene.add( mesh );
```

Oświetlenie

Światła oświetlają przedmioty znajdujące się na trójwymiarowej scenie. W bibliotece Three.js znajdują się definicje kilku klas oświetleniowych, podobnych do tych, które można znaleźć w typowych narzędziach do modelowania i bibliotekach grafów scen. Do najczęściej używanych rodzajów oświetlenia należą: **światło kierunkowe**, **światło punktowe**, **światło reflektowe** oraz **światło otaczające**.

Światło kierunkowe

Światło kierunkowe rzuca równoległe promienie w określonym, jednym kierunku. Nie ma pozycji, a jedynie kierunek, kolor i intensywność. (W istocie w bibliotece Three.js światła kierunkowe *mają* pozycję, ale jest ona używana wyłącznie do obliczania kierunku światła przy użyciu drugiego wektora, określającego pozycję docelową. Jest to niezgrabne i nieintuicyjne rozwiązanie, które — mam nadzieję — Mr.doob w przyszłości poprawi).

Światło punktowe

Światło punktowe ma pozycję, ale nie ma kierunku. Rzuca promienie we wszystkich kierunkach na określona odległość.

Światło reflektorowe

Światło reflektorowe ma pozycję i kierunek. Ponadto można określić jego parametry, takie jak rozmiar (kąt) wewnętrznego i zewnętrznego stożka reflektora oraz odległość, na jaką sięga oświetlenie.

Światło otaczające

Światło otaczające nie ma pozycji ani kierunku. Oświetla scenę równomiernie na całej powierzchni.

Wszystkie typy oświetlenia w Three.js mają własności `intensity` (definiuje intensywność światła) oraz `color` (wartość RGB).

Światła nie działają w pojedynkę. Ich wartości komponują się z właściwościami materiałów, w efekcie czego powstaje ostateczna postać powierzchni. Materiały `MeshPhongMaterial` i `MeshLambertMaterial` definiują następujące własności.

color

Własność zwana także **kolem rozproszonym** (ang. *diffuse color*) określa, jaką ilość światła świecącego w określonym kierunku (kierunkowego, punktowego, reflektorowego) odbija obiekt.

ambient

Ilość otaczającego światła sceny odbijana przez obiekt.

emissive

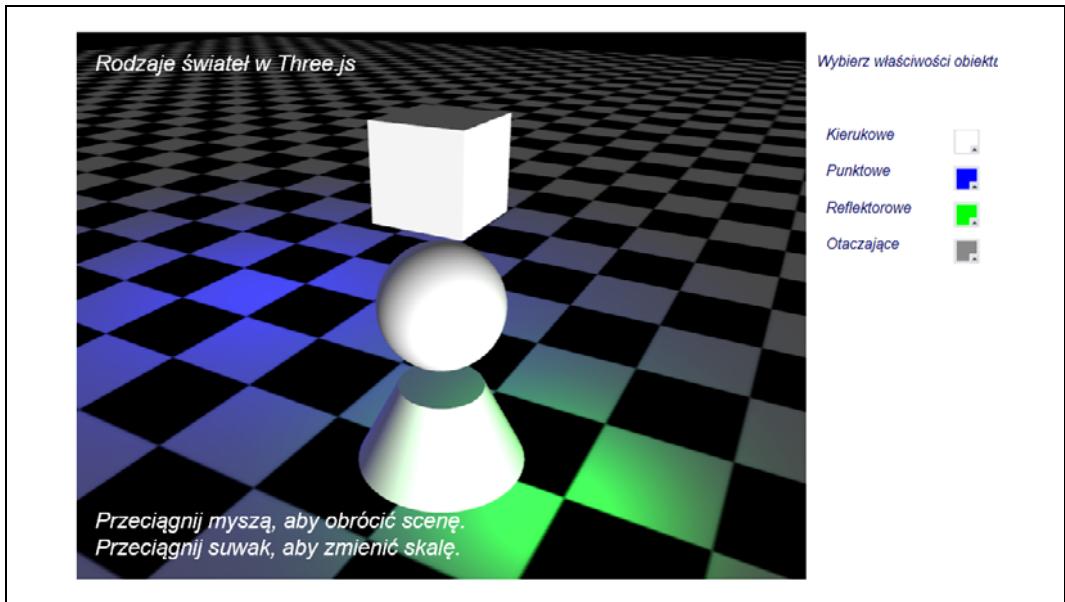
Ta własność materiału określa kolor emitowany przez obiekt, niezależnie od występujących na scenie źródeł światła.

Ponadto materiał `MeshPhongMaterial` obsługuje jeszcze kolor `specular` (refleks), który komponuje się z oświetleniem sceny w celu utworzenia refleksów odbitych od wierzchołków obiektu zwróconych ku źródłom światła.

Przypomnę tu, że `MeshBasicMaterial` w ogóle ignoriuje światło.

Na rysunku 4.12 przedstawiono eksperyment oświetleniowy zbudowany przy użyciu podstawowych typów oświetlenia. Aby go uruchomić, otwórz plik `r4/threelights.html`. Scena zawiera cztery źródła światła, po jednym każdego typu, ma tło pokryte tekstonią w czarno-białą kratę oraz zawiera trzy proste białe bryły geometryczne, na których można obserwować efekt działania różnych światel. Za pomocą próbników kolorów można zmieniać kolorystykę światel. Ustaw czarny kolor światła, a zobaczysz, że oświetlenie zostanie całkowicie wyłączone. Przeciągnij myszą po obszarze treści, aby obrócić scenę i zobaczyć efekt działania światel na różne części modelu.

Na poniższym listingu przedstawiony jest kod źródłowy dotyczący tworzenia światel. Białe światło kierunkowe umiejscowione przed sceną oświetla jaskrawe białe obszary znajdujące się na przedzie obiektów geometrycznych. Niebieskie światło punktowe świeci z tyłu modelu. Zwróci uwagę na niebieskie obszary na podłodze za obiektem. Niebieskie światło reflektorowe rzuca swój stożek w kierunku podłogi, w pobliżu przodu sceny, zgodnie z ustawieniem



Rysunek 4.12. Oświetlenie kierunkowe, punktowe, reflektorowe oraz otaczające

właściwości `spotLight.target.position`. W końcu światło otaczające oświetla nieznacznie, ale równomiernie wszystkie obiekty. Pozmieniaj kolory i poobracaj model, aby zobaczyć efekt działania różnych świateł osobno i połączeniu z innymi.

```
// Tworzy i dodaje wszystkie światła.  
directionalLight.position.set(.5, 0, 3);  
root.add(directionalLight);  
  
pointLight = new THREE.PointLight (0x0000ff, 1, 20);  
pointLight.position.set(-5, 2, -10);  
root.add(pointLight);  
  
spotLight = new THREE.SpotLight (0x00ff00);  
spotLight.position.set(2, 2, 5);  
spotLight.target.position.set(2, 0, 4);  
root.add(spotLight);  
  
ambientLight = new THREE.AmbientLight ( 0x888888 );  
root.add(ambientLight);
```

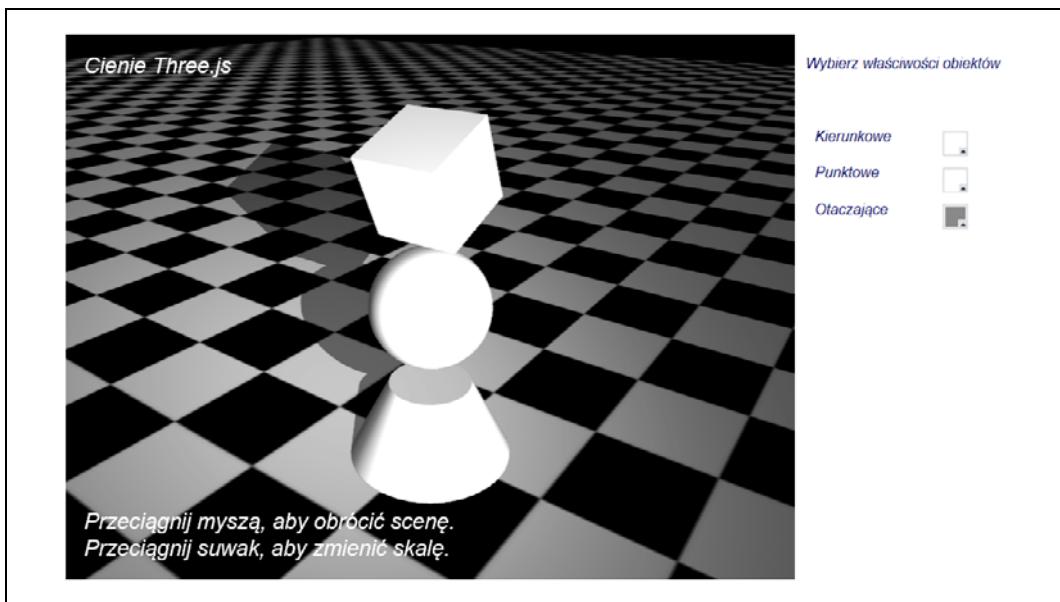


W tym momencie warto przypomnieć pewne znane już fakty. W WebGL oświetlenie, jak prawie wszystko, jest sztucznym tworem. Biblioteka zna tylko bufore i shadery, a zadaniem programisty jest syntezja efektów oświetleniowych za pomocą kodu shadera. Biblioteka Three.js zawiera osiąmaniającą ilość narzędzi do pracy z materiałami i światłem. A jeśli weźmie się pod uwagę, że wszystko to napisano w JavaScriptcie, tym bardziej należy docenić pracę twórcy. Oczywiście nic nie dałoby się zrobić, gdyby biblioteka WebGL nie dawała dostępu do GPU.

Cienie

Projektanci grafiki od lat wykorzystują cienie w celu uzyskania efektów wizualnych i zwiększenia realistmu obrazu. Najczęściej są to fałszywe, wyrenderowane zawczasu produkty i wystarczy poruszyć źródłem światła albo którymkolwiek z obiektów, aby zniweczyć cały efekt. Jednak w bibliotece Three.js istnieje możliwość renderowania cieni na bieżąco, w odniesieniu do bieżącego położenia światel i obiektów.

W przykładzie zawartym w pliku *r4/threejsshadows.html* zademonstrowany został sposób dodawania na bieżąco cieni do sceny. Spójrz na rysunek 4.13. Widoczne na nim obiekty rzucają cień na podłogę przy świetle reflektorowym świecącym z góry i przodu sceny. Cień przemieszcza się wraz z obracającą się kostką, ale nie ma na niego wpływu ruch podłogi. Gdyby cienie były tylko wcześniej wyrenderowaną imitacją, to byłyby „przyklejone” do podłogi i nie zmieniałyby położenia wraz z obracającym się obiektem. Pobaw się ustawieniami światel, zwłaszcza światła reflektorowego, aby zobaczyć, jak dynamicznie zmienia się cień.



Rysunek 4.13. Użycie światła reflektorowego i mapy cieni do tworzenia cieni na bieżąco

W bibliotece Three.js cienie powstają przy użyciu techniki o nazwie **mapowanie cieni** (ang. *shadow mapping*). Polega ona na tym, że renderer otrzymuje dodatkową teksturę, na której renderuje zaciemnione obszary, a następnie włącza ją do ostatecznego obrazu w shaderach fragmentów. Zatem włączenie cieni w Three.js wymaga wykonania kilku czynności. Oto one.

1. Włączenie mapowania cieni w rendererze.
2. Włączenie cieni i ustawienie parametrów cieniowania dla światel, które rzucają cienie. Można to zrobić dla typów światel THREE.DirectionalLight i THREE.SpotLight.
3. Wskazanie, które obiekty geometryczne rzucają i przyjmują cienie.

Zobaczmy, jak to wygląda w praktyce. Na listingu 4.3 przedstawiony jest kod dotyczący renderowania cieni (pogrubiony), dodany do funkcji `createScene()`.

Listing 4.3. Mapowanie cieni w bibliotece Three.js

```
var SHADOW_MAP_WIDTH = 2048, SHADOW_MAP_HEIGHT = 2048;

function createScene(canvas) {

    // Tworzy renderer Three.js i wiąże go z kanwą.
    renderer = new THREE.WebGLRenderer( { canvas: canvas, antialias: true } );

    // Ustawia rozmiar obszaru widoku.
    renderer.setSize(canvas.width, canvas.height);

    // Włącza cienie.
    renderer.shadowMapEnabled = true;
    renderer.shadowMapType = THREE.PCFSoftShadowMap;

    // Tworzy nową scenę Three.js.
    scene = new THREE.Scene();

    // Dodaje kamerę, aby można było oglądać scenę.
    camera = new THREE.PerspectiveCamera( 45, canvas.width / canvas.height,
        1, 4000 );
    camera.position.set(-2, 6, 12);
    scene.add(camera);

    // Tworzy grupę wszystkich obiektów.
    root = new THREE.Object3D();

    // Dodaje światło kierunkowe, aby pokazać obiekt.
    directionalLight = new THREE.DirectionalLight( 0xffffff, 1 );

    // Tworzy i dodaje wszystkie światła.
    directionalLight.position.set(.5, 0, 3);
    root.add(directionalLight);

    spotLight = new THREE.SpotLight ( 0xffffff );
    spotLight.position.set(2, 8, 15);
    spotLight.target.position.set(-2, 0, -2);
    root.add(spotLight);

    spotLight.castShadow = true;

    spotLight.shadowCameraNear = 1;
    spotLight.shadowCameraFar = 200;
    spotLight.shadowCameraFov = 45;

    spotLight.shadowDarkness = 0.5;

    spotLight.shadowMapWidth = SHADOW_MAP_WIDTH;
    spotLight.shadowMapHeight = SHADOW_MAP_HEIGHT;

    ambientLight = new THREE.AmbientLight ( 0x888888 );
    root.add(ambientLight);

    // Tworzy grupę wszystkich sfer.
    group = new THREE.Object3D();
    root.add(group);

    // Tworzy teksturę.
    var map = THREE.ImageUtils.loadTexture(mapUrl);
```

```

map.wrapS = map.wrapT = THREE.RepeatWrapping;
map.repeat.set(8, 8);

var color = 0xffffffff;
var ambient = 0x888888;
// Dodaje płaszczyzne podłogi, aby pokazać oświetlenie.
geometry = new THREE.PlaneGeometry(200, 200, 50, 50);
var mesh = new THREE.Mesh(geometry, new THREE.MeshPhongMaterial({color:color,
    ambient:ambient, map:map, side:THREE.DoubleSide}));
mesh.rotation.x = -Math.PI / 2;
mesh.position.y = -4.02;

// Dodaje siatkę do grupy.
group.add( mesh );
mesh.castShadow = false;
mesh.receiveShadow = true;

// Tworzy geometrię kostki.
geometry = new THREE.CubeGeometry(2, 2, 2);

// Dodaje geometrię i materiał do siatki.
mesh = new THREE.Mesh(geometry, new THREE.MeshPhongMaterial({color:color,
    ambient:ambient}));
mesh.position.y = 3;
mesh.castShadow = true;
mesh.receiveShadow = false;

// Dodaje siatkę do grupy.
group.add( mesh );

// Zapisuje kostkę, aby można było nią obracać.
cube = mesh;

// Tworzy geometrię sfery.
geometry = new THREE.SphereGeometry(Math.sqrt(2), 50, 50);

// Wstawia geometrię i materiał do siatki.
mesh = new THREE.Mesh(geometry, new THREE.MeshPhongMaterial({color:color,
    ambient:ambient}));
mesh.position.y = 0;
mesh.castShadow = true;
mesh.receiveShadow = false;

// Dodaje siatkę do grupy.
group.add( mesh );

// Tworzy geometrię cylindra.
geometry = new THREE.CylinderGeometry(1, 2, 2, 50, 10);

// Wstawia geometrię i materiał do siatki.
mesh = new THREE.Mesh(geometry, new THREE.MeshPhongMaterial({color:color,
    ambient:ambient}));
mesh.position.y = -3;

mesh.castShadow = true;
mesh.receiveShadow = false;

// Dodaje siatkę do grupy.
group.add( mesh );

// Dodaje grupę do sceny.
scene.add( root );
}

```

Najpierw włączymy cienie w rendererze za pomocą ustawienia `renderer.shadowMapEnabled` na `true` oraz jego własności `shadowMapType` na `THREE.PCFSoftShadowMap`. Biblioteka Three.js obsługuje trzy rodzaje algorytmu mapowania cieni: podstawowy, PCF (ang. *percentage close filtering*) oraz PCF soft shadows. Każdy kolejny pozwala uzyskać coraz bardziej realistyczny efekt, ale za cenę większej złożoności i obniżonej wydajności. Zmień w powyższym przykładzie ustawienie `shadowMapType` na `THREE.BasicShadowMap` i `Three.PCFShadowMap` i sprawdź, co się stanie. Jakość cieni znacznie się obniży, ponieważ zostaną zastosowane ustawienia niższej jakości. Przy renderowaniu skomplikowanych scen zastosowanie tej techniki może być konieczne ze względu na wydajność.

Następnie należy włączyć rzucanie cieni dla oświetlenia reflektorowego. W tym celu ustawiamy właściwość `castShadow` tego światła na `true`. Ponadto ustawiamy kilka parametrów wymaganych przez Three.js. Biblioteka ta renderuje cienie poprzez rzucanie promienia z pozycji światła w kierunku obiektu docelowego. Zasadniczo traktuje oświetlenie reflektorowe jak kolejną „kamerę” do renderowania sceny z pozycji. Dlatego musimy ustawiać parametry, tak jak dla kamery, włącznie z bliższą i dalszą płaszczyzną odcięcia oraz polem widzenia. Wartości płaszczyzn są w wysokim stopniu uzależnione od rozmiaru sceny i obiektów, więc zastosowaliśmy w miarę niewielkie liczby. Pole widzenia zostało określone metodą prób i błędów. Ponadto cieniowi należy ustawić wartość ciemności. Domyslnie wynosi ona `0.5` i jest to wartość odpowiednia w naszym przypadku. Następnie ustawiamy właściwości określające rozmiar mapy cieni. Jest to kolejna mapa bitowa tworzona przez Three.js, na której biblioteka renderuje zacienione obszary, które następnie mieszają z ostatecznym obrazem każdego obiektu. W naszym przykładzie ustawienia `SHADOW_MAP_WIDTH` i `SHADOW_MAP_HEIGHT` wynoszą `2048`, a więc o wiele więcej niż domyślna wartość biblioteki wynosząca `512`. Pozwoliło to uzyskać bardzo gładkie cienie. Niższe wartości dają bardziej poszarpane wyniki. Poeksperymentuj trochę z tymi wartościami w przykładzie, aby zobaczyć, jaki będzie efekt zastosowania map cieni o mniejszej rozdzielczości.

Na koniec musimy poinformować bibliotekę Three.js, które obiekty rzucają i przyjmują cienie. Domyslnie siatki nie obsługują cieni, więc trzeba to zmienić za pomocą odpowiednich ustawień. W tym przykładzie chcemy, aby obiekty geometryczne rzucaly cień na podłogę i podłoga przyjmowała cienie. W związku z tym, podłodze ustawiamy `mesh.castShadow` na `false`, a `mesh.receiveShadow` na `true`. Natomiast kostce, sferze i stożkowi definiujemy ustawienia `mesh.castShadow` na `true`, a `mesh.receiveShadow` na `false`.

Jako ostatni szlif ustawimy intensywność cienia w taki sposób, aby odpowiadała jasności rzucającego go oświetlenia reflektorowego. Algorytm mapowania cieni biblioteki Three.js nie uwzględnia automatycznie jasności źródeł światła. Zamiast tego używa właściwości `shadowDarkness` światła. Musimy zatem samodzielnie aktualizować tę właściwość wraz ze zmianą koloru oświetlenia przy użyciu interfejsu użytkownika. Poniżej znajduje się kod źródłowy funkcji pomocniczej `setShadowDarkness()`, która oblicza nową wartość ciemności cienia na podstawie średniej jasności składników czerwonego, zielonego i niebieskiego koloru światła. Gdy zmieni się kolor światła na ciemniejszy, cień stanie się bledszy.

```
function setShadowDarkness(light, r, g, b)
{
    r /= 255;
    g /= 255;
    b /= 255;
    var avg = (r + g + b) / 3;

    light.shadowDarkness = avg * 0.5;
}
```



Generowane na bieżąco cienie są fantastycznym dodatkiem do grafiki WebGL, a biblioteka Three.js znacznie ułatwia ich stosowanie. Jednak nie ma nic darmo. Po pierwsze, mapa cieni, która jest kolejną tektureą, zajmuje dodatkową pamięć graficzną. Mapa o rozmiarach 2048×2048 zajmuje 4 MB. Staraj się używać jak najmniejszych map cieni, które pozwalają uzyskać żądany efekt. Po drugie, w zależności od sprzętu graficznego, renderowanie pozaekranowe na mapie cieni może spowodować dodatkowe obciążenie systemu i znaczne zmniejszenie liczby klatek. Dlatego należy ostrożnie korzystać z tego udogodnienia. Przygotuj się na profilowanie i potencjalnie przejście na inne rozwiązanie, które nie wymaga obliczania cieni na bieżąco.

Shadery

Biblioteka Three.js zawiera bogaty zbiór gotowych materiałów zaimplementowanych przy użyciu shaderów GLSL. Shadery te służą do uzyskiwania typowych stylów cieniowania, takich jak cieniowanie bez oświetlenia, cieniowanie Phonga czy cieniowanie Lamberta. Jednak możliwości jest o wiele więcej. Ogólnie rzecz biorąc, materiały mogą implementować nieskończoną ilość efektów, używać najrozmaitszych właściwości, mogą też być bardzo skomplikowane. I tak shader imitujący trawę falującą na wietrze mógłby mieć parametry określające wysokość i gęstość trawy oraz szybkość i kierunek wiatru.

W miarę ewolucji grafiki komputerowej oraz obserwowanego od 20 lat wzrostu wartości produkcji — początkowo dotyczącej filmowych efektów specjalnych, a później także dla gier wideo — cieniowanie przestało być tylko artystycznym zajęciem i stało się ogólnym problemem programistycznym. Zamiast prób przewidywania każdej możliwej kombinacji właściwości materiałów i kodowania ich w silniku wykonawczym, specjalisci z branży opracowali programowalną technologię zwaną **programowalnymi shaderami** albo w skrócie po prostu **shaderami**. Shadery umożliwiają pisanie kodu implementującego skomplikowane efekty dla pojedynczych wierzchołków i pikseli przy użyciu kompilowanego języka podobnego do C i wykonywanego przez GPU. Za ich pomocą programista może utworzyć bardzo realistyczne i efektywne grafiki, wolne od ograniczeń wiążących się z używaniem wcześniej zdefiniowanych materiałów i modeli oświetlenia.

Klasa ShaderMaterial: zrób to sam

GL Shading Language (GLSL) to język cieniowania przeznaczony do użytku z bibliotekami OpenGL i OpenGL ES (na której bazuje API WebGL). Kod źródłowy w języku GLSL jest kompilowany i wykonywany na użytku WebGL przy użyciu metod obiektu kontekstu WebGL. Biblioteka Three.js ukrywa przed programistą kod GLSL, dzięki czemu można całkowicie ominąć krok pisania shaderów. W wielu aplikacjach gotowe typy materiałów są zupełnie wystarczające. Jeśli jednak chce się zastosować efekt, który nie jest standardowo dostępny, należy napisać własne shadery GLSL przy użyciu klasy THREE.ShaderMaterial.

Na rysunku 4.14 widać przykład działania klasy ShaderMaterial. Przykład ten, przedstawiający shader Fresnela, znajduje się w projekcie Three.js, w pliku *examples/webgl_materials_shaders_fresnel.html*. Cieniowanie Fresnela służy do symulowania odbicia i załamania światła przy zetknięciu z przezroczystym ciałem, takim jak woda i szkło.



Rysunek 4.14. Shader Fresnela pozwala uzyskać realistyczne efekty dzięki odbiciu i załamaniu światła



Shadery Fresnela zauważają nazwę efektowi Fresnela, zjawisku, które jako pierwszy opisał francuski fizyk Augustin-Jean Fresnel (1788 – 1827). Fresnel rozwinał teorię fal światła, badając sposób przechodzenia i rozprzestrzeniania się światła w różnych obiektach. Więcej informacji na ten temat można znaleźć w słowniku renderingu trójwymiarowego na stronie <http://www.3drender.com/glossary/fresneleffect.htm>.

Kod przygotowawczy w tym przykładzie tworzy obiekt klasy `ShaderMaterial` w następujący sposób: klonuje wartości `uniform` (parametry) obiektu szablonowego `FresnelShader` — każdy egzemplarz shadera musi mieć własną kopię tych danych — i przekazuje kod źródłowy GLSL dla shaderów wierzchołków i fragmentów. Następnie biblioteka `Three.js` automatycznie kompiluje i łączy shadery oraz wiąże własności JavaScript z wartościami `uniform`.

```
var shader = THREE.FresnelShader;
var uniforms = THREE.UniformsUtils.clone( shader.uniforms );

uniforms[ "tCube" ].value = textureCube;

var parameters = {
    fragmentShader: shader.fragmentShader,
    vertexShader: shader.vertexShader,
    uniforms: uniforms
};

var material = new THREE.ShaderMaterial( parameters );
```

Na listingu 4.4 pokazany jest kod GLSL shadera Fresnela (można go też znaleźć w pliku `examples/js/shaders/FresnelShader.js` projektu `Three.js`). Kod ten został napisany przez aktywnie wspierającego projekt `Three.js` programistę Branislava Ulicnego, lepiej znanego pod pseudonimem `AlteredQualia`. Przeanalizujmy ten kod, aby dowiedzieć się, jak działa.

Listing 4.4. Shader Fresnela biblioteki `Three.js`

```
/**
 * @author alteredq / http://alteredqualia.com/
 * Based on Nvidia Cg tutorial
 */
THREE.FresnelShader = {
```

```

uniforms: {

  "mRefractionRatio": { type: "f", value: 1.02 },
  "mFresnelBias": { type: "f", value: 0.1 },
  "mFresnelPower": { type: "f", value: 2.0 },
  "mFresnelScale": { type: "f", value: 1.0 },
  "tCube": { type: "t", value: null }
},

```

Własność uniforms klasy THREE.ShaderMaterial określa wartości, które Three.js przekaże do WebGL podczas używania shadera. Przypomnij, że kod shadera jest wykonywany dla każdego wierzchołka i piksela (fragmentu). Dane uniform (*jednolite*) shadera to wartości, które zgodnie z nazwą nie zmieniają się między wierzchołkami. Są to w istocie globalne zmienne o takiej samej wartości dla wszystkich wierzchołków i pikseli. Przedstawiony w tym przykładzie shader Fresnala definiuje dane jednolite sterujące odbiciem i załamaniem światła (np. mRefractionRatio i mFresnelScale). Ponadto shader ten definiuje zmienną jednolitą dla tekstury sześciennej używaną jako tło sceny. Podobnie jak w sześciennym mapowaniu środowiskowym przedstawionym w poprzednim podrozdziale, shader ten symuluje odbicie poprzez renderowanie pikseli z mapy sześciennej. Jednak w tym przypadku widoczne są nie tylko piksele odbite z mapy, ale również podlegające załamaniu światła.

Stosowanie kodu GLSL z biblioteką Three.js

Teraz należy zdefiniować shadery wierzchołków i fragmentów. Zaczniemy od shadera wierzchołków:

```

vertexShader: [

  "uniform float mRefractionRatio;",
  "uniform float mFresnelBias;",
  "uniform float mFresnelScale;",
  "uniform float mFresnelPower;",

  "varying vec3 vReflect;",
  "varying vec3 vRefract[3];",
  "varying float vReflectionFactor;",

  "void main() {",

    "vec4 mvPosition = modelViewMatrix * vec4( position, 1.0 );",
    "vec4 worldPosition = modelMatrix * vec4( position, 1.0 );",

    "vec3 worldNormal = normalize( mat3( modelMatrix[0].xyz, ",
    "      modelMatrix[1].xyz, modelMatrix[2].xyz ) * normal );",

    "vec3 I = worldPosition.xyz - cameraPosition;",

    "vReflect = reflect( I, worldNormal );",
    "vRefract[0] = refract( normalize( I ), worldNormal, ",
    "      mRefractionRatio );",
    "vRefract[1] = refract( normalize( I ), worldNormal, ",
    "      mRefractionRatio * 0.99 );",
    "vRefract[2] = refract( normalize( I ), worldNormal, ",
    "      mRefractionRatio * 0.98 );",
    "vReflectionFactor = mFresnelBias + mFresnelScale * ",
    "      pow( 1.0 + dot( normalize( I ), worldNormal ), ",
    "      mFresnelPower );",

```

```
        "gl_Position = projectionMatrix * mvPosition;",
    }"
].join("\n"),
```

Shader wierzchołków jest w przypadku tego materiału wołem roboczym. Wykorzystuje pozycję kamery i każdego wierzchołka modelu — w tym przypadku geometrię sfery tworzącą bańkę — do obliczenia wektora kierunkowego, który następnie zostaje użyty do obliczenia współczynników odbicia i załamania dla każdego wierzchołka. Zwróć uwagę na deklaracje varying, znajdujące się w shaderach wierzchołków i fragmentów. W odróżnieniu od zmiennych jednolitych (`uniform`), zmienne typu `varying` są obliczane osobno dla każdego wierzchołka i przekazywane z shadera wierzchołków do shadera fragmentów. W ten sposób shader wierzchołków może zwracać inne wartości oprócz wbudowanej `gl_Position`, której obliczanie jest jego podstawowym zadaniem. W shaderze Fresnela zwarcane dane `varying` dotyczą współczynników odbicia i załamania.

Shader wierzchołków Fresnela wykorzystuje ponadto kilka zmiennych typu `varying` i `uniform`, których nie widać w naszym kodzie, bo są zdefiniowane i automatycznie przekazywane do kompilatora GLSL przez bibliotekę Three.js. Są to: `modelMatrix`, `modelViewMatrix`, `projectionMatrix` oraz `cameraPosition`. Wartości tych nie trzeba i na dobrą sprawę nie powinno się jawnie deklarować w shaderze.

`modelMatrix (uniform)`

Macierz przekształcenia świata modelu (siatki). Jak napisałem w tym rozdziale, w podrozdziale „Graf sceny i hierarchia przekształceń”, macierz ta jest obliczana przez bibliotekę Three.js w każdej klatce, aby określić **pozycję obiektu w przestrzeni świata**. W shaderze macierz ta jest wykorzystywana do obliczania pozycji w przestrzeni świata każdego wierzchołka.

`modelViewMatrix (uniform)`

Przekształcenie reprezentujące pozycję każdego obiektu w przestrzeni kamery, tzn. we współrzędnych względnych do pozycji i orientacji kamery. Szczególnie przydatna do obliczania wartości odnoszących się do kamery (np. określania odbicia i załamania, co właśnie robimy w tym shaderze).

`projectionMatrix (uniform)`

Używana do obliczania rzutowania z trzech wymiarów na dwa wymiary, z przestrzeni kamery na przestrzeń ekranu.

`cameraPosition (uniform)`

Pozycja w przestrzeni świata kamery obsługiwana przez Three.js i przekazywana automatycznie.

`position (varying)`

Pozycja wierzchołka w przestrzeni modelu.

`normal (varying)`

Normalna wierzchołka w przestrzeni modelu.

Shader wierzchołków wykorzystuje również wbudowane funkcje GLSL, `reflect()` i `refract()`, do obliczania wektorów odbicia i załamania na podstawie kierunku kamery, normalnej i współczynnika załamania. (Funkcje te dodano do języka GLSL, ponieważ są bardzo przydatne w obliczeniach dotyczących oświetleniach, takich jak np. równania Fresnela).

Na końcu znajduje się jeszcze wywołanie funkcji `Array.join()` w celu konfiguracji shadera wierzchołków. Stanowi to ilustrację zastosowania kolejnej przydatnej techniki łączenia długich łańcuchów tekstowych, zawierających implementację shaderów w języku GLSL. Zamiast tworzyć symbole nowego wiersza na końcu każdej linijki kodu i stosować konkatenację, użyliśmy funkcji `join()`, aby wstawić znak nowego wiersza po każdym wierszu kodu.

Zadanie shadera fragmentów jest teraz oczywiste. Wykorzystuje on obliczone przez shader wierzchołków wartości odbicia i załamania do indeksowania w sześciennej teksturze przekazanej w zmiennej jednolitej `tCube`. Zmienna ta jest typu `samplerCube`, typu GLSL służącego do obsługi tekstur sześciennych. Mieszamy te dwa kolory przy użyciu funkcji GLSL `mix()`, aby otrzymać ostateczny piksel, który zapisujemy we wbudowanej zmiennej `gl_FragColor`.

```
fragmentShader: [  
  
    "uniform samplerCube tCube;,  
  
    "varying vec3 vReflect;,  
    "varying vec3 vRefract[3];,  
    "varying float vReflectionFactor;,  
  
    "void main() {",  
  
        "vec4 reflectedColor = textureCube( tCube, ",  
        "    " vec3( -vReflect.x, vReflect.yz ) );",  
        "    " vec4 refractedColor = vec4( 1.0 );",  
  
        "    refractedColor.r = textureCube( tCube, ",  
        "        " vec3( -vRefract[0].x, vRefract[0].yz ) ).r; ",  
        "    refractedColor.g = textureCube( tCube, ",  
        "        " vec3( -vRefract[1].x, vRefract[1].yz ) ).g; ",  
        "    refractedColor.b = textureCube( tCube, ",  
        "        " vec3( -vRefract[2].x, vRefract[2].yz ) ).b; ",  
  
        "    gl_FragColor = mix( refractedColor, ",  
        "        reflectedColor, clamp( vReflectionFactor, ",  
        "            0.0, 1.0 ) );",  
  
        "    }"  
    ].join("\n")  
};
```

Tworzenie własnego shadera może się wydawać pracochłonne, ale efekt wart jest tej dodatkowej pracy, ponieważ można otrzymać niezwykle realistyczną symulację optyki. Ponadto dodatkowe mechanizmy, które dostarcza biblioteka Three.js — aktualizowanie macierzy świata obiektów, śledzenie kamery, deklarowanie dziesiątek zmiennych GLSL, komplilowanie i łączenie kodu GLSL — oszczędzają dosłownie dni pracy i debugowania. Dzięki temu myśl o utworzeniu własnego shadera nie tylko nie jest odpychająca, ale wręcz zachęcająca. Mając ten szkielet, powinieneś podjąć próbę napisania własnego shadera. Zalecam na początek napisanie shadera Fresnela i innych dostępnych wśród przykładów biblioteki Three.js. Jest wiele różnych efektów i dużo do nauczenia.

Renderowanie

W tym rozdziale znacznie poszerzyliśmy zakres umiejętności tworzenia realistycznych aplikacji. Początkowo rysowaliśmy tylko proste figury geometryczne, a teraz umiemy już posługiwać się materiałami, teksturami, światłami, cieniami, a nawet potrafimy pisać własne shadery

w języku GLSL. Postawiliśmy sobie poprzeczkę wysoko, ale jeszcze do niej nie doskoczyliśmy. Musimy zrobić jeszcze jeden krok: nauczyć się renderowania.

Ostatecznym wynikiem pracy nad trójwymiarowym grafem sceny w bibliotece Three.js jest dwuwymiarowy obraz wyrenderowany na elemencie kanwy w oknie przeglądarki internetowej. Nieważne, czy używamy WebGL, dwuwymiarowego API rysunkowego kanwy, czy też technologii CSS. Ważne jest to, co ostatecznie będzie widać na ekranie. Korzystamy z biblioteki WebGL, ponieważ dzięki niej różne rzeczy da się wykonać szybko. Przy użyciu innych technologii też może udałoby się dojść do podobnych efektów, ale z pewnością nie osiągnelibyśmy zdowalającej szybkości zmiany klatek. Dlatego właśnie tak często korzystamy z biblioteki WebGL.

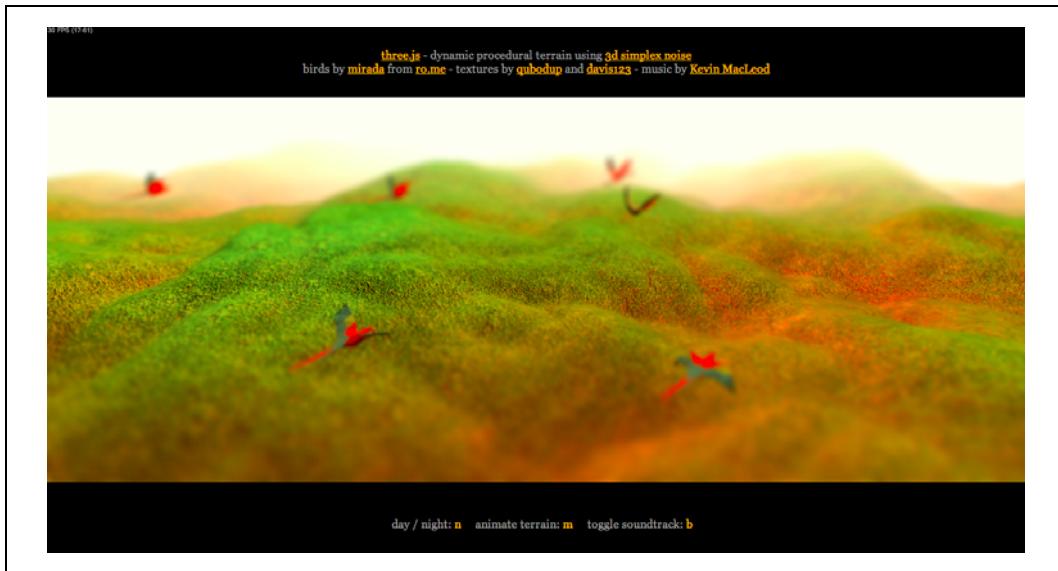
Sama biblioteka WebGL również oferuje kilka sposobów renderowania obrazów. Można np. za stosować renderowanie przy użyciu **bufora głębi** (ang. *z-buffer*), które polega na wykorzystaniu przez sprzęt dodatkowej pamięci, aby renderować na scenie wyłącznie piksele znajdujące się na przedzie. Wszystko zależy od nas. Jeśli nie użyjemy tej techniki, aplikacja będzie musiała sama sortować obiekty na scenie, możliwe, że sięgnie aż do poziomu samych trójkątów. Wydaje się to kłopotliwe, ale czasami dokładnie to programista chce robić. To tylko jeden z rodzajów wyborów, jakie musimy podejmować w odniesieniu do renderowania.

Biblioteka Three.js z założenia ma ułatwiać tworzenie podstawowej grafiki. Wbudowanego w nią renderera WebGL można używać do generowania wysokiej jakości grafiki bez wielkiego nacisku pracy ze strony programisty. Jak można się zorientować, siedząc po przestudiowanych do tej pory przykładach, aby wyrenderować grafikę wystarczy: 1) utworzyć renderer, 2) ustawić wymiary obszaru widoku, 3) wywołać funkcję `render()`. Jednak możemy znacznie dokładniej kontrolować proces renderowania i jeśli możliwość tę połączymy z zaawansowanymi technikami, takimi jak przetwarzanie końcowe (ang. *post-processing*), renderowanie wieloprzebiegowe (ang. *multipass rendering*) oraz renderowanie opóźnione (ang. *deferred rendering*), możemy uzyskać bardzo realistyczne efekty.

Przetwarzanie końcowe i renderowanie wieloprzebiegowe

Czasami jeden renderer nie wystarcza, a żeby uzyskać bardzo wysoką jakość realistyczny obraz, trzeba wykonać kilka renderingów sceny. Poszczególne renderingi, inaczej **przebiegi**, łączy się w celu utworzenia ostatecznej wersji obrazu w procesie zwanym **renderowaniem wieloprzebiegowym** (ang. *multipass rendering*). W wielu przypadkach proces taki zawiera także **przetwarzanie końcowe** (ang. *post-processing*), czyli czynność mającą na celu poprawienie jakości obrazu za pomocą specjalnych technik przetwarzania.

Przetwarzanie końcowe i wieloprzebiegowe to bardzo popularne techniki w renderowaniu grafiki trójwymiarowej i dlatego twórcy biblioteki Three.js ze szczególną starannością zaimplementowali ich obsługę. Na rysunku 4.15 widać subtelny, ale niezwykle sugestywny przykład zastosowania przetwarzania końcowego w bibliotece Three.js napisany przez AlteredQualia. Aby go obejrzeć w swojej przeglądarce, otwórz plik `examples/webgl_terrain_dynamic.html`. Ptaki mają statycznie przelatującą nad nieziemsko wyglądającą krainą, przecinając mgliste powietrze przychodząącym słońcu. Tak jakby proceduralnie wygenerowany za jednym razem z zastosowaniem szumu teren był niewystarczająco ujmujący, dodano jeszcze renderowanie wieloprzebiegowe z cieniowaniem poświaty, aby podkreślić rozproszenie jasnego światła słonecznego przez mgłę, oraz filtr Gaussa, aby przyjemnie rozmazać scenę i tym samym jeszcze bardziej spotęgować bajeczny efekt.



Rysunek 4.15. Przykład dynamicznego i proceduralnego generowania terenu, wyrenderowany przy użyciu kilku przebiegów przetwarzania końcowego — autor sceny: AlteredQualia, autor ptaków: Mirada (z RO.ME)

Przetwarzanie końcowe w Three.js bazuje na następujących składnikach.

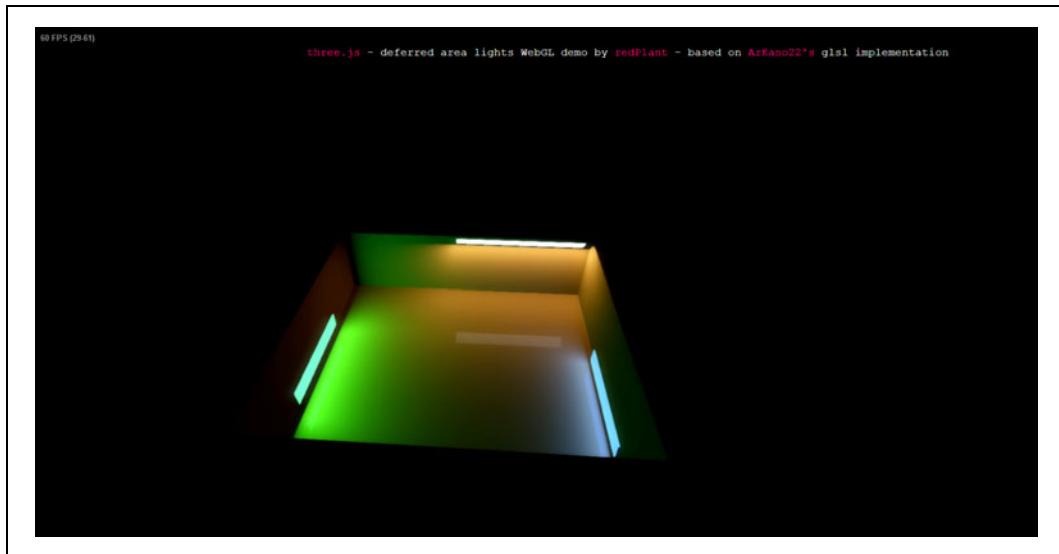
- Obsługa **wielu celów renderowania** poprzez obiekt THREE.WebGLRenderTarget. Dzięki wielu celom renderowania scenę można renderować wielokrotnie na pozaekranowych mapach bitowych, a następnie połączyć je wszystkie w ostateczny obraz. (Plik źródłowy: `src/renderers/WebGLRenderTarget.js`).
- Wieloprzebiegowa pętla renderowania zaimplementowana w klasie THREE.EffectComposer. Obiekt tej klasy zawiera przynajmniej jeden obiekt **przebiegu renderowania**, który wywołuje po kolej w celu wyrenderowania sceny. W każdym przebiegu dostępna jest cała scena i dane graficzne wygenerowane w poprzednim przebiegu, co umożliwia dodatkowe udoskonalenie obrazu.

Klasa THREE.EffectComposer i techniki wieloprzebiegowe, w których jest wykorzystywana, są zaimplementowane w folderach `examples/js/postprocessing/` i `examples/js/shaders/` projektu Three.js. Przeglądając ich zawartość, można znaleźć mnóstwo fantastycznych przykładów efektów specjalnych utworzonych za pomocą przetwarzania końcowego.

Renderowanie opóźnione

Pozostała jeszcze jedna technika renderowania — **renderowanie opóźnione** (ang. *deferred rendering*). Jak sama nazwa wskazuje, metoda ta polega na opóźnieniu renderowania grafiki na kanwie WebGL do czasu obliczenia ostatecznego obrazu przy użyciu kilku źródeł danych. W odróżnieniu od renderowania wieloprzebiegowego, które polega na wielokrotnym renderowaniu sceny i doskonaleniu obrazu, aby w końcu przenieść go na kanwę WebGL, w renderowaniu opóźnionym w pierwszym przebiegu wykorzystuje się wiele **buforów** (w istocie tekstur), w których gromadzi się dane potrzebne do obliczeń cieniowania. W następnym przebiegu obliczane są wartości pikseli przy użyciu wartości zgromadzonych w pierwszym przebiegu. Technika ta może

zużywać dużo pamięci i obciążać procesor, ale pozwala na uzyskanie bardzo realistycznych efektów, zwłaszcza świetlnych i dotyczących cieni. Przykład jej zastosowania pokazano na rysunku 4.16. ([examples/webgldeferred_arealights.html](#)).



Rysunek 4.16. Oświetlenie obliczone z dokładnością do jednego piksela przy użyciu renderowania opóźnionego

Podsumowanie

W tym rozdziale zawarto opisy wielu nowych pojęć i technik. Poznałeś w nim większość dostępnych w Three.js metod dotyczących rysowania i renderowania grafiki. Dowiedziałeś się, jak używać gotowych klas geometrii do tworzenia trójwymiarowych brył, siatek oraz parametryzowanych i ekstrudowanych kształtów. Wiesz już, czym są graf sceny i hierarchia przekształceń i jak ich używać do budowania złożonych scen. Przeszedłeś praktyczne przykłady wykorzystania materiałów, tekstur oraz światła. Na końcu dowiedziałeś się, jak za pomocą shaderów i zaawansowanych technik renderowania, takich jak przetwarzanie końcowe i opóźnione, zwiększyć realizm grafik. Biblioteka Three.js zawiera bogaty zestaw narzędzi zapakowany w przystępny i łatwy w użyciu pakiet. Narzędzia te w połączeniu z możliwościami WebGL umożliwiają uzyskanie prawie każdego efektu trójwymiarowego, jaki można sobie wyobrazić.

Animacje trójwymiarowe

Animacja to technika polegająca na zmienianiu wyświetlanego na ekranie obrazu w określonym czasie. Dzięki niej statyczna, trójwymiarowa scena może tętnić życiem. Istnieje wiele technik animacji i wiele koncepcji ich modelowania, ale w istocie wszystko sprowadza się do jednego, czyli do sprawienia, by poruszały się piksele na ekranie.

Biblioteka WebGL nie oferuje wbudowanych funkcji animacyjnych, ale jej moc i szybkość działania umożliwiają renderowanie fantastycznych grafik i zmienianie ich z prędkością 60 klatek na sekundę, można zatem programować trójwymiarową animowaną treść na kilka sposobów. Dodatkowo udoskonalenia architektury wykonawczej nowoczesnych przeglądarek sprawiają, że animacje takie można wstawać na strony internetowe wśród innych elementów, nie martwiąc się o artefakty i inne niepożądane efekty.

Z pomocą animacji można zmieniać na scenie WebGL praktycznie wszystko: przekształcenia, geometrię, tekstury, materiały, światła i kamery. Obiekty mogą być przesuwane, obracane, skalowane i poruszane wzduż wyznaczonych ścieżek; geometria może być wyginana, skręcana i przemieniana w różne kształty; tekstury mogą być przesuwane, skalowane, obracane i przewijane, a ich piksele można zmieniać w każdej klatce; można zmieniać w czasie kolory materiałów, refleksy oraz poziom przezroczystości; kamery można przesuwać i obracać, aby uzyskać efekty filmowe. Możliwości są praktycznie nieograniczone.

W tym rozdziale poznasz kilka różnych technik animacji oraz służących do ich implementowania narzędzi i bibliotek. Techniki te są mocno zakorzenione w przemyślach gier i filmowym, a ich działanie opiera się na solidnych matematycznych podstawach. Animacja przy użyciu WebGL to dynamicznie rozwijająca się dziedzina, a więc poznawanie jej — siłą rzeczy — wymaga zbadania wielu różnych rozwiązań. Biblioteka Three.js zawiera narzędzia do animacji, które sprawdzają się bardzo dobrze w niektórych sytuacjach. Tutaj przyjrzymy się też innej otwartej bibliotece o nazwie Tween.js. Jest to niewielka i łatwa w obsłudze biblioteka do tworzenia prostych przejść, które jednak nie zastąpią bardziej kompleksowych produktów. Jeśli więc potrzebujesz czegoś skomplikowanego, prawdopodobnie będziesz musiał zbudować własny mechanizm animacyjny.

Animacje przy użyciu WebGL realizuje się za pomocą przynajmniej jednej z wymienionych poniżej technik, których szczegółowy opis znajduje się dalej w tym rozdziale. Oto one.

- Funkcja `requestAnimationFrame()` służąca do sterowania pętlą wykonawczą.
- Programowe **modyfikowanie własności** obiektów wizualnych przy użyciu pętli wykonawczej. Jest to dobry sposób na tworzenie prostych animacji, takich jak obiekty obraca-

jące się wokół jednej osi. Technika ta jest także przydatna, gdy można wyrazić pozycję, orientację lub jakąś inną cechę obiektu jako funkcję zmiennej, takiej jak czas. Ogólnie rzecz biorąc, jest to najprostsza do zaimplementowania technika animacji, ale jej możliwości są ograniczone do szczególnych przypadków.

- Wykorzystanie **miedzyklatek** (ang. *tween*) w celu uzyskania płynnego przejścia od jednej do drugiej wartości cechy. Miedzyklatki doskonale nadają się do tworzenia prostych, jednorazowych efektów (np. przesuwania obiektów z jednego miejsca w inne po prostej linii).
- Wykorzystanie **klatek kluczowych** w taki sposób, że struktury danych reprezentują indywidualne wartości na osi czasu, a mechanizm oblicza (interpoluje) wartości pośrednie w celu uzyskania płynnego efektu. Klatki kluczowe najlepiej nadają się do tworzenia prostego animowania przesunięć, obrotów i skalowania oraz właściwości, takich jak kolory materiałów. W odróżnieniu od miedzyklatek, które umożliwiają tworzenie pojedynczych przejść od jednej wartości do innej, klatki kluczowe pozwalają na opracowywanie kilku przejść.
- Animowanie obiektów po linii **ścieżek** — wygenerowanych przez użytkownika odcinków prostych — umożliwia tworzenie złożonych i realistycznych ruchów na podstawie wzorów i wcześniej utworzonych danych ścieżkowych.
- Wykorzystanie **celów morfingu** (ang. *morph target*) w celu zdeformowania geometrii poprzez zmieszanie kilku różnych kształtów. Jest to doskonała technika do animowania mimiki oraz tworzenia bardzo prostych animacji postaci.
- Zastosowanie **skinningu** w celu deformowania geometrii na podstawie animacji znajdującej się pod spodem szkieletu. Jest to preferowany sposób animowania postaci i innych złożonych kształtów.
- Użycie **shaderów** do deformowania wierzchołków i zmieniania wartości pikseli w czasie. Czasami żądany animowany efekt najlepiej obliczyć wg wierzchołków lub pikseli, co oznacza, że dobrym sposobem implementacji jest użycie języka GLSL. Shaderów można też używać w celu zwiększenia wydajności innych technik — zwłaszcza morfingu i skinningu, które mogą znacznie obciążać CPU.

W aplikacjach często wykorzystuje się nie jedną, ale kilka lub nawet wszystkie z wymienionych technik. Nie istnieją żadne sztywne reguły określające, kiedy stosować daną metodę, chociaż oczywiście każda z nich najlepiej sprawdza się w określonych sytuacjach. Wybór techniki jest też często podyktowany względami produkcyjnymi. Jeśli np. w zespole nie ma artysty grafika, być może łatwiej będzie uzyskać animacje poprzez ich zaprogramowanie. Czasami też wybór jest kwestią osobistych preferencji. Animacja trójwymiarowa to w takim samym stopniu sztuka, co rzemiosło.

Sterowanie animacją za pomocą funkcji `requestAnimationFrame()`

W poprzednich rozdziałach napisalem już, jak uruchamiać pętlę wykonawczą aplikacji przy użyciu funkcji `requestAnimationFrame()`, która jest względnie nowym dodatkiem do API przeglądarek internetowych.

Funkcja `requestAnimationFrame()` umożliwia aplikacjom sieciowym dostarczanie spójnej i niezawodnej prezentacji treści wizualnej przy użyciu kodu w języku JavaScript. Treścią tą może być zmiana DOM strony, modyfikacja układu, zmiana stylów przy użyciu CSS lub tworzenie dowolnej grafiki z pomocą jednego z API rysunkowych, np. WebGL i kanwy. Po ukazaniu się w przeglądarce Firefox 4 funkcja ta szybko została wprowadzona także do wszystkich pozostałych przeglądarek. Historia powstania tej funkcji jest taka, że Robert O'Callahan z Mozilli szukał sposobu na synchronizację obsługiwanych przez przeglądarkę wbudowanych animacji, takich jak przejścia CSS i SVG z kodem JavaScript.

W przeszłości w aplikacjach sieciowych do animowania treści stron stosowano zegary, wykorzystując funkcje `setTimeout()` i `setInterval()`. W miarę jak zaczęto tworzyć coraz bardziej skomplikowane animacje i funkcje interaktywne, stało się jasne, że technika ta ma kilka poważnych wad.

- Funkcje zegarowe wywołują funkcje zwrotne w możliwie jak najdokładniejszych, określonych odstępach czasu, niezależnie od tego, czy dany moment jest dobry na narysowanie czegoś, czy nie.
- Kodu JavaScript wykonywanego w funkcji zwrotnej nie da się w niezawodny sposób zsynchronizować z inną generowaną na stronie przez przeglądarkę animacją (np. SVG albo przejściami CSS).
- Zegary są wykonywane, niezależnie od tego, czy strona lub karta jest widoczna, czy zminimalizowana, przez co istnieje możliwość niepotrzebnego wykonywania wywołań.
- Kod JavaScript nie dysponuje informacjami na temat częstotliwości odświeżania ekranu, więc aplikacja musi arbitralnie wybrać interwał: jeśli będzie to $1/24$ sekundy, użytkownika korzystającego z ekranu o częstotliwości 60 Hz pozbawimy rozdzielczości; a jeśli będzie to $1/60$, w przypadku ekranów o niskiej częstotliwości odświeżania marnowane będą cykle procesora na rysowanie treści, która nigdy nie będzie widoczna.

Funkcja `requestAnimationFrame()` ma rozwiązywać wszystkie wymienione problemy. W przykładach przedstawionych w poprzednich rozdziałach pętla wykonawcza wyglądała mniej więcej tak:

```
function run() {  
    // Żąda następnej klatki animacji.  
    requestAnimationFrame(run);  
  
    // Wykonuje animacje.  
    animate();  
  
    // Renderuje scenę.  
    renderer.render( scene, camera );
```

Należy tu zwrócić uwagę na brak wartości określającej czas w wywołaniu funkcji `requestAnimationFrame()`. Nie nakazujemy przeglądarki wywołania naszej animacji ani narysowania kodu w określonym czasie, ani robienia tego, co jakiś określony czas. Zamiast tego nakazujemy wywołanie animacji, gdy przeglądarka będzie **gotowa do ponownego przedstawienia strony**. To bardzo ważna różnica, ponieważ teraz przeglądarka może wywoływać kod rysujący napisany przez użytkownika podczas własnego cyklu rysowania. Korzyści z tego postępowania jest kilka. Po pierwsze, przeglądarka wywołuje animację tak często — albo tak *nieczęsto* — jak chce. Gdy program dysponuje wystarczającą ilością wolnych cykli, może spróbować zapewnić jak największą szybkość zmiany klatek odpowiadającą częstotliwości odświeżania ekranu. A gdy

strona lub karta jest niewidoczna albo cała przeglądarka zostanie zminimalizowana, program może zredukować liczbę wywołań, optymalizując w ten sposób wykorzystanie zasobów komputera. Po drugie, przeglądarka może seryjnie wywoływać kod rysujący użytkownika i w ten sposób zmniejszać liczbę procesów malowania ekranu, a co za tym idzie, oszczędzać zasoby. Po trzecie, każdy kod rysujący użytkownika wykonywany przez funkcję `requestAnimationFrame()` zostanie zmieszany, **złożony**, z innymi wywołaniami rysunkowymi, włącznie z wewnętrznyimi. Efektem jest płynniejsze, szybsze i bardziej efektywne rysowanie stron oraz animacji.

Używanie funkcji `requestAnimationFrame()` we własnych aplikacjach

Jak wiele nowości z pakietu technologii HTML5, funkcja `requestAnimationFrame()` nie jest obsługiwana przez wszystkie wersje każdej przeglądarki, chociaż sytuacja ta cały czas się zmienia. Ponadto funkcja ta powstała jako eksperymentalny dodatek dla jednej przeglądarki, a dopiero potem przeszła przez proces rekomendacji W3C, dlatego w każdej przeglądarce występuje z innym przedrostkiem. Na szczęście, dostępna jest świetna podkładka autorstwa Paula Irisha z Google. Jej kod, pokazany na listingu 5.1, znajduje się w pliku *libs/requestAnimationFrame/RequestAnimationFrame.js*, w folderze z plikami towarzyszącymi tej książce. Kod ten próbuje znaleźć właściwą nazwę funkcji `requestAnimationFrame()` w danej przeglądarce, a jeśli mu się nie uda, stosuje rozwiązanie awaryjne w postaci funkcji `setTimeout()` z 60 klatkami na sekundę.

Listing 5.1. Podkładka *RequestAnimationFrame.js* Paula Irisha

```
/*
 * Provides requestAnimationFrame in a cross browser way.
 * http://paulirish.com/2011/requestanimationframe-for-smart-animating/
 */
if ( !window.requestAnimationFrame ) {

    window.requestAnimationFrame = ( function() {

        return window.webkitRequestAnimationFrame ||
        window.mozRequestAnimationFrame ||
        window.oRequestAnimationFrame ||
        window.msRequestAnimationFrame ||
        function( /*function FrameRequestCallback */ callback,
        /*DOMElement Element */ element ) {

            window.setTimeout( callback, 1000 / 60 );
        };
    } )();
}
```



Termin **podkładka** (ang. *polyfill*) oznacza kod (najczęściej w języku JavaScript) dodający do przeglądarki implementację brakujących udogodnień. Podkładek po-wszechnie używa się w starszych wersjach przeglądarek, które nie obsługują nowych i eksperymentalnych funkcji. Autorem terminu jest Remy Sharp z Wielkiej Brytanii. Więcej informacji na temat tego słowa i jego pochodzenia można znaleźć na stronie <http://remysharp.com/2010/10/08/what-is-a-polyfill/>.

Jedną z kluczowych kwestii podczas używania funkcji `requestAnimationFrame()` jest dbanie o to, by żądać kolejnej klatki *przed wywołaniem* jakiegokolwiek innego kodu użytkownika, jak zostało to zrobione w pokazanej wcześniej przykładowej pętli wykonawczej. Ma to duże znaczenie dla obsługi wyjątków. Jeśli cała aplikacja trójwymiarowa jest sterowana w wywołaniu zwrtnym animacji i zostanie wygenerowany wyjątek gdzieś przed zażądaniem następnej klatki, aplikacja ulega awarii. Jeśli jednak klatki zażąda się przed wykonaniem innych czynności, ma się przynajmniej gwarancję dalszego wykonywania kodu. To umożliwia aplikacji dalsze funkcjonowanie i rysowanie elementów, nawet jeśli gdzieś indziej coś się popsuje.

Funkcja `requestAnimationFrame()` a wydajność

Funkcja `requestAnimationFrame()` bardzo pomaga zwiększyć wydajność aplikacji, ale wiążą się z tym dodatkowe obowiązki dla programisty. Jeśli przeglądarka wywołuje naszą funkcję zwrtną 60 razy na sekundę, musimy pisać takie funkcje zwrotne, których wykonywanie zajmuje nie więcej niż 16 milisekund. W przeciwnym przypadku aplikacja może nie reagować na działania użytkownika. A ponieważ 16 milisekund to bardzo mało czasu, należy starannie zaplanować wykonywanie jak najmniejszej ilości pracy w jednym wywołaniu. W najwyższej jakości aplikacjach trójwymiarowych w połączeniu z funkcją `requestAnimationFrame()` mogą być stosowane zegary, workery i inne techniki animacyjne, takie jak przekształcenia i przejścia CSS, aby zapewnić interaktywność i maksymalną wydajność programu.



Funkcja `requestAnimationFrame()` jest jedną z najważniejszych nowości wprowadzonych w HTML5. W tym podrozdziale opisałem zaledwie niewielki wycinek całego problemu, ale jeśli chcesz dowiedzieć się więcej, możesz poszukać informacji w internecie. Poszukaj nazwy tej funkcji, a znajdziesz mnóstwo artykułów, poradników i szczegółowych objaśnień.

Animacje klatkowe a animacje czasowe

Pierwsze systemy animacji komputerowej imitowały filmowe techniki animacji poprzez wyświetlanie po kolei serii nieruchomych obrazów albo, w przypadku grafiki trójwymiarowej, serii obrazów wektorowych generowanych przez program. Obraz taki nazywany jest **klatką** (ang. *frame*). Pierwotnie filmy kręcono i odtwarzano z prędkością 24 obrazów na sekundę, czyli z **szybkością zmiany klatek** wynoszącą 24 klatki na sekundę (24 fps). Prędkość taka była odpowiednia dla dużych ekranów projekcyjnych, na których filmy oglądano przy słabym oświetleniu. Jednak w animacjach i trójwymiarowych grach komputerowych ludzkie zmysły są w stanie zarejestrować znacznie szybsze zmiany, od 30 do 60, a nawet więcej klatek na sekundę. Mimo to, w wielu systemach animacyjnych, takich jak np. Adobe Flash, początkowo przyjęto konwencję wyświetlania 24 klatek na sekundę, ponieważ była znana animatorom. Obecnie wartości te znacznie się zmieniły — Flash obsługuje już 60 klatek na sekundę, jeśli programista tego zażąda — ale sama koncepcja pozostaje niezmieniona. Technika animacji przy użyciu serii osobnych klatek nazywa się **animacją klatkową**.

Animacja klatkowa ma jedną poważną wadę: częstotliwość zmiany klatek jest ustawiona na stałe i nie można jej zwiększyć nawet wtedy, kiedy możliwości sprzętu na to pozwalają. W filmie nie stanowiło to problemu, ponieważ w branży wszyscy używali bardzo podobnego sprzętu. Jednak poszczególne urządzenia do oglądania animacji komputerowej mogą radykalnie różnić się pod względem mocy obliczeniowej. Jeśli utworzymy animację z prędkością 24 fps, a komputer

będzie odświeżał ekran z częstotliwością 60 Hz, pozbawimy użytkownika pewnych szczegółów i płynności odtwarzania.

Opisane problemy rozwiązuje inna technika zwana **animacją czasową** (ang. *time-based animation*). W metodzie tej serie grafik wektorowych łączy się z punktami w czasie, a nie klatkami w sekwencji klatek o określonej częstotliwości. Dzięki temu komputer może prezentować obrazy i interpolowane między nimi klatki zawsze z odpowiednią prędkością, zapewniając najlepszą jakość i płynność przejścia. W przykładach przedstawionych w poprzednich rozdziałach zastosowano właśnie animacji czasowe. W każdym cyklu pętli wykonawczej funkcja `animate()` oblicza różnicę czasową między bieżącą a poprzednią klatką i otrzymanej wartości używa do obliczenia kąta obrotu. Także w tym rozdziale i następnych wszystkie przykłady są animacjami czasowymi. Dlatego mimo obecności słowa **frame** (klatka) w nazwie funkcji `requestAnimationFrame()` nie martw się, że nie będzie się ona nadawała do użycia.

Animowanie przy użyciu programowego aktualizowania właściwości

Najprostszym sposobem na rozpoczęcie animowania sceny WebGL jest napisanie kodu zmieniającego właściwości obiektu w każdym cyklu działania pętli wykonawczej. Przykłady widziałeś już w poprzednich rozdziałach. Aby obrócić kostkę Three.js w rozdziale 3., zmienialiśmy w każdej klatce własność `rotation.y` tej bryły — czyli jej kąt obrotu wokół osi `y`. Poniżej dla przypomnienia przedstawiam jeszcze raz ten sam kod.

```
var duration = 5000; // ms
var currentTime = Date.now();
function animate() {
    var now = Date.now();
    var deltat = now - currentTime;
    currentTime = now;
    var fract = deltat / duration;
    var angle = Math.PI * 2 * fract;
    cube.rotation.y += angle;
}
```

Zmienne `duration`, `currentTime`, `now` i `deltat` służą do obliczenia wartości czasowej animacji dla obrotu. W tym przypadku chcemy wykonać pełny obrót wokół osi `y` w czasie pięciu sekund. Obliczony kąt (`angle`) jest ułamkiem jednego pełnego obrotu, wartością, którą należy dodać do bieżącej wartości właściwości `rotation.y` kostki. Przypomnij, że w bibliotece Three.js obrót wyraża się w **radianach**, czyli `Math.PI * 2` równa się pełnemu obrotowi (o 360 stopni).

W ten sposób można animować na scenie cechę przedmiotu, taką jak pozycja, obrót, skala, kolory materiałów, przezroczystość itd. Ponadto technika ta jest bardzo ogólna, tzn. dzięki aktualizowaniu własności za pomocą kodu JavaScript można zastosować dowolne obliczenia. Animacjami można sterować przy użyciu wzorów matematycznych, logiki Boole'a, wartości statystycznych, strumieni danych, bieżących wskazań czujników itd. Jest to więc doskonała metoda do prezentowania danych naukowych i tworzenia wizualizacji. Można za jej pomocą przedstawić jakiś układ słoneczny, procesy fizyczne, zjawiska przyrodnicze, informacje zmieniające się w czasie, analizy statystyczne, dane geograficzne, dane dotyczące ruchu na stronie internetowej i inne dynamiczne rodzaje informacji pochodzących z baz danych. Ponadto technika ta jest doskonała do tworzenia zabawnych aplikacji, np. zawierających muzyczne wizualizacje.

Na rysunku 5.1 widać fantastyczną grę świateł dla piosenki *Lights* artystki Ellie Goulding. Jest to muzyczna wizualizacja utworzona przy użyciu WebGL przez brytyjską agencję interaktywną Hello Enjoy (<http://helloenjoy.com/>). Opracowanie to jest już znane od jakiegoś czasu, ale wciąż robi wrażenie. Widac żarzące się i migające kule, przelatujące po łuku komety, pojawiające się i znikające kolorowe piłki, szaleńczo kręcące się reflektory oraz balony w kształcie leż wyłaniające się z kolorowego, falującego terenu. Wszystko to jest zsynchronizowane z muzyką. To najwyższej klasy błyskotka, w której wszystkie efekty zostały wygenerowane programowo.



Rysunek 5.1. *Lights* artystki Ellie Goulding (<http://lights.elliegoulding.com/>) — muzyczna wizualizacja zbudowana przy użyciu programowej animacji; obraz opublikowany dzięki uprzejmości Hello Enjoy, Inc.

Na listingu 5.2 pokazano fragment kodu źródłowego tej animacji. Metoda `update()` aplikacji jest wywoływana w każdej iteracji pętli wykonawczej. Ta z kolei wywołuje metodę `update()` na wszystkich obiektach na scenie. Poniższy fragment pochodzi z metody `LIGHTS.StarManager.update()`, animującej gwiazdy w tle. Gwiazdy są renderowane jako cząsteczki `Three.js` należące do obiektu `THREE.ParticleSystem`. Pogrubione wiersze pokazują sposób aktualizacji koloru RGB każdej gwiazdy w odniesieniu do upływającego czasu, współczynnika rozkładu oraz operatora modulo (%) w celu utworzenia efektu migotania.

Listing 5.2. Animacja muzyki — fragment kodu z wizualizacji *Lights* artystki Ellie Goulding

```
update: function() {  
  
    var stars = this.stars,  
        deltaTime = LIGHTS.deltaTime,  
        star, brightness, i, il;  
  
    for( i = 0, il = stars.length; i < il; i++ ) {  
  
        star = this.stars[ i ];  
  
        star.life += deltaTime;  
  
        brightness = (star.life * 2) % 2;
```

```

        if( brightness > 1 )
            brightness = 1 - (brightness - 1);

        star.color.r =
        star.color.g =
        star.color.b = (Math.sin( brightness * rad90 - rad90 ) + 1) * 4;
    }

    this.particles._dirtyColors = true;
},

```

Metoda ta, mimo wielu zalet, ma też jednak wady. Wymaga ręcznego kodowania każdego efektu, przez co w konsekwencji trudno animować wiele różnych rodzajów przedmiotów. Ponadto kod źródłowy tej techniki zwykle jest obszerniejszy niż analogiczny w innych metodach, o których będzie mowa niebawem. W końcu w centrum uwagi został postawiony programista zamiast artysty, który prawdopodobnie jest znacznie lepszą osobą do tworzenia odpowiednich efektów wizualnych. Mimo to, programowa animacja jest doskonałym sposobem na szybkie i łatwe ożywienie sceny. Efekty mogą być imponujące, czego dowodem jest przypadek światła Ellie Goulding.

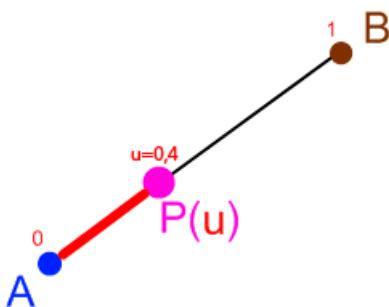
Animowanie przejść przy użyciu międzyklatek

Wiele animowanych efektów lepiej reprezentować jako struktury danych, nie zaś jako wartości programowo generowane w każdej iteracji pętli wykonawczej. Aplikacja zawiera zbiór wartości i szereg czasowy, a ogólny mechanizm obliczeniowy oblicza dla poszczególnych klatek wartości służące do modyfikacji wybranych własności. Jedną z tego rodzaju technik jest tzw. **wstawianie międzyklatek** albo **klatek pośrednich** (ang. *tweening*).

Tweening to proces polegający na generowaniu wartości pośrednich między parami innych wartości. Animator dostarcza tylko wartości początkowe i końcowe animacji, a program oblicza wartości pośrednie dla poszczególnych punktów czasowych. Jest to doskonała metoda do tworzenia jednorazowych przejść między stanami, takich jak przesunięcie obiektu w reakcji na kliknięcie myszą.

Interpolacja

Istotą tweeningu jest matematyczna technika zwana **interpolacją**. Polega ona na generowaniu wartości mieszczącej się między dwiema innymi wartościami na podstawie informacji skalarnej, takiej jak czas lub ułamek. Na rysunku 5.2 przedstawiono ilustrację zasady działania interpolacji. Dla dowolnej pary wartości A i B oraz ułamka u z przedziału od 0 do 1, wartość interpolacji P można obliczyć za pomocą wzoru $A+u*(B-A)$. W przypadku przedstawionym na rysunku otrzymujemy interpolowaną wartość $P(u) = 0,4$. Jest to najprostsza postać interpolacji, zwana **interpolacją liniową**, ponieważ wykres funkcji matematycznej używanej do obliczenia wyniku jest linią prostą. W powszechnym użyciu są też inne, bardziej skomplikowane funkcje interpolacji, np. sklejane (rodzaj krzywej) i wielomianowe. Nieco dalej omawiam też animacje oparte na funkcjach sklejanych.



Rysunek 5.2. Interpolacja liniowa (<http://bit.ly/gpwiki-linear-interlopation>)

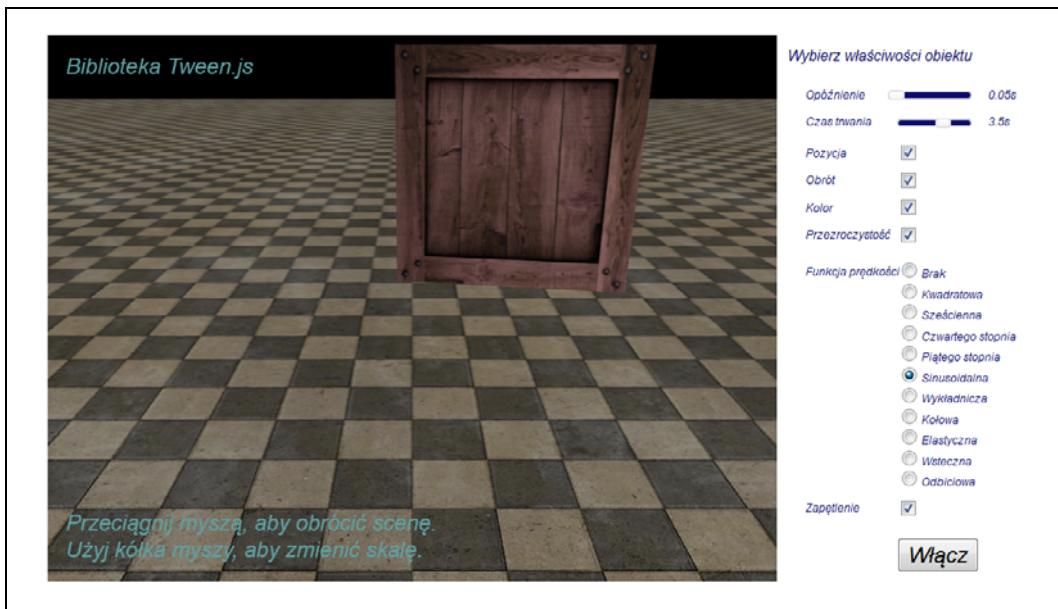
Interpolację wykorzystuje się do obliczania wartości pośrednich pozycji, obrotów, kolorów, wartości skalarnych (np. przezroczystości) itd. W przypadku wartości wieloskładnikowych, takich jak trójwymiarowe wektory, interpoluje się każdy składnik osobno. Przykładowo wartość interpolacji P dla $u = 0,5$ dla trójwymiarowego wektora AB w przedziale od $(0,0,0)$ do $(1,2,3)$ wynosi $(0,5;1;1,5)$.

Biblioteka Tween.js

Samodzielne zaimplementowanie prostego tweeningu nie jest trudne, ale jeśli chcesz korzystać z nieliniowych funkcji interpolacji i różnych fantastycznych efektów, takich jak przyspieszanie i zwalnianie animacji w wybranych momentach, to problem może stać się skomplikowany. Dlatego zamiast tworzyć własny system, można użyć jednej z gotowych bibliotek. Popularnym otwartym narzędziem tego typu jest opracowana przez Soledad Penadés biblioteka Tween.js (<https://github.com/sole/tween.js>). Znajduje ona zastosowanie w wielu projektach WebGL budowanych na bazie Three.js, np. w znanyju już RO.ME (<http://www.ro.me/>), WebGL Globe (<http://workshop.chromeexperiments.com/globe/>) oraz Mine3D (<http://egraether.com/mine3d/>) — internetowej wersji klasycznej gry Minesweeper.

Plik *r5/tweenjs/tweens.html* zawiera system do testowania różnych opcji Tween.js (rysunek 5.3). W programie biblioteka ta jest wykorzystywana do stosowania różnych rodzajów przejść do pokrytej teksturą kostki: pozycji, obrotu, koloru materiału oraz przezroczystości. Dostępne są suwaki do zmieniania czasu przejścia i opóźnienia (czasu, po upływie którego ma się rozpoczęć animacja), pola wyboru do włączania i wyłączania różnych algorytmów obliczania wartości pośrednich oraz opcja **zapętlania** animacji (powtarzania jej w nieskończoność). Ponadto można też zmieniać funkcje prędkości animacji, ale o nich będzie mowa w następnym podrozdziale. Wypróbuj różne ustawienia, aby zobaczyć, jak działają.

Biblioteka Tween.js jest bardzo łatwa w użyciu. Ma prostą składnię, a dzięki polimorfizmowi języka JavaScript można przetwarzać każdą własność przy użyciu tych samych wywołań metod. Ponadto wykorzystywana jest w niej taka sama technika wywołań łańcuchowych, jakiej użyto w jQuery, co pozwala na pisanie bardzo zwięzłego kodu. Zobaczmy, jak to wygląda w praktyce. Na listingu 5.3 przedstawiono fragment funkcji `playAnimations()` służącej do obliczania wartości pośrednich w reakcji na zmianę wartości.



Rysunek 5.3. Animowanie przejść przy użyciu biblioteki Tween.js

Listing 5.3. Kod biblioteki Tween.js do animacji pozycji obiektu

```
positionTween =  
  
    new TWEEN.Tween( group.position )  
        .to({ x: 2, y: 2, z:-3 }, duration * 1000)  
        .interpolation(interpolationType)  
        .delay( delayTime * 1000 )  
        .easing(easingFunction)  
        .repeat(repeatCount)  
        .start();
```

Animację pozycji tworzy jeden łańcuch połączonych metod. Oto on.

- Wywołanie konstruktora `new TWEEN.Tween` przyjmuje jeden argument — obiekt, którego właściwości mają być poddane animacji.
- Wywołanie metody `to()`, która przyjmuje obiekt JavaScript definiujący właściwości do animowania oraz czas animacji w milisekundach.
- Wywołanie metody `interpolation()` (nie jest obowiązkowe) określającej typ interpolacji. Jeśli się opuści tę metodę, domyślnie zostanie zastosowana interpolacja liniowa (`TWEEN.Interpolation.linear`).
- Wywołanie metody `delay()` (nie jest obowiązkowe) określającej opóźnienie rozpoczęcia animacji.
- Wywołanie metody `easing()` (nie jest obowiązkowe) określającej funkcję szybkości (opisaną w następnym podrozdziale).
- Wywołanie metody `repeat()` (nie jest obowiązkowe) określającej liczbę powtórzeń animacji (domyślna wartość wynosi zero).
- Wywołanie metody `start()` uruchamiającej animację.

Wszystkie te metody można też wywoływać osobno. Każdy z rodzajów animacji — pozycja, obrót, kolor materiału i przezroczystość — konfiguruje się w podobny sposób. Wielką zaletą biblioteki Tween.js jest to, że do metody `to()` nie trzeba podawać wszystkich wartości obiektu, a jedynie te, które mają być zmieniane. Przykładowo animacja obrotu zmienia tylko obrót wokół osi `y`, a więc jest utworzona następująco:

```
rotationTween =  
  
  new TWEEN.Tween( group.rotation )  
    .to( { y: Math.PI * 2 }, duration * 1000 )  
    .interpolation(interpolationType)  
    .delay( delayTime * 1000 )  
    .easing(easingFunction)  
    .repeat(repeatCount)  
    .start();
```

Po skonfigurowaniu i uruchomieniu animacji pozostało dopilnować, by biblioteka Tween.js aktualizowała ją w każdej klatce. To zadanie aplikacji, więc do naszej funkcji `run()` dodajemy poniższy wiersz kodu:

```
TWEEN.update();
```

Wewnętrznie biblioteka Tween.js zawiera listę wszystkich swoich uruchomionych obiektów `tween` i kolejno wywołuje ich metodę `update()`. Metoda ta oblicza upływający czas, stosuje funkcje prędkości i opóźnienia oraz włącza lub wyłącza opcje powtarzania, a na koniec ustawia właściwości docelowego obiektu zgodnie z ustawieniami w metodzie `to()`. Jest to bardzo eleganckie i proste rozwiązanie umożliwiające zmianę właściwości obiektów w czasie bez ręcznego kodowania zmian w każdej klatce.

Funkcja prędkości animacji

Najprostsze animacje oparte na interpolacji liniowej mogą dawać dość sztywny, nienaturalny efekt, ponieważ zmiany właściwości obiektów odbywają się ze stałą prędkością. Jest to sprzeczne z tym, jak przedmioty zachowują się w realnym świecie, w którym rządzą prawa bezwładności, pędu, przyspieszenia itd. Korzystając z biblioteki Tween.js, można uzyskać bardziej realistyczne efekty. Służą do tego **funkcje prędkości animacji** (ang. *easing*). Są to nieliniowe funkcje stosowane do początku i końca przejścia, a efekt ich działania może być bardzo realistyczny. Za ich pomocą można nawet uzyskać namiastkę fizyki bez pracochłonnego dołączania całego systemu fizycznego do aplikacji.

Wypróbuj w przykładowej aplikacji różne funkcje prędkości, aby zobaczyć, jak działają. Niektóre z nich powodują proste przyspieszanie i zwalnianie animacji, inne z kolei dają efekty podobne do odbijającej się piłki lub skokowe. Funkcje wielomianowe, kwadratowa (`Quadratic`), sześcienne (`Cubic`), czwartego stopnia (`Quartic`) oraz piątego stopnia (`Quintic`) animują przejścia w sposób zgodny z ich nazwami, czyli przy użyciu funkcji drugiego, trzeciego, czwartego i piątego stopnia. W pozostałych funkcjach stosowane są efekty w postaci fal sinusoidalnej, odbicia oraz nagłych ruchów. Każdej z tych funkcji można użyć na początku, na końcu animacji albo z obu jej stron.

Funkcje prędkości w istocie modyfikują **czas**. Na listingu 5.4 przedstawiono kod źródłowy funkcji `TWEEN.Easing.Cubic`. Dane wejściowe mieszczą się w przedziale zamkniętym od 0 do 1 (czyli są ułamkiem pełnego czasu trwania animacji). Wartość wejściowa, `k`, jest podnoszona przez funkcję do potęgi trzeciej, przez co im mniejsza wartość wejściowa, tym mniejsza wartość wyjściowa. Analogicznie, im wartość `k` jest bliższa 1, tym bliżej do tej wartości wynikowi.

Listing 5.4. Sześcienna funkcja prędkości animacji z biblioteki Tween.js

```
Cubic: {

  In: function ( k ) {

    return k * k * k;

  },

  Out: function ( k ) {

    return --k * k * k + 1;

  },

  InOut: function ( k ) {

    if ( ( k *= 2 ) < 1 ) return 0.5 * k * k * k;
    return 0.5 * ( ( k -= 2 ) * k * k + 2 );
  }
},
```



Budowa funkcji prędkości animacji biblioteki Tween.js jest oparta na bazie nowatorskich prac nad animacjami Roberta Pennera (<http://robertpenner.com/index2.html>). Można wśród nich znaleźć wiele przydatnych równań, włącznie z liniowymi, kwadratowymi, czwartego stopnia, sinusoidalnymi i wykładniczymi. Prace Pennera zostały przeniesione z języka ActionScript, w którym powstały, do wielu innych języków, takich jak JavaScript, Java, CSS, C++ czy C#, oraz wykorzystane w narzędziach do animacji biblioteki jQuery.

Jak właśnie pokazałem, tweening to doskonała technika do szybkiego i łatwego tworzenia prostych, naturalnie wyglądających efektów. Biblioteka Tween.js umożliwia nawet łączenie animacji w sekwencje, aby można było opracowywać bardziej złożone efekty. Gdy jednak zacznesz tworzyć skomplikowane animacje, będziesz potrzebować ogólniejszego rozwiązania. Wówczas przydatne staną się klatki kluczowe.

Tworzenie skomplikowanych animacji przy użyciu klatek kluczowych

Tweening to doskonała technika do budowania prostych efektów przejść. Żeby tworzyć bardziej skomplikowane animacje, należy wznieść się na wyższy poziom i użyć tzw. **klatek kluczowych**. W metodzie tej, zamiast definiować pojedynczą parę wartości, dla której mają być wygenerowane wartości pośrednie, tworzona jest lista wartości, na dodatek między poszczególnymi wartościami mogą występować różne czasy trwania. Zwrót uwagę, że określenie **animacja oparta na klatkach kluczowych** jest używane zarówno w odniesieniu do systemów klatek kluczowych, jak i czasowych — jest to pozostałość po starej, klatekowej nomenklaturze.

Dane klatki kluczowej składają się z dwóch części: listy wartości czasowych (**kluczy**) i listy wartości własności, które mają zostać zastosowane w odpowiednich punktach czasowych. System animacyjny oblicza wartości pośrednie dla wartości czasowych leżących między każdą parą kluczów.

Poniższy fragment kodu (z hipotetycznego systemu animacyjnego) przedstawia przykładowe dane klatki kluczowej dla animacji przesuwającej przedmiot z pierwotnego miejsca poza pole widzenia kamery. Cała animacja trwa sekundę. W czasie jednej czwartej sekundy obiekt przesuwa się do góry, a następnie w pozostałym czasie przesuwa się jeszcze trochę w górę i poza pole widzenia kamery. System animacyjny oblicza wartości pośrednie dla punktów od (0,0,0) do (0,1,0) w czasie pierwszej čwiartki sekundy, a następnie dla punktów od (0,1,0) do (0,2,5) w czasie pozostałych trzech czwartych sekundy.

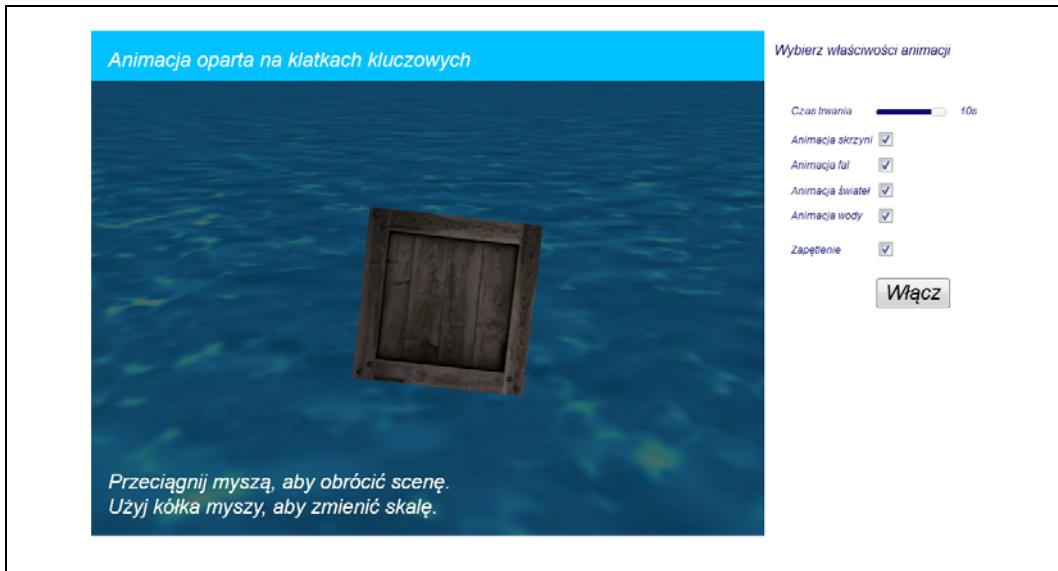
```
var keys = [0, 0.25, 1];
var values = [[0, 0, 0],
              [0, 1, 0],
              [0, 2, 5]
            ];
```

Animacje oparte na klatkach kluczowych mogą działać z interpolacją liniową lub bardziej skomplikowanymi rodzajami interpolacji, np. bazującymi na funkcji sklejanej. Innymi słowy, punkty danych reprezentujące klucze można traktować jak punkty na wykresie liniowym lub jak wykres bardziej skomplikowanej funkcji. Zarówno w tweeningu, jak i animacji opartej na klatkach kluczowych stosuje się interpolację, jednak te dwa rodzaje animacji różnią się pod dwoma ważnymi względami: po pierwsze, animacje oparte na klatkach kluczowych mogą zawierać więcej niż dwie wartości i po drugie, interwał czasowy między kluczami może być różny. To daje programiście większe możliwości i pozwala uzyskać bardziej różnorodne efekty.

Keyframe.js – proste narzędzie do tworzenia animacji opartych na klatkach kluczowych

Zanim przeanalizujemy przykład animacji opartej na klatkach kluczowych, musimy znaleźć bibliotekę animacyjną umożliwiającą korzystanie z tej techniki. Biblioteka Tween.js zawiera najprostsze narzędzia w postaci list wartościami zamiast tylko ich par. Jednak uważam, że składnia ich definiowania jest niezgrabna. Ponadto nie ma możliwości różnicowania interwału między kolejnymi kluczami. Biblioteka Three.js zawiera odpowiednie wbudowane klasy animacyjne, ale nie nadają się one do szybkiego kodowania efektów. Ich głównym zadaniem jest obsługa narzędzi do ładowania plików w formatach JSON, COLLADA i innych. Chociaż w istocie klatki kluczowe powinno się opracowywać w specjalnych programach, takich jak 3ds Max, Maya czy Blender, a nie pisać ręcznie, warto mieć narzędzie programistyczne do łatwego tworzenia takich klatek. Problem ze znalezieniem łatwego w użyciu rozwiązania do tworzenia klatek kluczowych dla WebGL sprawił, że sam napisałem odpowiednie narzędzie o nazwie Keyframe.js.

Keyframe.js to bardzo prosty program. Zawiera dwie klasy, KeyFrameAnimator, kontrolującą stan animacji (uruchomienie, zatrzymanie, zapętlenie itd.), oraz Interpolator, obliczającą wartości pośrednie dla każdej pary kluczów. Aktualnie biblioteka ta obsługuje tylko interpolację liniową, ale za to umożliwia dostarczanie funkcji szybkości animacji, dzięki czemu ktoś, kto potrafi pozytywne równania Pennera zaimplementowane w Tween.js, może zaoszczędzić mnóstwo pracy. Aby zobaczyć bibliotekę Keyframe.js w akcji, otwórz plik [r5/keyframeanimation.html](#), aby wyświetlić stronę widoczną na rysunku 5.4. Widać na niej fragment przygody na morzu: drewniana skrzynia unosi się na wzburzonych wodach, podczas gdy niebo naprzemiennie jaśnieje i ciemnieje, co oznacza zbliżający się sztorm. Znajdujące się po prawej przyciski kontrolki służą do zmieniania czasu oraz włączania i wyłączania różnych rodzajów animacji, a także do włączania i wyłączania pętli.



Rysunek 5.4. Skomplikowane animacje utworzone przy użyciu klatek kluczowych

Na listingu 5.5 przedstawiony jest kod źródłowy animacji skrzyni. Najpierw tworzymy obiekt klasy KF.KeyFrameAnimator i inicjujemy go przy użyciu następujących parametrów: zapętlenie, czas trwania (w milisekundach), funkcja prędkości (pożyczona z Tween.js) oraz zbiór danych interpolacji klatek kluczowych w parametrze interps. W odróżnieniu od biblioteki Tween.js, tutaj klucze i wartości są listami, nie parami. Ponadto interwały między kolejnymi klateczkami są różne. Zgodnie z interpolatorem pozycji (target:group.position), skrzynia porusza się w lewo i do przodu w czasie od $t = 0$ do $t = 0,2$, a następnie szybko wraca do początkowego położenia ($t = 0,2$ do $0,25$), po czym na chwilę zanurza się głębiej ($t = 0,25$ do $0,375$). Później wraca na powierzchnię w czasie $t = 0,5$, powoli się zanurza ($t = 0,5$ do $0,9$) i na koniec wraca do góry w czasie $1,0$. Zwróci uwagę, że w bibliotece Keyframe.js klucze określa się jako ułamki czasu trwania animacji. Znaczy to, że ich wartości zawsze mieszczą się w przedziale od 0 do 1, w związku z czym rzeczywisty czas klatki wynosi:

$$\text{czas} = t \times \text{czas trwania}$$

Listing 5.5. Oparta na klatek kluczowych animacja drewnianej skrzyni

```
if (animateCrate)
{
    crateAnimator = new KF.KeyFrameAnimator;
    crateAnimator.init({
        interps:
        [
            {
                keys:[0, .2, .25, .375, .5, .9, 1],
                values:[
                    { x : 0, y:0, z: 0 },
                    { x : .5, y:0, z: .5 },
                    { x : 0, y:0, z: 0 },
                    { x : .5, y:-.25, z: .5 },
                    { x : 0, y:0, z: 0 },
                    { x : .5, y:-.25, z: .5 },
                    { x : 0, y:0, z: 0 },
                ],
            }
        ]
    });
}
```

```

        target:group.position
    },
    {
        keys:[0, .25, .5, .75, 1],
        values:[
            { x : 0, z : 0 },
            { x : Math.PI / 12, z : Math.PI / 12 },
            { x : 0, z : Math.PI / 12 },
            { x : -Math.PI / 12, z : -Math.PI / 12 },
            { x : 0, z : 0 },
        ],
        target:group.rotation
    },
],
loop: loopAnimation,
duration:duration * 1000,
easing:TWEEN.Easing.Bounce.InOut,
});
crateAnimator.start();
}

```

Na listingu znajduje się też drugi interpolator dotyczący obrotu, który obraca skrzynię wokół osi x. Zauważ, że ma on inną liczbę kluczy. Nie jest to błędem, a zamierzoną cechą. Animacje pozycji i obrotu celowo nie zostały zsynchronizowane, aby ruch przedmiotu był bardziej chaotyczny. Ostatni element to funkcja prędkości TWEEN.Easing.Bounce.InOut. Kombinacja niezależnych i nieskoordynowanych przesunięć i obrotów ze „skokową” funkcją prędkości dała dobry efekt: skrzynia dość realistycznie podskakuje na wodzie. Pozostało jeszcze tylko uruchomić animację za pomocą metody start().

Animacje wody i burzy są obsługiwane w podobny sposób, chociaż w żadnej z nich nie użyto funkcji do określania prędkości animacji. Zostały zastosowane animacje, z których jedna wprowadza w ruch powierzchnię wody (prosta rotacja płaszczyzny wody wokół osi x), druga tworzy imitację fal (poprzez „przewijanie” tekstuury za pomocą interpolacji jej własności offset), a ostatnia włącza migoczące światła przy użyciu interpolacji wartości RGB koloru.

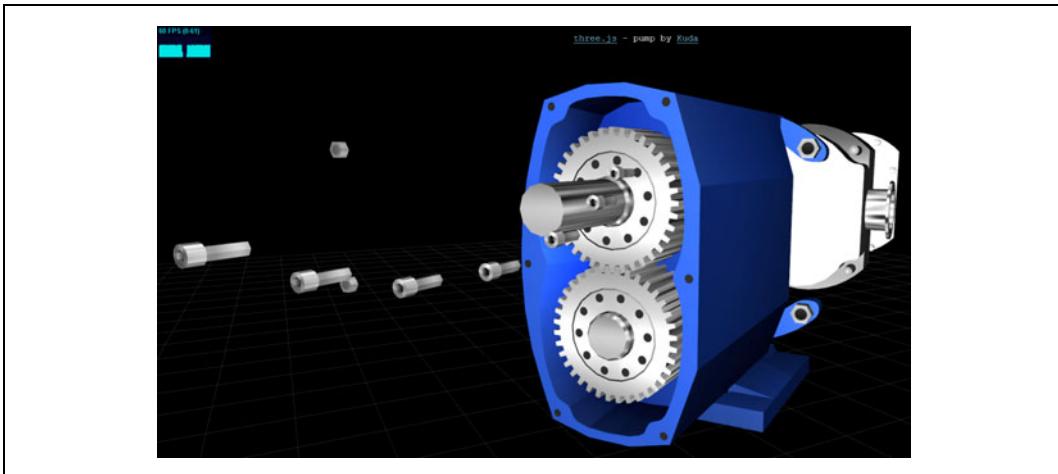
Jest to prosta ilustracja tworzenia za pomocą klatek kluczowych efektów ciekawych niż dostępne w Tween.js proste przejścia. Klatki kluczowe można łatwo reprezentować jako tablice kluczy i wartości, aby animator mógł ustawać po kolei klatki o różnym czasie trwania. W praktyce programiści rzadko budują tego rodzaju animacje ręcznie. Najczęściej pracę tę wykonują artyści przy użyciu specjalnych narzędzi. Tak powinno się tworzyć skomplikowane efekty, zwłaszcza angażujące wiele obiektów. Są one tematem kolejnego podrozdziału.

Animacje obiektów połączonych z użyciem klatek kluczowych

Opisane do tej pory techniki animacji służą do poruszania pojedynczymi obiektami w miejscu (np. obracania) i w obrębie sceny. Jednak za ich pomocą i z wykorzystaniem hierarchii przekształceń można tworzyć skomplikowane ruchy złożonych obiektów.

Załóżmy, że chcemy utworzyć robota, który idzie do przodu i macha rękami. Jako modelu możemy użyć struktury hierarchicznej: ciało robota składa się z górnej i dolnej części. Część górną zawiera ręce i tors, ręce składają się z części górnej, dolnej itd. Jeśli odpowiednio skonstruujemy hierarchię i poddamy animacji odpowiednie części robota, sprawimy, że będzie ruszał nogami i rękami. Technika polegająca na łączeniu hierarchii osobnych części i animowaniu ich kombinacji nazywa się **animacją obiektów połączonych** (ang. *articulated animation*).

Ciekawą demonstrację tego rodzaju animacji można znaleźć w przykładach do biblioteki Three.js, w pliku *examples/webgl_loader_collada_keyframe.html*. Gdy go otworzysz w przeglądarce, zobaczysz animację obrazującą działanie pompy. Pompa obraca się i widać, jak jest składana oraz rozkładana na części. Można zobaczyć zawory, uszczelki, koła zębate, obudowy oraz śruby. Każda z części jest osobno animowana, ale dzięki hierarchii przekształceń biblioteki Three.js wszystkie poruszają się razem ze swoimi elementami nadzorowanymi. To pozwoliło na przedstawienie sposobu składania i rozkładania całego urządzenia. Na rysunku 5.5 przedstawiony jest zrzut ekranu.



Rysunek 5.5. Animacja obiektów połączonych: działanie pompy przedstawione przy użyciu klatek kluczowych w połączeniu z hierarchią przekształceń (model COLLADA utworzony przy użyciu otwartego systemu Kuda (<https://code.google.com/p/kuda/>))

Model ten jest wczytywany z pliku w formacie COLLADA (o rozszerzeniu *.dae*). Jest to format tekstowy XML służący do opisu treści trójwymiarowej. Przy jego użyciu reprezentować można i pojedyncze modele, i całe sceny. Obsługuje materiały, światła, kamery oraz animacje. Nie będę go dogłębnie opisywać, ale dane klatki kluczowej COLLADA wyglądają podobnie do przedstawionych poniżej, które pochodzą z modelu pompy (plik *examples/models/collada/pump/pump.dae*).

```
<animation id="camTrick_G.translate_camTrick_G">
  <source id="camTrick_G..." name="camTrick_G...">
    <float_array id="camTrick_G..." count="3">0.04166662 ... </float_array>
    <source id="camTrick_G..." name="camTrick_G...">
      <float_array id="camTrick_G..." count="3">8.637086 ... </float_array>
```

Element `<animation>` definiuje animację. Widoczne w nim elementy potomne `<float_array>` w tym przypadku definiują odpowiednio klucze i wartości potrzebne do animowania składnika *x* przekształcenia obiektu o nazwie *camTrick_G*. Klucze są podane w sekundach. W czasie 7,08333 sekundy *camTrick_G* przesunie się po osi *x* od 8,637086 do 0. Jest jeszcze dodatkowy klucz na sekundzie 6,5, który określa przesunięcie po osi *x* o 7,794443. Zatem w tej animacji mamy dość wolne przesunięcie wg osi *x* w czasie 6,5 sekundy, po którym następuje przyspieszenie w czasie pozostałych 0,58333 sekundy. W pliku tym znajduje się kilkadesięć takich elementów animacyjnych (dokładnie 74) dla różnych obiektów składających się na model pompy.

Na listingu 5.6 przedstawiony jest wyciąg z kodu dotyczącego konfigurowania animacji. Wykorzystano w nim wbudowane klasy Three.js THREE.KeyFrameAnimation i THREE.AnimationHandler. Klasa THREE.KeyFrameAnimation implementuje ogólną animację klatkową do użytku z formatem COLLADA i innymi tego typu formatami. Natomiast THREE.AnimationHandler to klasa singletonowa zarządzająca listą animacji na scenie i odpowiedzialna za ich aktualizowanie w każdej iteracji pętli wykonawczej programu. (Kod źródłowy tych klas znajduje się w projekcie Three.js w folderze *src/extras/animation*).

Listing 5.6. Inicjacja animacji klatkowych w Three.js

```
var animHandler = THREE.AnimationHandler;

for ( var i = 0; i < kfAnimationsLength; ++i ) {

    var animation = animations[ i ];
    animHandler.add( animation );

    var kfAnimation = new THREE.KeyFrameAnimation(
        animation.node, animation.name );
    kfAnimation.timeScale = 1;
    kfAnimations.push( kfAnimation );
}

}
```

W kodzie tym wykonywanych jest kilka czynności przygotowawczych przed wywołaniem metody `play()` każdej z animacji. Metoda ta przyjmuje dwa argumenty: znacznik `loop` oraz opcjonalny czas rozpoczęcia (zerowa wartość domyślna oznacza natychmiastowe uruchomienie):

```
animation.play( false, 0 );
```

Przykład ten stanowi ilustrację łączenia animacji opartej na klatkach kluczowych z hierarchią przekształceń w celu utworzenia złożonych, połączonych efektów. Animacja obiektów połączonych to typowa technika wykorzystywana do animowania przedmiotów mechanicznych. Dalej w tym rozdziale piszę, że stanowi ona również podstawę mechanizmu poruszającego szkieletem w animacji obiektów pokrytych powłoką (ang. *skinned animation*).



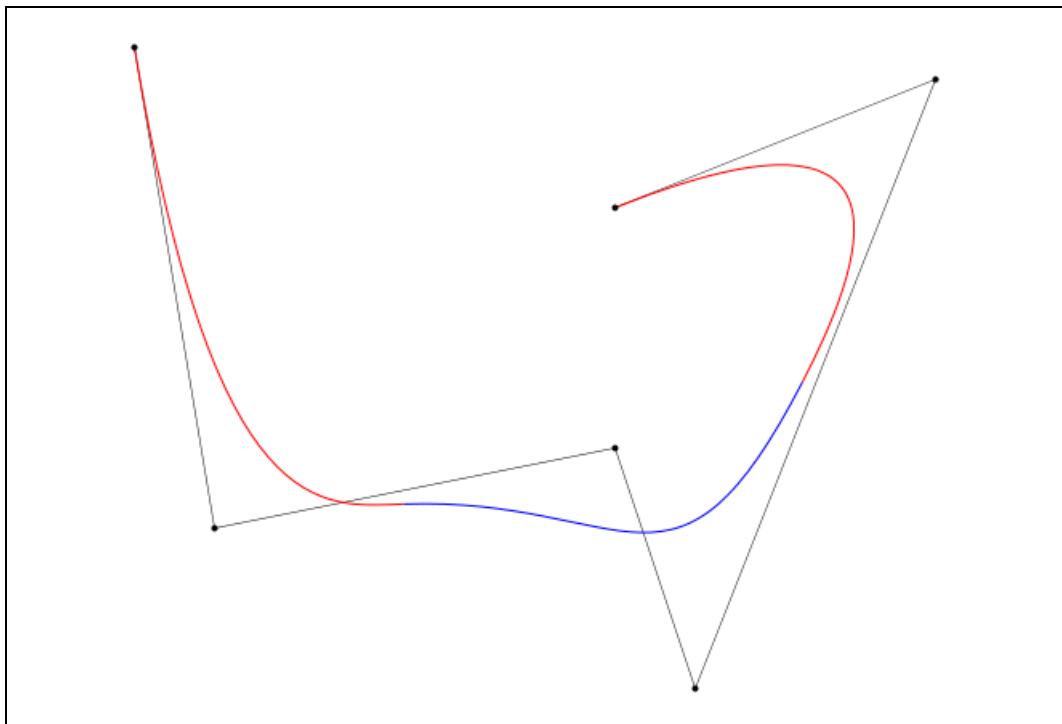
Mechanizm wczytywania plików COLLADA, podobnie jak inne takie mechanizmy w Three.js, nie należy do głównego pakietu, tylko znajduje się w przykładach. Kod źródłowy tego programu znajduje się w pliku *examples/js/loaders/ColladaLoader.js*. Szczegółowy opis formatu COLLADA podaje w rozdziale 8.

Tworzenie wrażenia płynnego ruchu przy użyciu krzywych i śledzenia ścieżki

Klatki kluczowe są doskonałym sposobem na określenie sekwencji przemian następujących w zmieniających się odstępach czasu. Dodając animację obiektów łączonych do hierarchii, możemy uzyskać skomplikowane interakcje. Jednak przedstawione do tej pory przykłady wyglądają nienaturalnie, ponieważ używane w nich są liniowe funkcje interpolacyjne. W rzeczywistym świecie ruch odbywa się po krzywych: samochody jeżdżą po drogach z zakrętami, samoloty latają po zakrzywionych trajektoriach, pociski spadają po łuku itd. Próba symulowania tych efektów przy użyciu interpolacji liniowej daje nieprzyjemne i nienaturalne efekty.

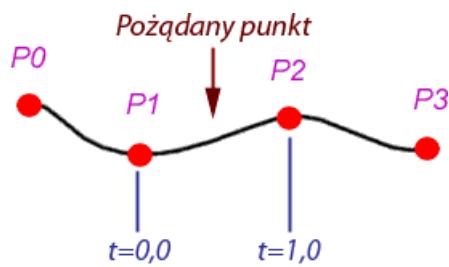
Moglibyśmy użyć silnika fizycznego, ale w wielu przypadkach byłby to przerost formy nad treścią. Czasami potrzebujemy tylko predefiniowanej, naturalnie wyglądającej animacji bez wykonywania skomplikowanych i obciążających systemów obliczeń fizycznych.

Z pomocą danych klatek kluczowych można opisywać nie tylko liniowe animacje. Można je też traktować jak punkty definujące krzywą. Najczęściej w animacjach używana jest gładka, ciągła linia, zwana **krzywą klejaną** lub **krzywą składaną** (ang. *spline curve*). W grafice komputerowej szczególnie często używane są tzw. **krzywe B-sklejane** (ang. *B-spline*), które można względnie szybko obliczać. Krzywą taką definiuje się przy użyciu zbioru punktów danych określających jej ogólny kształt oraz dodatkowych **punktów kontrolnych** dostosowujących go szczegółowo. Na rysunku 5.6 przedstawiona jest nieskomplikowana krzywa B-sklejana z punktami kontrolnymi oznaczonymi czarnymi kropkami. Każdy, kto do rysowania używa profesjonalnych programów, takich jak np. Adobe Illustrator, dobrze wie, czym są punkty kontrolne do ustawiania kształtu krzywej.



Rysunek 5.6. Krzywa B-składana Wojciecha Muły (licencja Creative Commons CC0 1.0 Universal Public Domain Dedication)

Interpolacja krzywej sklejanej jest nieco bardziej skomplikowana niż zwykła interpolacja liniowa. Należy posłużyć się wzorami wielomianowymi, podobnymi do użytych w funkcjach określania prędkości animacji w bibliotece Tween.js, oraz dodatkowymi wartościami, po jednej z każdej strony kluczowej wartości. Opis interpolacji krzywych sklejanych wychodzi poza ramy tematyczne tej książki, ale rysunek 5.7 pozwala zrozumieć z grubsza, jak to działa: aby obliczyć interpolowaną wartość na krzywej między punktami P₁ i P₂, używamy także punktów kontrolnych P₀ i P₃.



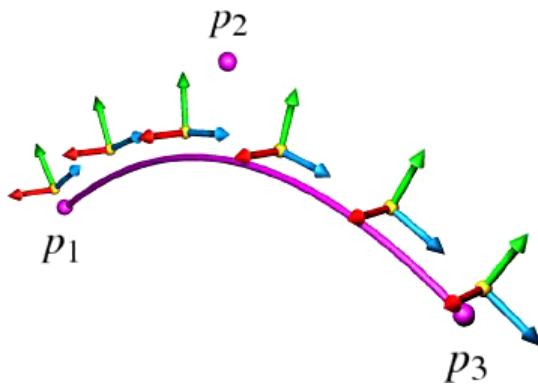
Rysunek 5.7. Interpolacja na krzywej sklejanej (<http://www.mvps.org/directx/articles/catmull/>); publikacja za pozwoleniem autora



Istnieje wiele rodzajów krzywych sklejanych, np. **krzywe B-sklejane**, **sześcienne krzywe sklejane Béziera** oraz **krzywe sklejane Catmulla-Roma** (nazwa pochodzi od nazwiska genialnego założyciela wytwórni Pixar Eda Catmulla). Krzywe Catmulla-Roma zyskały popularność dzięki temu, że łatwiej się je tworzy i oblicza niż krzywe Béziera. Biblioteka Three.js zawiera gotową klasę animacyjną wykorzystującą interpolację tych krzywych. Kod źródłowy tej klasy znajduje się w pliku `src/extras/animations/animation.js`.

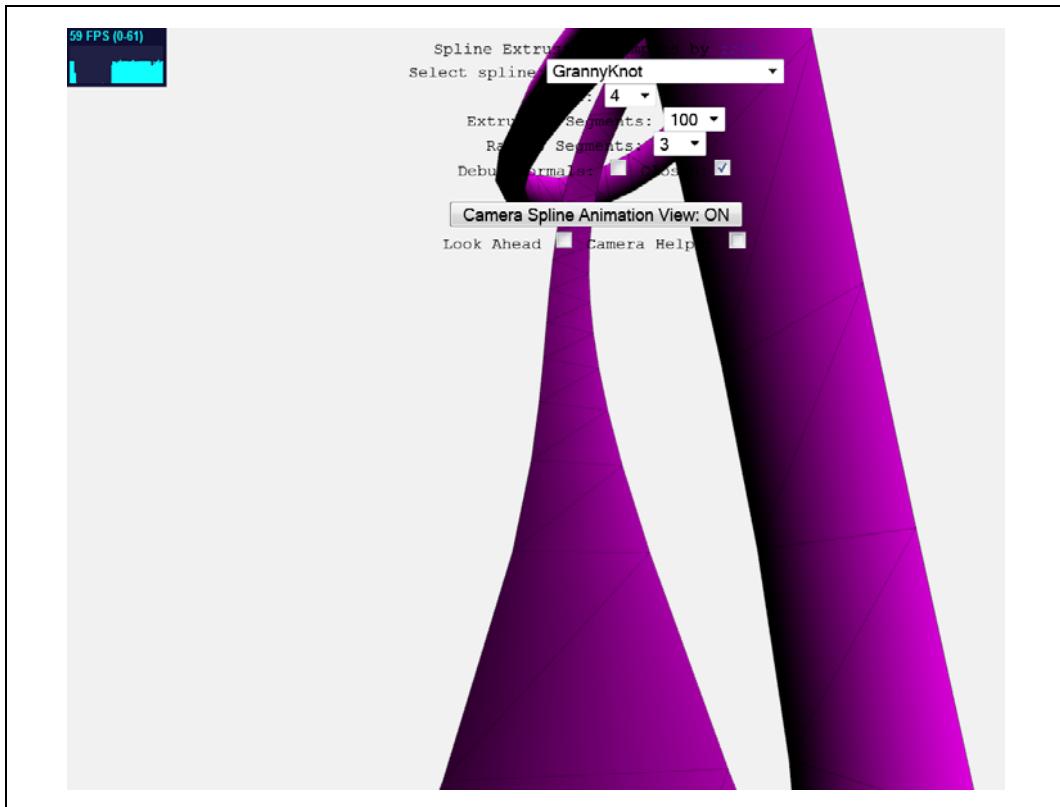
W internecie można znaleźć kilka dobrych samouczków dotyczących używania krzywej Catmulla-Roma, np. <http://flashcove.net/795/cubic-spline-generation-in-as3-catmull-romcurves/> i <http://www.mvps.org/directx/articles/catmull/>.

W animacji opartej na krzywych sklejanych często uwzględnia się orientację i pozycję. Jeśli np. chcemy naturalnie animować obiekt poruszający się po krzywej, musimy sprawić, żeby się obracał, pochylał i toczył. Proces ten jest przedstawiony na rysunku 5.8. W każdym punkcie oznaczonym na tej krzywej obliczane są **styczna**, **normalna** oraz **dwunormalna (binormalna)**. Styczna to prosta linia biegąca w tym samym kierunku, co krzywa i przecinającą ją tylko w jednym miejscu. Normalna to linia prostopadła do kierunku krzywej (a więc i do stycznej). Dwunormalna to iloczyn wektorowy tych dwóch linii. Razem te trzy wektory definiują ramę odniesienia zwaną **ramą TNB** (ang. *TNB frame*), która ustala orientację dla obiektu poruszającego się po ścieżce.



Rysunek 5.8. Ramki współrzędnych animacji opartej na krzywej sklejanej (<http://circecharacterworks.wordpress.com/skinning/>); styczne, normalne i binormalne są oznaczone strzałkami niebieską (do przodu), zieloną (do góry) oraz czerwoną (w prawo). Obraz opublikowany dzięki uprzejmości Cedrika Bazillou

Wśród przykładowych plików biblioteki Three.js znajduje się ciekawy przykład animacji ścieżkowej. Otwórz plik *examples/webgl_geometry_extrude_splines.html* i naciśnij przycisk *Camera Spline Animation View* (widok z kamery animacji opartej na krzywej sklejanej), aby zobaczyć animację pokazaną na rysunku 5.9. Kamera porusza się po krzywej sklejanej, ciągle korygując swoją pozycję i orientację. Przykład ten jest animacją w pełni komputerową utworzoną przez obliczanie interpolacji na krzywej sklejanej oraz przy użyciu ramy TNB. Gdyby było trzeba, można by utworzyć z niej klasę.

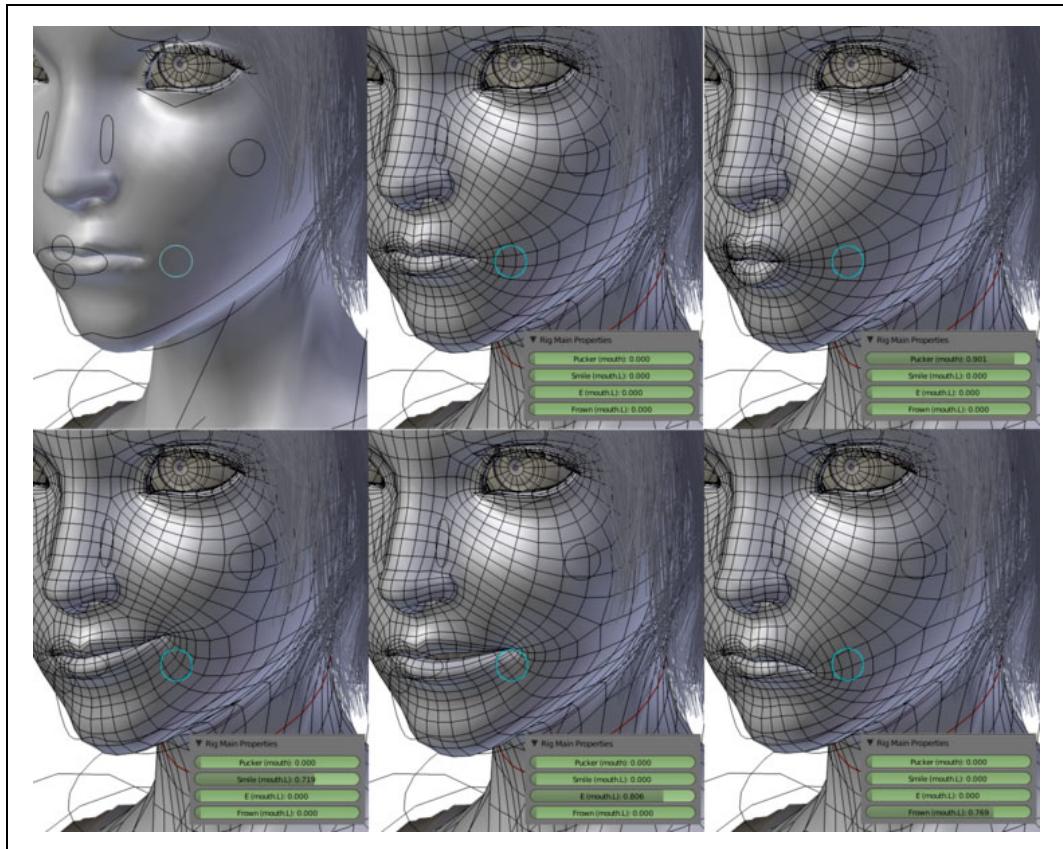


Rysunek 5.9. Animacja widoku z kamery podążającej po ścieżce

Animacja postaci i twarzy przy użyciu morfingu

Klatki kluczowe i animacja obiektów połączonych to techniki doskonale nadające się do poruszania obiektami w obrębie sceny, ale wiele efektów animacyjnych wymaga zmiany geometrii animowanego obiektu. Jednym ze sposobów realizacji takiego pomysłu jest zastosowanie tzw. **animacji opartej na celach morfingu** (ang. *morph target animation*) albo krócej po prostu **morfingu**. W metodzie tej wykorzystuje się interpolacje obliczane na podstawie wierzchołków do zmiany wierzchołków siatki. Zazwyczaj zapisuje się podzbior wierzchołków siatki, wraz z indeksami, jako zbiór **celów morfingu**, które mają zostać użyte w międzyklatce. Międzyklatka oblicza wartość interpolowaną dla każdej pary wartości wierzchołków w celach morfingu, a następnie te interpolowane wartości są używane do deformowania wierzchołków w siatce.

Cele morfingu doskonale nadają się do animowania mimiki i innych drobnych szczegółów, których nie da się łatwo zaimplementować przy użyciu animacji szkieletowej (następny podrozdział). Są one kompaktowe i nie jest potrzebny bardzo szczegółowy szkielet zawierający wiele kości twarzy. Ponadto przy użyciu celów morfingu animator może utworzyć bardzo specyficzne grymasły, ponieważ ma możliwość modyfikowania siatki na poziomie wierzchołków. Na rysunku 5.10 można zobaczyć efekt użycia morfingu w celu uzyskania różnych wyrazów twarzy. Każdy wyraz, np. wygięte wargi albo uśmiech, jest reprezentowany przez zbiór wierzchołków obejmujących usta i obszar wokół nich.



Rysunek 5.10. Morfingi wyrazu twarzy (<http://en.wikipedia.org/wiki/File:Sintel-face-morph.png>); licencja Uznanie autorstwa-Na tych samych warunkach 3.0 Unported

Morfingu można używać także do innych celów animacyjnych. W bibliotece Three.js znajduje się kilka przykładów animacji całych postaci wykonanych tą techniką. Na rysunku 5.11 widać utworzone w ten sposób potwory. Modele utworzono w pliku w formacie id Software MD2, który jest często używany do przechowywania postaci z gier tej firmy, np. *Quake II*. Plik MD2 został następnie przekonwertowany na format JSON biblioteki Three.js (rozdział 8.).



Rysunek 5.11. Postaci animowanie przy użyciu celów morfingu; modele zostały przekonwertowane z formatu MD2 na format JSON biblioteki Three.js (ogry autorstwa Magarnigala — <http://bit.ly/L0ppGl>)

Aby obejrzeć te animacje w przeglądarce internetowej, otwórz plik `examples/webgl_morphtargets_md2_control.htm`. Zobaczysz kilka człapiących, kręcących się i rozglądających na wszystkie strony ogrów. Za pomocą klawiszy strzałek i przycisków WASD można nimi poruszać. Cały efekt wygląda naprawdę bardzo interesująco.

Jeśli chcesz zobaczyć, jak wyglądają dane w formacie MD2, otwórz przekonwertowany plik MD2 `examples/models/animated/ogro/ogro-light.js`. Około 18. wiersza znajdziesz własność JSON o następującym początku:

```
"morphTargets": [  
  { "name": "stand001", "vertices": [0.6,-2.7,1.5,-5.5,-3.3,-0.6 ...
```

Dane te ciągną się przez kilka linijek. Każdy element tablicy `morphTargets` jest celem morfingu. Każdy taki cel zawiera kompletny zbiór wierzchołków siatki ogra, a poszczególne cele różnią się wartościami pozycji. Biblioteka Three.js wykonuje animację morfingu w ten sposób, że przegląda cyklicznie zbiór celów dla modelu i interpoluje wartości wierzchołków, aby utworzyć płynne przejścia od jednego celu do następnego. Kod źródłowy dotyczący ładowania, konfigurowania i animowania postaci MD2 znajduje się w klasie `THREE.M2DCharacterComplex` w pliku `examples/js/MD2CharacterComplex.js`.

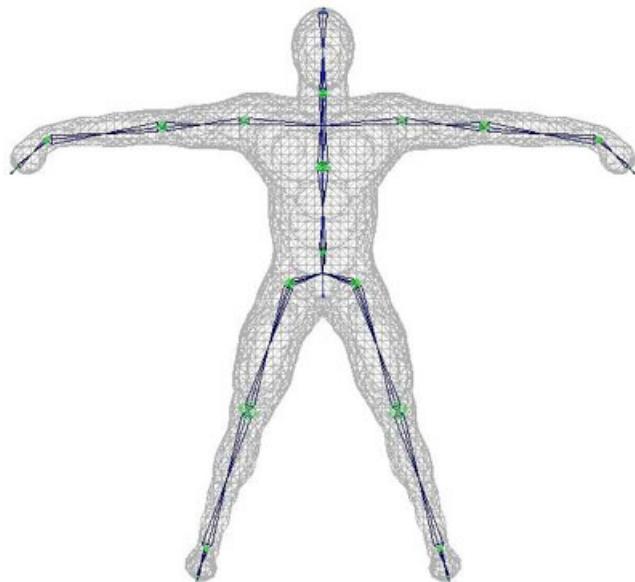


Plik MD2 dla tego przykładu został przekonwertowany na format JSON biblioteki Three.js przy użyciu fantastycznego internetowego narzędzia autorstwa Klasa, znanego również pod pseudonimem OutsideOfSociety (<http://twitter.com/oosmoxiecode>), pracownika szwedzkiej agencji interaktywnej North Kingdom. Szczegółowe informacje na temat obsługi konwertera znajdują się we wpisie na blogu Klasa (<http://oosmoxiecode.com/blog/index.php/2012/01/md2-to-json-converter>).

Animowanie postaci przy użyciu animacji szkieletowej

Animacja obiektów połączonych jest techniką odpowiednią do animowania przedmiotów nie-organicznych — robotów, samochodów, maszyn itd. Jednak dla obiektów ożywionych sprawdza się bardzo słabo. Bujanie się roślin na wietrze, skoki zwierząt i taniec ludzi to czynności wymagające modyfikacji geometrii siatki: gałęzie się skręcają, na skórze powstają zmarszczki, mięśnie się napinają. Osiągnięcie tak skomplikowanych efektów przy użyciu prostej techniki, czyli animacji obiektów połączonych, jest praktycznie niemożliwe. Dlatego skorzystamy z metody o nazwie **animacja szkieletowa** (ang. *skeletal animation*), która jest też znana pod angielskimi nazwami *skinning* oraz *single mesh animation*.

Animacja szkieletowa polega na odkształcaniu wierzchołków siatki, powłoki (ang. *skin*) w określonym czasie. Działa na bazie hierarchii połączonych obiektów zwanej **szkieletem** (ang. *skeleton*). Szkielet służy tylko jako ukryty mechanizm, na którym bazuje cała animacja, i na ekranie go nie widać. Działanie animacji opiera się na zmianach szkieletu, w połączeniu z dodatkowymi danymi opisującymi jego wpływ na zmiany powłoki w różnych miejscach siatki. Na rysunku 5.12 pokazano prosty szkielet wraz z powłoką.

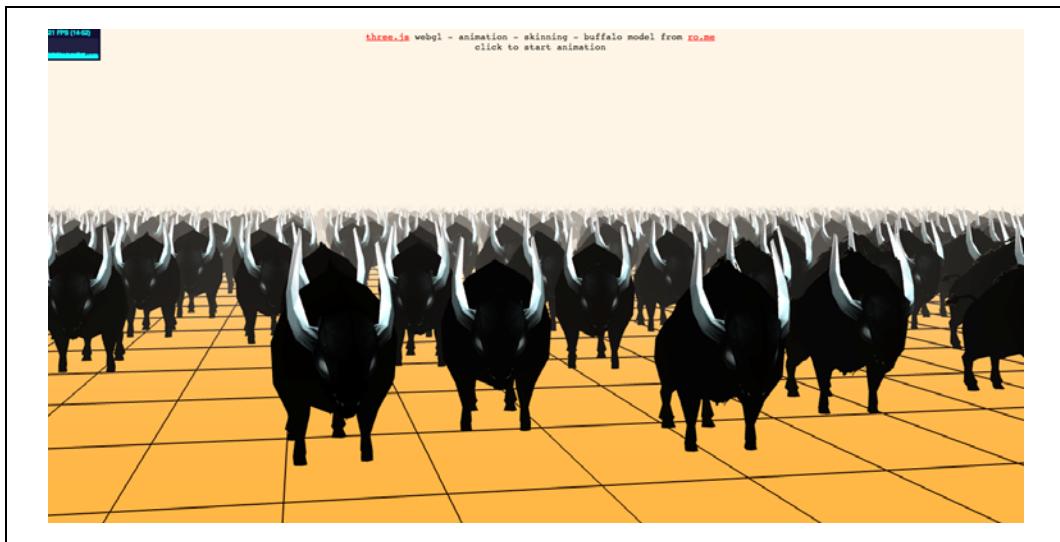


Rysunek 5.12. Siatka postaci z zaznaczonym szkieletem odpowiednia do użycia w animacji szkieletowej — z samouczka Franka A. Rivery (<http://www.animationartist.com/2000/Tutorials/trueSpaceBones/Bones.html>)

Szkielet składa się z **kości** (ang. *bones*), które są uporządkowane w strukturze hierarchicznej w intuicyjny sposób, tzn. jak w normalnym szkieletie kość stopy jest połączona z podudziem, podudzie z kolanem itd. Podobnie jak w animacji obiektów połączonych, przekształcenie kości powoduje przesunięcie wszystkich jej kości podległych. Jednak w tym przypadku szkielet jest niewidoczny.

Każda kość szkieletu jest powiązana ze zbiorem wierzchołków siatki wraz **wagą mieszania** (ang. *blend weight*) dla każdego wierzcholka. Waga mieszania określa, jak duży wpływ dana kość ma na powiązane z nią wierzchołki. Wierzchołki można wiązać z wieloma kośćmi, przez co ostateczna pozycja i orientacja wierzcholka jest określona przez kombinację wielu przekształceń i wag mieszania. Jeśli myślisz, że to skomplikowane, masz rację. Animacje szkieletowe prawie zawsze tworzy się za pomocą specjalnych narzędzi, a nie ręcznie. Algorytmy do ich obsługi również są skomplikowane. Obecnie większość silników wykonawczych animuje powłoki przy użyciu GPU, jeśli jest to możliwe. Biblioteka Three.js nie jest tu wyjątkiem.

Jeśli chcesz obejrzeć prawdziwy przykład animacji szkieletowej, otwórz plik *examples/webgl_animation_skinning.html* z folderu biblioteki Three.js. Na ekranie zobaczysz stado modeli byków. Kliknij, aby rozpocząć animację. Byki będą biegły w miejscu, poruszając się w naturalny sposób (rysunek 5.13).



Rysunek 5.13. Animacja szkieletowa zrealizowana przy użyciu biblioteki Three.js; model byka wzięty z RO.ME

Przeanalizujemy teraz fragment kodu źródłowego tego przykładu, aby dowiedzieć się, w jaki sposób animacje szkieletowe są zaimplementowane w bibliotece Three.js. Najpierw wczytujemy model byka. W tym celu tworzymy obiekt klasy THREE.JSONLoader i wywołujemy jego metodę `load()`. Klasa ta wczytuje pliki w formacie JSON biblioteki Three.js. Format ten zawiera informacje powłokowe oraz geometryczne.

```
var loader = new THREE.JSONLoader();
loader.load( "obj/buffalo/buffalo.js", createScene );
```

Metoda `load()` pobiera jako drugi argument funkcję zwrotną, która zostanie wywołana po pobraniu i przetworzeniu pliku. Na listingu 5.7 pokazany jest fragment funkcji zwrotnej `createScene()`, w którym najważniejsze wiersze zaznaczono pogrubieniem.

Listing 5.7. Funkcja zwrotna konfigurująca animację szkieletową po załadowaniu pliku

```
function createScene( geometry, materials ) {

    buffalos = [];
    animations = [];
```

```

var x, y,
    buffalo, animation,
    gridx = 25, gridz = 15,
    sepX = 150, sepZ = 300;

var material = new THREE.MeshFaceMaterial( materials );

var originalMaterial = materials[ 0 ];

originalMaterial.skinning = true;
originalMaterial.transparent = true;
originalMaterial.alphaTest = 0.75;

THREE.AnimationHandler.add( geometry.animation );

for( x = 0; x < gridx; x ++ ) {

    for( z = 0; z < gridz; z ++ ) {

        buffalo = new THREE.SkinnedMesh( geometry,
            material, false );

        buffalo.position.x = - ( gridx - 1 ) * sepX * 0.5 +
            x * sepX + Math.random() * 0.5 * sepX;

        buffalo.position.z = - ( gridz - 1 ) * sepZ * 0.5 +
            z * sepZ + Math.random() * 0.5 * sepZ - 500;

        buffalo.position.y =
            buffalo.geometry.boundingSphere.radius * 0.5;
        buffalo.rotation.y = 0.2 - Math.random() * 0.4;

        scene.add( buffalo );

        buffalos.push( buffalo );

        animation = new THREE.Animation( buffalo, "take_001" );
        animations.push( animation );

        offset.push( Math.random() );
    }
}

```

Funkcja `createScene()` wykonuje pętlę tworzącą wiele egzemplarzy siatki byka z jednej załadowanej porcji geometrii. Zwróć uwagę na typ tworzonej siatki: nie jest to znany już z poprzednich przykładów typ `THREE.Mesh`, tylko `THREE.SkinnedMesh`. Typ ten jest renderowany przez specjalny shader wierzchołków, który dla optymalizacji wydajności wykonuje animację szkieletową w GPU.

Funkcja `createScene()` wykorzystuje też wbudowane klasy animacyjne biblioteki Three.js `THREE.Animation` i `THREE.AnimationHandler`. Klasa `THREE.Animation` implementuje ogólną animację opartą na klatkach kluczowych, która w przypadku animacji szkieletowej jest wykorzystywana do obsługi szkieletu. Natomiast `THREE.AnimationHandler` to singletonowy obiekt przechowujący wszystkie animacje sceny i obsługujący ich aktualizowanie w każdej iteracji pętli wykonawczej programu. Nasza funkcja najpierw dodaje dane animacji do listy „handlera” animacji, wywołując metodę `THREE.AnimationHandler.add()`, potem przekazuje mu dane animacyjne geometrii, które zostały automatycznie załadowane przez mechanizm ładowania danych JSON biblioteki Three.js. Nieco później program tworzy dla każdego byka nowy obiekt klasy `THREE.Animation`, wiążąc instancję zapisaną w zmiennej `buffalo` z animacją o nazwie `take_001` z pliku JSON.

Po zakończeniu czynności związanych z przygotowywaniem animacji możemy je uruchomić. Robimy to, wywołując funkcję `startAnimation()` w reakcji na kliknięcie myszą. Spójrz na listing 5.8. Funkcja `startAnimation()` zawiera pętlę przeglądającą tablicę animacji i dla każdej z nich wywołującą funkcję `play()`. Każdej animacji przypisywane jest też inne losowo wybrane opóźnienie, aby zwierzęta nie były ze sobą idealnie zsynchronizowane.

Listing 5.8. Uruchamianie animacji szkieletowej

```
function startAnimation() {  
    for( var i = 0; i < animations.length; i ++ ) {  
        animations[ i ].offset = 0.05 * Math.random();  
        animations[ i ].play();  
    }  
  
    dz = dstep;  
    playback = true;  
}
```

Jeśli chcesz zobaczyć, jak dokładnie wyglądają dane w formacie animacyjnym JSON, otwórz plik `examples/obj/buffalo.js`. Poszukaj w nim właściwości `bones`, `skinWeights` oraz `skinIndices`, aby dowiedzieć się, w jaki sposób rozłożone są dane szkieletu. Poszukaj też właściwości `animation`, która zawiera hierarchię klatek kluczowych użytych do animacji szkieletu. W wewnętrznych mechanizmach programu dużo się dzieje, a biblioteka Three.js bardzo w tym wszystkim pomaga, np. poprzez dostarczenie opartej na shaderze implementacji animacji szkieletowej wykorzystującej do obliczeń GPU.

Animowanie przy użyciu shaderów

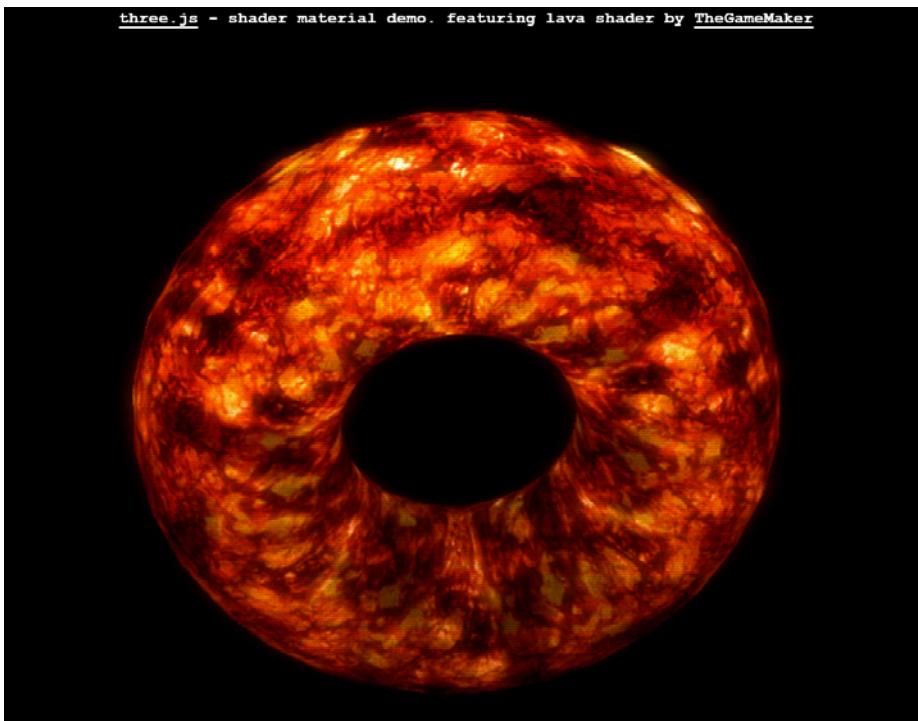
Opisane do tej pory techniki, np. klatki kluczowe, klatki pośrednie i animacja szkieletowa, można zaimplementować w języku JavaScript, ale jeśli chce się wykorzystać pełnię możliwości sprzętu, należy skorzystać z shaderów napisanych w języku GLSL. W bibliotece Three.js stosowane są obie te techniki: system oparty na klatkach kluczowych jest napisany w JavaScriptie, a morfing i animacje szkieletowe zaimplementowano we wbudowanych shaderach dla typów materiałów, takich jak *Phong* i *Lambert*. Jeśli w siatce obecne są dane dotyczące animacji szkieletowej lub morfingu (opisane wcześniej typy `THREE.SkinnedMesh` lub `THREE.MorphAnimMesh`), shader biblioteki Three.js wykorzystuje je do obliczania nowych pozycji wierzchołków.



Jeżeli chcesz zobaczyć, jak w bibliotece Three.js wygląda kod GLSL dotyczący animacji szkieletowej i morfingu, otwórz plik `src/renderers/WebGLShaders.js` i poszukaj słów `skin` oraz `morph`. Wiedz jednak, że w kodzie tym zastosowano zaawansowane techniki języka GLSL oraz implementacji biblioteki Three.js. Jeśli potrapisz się, o co w tym wszystkim chodzi, warto tam zajrzeć, gdyż znajdziesz mnóstwo cennych informacji.

Z pomocą kodu GLSL można nie tylko zoptymalizować wydajność typowych technik, takich jak animacja szkieletowa, ale również tworzyć dowolne efekty wizualne, np. sprawić, aby tafla oceanu polyskiwała, aby światło odbijało się od fal i załamywało się na nich albo można utworzyć ląkę kołyszącą się na lekkim wietrzu. Wszystkie te efekty można otrzymać przy użyciu JavaScriptu, ale język GLSL jest znacznie lepiej przystosowany do przetwarzania dużych ilości danych wierzchołkowych i graficznych. W bibliotece Three.js znajduje się fantastyczny przykład

animacji utworzonej na bazie shaderów. Otwórz plik *examples/webgl_shader_lava.html*, aby wyświetlić w oknie przeglądarki wolno obracający się torus o powierzchni pokrytej rozpaloną lawą (rysunek 5.14).



Rysunek 5.14. Animowany efekt lawy utworzony przy użyciu shadera GLSL; autorem kodu shadera jest TheGameMaker (<http://irrlicht.sourceforge.net/forum/viewtopic.php?t=21057>)

Animacja lawy jest wykonana przy użyciu klasy THREE.ShaderMaterial i kodu źródłowego w języku GLSL. Przyjrzyjmy się mu. Na listingu 5.9 znajduje się kod tworzący obiekt ShaderMaterial. Do shadera przekazywanych jest kilka jednolitych wartości. Dla nas w tej chwili najważniejsze są `time` oraz tekstury `texture1` i `texture2`. Jak wkrótce zobaczysz, te trzy parametry w połączeniu zmagicznymi obliczeniami pozwalają otrzymać realistycznie wyglądającą lawę.

Listing 5.9. Tworzenie siatki torusa i obiektu ShaderMaterial

```
uniforms = {  
    fogDensity: { type: "f", value: 0.45 },  
    fogColor: { type: "v3",  
        value: new THREE.Vector3( 0, 0, 0 ) },  
    time: { type: "f", value: 1.0 },  
    resolution: { type: "v2",  
        value: new THREE.Vector2() },  
    uvScale: { type: "v2",  
        value: new THREE.Vector2( 3.0, 1.0 ) },  
    texture1: { type: "t",  
        value: THREE.ImageUtils.loadTexture(  
            "textures/lava/cloud.png" ) },
```

```

    texture2: { type: "t",
      value: THREE.ImageUtils.loadTexture(
        "textures/lava/lavatile.jpg" ) }
};

uniforms.texture1.value.wrapS =
  uniforms.texture1.value.wrapT = THREE.RepeatWrapping;
uniforms.texture2.value.wrapS =
  uniforms.texture2.value.wrapT = THREE.RepeatWrapping;

```

Mając gotowe zmienne typu uniform, możemy utworzyć materiał dla shadera. W tym celu musimy przekazać kod GLSL shaderów fragmentów i wierzchołków do konstruktora. Zwróć uwagę na zastosowaną do tego technikę: w kodzie HTML kod źródłowy GLSL zapisaliśmy w elementach <script>, a następnie pobraliśmy go z właściwości.textContent tych elementów. Porównaj to z wcześniejszymi przykładami shaderów. Zamiast tworzyć wielowierszowe łańcuchy tekstu ze znakami nowego wiersza, napisaliśmy kod shadera w prostszy sposób. Kolejnym GLSL zajmiemy się za chwilę.

```

var size = 0.65;

material = new THREE.ShaderMaterial( {

  uniforms: uniforms,
  vertexShader: document.getElementById(
    'vertexShader' ).textContent,
  fragmentShader: document.getElementById(
    'fragmentShader' ).textContent
} );

```

Następnie tworzymy siatkę torusa przy użyciu nowego obiektu klasy THREE.ShaderMaterial i dodajemy ją do sceny:

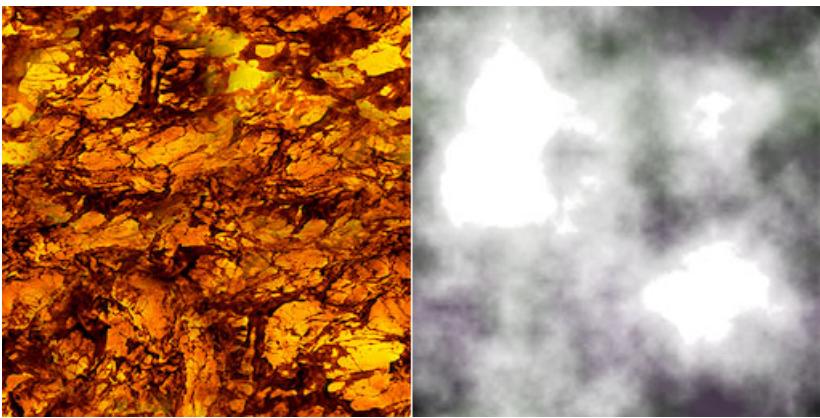
```

mesh = new THREE.Mesh(
  new THREE.TorusGeometry( size, 0.3, 30, 30 ),
  material );
mesh.rotation.x = 0.3;
scene.add( mesh );

```

Algorytm shadera jest bardzo sprytny. Łączy dwie tekstury, z których jedna zawiera podstawowy kolor lawy i wzorzec wizualny, a druga zawiera chmury „zaklaczające” podstawową strukturę w czasie, tworząc efekt przepływu. Obie te tekstury są przedstawione na rysunku 5.15.

Pokazany na listingu 5.10 kod GLSL shadera wierzchołków jest bardzo prosty. Jak większość shaderów, wykonuje obliczenia dotyczące przekształceń, polegające na pomnożeniu wierzchołków przez macierze modelu, widoku i rzutowania, aby przenieść te wierzchołki do przestrzeni ekranu i zapisać je we wbudowanej zmiennej GLSL o nazwie gl_Position. Dodatkowo zadeklarowaliśmy parametr rodzaju varying o nazwie vUv. Jest to współrzędna teksturowa dla każdego wierzchołka, którą shader wierzchołków zwraca, aby można jej było użyć w shaderze fragmentów, o którym będzie mowa za chwilę. Ten shader przyjmuje także parametr skali, którego używa do skalowania współrzędnych teksturowych.



Rysunek 5.15. Tekstury lawy i zakłceń

Listing 5.10. Shader wierzchołków osadzony w elemencie `<script>` języka HTML

```
<script id="vertexShader" type="x-shader/x-vertex">
```

```
uniform vec2 uvScale;
varying vec2 vUv;

void main()
{
    vUv = uvScale * uv;
    vec4 mvPosition = modelViewMatrix * vec4( position, 1.0 );
    gl_Position = projectionMatrix * mvPosition;
}
</script>
```

Jak napisalem, kod GLSL znajduje się w elemencie `<script>`, a więc można go łatwo odczytać bez posługiwania się cudzysłowami, znakami nowego wiersz itd. Sztuczka polega na użyciu innej niż normalnie wartości atrybutu `type` skryptu: `x-shader/x-vertex`. Przeglądarki internetowe nie rozpoznają tego typu i użyliśmy go tylko po to, by zaznaczyć, że dany element *nie zawiera* kodu JavaScript.

Większość pracy wykonuje kod GLSL shadera fragmentów, który jest przedstawiony na listingu 5.11. Najpierw deklarujemy parametry `uniform` odpowiadające użytkom w kodzie JavaScript, a następnie deklarujemy parametr `varying vUv`, pasujący do wyjścia shadera wierzchołków.

Listing 5.11. Kod shadera fragmentów animacji

```
<script id="fragmentShader" type="x-shader/x-fragment">

uniform float time;
uniform vec2 resolution;

uniform float fogDensity;

uniform vec3 fogColor;

uniform sampler2D texture1;
uniform sampler2D texture2;

varying vec2 vUv;
```

Teraz przyjrzymy się głównemu programowi cieniującemu. Jego istotą jest to, że tekstura przedstawiająca chmurę, `texture1`, służy jako źródło zakłóceń powodujące nieznaczne przesunięcie wartości współrzędnej teksturowej używanej do pobierania wartości kolorów z `texture2`, czyli tekstury przedstawiającej lawę. (Funkcja `texture2D()` języka GLSL pobiera dane koloru z teksturowy na podstawie dwuwymiarowej współrzędnej teksturowej). Mnożąc współrzędną teksturowy zakłóceniowej przez bieżącą wartość zmiennej `time` oraz dodając parę doświadczalnie określonych przesunięć (np. 1.5, -1.5), otrzymujemy efekt przepływającej lawy. Następnie wartość koloru piksela jest zapisywana we wbudowanej zmiennej GLSL `gl_FragColor`.

```
void main( void ) {
    vec2 position = -1.0 + 2.0 * vUv;

    vec4 noise = texture2D( texture1, vUv );
    vec2 T1 = vUv + vec2( 1.5, -1.5 ) * time * 0.02;
    vec2 T2 = vUv + vec2( -0.5, 2.0 ) * time * 0.01;

    T1.x += noise.x * 2.0;
    T1.y += noise.y * 2.0;
    T2.x -= noise.y * 0.2;
    T2.y += noise.z * 0.2;

    float p = texture2D( texture1, T1 * 2.0 ).a;

    vec4 color = texture2D( texture2, T2 * 2.0 );
    vec4 temp = color * ( vec4( p, p, p, p ) * 2.0 ) +
        ( color * color - 0.1 );

    if( temp.r > 1.0 ){ temp.bg += clamp( temp.r - 2.0, 0.0, 100.0 ); }
    if( temp.g > 1.0 ){ temp.rb += temp.g - 1.0; }
    if( temp.b > 1.0 ){ temp.rg += temp.b - 1.0; }

    gl_FragColor = temp;
```

W tym momencie efekt płynącej lawy jest gotowy. Jednak w shaderze tym dodana jest też mgła. Wartość zapisana w zmiennej `gl_FragColor` jest mieszana z wartością mgły *obliczoną* przy użyciu przekazanych do shadera parametrów mgły. Ostateczna wartość koloru piksela zostaje zapisana we wbudowanej zmiennej GLSL `gl_FragColor`.

```
float depth = gl_FragCoord.z / gl_FragCoord.w;
const float LOG2 = 1.442695;
float fogFactor = exp2( - fogDensity * fogDensity * depth * depth * LOG2 );
fogFactor = 1.0 - clamp( fogFactor, 0.0, 1.0 );
gl_FragColor = mix( gl_FragColor, vec4( fogColor, gl_FragColor.w ), fogFactor );
}
```

`</script>`

Pozostało już tylko włączyć sterowanie animacją w pętli wykonawczej poprzez aktualizowanie wartości `time` w każdej iteracji. W bibliotece Three.js jest to banalnie łatwe, ponieważ automatycznie przekazuje ona wszystkie wartości rodzaju `uniform` do shaderów GLSL przy każdej aktualizacji renderera. Programista musi tylko ustawić własność w JavaScriptie. W tym przykładzie funkcja `render()` jest wywoływana w każdej klatce animacji. Spójrz na pogrubiony wiersz kodu.

```
function render() {
    var delta = 5 * clock.getDelta();

    uniforms.time.value += 0.2 * delta;
```

```
    mesh.rotation.y += 0.0125 * delta;  
    mesh.rotation.x += 0.05 * delta;  
  
    renderer.clear();  
    composer.render( 0.01 );  
}
```

Bez wątpienia do kodowania takich animacji potrzebny jest zmysł artystyczny. Trzeba nie tylko znać składnię języka GLSL i jego wbudowane funkcje, ale również opanować pewne osobliwe algorytmy grafiki komputerowej. Jeśli sobie poradzisz, możesz wiele zyskać. W internecie jest mnóstwo cennych informacji i gotowych przykładów, od których można zacząć naukę.

Podsumowanie

W rozdziale tym dowiedziałeś się, że istnieje wiele technik animacji grafiki trójwymiarowej przy użyciu biblioteki WebGL. W istocie każdą animacją steruje nowa przeglądarkowa funkcja o nazwie `requestAnimationFrame()`, która zapewnia, że rysowanie na stronie internetowej odbywa się w odpowiednim czasie i we właściwy sposób. Dodatkowo istnieje wiele metod animowania obiektów, od prostych do skomplikowanych. Wybór jednej z nich zależy od tego, jaki efekt chce się uzyskać. Treść można animować programowo w klatkach albo używać metod opartych na wykorzystaniu danych, do których zaliczają się generowanie klatek pośrednich, stosowanie klatek kluczowych, morfing oraz animowanie szkieletowe. Naturalnie wyglądający ruch można uzyskać, łącząc klatki kluczowe z podążaniem po ścieżce. Aby obsługiwać animację przez GPU, można użyć shaderów, które dają programiście jeszcze większe możliwości działania. Narzędzia i biblioteki służące do tworzenia animacji przy użyciu biblioteki WebGL ciągle się rozwijają, więc nie da się powiedzieć, które jest najlepsze. Jednak mamy wiele możliwości, a dzięki językowi JavaScript i społeczności open source nie musimy pokonywać zbyt wielu barier.

Tworzenie zaawansowanych efektów na stronach przy użyciu CSS3

W kilku poprzednich rozdziałach pokazałem, jak za pomocą WebGL i przy wykorzystaniu pełni możliwości sprzętu tworzyć fantastyczne trójwymiarowe obiekty, sceny i animacje. Jednak, mimo wielu zalet, biblioteka WebGL ma jedną podstawową wadę, która polega na tym, że nie można nakładać treści HTML jako tekstury na powierzchnię trójwymiarowych obiektów. Gdybyśmy chcieli zastosować opisane w poprzednich rozdziałach techniki do elementów strony internetowej, musimy skorzystać z innej nowości, jaką jest CSS3.

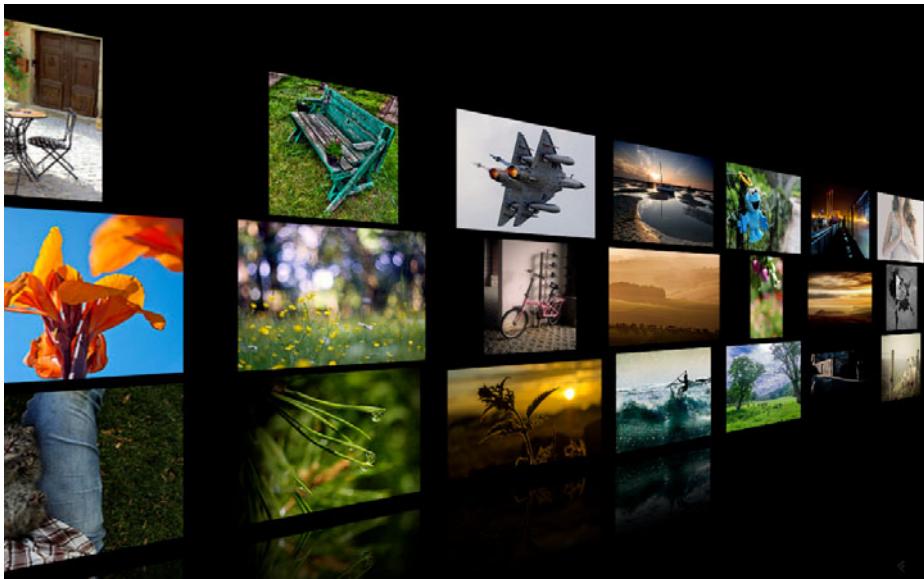
Dzięki CSS3 można ożywiać za pomocą animacji, filtrów oraz dwu- i trójwymiarowych przekształceń wybrane elementy albo całe strony internetowe. Techniki te umożliwiają tworzenie rozmaitych efektów wizualnych, które można wykorzystać w prostych grach, do opracowywania przykuwających uwagę reklam oraz intuicyjnych w obsłudze interfejsów użytkownika. W odróżnieniu od biblioteki WebGL, do obsługi której potrzebna jest przynajmniej znajomość podstawowych zagadnień dotyczących programowania grafiki trójwymiarowej i jakiejś dodatkowej biblioteki, np. Three.js, do używania CSS3 wystarczy znajomość języków HTML, CSS oraz podstaw JavaScriptu. Dodatkowo pomocna jest umiejętność posługiwania się bibliotekami pomocniczymi, takimi jak np. jQuery. To sprawia, że praca z CSS3 jest znacznie łatwiejsza niż z WebGL, chociaż ma się dostęp tylko do tego, co zostało udostępnione przez przeglądarkę. Krótko mówiąc, techniki trójwymiarowe w CSS są proste, ale za cenę elastyczności i redukcji możliwości.

Funkcje trójwymiarowe języka CSS3 sięgają korzeniami trójwymiarowych przejść opracowanych przez firmę Apple dla systemu szkieletowego Core Animation, na bazie którego utworzono po-wszechnie dziś znane efekty interfejsu użytkownika, takie jak przejścia ekranowe w aplikacji iOS Weather widocznej na rysunku 6.1. Trójwymiarowe dodatki do CSS3 zostały zaproponowane przez zespół programistów ds. WebKit w latach 2009 i 2010 i po raz pierwszy ich obsługa pojawiła się w przeglądarce Safari dla systemów Mac OS oraz iOS. Później dodano je do przeglądarki Chrome i w końcu także do innych przeglądarek internetowych.

Dostępność trójwymiarowych efektów dla elementów HTML otwiera przed programistą wiele możliwości. Na rysunku 6.2 można zobaczyć Snowstack (<http://www.satine.org/research/webkit/snowleopard/snowstack.html>), pokazowy model opracowany przez zespół programistów przeglądarki Safari. Jest to przeglądarka zdjęć wyświetlająca zdjęcia z kanału Flickr przy użyciu wyłącznie kodu HTML, CSS oraz JavaScriptu. Użytkownik za pomocą klawiszy strzałek może



Rysunek 6.1. Przejścia ekranowe w aplikacji Weather dla systemu iOS



Rysunek 6.2. Snowstack, czyli przeglądarka zdjęć utworzona przy użyciu funkcji trójwymiarowych technologii CSS (<http://www.satine.org/research/webkit/snowleopard/snowstack.html>)

przeglądać pozornie nieskończoną galerię kafelków zdjęć. Aplikacja działa we wszystkich przeglądarkach i urządzeniach. Snowstack to wprawdzie tylko pokaz możliwości technologii, ale dobrze ukazuje potencjał, jaki drzemie w CSS, dotyczący wizualizacji i badania dużych ilości danych.

Wielu programistów posługuje się trójwymiarowymi efektami CSS w celu tworzenia innowacyjnych rodzajów treści internetowej. Niektórzy nie ograniczyli się tylko do przekształcania płaskich kafelków i opanowali symulowanie renderowania pełnych trójwymiarowych obiektów, a pewien przedsiębiorczy jegomość, o którym jeszcze napiszę, użył nawet tej technologii do budowy prototypów gier FPS! CSS można też używać w połączeniu z WebGL, przy czym CSS wykorzystuje się do integrowania elementów HTML interfejsu użytkownika, a WebGL do obsługi renderowania obiektów trójwymiarowych.

CSS3 to zbiór specyfikacji opisujących techniki stosowania dynamicznych efektów do elementów stron internetowych. W tym rozdziale znajduje się opis tych technik CSS, które służą do tworzenia efektów trójwymiarowych.

Przekształcenia CSS

Operacje trójwymiarowe (przesunięcie, obrót, skalowanie) zastosowane do całych elementów.

Przejścia CSS

Proste zmiany własności CSS w czasie. Podobnie jak klatki pośrednie (opisane w poprzednim rozdziale), przejścia CSS bardzo dobrze nadają się do tworzenia jednorazowych efektów.

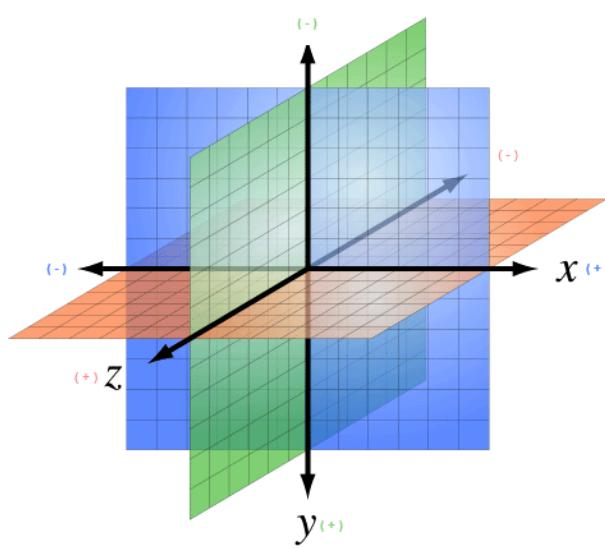
Animacje CSS

Skomplikowane zmiany własności CSS w czasie realizowane przy użyciu klatek kluczowych.

Przekształcenia CSS

Podstawą technik tworzenia trójwymiarowych efektów w CSS jest przetwarzanie elementów stron internetowych za pomocą przekształceń CSS. Ich specyfikacja, o nazwie CSS Transforms (<http://www.w3.org/TR/css3-transforms/>), stanowi połączenie wcześniejszych prac dotyczących zastosowania CSS do modyfikacji pozycji, orientacji, skali i innych właściwości elementów strony internetowej za pomocą przekształceń zamiast prostych określeń szerokości i wysokości oraz odstępu od góry i lewej krawędzi okna przeglądarki.

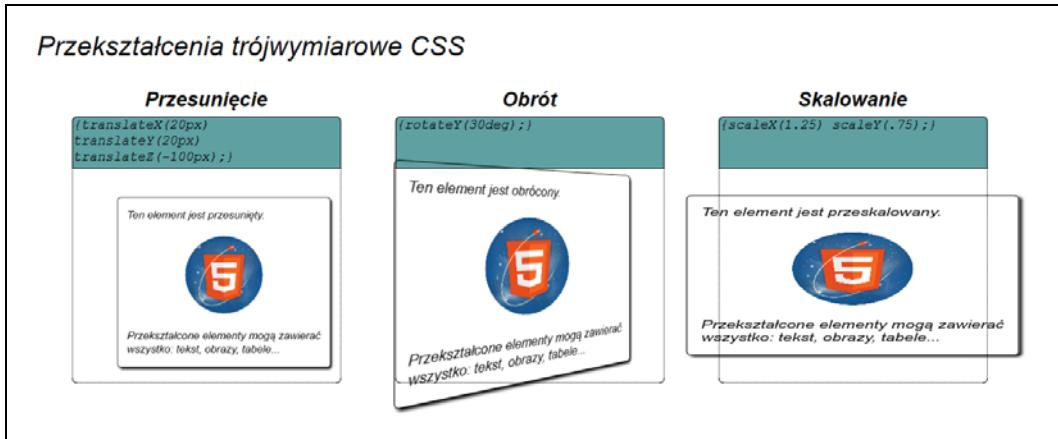
Przypominam, że w grafice trójwymiarowej używa się trójwymiarowego układu współrzędnych zawierającego oś z reprezentującą pozycję na zewnątrz i do wnętrza ekranu, dzięki którym uzyskuje się wrażenie głębi. Na rysunku 6.3 pokazuję trójwymiarowy układ współrzędnych wykorzystywany w CSS. Należy zwrócić uwagę, że, w odróżnieniu od innych systemów trójwymiarowych, w tym przypadku dodatnia część osi y biegnie w dół, a nie do góry. Zrobiono tak, aby zachować zgodność z dwuwymiarowym układem xy stosowanym w przeglądarkach internetowych i oknach programów.



Rysunek 6.3. Trójwymiarowy układ współrzędnych używany w CSS. Dodatnia część osi y jest skierowana w dół zamiast do góry (adaptacja rysunku <http://bit.ly/wikimedia-3d-coordinate>; licencja Creative Commons Attribution Share Alike 3.0 unported)

Przekształcenia trójwymiarowe w praktyce

Przekształcenia trójwymiarowe w CSS definiuje się w normalny sposób, tzn. przy użyciu własności. W specyfikacji CSS3 zdefiniowano kilka własności do przekształcania elementów. Omawianie ich rozpocznie od przykładu przedstawionego na rysunku 6.4. Widać na nim trzy elementy, z których każdy został przekształcony w inny sposób: przesunięty, obrócony oraz przeskalowany.



Rysunek 6.4. Przekształcenia trójwymiarowe CSS: przesunięcie, obrót oraz skalowanie

Kod źródłowy tego przykładu znajduje się w plikach *r6/css3dtransforms.html* (HTML) i *css/css3dtransforms.css* (CSS). Pokazany na listingu 6.1 fragment kodu HTML zawiera definicję pierwszego elementu `<div>`, do którego zastosowano przesunięcie.

Listing 6.1. Element, do którego zastosowano przekształcenie trójwymiarowe CSS

```
<div id="card1" class="container perspective">
  <div class="legend">
    Przesunięcie
  </div>
  <div class="code">{translateX(20px) translateY(20px) translateZ(-100px);}</div>
  <div class="cardBorder">
    <div class="card translate">
      <p>Ten element jest przesunięty.</p>
      </img>
      <p>Przekształcone elementy mogą zawierać wszystko: tekst, obrazy, tabele...</p>
    </div>
  </div>
</div>
```

Pogrubiony fragment to deklaracje dwóch klas dla położonego najgłębiej w hierarchii elementu `<div>`; są to klasy `card` i `translate`. Pierwsza definiuje własności typowe dla wszystkich elementów reprezentujących „karty” na stronie — np. ciągłe obramowanie, cień i zaokrąglone rogi. Klasa `translate` definiuje przesunięcie. Na listingu 6.2 podaję definicje CSS tych dwóch klas oraz kod klasy `cardBorder`, która ma zastosowanie do elementu nadrzędnego każdej karty i sprawia, że wyświetlane jest przerywane obramowanie, ukazujące pozycję karty przed jej przekształceniem. Na razie można zignorować własności zaczynające się od przedrostka `-moz-transform-style`. Są potrzebne przeglądarki Firefox, o czym będzie mowa w następnym podrozdziale, dotyczącym perspektywy.

Listing 6.2. Kod CSS zawierający definicje przesunięcia

```
.cardBorder {  
    position: absolute;  
    width: 100%;  
    height: 80%;  
    top:30%;  
    border:1px dotted;  
    border-radius:0 0 4px 4px;  
    -moz-transform-style: preserve-3d;  
}  
  
.card {  
    position: absolute;  
    width: 99%;  
    height: 99%;  
    border:1px solid;  
    border-radius: 4px;  
    box-shadow: 2px 2px 2px;  
    -moz-transform-style: preserve-3d;  
}  
  
.translate {  
    -webkit-transform: translateX(20px) translateY(20px) translateZ(-100px);  
    -moz-transform: translateX(20px) translateY(20px) translateZ(-100px);  
    -o-transform: translateX(20px) translateY(20px) translateZ(-100px);  
    transform: translateX(20px) translateY(20px) translateZ(-100px);  
}
```

Klasa `translate` zawiera definicję trójwymiarowego przekształcenia CSS w postaci definicji właściwości `transform`. W tym przykładzie element zostaje przesunięty o 20 pikseli na osiach x i y oraz o 100 pikseli na osi z (w głęb ekranu). Korzystając z właściwości `transform`, można stosować kilka rodzajów przekształceń elementów. Oprócz przesunięcia, dostępne są też obrót i skalowanie, przekształcenia przy użyciu macierzy oraz rzutowanie perspektywy. W tabeli 6.1 znajduje się zestawienie wszystkich metod trójwymiarowych przekształceń CSS.

Tabela 6.1. Metody przekształceń trójwymiarowych CSS

Metoda	Opis
<code>translateX(x)</code>	Przesunięcie wzdłuż osi x
<code>translateY(y)</code>	Przesunięcie wzdłuż osi y
<code>translateZ(z)</code>	Przesunięcie wzdłuż osi z
<code>translate3d(x, y, z)</code>	Przesunięcie wzdłuż osi x , y i z
<code>rotateX(angle)</code>	Obrót wokół osi x
<code>rotateY(angle)</code>	Obrót wokół osi y
<code>rotateZ(angle)</code>	Obrót wokół osi z
<code>rotate3d(x, y, z, angle)</code>	Obrót wokół dowolnej osi
<code>scaleX(x)</code>	Skalowanie wzdłuż osi x
<code>scaleY(y)</code>	Skalowanie wzdłuż osi y
<code>scaleZ(z)</code>	Skalowanie wzdłuż osi z
<code>scale3d(x, y, z)</code>	Skalowanie wzdłuż osi x , y i z
<code>matrix3d(...)</code>	Definicja macierzy przekształceń o wymiarach 4×4 , a więc z 16 wartościami
<code>perspective(depth)</code>	Definicja rzutowania perspektywicznego <code>depth</code> pikseli

Karty druga i trzecia zostały przekształcone w podobny sposób, tylko przy użyciu klas CSS `rotate` i `scale`.

```
.rotate {  
    -webkit-transform: rotateY(30deg);  
    -moz-transform: rotateY(30deg);  
    -o-transform: rotateY(30deg);  
    transform: rotateY(30deg);  
}  
  
.scale {  
    -webkit-transform: scaleX(1.25) scaleY(.75);  
    -moz-transform: scaleX(1.25) scaleY(.75);  
    -o-transform: scaleX(1.25) scaleY(.75);  
    transform: scaleX(1.25) scaleY(.75);  
}
```

Wartości obrotu można podawać w stopniach, radianach lub **gradach** (grad to jednostka równa $\frac{1}{400}$ kąta pełnego), np. `90deg`, `1.57rad` lub `100grad`. Wartość skalowania to skalar oznaczający mnożnik wartości dla każdej z osi (tzn. nieprzeskalowany element ma skalę 1 wzdłuż każdej osi).



Warto zwrócić uwagę na zastosowanie w kodzie CSS przedrostków specyficznych dla różnych przeglądarek (np. `-webkit-transform`). Ich użycie pozwala na zapewnienie działania przekształceń w starszych przeglądarkach, w których były one przez kilka lat tylko funkcją eksperymentalną. Dodawanie przedrostków jest uciążliwe, ale wymaga ich wiele nowych właściwości CSS, dzięki czemu projektanci stron zdążyli już do nich się przyzwyczaić. Jeżeli znuży Cię ciągłe przepisywanie prawie takiego samego kodu, możesz użyć jednego z narzędzi do generowania arkuszy stylów, np. LESS (<http://lesscss.org/>). Od czasu do czasu będę opuszczał przedrostki w przykładach dla uproszczenia kodu. Jednak trzeba pamiętać, by dodać je w ostatecznej wersji kodu.

W CSS dostępna jest jeszcze jedna właściwość, o nazwie `transform-origin`, która umożliwia ustawienie początku przekształceń. Jej domyślna wartość to `50% 50%`, czyli środek układu współrzędnych. Zmieniając ją, można sprawić, że obiekt obróci się wokół innego punktu niż jego środek. Wartością właściwości `transform-origin` może być każde słowo kluczowe CSS oznaczające przesunięcie, a więc `left`, `center`, `right` oraz wartość wyrażona w jednostce długości (piksel, cal, em itd.) lub `%`.

Perspektywa

Zapewne zauważyłeś w poprzednim przykładzie, że każdemu z elementów `<div>` najwyższego poziomu przypisano klasę `perspective`. Przekształcenia trójwymiarowe CSS można stosować z rzutowaniem perspektywicznym lub bez niego, chociaż lepiej go stosować.

Rzuty perspektywiczne w CSS3 definiuje się bardzo łatwo. Na listingu 6.3 pokazano kod CSS definiujący perspektywę.

Listing 6.3. Właściwość perspective CSS

```
.perspective {  
    -webkit-perspective: 400px;  
    -moz-perspective: 400px;  
    -o-perspective: 400px;  
    perspective: 400px;  
}
```

```
.noperspective {  
    -webkit-perspective: 0px;  
    -moz-perspective: 0px;  
    -o-perspective: 0px;  
    perspective: 0px;  
}
```

Zdefiniowana została tu klasa CSS o nazwie `perspective`. Przypiszemy ją tym elementom, do których będziemy chcieli zastosować rzutowanie perspektywiczne. Podana wartość reprezentuje odległość od płaszczyzny widzenia do płaszczyzny *xy* ($z=0$). Do definiowania perspektywy można używać dowolnej jednostki długości CSS, np. pikseli, punktów, cali, em itd. Dodatkowo w tym samym pliku znajduje się deklaracja klasy `noperspective`, która służy do renderowania elementów bez perspektywy. Wartości w niej są ustawione na zero, bo taka jest wartość domyślna własności `perspective`.



W CSS perspektywy używa się nieco inaczej niż w WebGL, ale ogólnie same pojęcia są takie same. Jeśli chcesz odświeżyć sobie pamięć na ten temat, przeczytaj jeszcze raz odpowiednią część rozdziału 1.

Aby zobaczyć, czym różni się obraz wyrenderowany z perspektywą od obrazu wyrenderowanego bez niej, przeanalizujemy przykład. Otwórz plik `r6/css3dperspective.html`. W oknie przeglądarki zostaną wyświetlane dwie karty. Karta po lewej jest wyrenderowana z perspektywą, a po prawej — bez niej. Jedyna różnica między tymi elementami polega na użyciu własności CSS `perspective`. Każda z kart jest obrócona o 30 stopni wokół osi *y*, ale bez perspektywy obraz (po prawej) wydaje się raczej zgnieciony poziomo, a nie obrócony (rysunek 6.5).

Przekształcenia trójwymiarowe CSS - perspektywa

Z perspektywą

(`perspective: 400px;`)

Ten element został wyrenderowany z perspektywą.

Bez perspektywy

(`perspective: 0px;`)

Ten element został wyrenderowany bez perspektywy.

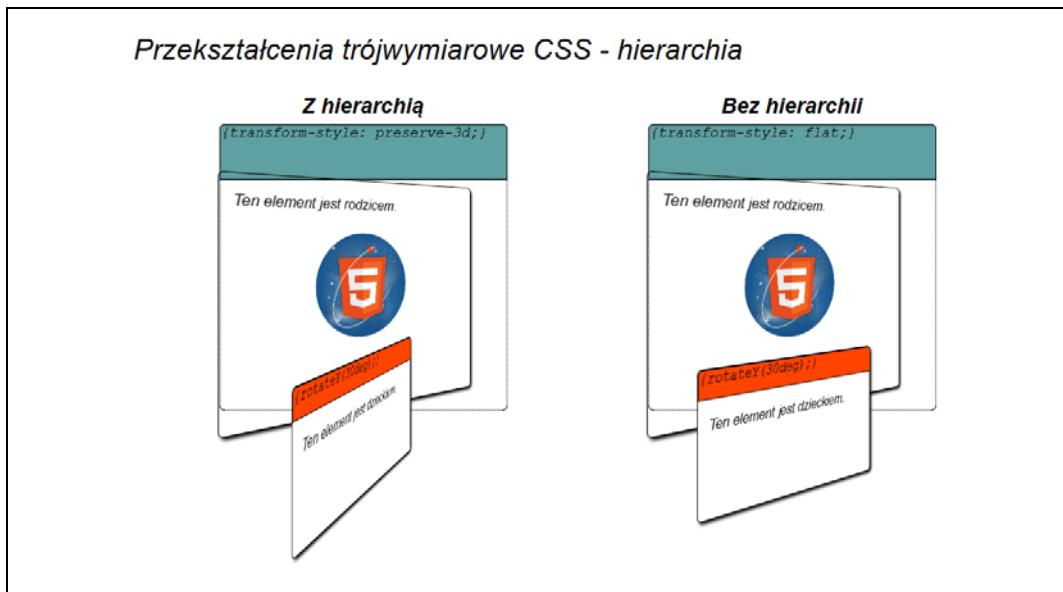
Rysunek 6.5. Przekształcenia CSS i perspektywa: element po lewej został wyrenderowany przy użyciu perspektywy, a element po prawej bez niej (logo HTML Rawkes autorstwa Phila Banksa — <http://twitter.com/emirpprime>)

Perspektywę można też stosować przy użyciu funkcji przekształceniowej `perspective()`, której opis znajduje się w tabeli 6.1. Jednak w praktyce zazwyczaj lepiej oddzielić wartość perspektywy od wartości przekształcenia, używając dwóch różnych własności. W przeciwnym razie wartość perspektywy trzeba będzie powtarzać za każdym razem, gdy zechce się zmienić inne wartości przekształceń.

Tworzenie hierarchii przekształceń

W CSS3 przekształcenia trójwymiarowe mogą być dziedziczone przez hierarchię obiektów DOM. Element, który ma zdefiniowane przekształcenia trójwymiarowe, może dziedziczyć przekształcenia po elementach nadrzędnych lub je ignorować, w zależności od ustawienia właściwości `transform-style`.

Na rysunku 6.6 pokazano, jak można wykorzystać właściwość `transform-style` do utworzenia hierarchii przekształceń. Każda z kart jest obrócona o 30 stopni wokół osi *y*. Ponadto każda zawiera kartę potomną, `childCard`, która również jest obrócona o 30 stopni wokół osi *y*. Zwróć uwagę, że karta znajdująca się w karcie po lewej jest obrócona o 30 stopni w stosunku do płaszczyzny swojego rodzica. Natomiast po prawej obie karty znajdują się w tej samej płaszczyźnie.



Rysunek 6.6. Hierarchia przekształceń trójwymiarowych w CSS

Kod źródłowy tego przykładu zamieszczony został w plikach `r6/css3dhierarchy.html` i `css/css3dhierarchy.css`. W kodzie HTML znajdują się definicje dwóch prawie identycznych hierarchii DOM elementów. Jedyna różnica jest taka, że pierwsza karta ma klasę `hierarchy`, a druga `nohierarchy`.

```
<div id="hierarchy1" class="container perspective">
  <div class="legend">
    Z hierarchią
  </div>
  <div class="code">{transform-style: preserve-3d;}</div>
  <div class="cardBorder">
    <div class="card hierarchy rotate">
      <p>Ten element jest rodzicem.</p>
      </img>
      <p></p>
      <div class="childCard rotate">
        <div class="code">{rotateY(30deg);}</div>
        <p>Ten element jest dzieckiem.</p>
      </div>
    </div>
  </div>
</div>
```

```

        </div>
    </div>
</div>

<div id="hierarchy2" class="container perspective">
    <div class="legend">
        Bez hierarchii
    </div>
    <div class="code">{transform-style: flat;}</div>
    <div class="cardBorder">
        <div class="card nohierarchy rotate">
            <p>Ten element jest rodzicem.</p>
            </img>
            <p></p>
            <div class="childCard rotate">
                <div class="code">{rotateY(30deg);}</div>
                <p>Ten element jest dzieckiem.</p>
            </div>
        </div>
    </div>
</div>
</div>

```

Oto definicje CSS klas hierarchy i nohierarchy.

```

.hierarchy {
    -webkit-transform-style: preserve-3d;
    -moz-transform-style: preserve-3d;
    -o-transform-style: preserve-3d;
    transform-style: preserve-3d;
}

.nohierarchy {
    -webkit-transform-style: flat;
    -moz-transform-style: flat;
    -o-transform-style: flat;
    transform-style: flat;
}

```

Własność `transform-style` przyjmuje dwie wartości; pierwsza `flat` (domyślna) oznacza, że przekształcenia mają nie być stosowane w elementach potomnych, druga `preserve-3d` nakazuje zastosować przekształcenia w elementach potomnych. Stosując wartość `preserve-3d`, można utworzyć głęboką hierarchię obiektów trójwymiarowych zwłaszcza wtedy, kiedy dodatkowo użyje się innych technik opisanych w tym rozdziale.



Ostrzeżenie dotyczące przeglądarki: w pierwszym przykładzie w tym podrozdziale pominąłem jeden szczegół dotyczący definicji klas `card` i `cardBorder`. Zawierają one następującą deklarację:

```
-moz-transform-style: preserve-3d;
```

Przeglądarka Firefox, w odróżnieniu od przeglądarek opartych na WebKit, nie przekazuje wartości właściwości `transform-style` elementom potomnym. Jeśli nie zdefiniuje się jej dla każdego elementu osobno, nie zadziałają przekształcenia ani nie zostanie zastosowana perspektywa. Rozwiązaniem jest ustawienie właściwości `transform-style` na `preserve-3d` każdemu elementowi potomnemu w drzewie DOM. Jest to nieeleganckie, ale konieczne.

Najgorsze jest jednak to, że przeglądarki różnie interpretują opisywaną właściwość — np. Internet Explorer w wersji 11 w ogóle jej nie obsługuje.

Kontrolowanie renderowania tylnej ściany obiektów

W klasycznym renderowaniu trójwymiarowym, gdy wielokąt jest zwrócony tyłem do użytkownika, system może wyświetlić jego **tylną ścianę** (ang. *backface*) lub nie wyświetlać jej w ogóle, w zależności od ustawień zdefiniowanych przez programistę. Podobne możliwości dostępne są w CSS3. Jeśli element zostanie odwrócony tyłem do użytkownika, o tym, czy zostanie pokazany, decyduje ustawienie własności `backface-visibility`.

W CSS3 renderowanie na odwrocie jest potrzebne do tworzenia obiektów dwustronnych. Powiedzmy, że chcemy utworzyć obrotowe przejście zmiany ekranu, takie jak w aplikacji Weather dla systemu iOS widocznej na rysunku 6.1. Aby uzyskać taki efekt, należy dobrze skonstruować kod HTML oraz poprawnie użyć własności `backface-visibility`. Na rysunku 6.7 pokazano, jak to wygląda w praktyce.

Przekształcenia trójwymiarowe CSS - widoczność tylnej ściany



Rysunek 6.7. Modyfikowanie widoczności tylnej ściany w celu utworzenia obiektów dwustronnych

Otwórz plik `r6/css3dbbackfaces.html` w przeglądarce. Na ekranie zobaczysz cztery karty. Na górze znajdują się dwie jednostronne karty, wygenerowane odpowiednio przy włączonej i wyłączonej własności `backface-visibility`. Karta po lewej jest obrócona tyłem do użytkownika i widać jej tylną ścianę. Karta po prawej też jest odwrócona tyłem, ale ma wyłączone renderowanie tylnej ściany. Kartę po lewej widać, tylko tekst „PRZÓD” jest napisany odwrotnie, natomiast karty po prawej nie widać.

Na dole znajdują się dwie karty dwustronne, odpowiednio z włączoną i wyłączoną tylną ścianą. Obiekty te również są obrócone tyłem do użytkownika. Ale karty te mają zdefiniowany dodatkowy element, zawierający tekst „TYŁ”, który jest obrócony w kierunku użytkownika, tak aby imitować drugą stronę karty. Karta po lewej ma włączoną widoczność tylnej ściany, a dzięki ustawieniu przezroczystości na 0.8 przez przednią ścianę widać odwrócony tekst „PRZÓD”. Natomiast karta po prawej ma wyłączoną widoczność tylnej ściany, a więc nie widać na niej przedniej ściany. W ten właśnie sposób tworzy się w CSS obiekty dwustronne. Przeanalizujmy teraz kod źródłowy.

Na listingu 6.4 pokazano kod HTML opisywanej strony. Elementy, których tylna ściana jest widoczna, są przypisane do klasy `backface`; elementy, których tylna ściana jest niewidoczna, są przypisane do klasy `nobackface`. Znajdujące się na dole karty dwustronne są w istocie utworzone z dwóch elementów: po jednym dla przedniej i tylnej ściany, które są odpowiednio przypisane do klas CSS `frontside` i `backside`. Karta znajdująca się na dole po prawej, oprócz tych dwóch, zawiera jeszcze klasę `nobackface`, dzięki czemu jest wyświetlana poprawnie, niezależnie od tego, która jej ściana jest widoczna.

Listing 6.4. Tworzenie dwustronnego elementu HTML

```
<div id="backface1" class="container perspective ">
    <div class="legend">
        Jednostronny, tylna ściana widoczna
    </div>
    <div class="code">{backface-visibility: visible;}</div>
    <div class="cardBorder">
        <div class="card backface frontside">
            PRZÓD
        </div>
    </div>
</div>

<div id="backface2" class="container perspective ">
    <div class="legend">
        Jednostronny, tylna ściana ukryta
    </div>
    <div class="code">{backface-visibility: hidden;}</div>
    <div class="cardBorder">
        <div class="card nobackface frontside">
            PRZÓD
        </div>
    </div>
</div>

<div id="backface3" class="container perspective ">
    <div class="legend">
        Dwustronny, tylna ściana widoczna
    </div>
    <div class="code">{backface-visibility: visible;}</div>
    <div class="cardBorder">
        <div class="card backface frontside">
            PRZÓD
        </div>
        <div class="card backface backside">
            TYŁ
        </div>
    </div>
</div>
```

```

<div id="backface4" class="container perspective ">
  <div class="legend">
    Dwustronny, tylna ściana ukryta
  </div>
  <div class="code">{backface-visibility: hidden;}</div>
  <div class="cardBorder">
    <div class="card nobackface frontside">
      PRZÓD
    </div>
    <div class="card nobackface backside">
      TYŁ
    </div>
  </div>
</div>

```

Na listingu 6.5 znajdują się deklaracje reguł stylistycznych z pliku *css/css3dbackfaces.css*. Najpierw zdefiniowaliśmy klasy *frontside* i *backside*. Pierwsza zawiera deklaracje dotyczące przedniej strony karty, ale ponieważ w tym przykładzie interesuje nas renderowanie tylnej ściany, obracamy ją o 210 stopni wokół osi *y*. Natomiast tylną ścianę karty obracamy do użytkownika o 30 stopni. Karty znajdują się w jednej płaszczyźnie, ponieważ różnica obrotu między nimi wynosi 180 stopni. Gdy ukryjemy tylną ścianę za pomocą klasy *nobackface*, otrzymamy doskonałą dwustronną kartę, taką jak na dole po prawej na rysunku. W klasie *nobackface* ustawiliśmy własność *backface-visibility* na *hidden*, aby otrzymać żądzany efekt.

Listing 6.5. Deklaracje CSS potrzebne do utworzenia dwustronnych obiektów

```

.frontside {
  -webkit-transform: rotateY(210deg);
  -moz-transform: rotateY(210deg);
  -o-transform: rotateY(210deg);
  transform: rotateY(210deg);
  line-height:160px;
  font-size:40px;
  color:White;
  background-color:DarkCyan;
  border-color:Black;
  box-shadow:2px 2px 2px Black;
}

.backside {
  -webkit-transform: rotateY(30deg);
  -moz-transform: rotateY(30deg);
  -o-transform: rotateY(30deg);
  transform: rotateY(30deg);
  line-height:160px;
  font-size:40px;
  color:White;
  background-color:DarkRed;
  border-color:Black;
  box-shadow:2px 2px 2px Black;
  opacity:0.8;
}

.backface {
  -webkit-backface-visibility: visible;
  -moz-backface-visibility: visible;
  -o-backface-visibility: visible;
  backface-visibility: visible;
}

```

```

.nobackface {
    -webkit-backface-visibility: hidden;
    -moz-backface-visibility: hidden;
    -o-backface-visibility: hidden;
    backface-visibility: hidden;
}

```

Zestawienie właściwości przekształceniowych CSS

W tym punkcie opisałem właściwości CSS służące do tworzenia efektów trójwymiarowych na elementach HTML. W tabeli 6.2 znajduje się zestawienie tych właściwości.

Tabela 6.2. Właściwości przekształceń CSS

Właściwość	Opis
transform	Słosuje przekształcenie przy użyciu przynajmniej jednej z metod przekształceniowych (tabela 6.1).
transform-origin	Definiuje początek wszystkich przekształceń (domyślne ustawienie: 50%, 50%, 0).
perspective	Określa głębię perspektywy w jednostkach odległości CSS (domyślne ustawienie: 0 oznaczające brak perspektywy).
perspective-origin	Określa punkt zbiegu perspektywy we współrzędnych xy.
transform-style	Określa, czy elementy potomne elementu trójwymiarowego są renderowane płasko, czy trójwymiarowo.
backface-visibility	Określa, czy elementy odwrócone tyłem do ekranu mają być renderowane.

Przekształcenia CSS to technika bardzo przydatna do tworzenia trójwymiarowych efektów na stronach internetowych. A jeśli połączy się ją z przejściami i animacjami, można budować fantastyczne dynamiczne efekty.



Przykłady przedstawione w tym podrozdziale były inspirowane wpisami na świetnym blogu Davida DeSandra *24 Ways to impress your friends*. DeSandro pozwolił mi na zaadaptowanie jego prac do własnych potrzeb. Na stronie <http://24ways.org/2010/intro-to-css-3d-transforms/> i wielu innych jego stronach można znaleźć mnóstwo cennych informacji o tworzeniu trójwymiarowych efektów przy użyciu CSS.

Przejścia CSS

Przejścia CSS (ang. *CSS transitions*) umożliwiają stopniowe zmienianie właściwości w czasie i ich zastosowanie jest pod wieloma względami podobne do użycia opisanej w poprzednim rozdziale biblioteki Tween.js. Jednak efekty przejść są wbudowane w przeglądarki internetowe, dzięki czemu nie trzeba używać JavaScriptu. Należy jednak podkreślić, że za pomocą przejść można animować większość właściwości CSS, ale nie wszystkie, np. można animować szerokość, położenie, kolor, położenie na osi z, przezroczystość itd.

Podstawowa składnia przejścia CSS jest następująca:

transition: nazwa-właściwości czas-trwania funkcja-czasowa opóźnienie;

nazwa-właściwości

Może być nazwą konkretnej właściwości, słowem kluczowym all oznaczającym, że przejście dotyczy wszystkich zmienianych właściwości albo słowem kluczowym none oznaczającym, że przejście nie dotyczy żadnej właściwości.

czas-trwania

Liczba sekund lub milisekund trwania przejścia.

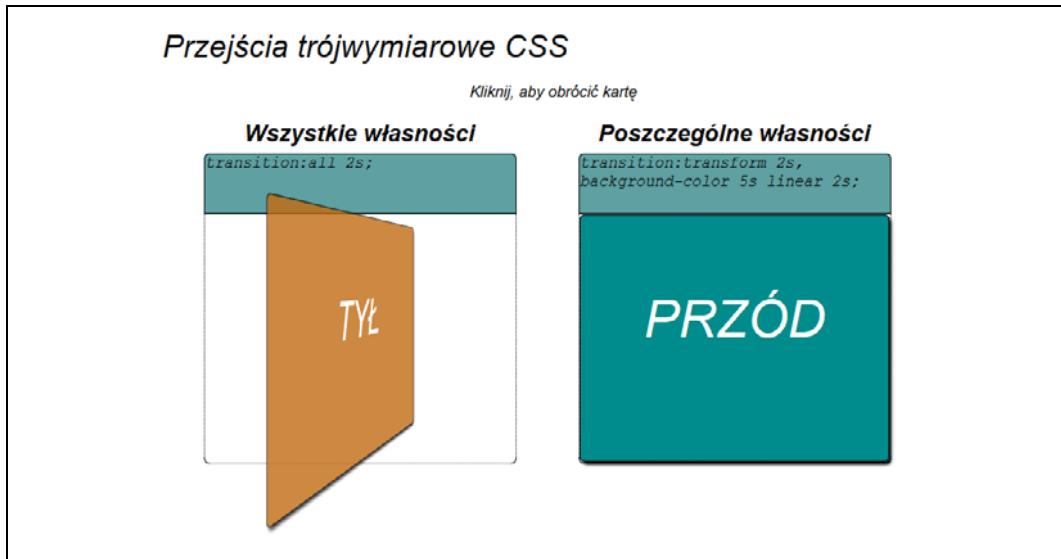
funkcja-czasowa

Nazwa funkcji czasowej, która ma być użyta do animacji przejścia. Dostępne są następujące funkcje: linear, ease, ease-in, ease-out, ease-in-out oraz cubic-bezier.

opóźnienie

Liczba sekund lub milisekund, jaką należy odczekać przed rozpoczęciem przejścia.

Własność transition to w istocie skrót pozwalający definiować w jednym miejscu cztery właściwości CSS: transition-property, transition-duration, transition-timing-function oraz transition-delay. Na konkretnym przykładzie zobaczymy, jak to działa. Otwórz plik *r6/css3dtransitions.html*. Zobaczysz stronę internetową widoczną na rysunku 6.8. Kliknięcie jednej z kart spowoduje obrócenie jej na drugą stronę, która została utworzona za pomocą techniki opisanej w poprzednim podrozdziale. Przejście obrotu trwa dwie sekundy i początkowo przyspiesza, potem zwalnia, a następnie znowu przyspiesza. Ponadto karty zmieniają kolor z początkowego morskiego na złoty, ale karta po lewej zmienia kolor w czasie obrotu, a karta po prawej zmienia go dopiero po zakończeniu obrotu.



Rysunek 6.8. Zastosowanie przejść CSS do animowania właściwości

Definicje HTML przodu i tyłu kart są podobne. Najważniejsza różnica między nimi polega na użyciu klasy `easeAll2sec` dla karty po lewej i `easeTransform2secColor5secDelay` dla karty po prawej. Za chwilę przyjrzymy się tym klasom.

```
<div id="transition1" class="container perspective">
  <div class="legend">
    Wszystkie właściwości
  </div>
  <div class="code">transition:all 2s;</div>
  <div class="cardBorder">
    <div id="front1"
        class="card nobackface frontside clickable easeAll2sec">
      PRZÓD
    </div>
  </div>
</div>
```

```

</div>
<div id="back1"
      class="card nobackface backside clickable easeAll2sec">
    TYŁ
</div>
</div>
</div>

<div id="transition2" class="container perspective ">
  <div class="legend">
    Poszczególne właściwości
  </div>
  <div class="code">transition:transform 2s,
    background-color 5s linear 2s;</div>
  <div class="cardBorder">
    <div id="front2"
        class="card nobackface frontside clickable easeTransform2secColor5secDelay">
      PRZÓD
    </div>
    <div id="back2"
        class="card nobackface backside clickable easeTransform2secColor5secDelay">
      TYŁ
    </div>
  </div>
</div>

```

Efekt jest wywoływany przez kliknięcie myszą dzięki zastosowaniu procedur obsługi kliknięć z biblioteki jQuery. Procedura przechowuje dla każdej karty wartość logiczną oznaczającą, która jej strona jest aktualnie wyświetcona, oraz dodaje lub usuwa klasy `flip` i `goGold` zgodnie z potrzebą. Klasa `flip` obraca kartę o 180 stopni. Natomiast klasa `goGold` ustawia kolor na złoty. Bez przejęcia CSS zmiany te następowaliby natychmiast, zamiast dokonywać się płynnie w określonym czasie.

```

<script type="text/javascript">

  var front1 = true;
  var front2 = true;
  $(document).ready(
    function() {
      $('#transition1 .clickable').click(function(){
        //alert("Kliknięto");
        if (front1)
        {
          $('#front1').addClass('flip');
          $('#back1').addClass('flip');
          $('#front1').addClass('goGold');
          $('#back1').addClass('goGold');
        }
        else
        {
          $('#front1').removeClass('flip');
          $('#back1').removeClass('flip');
          $('#front1').removeClass('goGold');
          $('#back1').removeClass('goGold');
        }

        front1 = !front1;
      });

      $('#transition2 .clickable').click(function(){
        if (front2)
        {

```

```

        $('#front2').addClass('flip');
        $('#back2').addClass('flip');
        $('#front2').addClass('goGold');
        $('#back2').addClass('goGold');
    }
} else {
    $('#front2').removeClass('flip');
    $('#back2').removeClass('flip');
    $('#front2').removeClass('goGold');
    $('#back2').removeClass('goGold');
}

front2 = !front2;
});
);
};

</script>

```

Kod CSS do tego przykładu znajduje się w pliku *css/css3dtransitions.css* oraz częściowo na liście 6.6.

Listing 6.6. Definicje przejść CSS

```

.frontside {
    -webkit-transform: rotateY(0deg);
    -moz-transform: rotateY(0deg);
    -o-transform: rotateY(0deg);
    transform: rotateY(0deg);
...
}

.backside {
    -webkit-transform: rotateY(180deg);
    -moz-transform: rotateY(180deg);
    -o-transform: rotateY(180deg);
    transform: rotateY(180deg);
...
}

.frontside.flip {
    -webkit-transform: rotateY(-180deg);
    -moz-transform: rotateY(-180deg);
    -o-transform: rotateY(-180deg);
    transform: rotateY(-180deg);
}

.backside.flip {
    -webkit-transform: rotateY(0deg);
    -moz-transform: rotateY(0deg);
    -o-transform: rotateY(0deg);
    transform: rotateY(0deg);
}

.goGold {
    background-color:Goldenrod;
}

.easeAll2sec {
    -webkit-transition:all 2s;
    -moz-transition:all 2s;
    -o-transition:all 2s;
}

```

```
        transition:all 2s;
    }

.easeTransform2secColor5secDelay {
    -webkit-transition:-webkit-transform 2s, background-color 5s linear 2s;
    -moz-transition:-moz-transform 2s, background-color 5s linear 2s;
    -o-transition:-o-transform 2s, background-color 5s linear 2s;
    transition:transform 2s, background-color 5s linear 2s;
}
```

Przód i tył karty mają zdefiniowane odpowiednie obroty w klasach frontside i backside. Gdy dołączy się klasę flip, karty obracają się o 180 stopni. Klasa goGold służy do zmieniania koloru tła elementu na złoty. Klasy oznaczone pogrubieniem definiują dwa różne przejścia. Przejście easeAll2sec jest proste: zmienia wszystkie określone właściwości w ciągu dwóch sekund, stosując domyślną funkcję ease.

Klasa easeTransform2secColor5secDelay jest nieco bardziej skomplikowana. Zawiera dwa osobne przejścia, jedno dotyczące właściwości transform, a drugie koloru tła (background-color), rozdzielone przecinkiem. Przejście transform jest identyczne z przejściem w klasie easeAll2sec, tzn. jest to dwusekundowe przejście zdefiniowane przy użyciu funkcji ease. Natomiast przejście koloru tła jest inne: trwa pięć sekund, zaczyna się po dwóch sekundach oraz odbywa się liniowo (linear).



W podrozdziale tym zaledwie dotknęliśmy wiedzy na temat przejść w CSS. Jeśli chcesz dowiedzieć się więcej, możesz przeczytać znakomity artykuł specjalisty CSS Microsoftu, Kirupy Chinnathambiego (http://www.kirupa.com/html5/all_about_css_transitions.htm).

Przejścia są łatwe w użyciu, ale ich zastosowanie jest ograniczone do prostych jednorazowych efektów. Jeśli chcesz utworzyć bardziej skomplikowane sekwencje i pętle, musisz użyć innej technologii CSS3, czyli animacji.

Animacje CSS

Animacje CSS to technika animacyjna bardziej ogólna od przejść. Podobnie jak w opisanych w poprzednim rozdziale animacjach klatkowych, w animacjach CSS używa się klatek kluczowych oraz parametrów określających czas trwania, funkcję prędkości, czas opóźnienia oraz zapętlenie animacji. Obejrzyjmy kilka przykładów.

Otwórz plik *r6/css3danimations.html*. Na stronie tej znajdują się trzy karty. Kliknięcie każdej z nich powoduje uruchomienie innej animacji (rysunek 6.9). Karta po lewej wykonuje prosty jednorazowy obrót wokół osi *y*. Karta po prawej kołysze się na lewo i prawo w nieskończoność. Natomiast karta na dole leci do góry i w prawo, obracając się wokół osi *y*, a potem wraca w taki sam sposób na pierwotne miejsce.

Kod CSS tworzący animację składa się z dwóch części: reguły @keyframes, w której umieszcza się dane dotyczące klatek kluczowych, oraz kilku właściwości, jakie można zdefiniować dla elementu. Oto one.

Trójwymiarowe animacje CSS

Kliknij kartę, aby włączyć animację

Obrót (raz)

```
@keyframes kfRotateY {  
    from {  
        transform: rotateY(0deg);  
    }  
    to {  
        transform: rotateY(360deg);  
    }  
}  
  
.animRotateY {  
    animation-duration: 2s;  
    animation-name: kfRotateY;  
    animation-iteration-count: infinite;  
    animation-timing-function: linear;  
}
```

Lot (przemienne)

```
@keyframes kfSweep {  
    0% {  
        transform: translate3d(0, 0, 0) rotateZ(0deg);  
    }  
    25% {  
        transform: translate3d(50px, -100px, 0);  
    }  
    50% {  
        transform: translate3d(100px, -200px, 0);  
    }  
    75% {  
        transform: translate3d(150px, -300px, 0);  
    }  
    100% {  
        transform: translate3d(200px, -400px, 0);  
    }  
}  
  
.animFly {  
    animation-duration: 1s;  
    animation-name: kfSweep;  
    animation-iteration-count: infinite;  
    animation-timing-function: cubic-bezier(0, 1,  
    0.5, 1);  
}
```

Patrzdasanie (ciągłe)

```
@keyframes kfShake {  
    0% {  
        transform: translate3d(0, 0, 0) rotateZ(0deg);  
    }  
    10% {  
        transform: translate3d(0, -20px, 0) rotateZ(-10deg);  
    }  
    20% {  
        transform: translate3d(0, 20px, 0) rotateZ(10deg);  
    }  
    30% {  
        transform: translate3d(0, -40px, 0) rotateZ(-20deg);  
    }  
    40% {  
        transform: translate3d(0, 40px, 0) rotateZ(20deg);  
    }  
    50% {  
        transform: translate3d(0, -60px, 0) rotateZ(-30deg);  
    }  
    60% {  
        transform: translate3d(0, 60px, 0) rotateZ(30deg);  
    }  
    70% {  
        transform: translate3d(0, -80px, 0) rotateZ(-40deg);  
    }  
    80% {  
        transform: translate3d(0, 80px, 0) rotateZ(40deg);  
    }  
    90% {  
        transform: translate3d(0, -100px, 0) rotateZ(-50deg);  
    }  
    100% {  
        transform: translate3d(0, 100px, 0) rotateZ(50deg);  
    }  
}
```

Rysunek 6.9. Trójwymiarowe animacje CSS

animation-name

Nazwa zbioru klatek kluczowych zadeklarowanych w regule @keyframes, który będzie służył jako źródło danych dotyczących klatek kluczowych.

animation-duration

Określa długość trwania animacji w sekundach lub milisekundach.

animation-timing-function

Nazwa funkcji czasowej do animowania klatek kluczowych. Dostępne są następujące funkcje: linear, ease, ease-in, ease-out, ease-in-out oraz cubic-bezier.

animation-delay

Określa, po ilu sekundach lub milisekundach ma się rozpocząć animacja.

animation-iteration-count

Określa liczbę powtórzeń animacji. Domyślana wartość to 1. Aby animacja była powtarzana w nieskończoność, należy wpisać słowo kluczowe infinite.

animation-direction

Określa, czy animacja ma być wykonywana do przodu, czy do tyłu albo na zmianę, jeśli ustawiiono wiele powtórzeń. Możliwe wartości to normal (do przodu), reverse (do tyłu) alternate (najpierw do przodu, potem do tyłu) oraz alternate-reverse (najpierw do tyłu, a potem do przodu).

Wszystkie wymienione właściwości można zdefiniować w jednej właściwości zbiorczej o nazwie **animation**:

```
animation: animation-name animation-duration animation-timing-function animation-delay animation-iteration-count animation-direction;
```

Kod CSS do przykładu przedstawionego na rysunku 6.9 znajduje się w pliku *css/css3danimations.css*. Na listingu 6.7 pokazano jego najważniejsze fragmenty. Znajdują się w nim reguły @keyframes zawierające definicje klatek kluczowych kfRotateY i kfRotateMinusY (obracające odpowiednio przód i tył karty), kfShake do animacji z kołysaniem oraz kfFly do animacji z lataniem. Następnie zdefiniowane zostały indywidualne klasy z różnymi parametrami dla każdej animacji. Klasy animRotateY i animRotateMinusY definiują nieskończone liniowe animacje obracające element wokół osi y. Są to proste animacje w postaci przejścia od klatki początkowej do końcowej.

Listing 6.7. Deklaracje CSS tworzące animacje klatkowe

```
@-webkit-keyframes kfRotateY {  
    from {  
        -webkit-transform: rotateY(0deg);  
    }  
  
    to {  
        -webkit-transform: rotateY(360deg);  
    }  
}  
  
.animRotateY {  
    -webkit-animation-duration: 2s;  
    -webkit-animation-name: kfRotateY;  
    -webkit-animation-iteration-count: infinite;  
    -webkit-animation-timing-function: linear;  
}  
  
@-webkit-keyframes kfRotateMinusY {  
    from {  
        -webkit-transform: rotateY(-180deg);  
    }  
  
    to {  
        -webkit-transform: rotateY(180deg);  
    }  
}  
  
.animRotateMinusY {  
    -webkit-animation-duration: 2s;  
    -webkit-animation-name: kfRotateMinusY;  
    -webkit-animation-iteration-count: infinite;  
    -webkit-animation-timing-function: linear;  
}  
  
@-webkit-keyframes kfShake {  
    0% {  
        -webkit-transform: translate3d(0, 0, 0) rotateZ(0deg);  
    }  
    25% {  
        -webkit-transform: translate3d(0, -20px, 0) rotateZ(20deg);  
    }  
    50% {  
        -webkit-transform: translate3d(0, 0, 0) rotateZ(-20deg);  
    }  
    100% {  
        -webkit-transform: translate3d(0, -20px, 0) rotateZ(-20deg);  
    }  
}
```

```

.animShake
{
    -webkit-animation-duration: .5s;
    -webkit-animation-name: kfShake;
    -webkit-animation-iteration-count: infinite;
    -webkit-animation-timing-function:ease-in-out;
}

@-webkit-keyframes kfFly {
    0% {
        -webkit-transform:translate3d(0, 0, 0);
    }
    25% {
        -webkit-transform: translate3d(100px, -100px, 20px0);
    }
    50% {
        -webkit-transform: translate3d(200px, -200px, 40px);
    }
    100% {
        -webkit-transform: translate3d(400px, -300px, 20px);
    }
}

.animFly
{
    -webkit-animation-duration: 2s;
    -webkit-animation-name: kfFly;
    -webkit-animation-iteration-count: 2;
    -webkit-animation-timing-function:cubic-bezier(0.1, 0.2, 0.8, 1);
    -webkit-animation-direction:alternate;
}

```

Klasa `kfShake` jest nieco bardziej skomplikowana, ponieważ użyto w niej czterech klatek kluczowych, 0%, 25%, 50% oraz 100%, zawierających definicje przesunięć na osiach x i y oraz obrotu wokół osi z . Na końcu znajduje się najbardziej skomplikowana klasa `animFly`. Wykorzystuje ona definicje szeregu przesunięć w klatkach kluczowych, niestandardową sześcienną funkcję interpolacji Béziera oraz iteracje o zmiennym kierunku. Jednak klasa ta definiuje tylko ścieżkę lotu elementu, który sprawia wrażenie, jakby „machał skrzydłami”. Jest to spowodowane dodaniem klas `animRotateY` i `animRotateMinusY` w reakcji na kliknięcie myszą. Zatem do karty na dole stosowane są zagnieżdżone animacje.

Jak można zauważyc, karta znajdująca się na górze po lewej obraca się tylko raz, mimo iż w klasach `animRotateY` i `animRotateMinusY` własność `animation-iteration-count` jest ustawiona na `infinite`. Jest tak, ponieważ kod jQuery obsługujący kliknięcie zatrzymuje animację po jednym powtórzeniu (pogrubiony fragment).

```

$('#front1').click(function(){
    $('#front1').addClass('animRotateY');
    $('#back1').addClass('animRotateMinusY');
    setTimeout(function(){
        $('#front1').removeClass('animRotateY');
        $('#back1').removeClass('animRotateMinusY');

    }, 2000);
});

```

Klasy te definiują nieskończone animacje, dzięki czemu mogą być używane do tworzenia różnych efektów, np. w połączeniu z klasą `animFly`. A jeśli trzeba, można te animacje zatrzymać za pomocą kodu JavaScript.

Zaawansowane funkcje CSS

Większość przedstawionych do tej pory przykładów zawierała płaskie ruchome obiekty. Użyte do ich budowy techniki są doskonałe do tworzenia trójwymiarowych elementów interfejsu użytkownika i efektów przejść, ale nie spełniają wysokich wymagań, jakie obecnie stawia się trójwymiarowym grom i innym aplikacjom tego typu, dlatego programiści próbują przesunąć granice możliwości technologii CSS. W pozostałej części tego rozdziału znajduje się opis technik tworzenia fantastycznych trójwymiarowych efektów przy użyciu CSS3.

Renderowanie obiektów trójwymiarowych

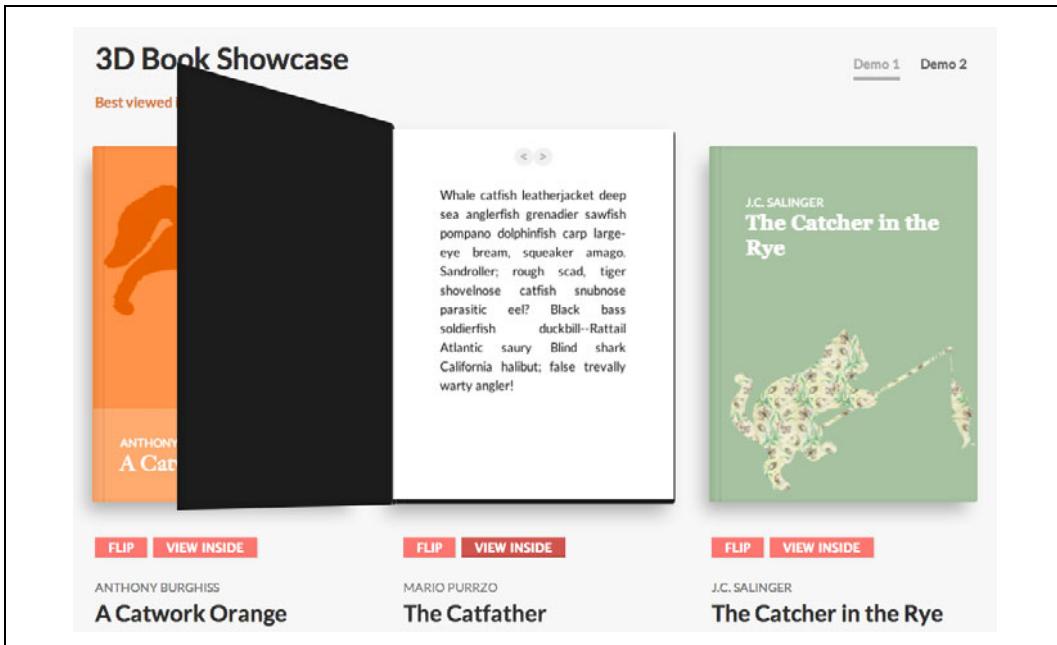
W poprzednich podrozdziałach pokazałem, że aby utworzyć dwustronny płaski obiekt, należy napisać trochę kodu HTML i CSS. A jeśli chce się otrzymać przedmiot mający głębię, np. kostkę, trzeba wysilić się jeszcze bardziej. Istnieje kilka stron internetowych poświęconych CSS3, na których znajdują się przykłady rozwiązań tego problemu. Na rysunku 6.10 widać trójwymiarowe wirtualne pudełko utworzone przez niemieckiego programistę Dirka Webera dla wirtualnej strony <http://www.eleqtriq.com>. Pudełko ma przód i tył, boki, góre i dół oraz może być obracane. Widać nawet refleksy!



Rotated packshot

Rysunek 6.10. Obracany trójwymiarowy obiekt utworzony przy użyciu CSS (<http://www.eleqtriq.com/2010/11/natural-object-rotation-with-css3-3d/>)

Zespół projektantów i programistów Codrops (<http://tympanus.net/codrops/>) rozwinał pomysł pudełka i utworzył trójwymiarową wirtualną książkę. Można ją otworzyć i zajrzeć do środka, a nawet przewracać strony (rysunek 6.11).



Rysunek 6.11. Trójwymiarowa wirtualna książka (<http://tympanus.net/codrops/2013/01/08/3d-book-showcase/>)

Aby uzyskać takie pełne trójwymiarowe efekty, należy utworzyć przynajmniej jeden element HTML dla każdego boku, zdefiniować kilka klas CSS oraz zazwyczaj dodać trochę kodu JavaScript. Nie jest to łatwe, ale rezultat może być „wart zachodu”. Jeśli chcesz zobaczyć, jakie trójwymiarowe perelki potrafią wyczarować przy użyciu CSS3 najbardziej kreatywni programiści, przejrzyj strony wymienione w tym rozdziale i w dodatku. W większości z nich kod źródłowy jest dostępny bez ograniczeń, więc możesz go wykorzystać jako bazę dla własnych prac.

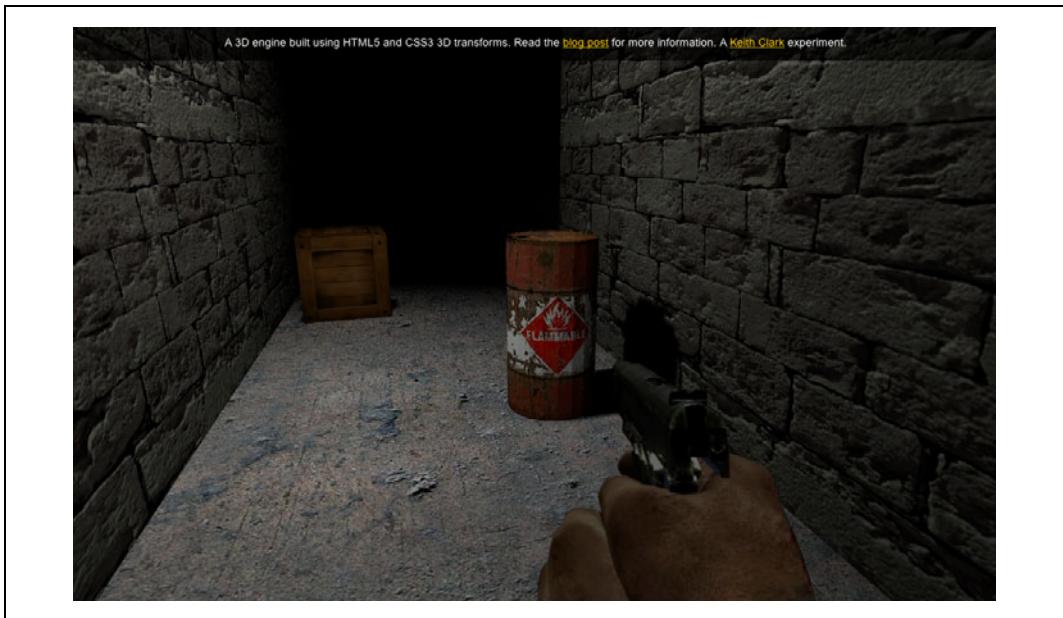
Renderowanie środowisk trójwymiarowych

Biorąc pod uwagę to, do czego technologie trójwymiarowe CSS zostały opracowane — czyli głównie przetwarzanie prostokątnych obiektów — może się wydawać, że utworzenie z ich pomocą wszechotaczającego środowiska gry jest niemożliwe. A jednak programista z Wielkiej Brytanii Keith Clark dokonał niemożliwego, czyli utworzył pierwszą strzelankę w stylu gry *Doom* przy użyciu wyłącznie kodu JavaScript i przekształceń CSS3. Efekt jego pracy można zobaczyć na rysunku 6.12.

Clark zastosował w tej grze takie efekty, których uzyskanie przy użyciu CSS3, nawet z przekształceniemi trójwymiarowymi, może wydawać się nieosiągalnym marzeniem. Oto one.

Trójwymiarowa geometria

W CSS można używać wyłącznie prostokątów, co w pierwszej chwili może się wydawać ograniczeniem, ale przecież w prawdziwych systemach do renderowania grafiki trójwymiarowej również używa się płaskich wielokątów — najczęściej trójkątów lub czworokątów. Ponadto w formacie PNG można zastosować kanał alfa, który można wykorzystać do „wycinania” kształtów w czworokątach. Te dwa składniki pozwoliły Clarkowi zbudować cylindry, pistolety, rękę gracza oraz realistyczną geometrię trójwymiarową.



Rysunek 6.12. Strzelanka utworzona przy użyciu technologii trójwymiarowych CSS i języka JavaScript (<http://blog.keithclark.co.uk/creating-3d-worlds-with-html-and-css/>)

Kamera, nawigacja oraz kolizje

Narzędzia dotyczące perspektywy w CSS są bardzo proste, ale Clark znalazł sposób na poruszanie kamerą na bieżąco za pomocą klawiszy oraz na obliczanie kolizji poprzez rzutowanie pozycji gracza na dwuwymiarową płaszczyznę podłożu i porównywanie jej z ręcznie zrobioną dwuwymiarową mapą wysokości.

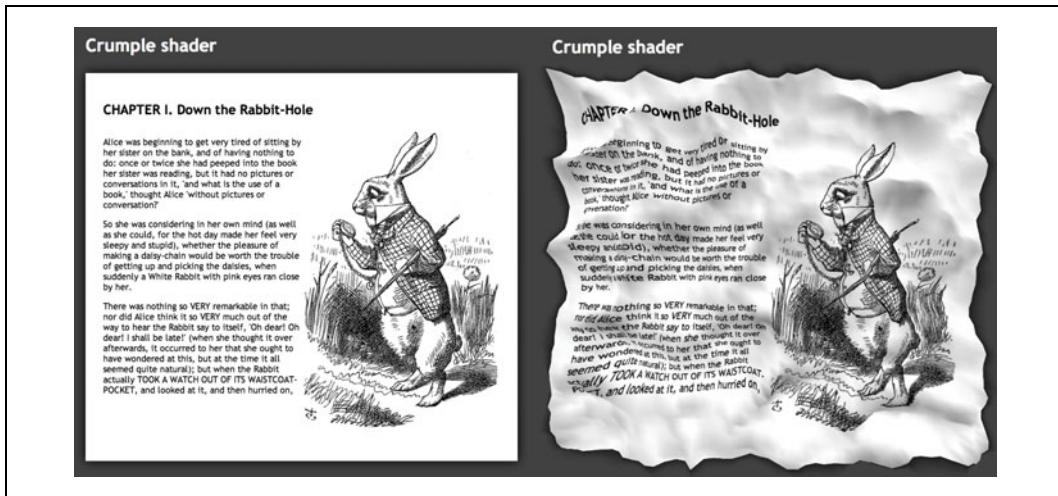
Oświetlenie i cienie

W CSS nie ma narzędzi do tworzenia efektów świetlnych. Dlatego Clark, aby otrzymać realistyczny model oświetlenia, musiał opracować wektory normalne dla każdego czworokąta i wirtualne źródła światła w JavaScriptie oraz renderować pozaekranowo tekstury na elemencie <canvas>, które następnie mieszał z podstawową tekstrurą.

Keith Clark zrobił to, czego podjęłoby się niewielu programistów. Utworzenie takiego środowiska przy użyciu WebGL i biblioteki w rodzaju Three.js byłoby znacznie łatwiejsze, ale projekt ten jest ważnym studium przypadku dotyczącym możliwości technologii CSS3. Więcej informacji na ten temat znajduje się na stronie <http://blog.keithclark.co.uk/creating-3d-worlds-with-html-and-css/>.

Tworzenie zaawansowanych efektów przy użyciu filtrów CSS

W niektórych przeglądarkach eksperymentuje się z możliwością używania języka GLSL w celu stosowania efektów trójwymiarowych do elementów CSS. Technologia ta, której pionierem jest firma Adobe Systems, nazywa się *CSS Custom Filters* (własne filtry CSS), a kiedyś nazywała się *CSS Shaders* (shadery CSS). Na rysunku 6.13 pokazano element DOM przed zastosowaniem efektu filtra i po jego zastosowaniu. Gdy użytkownik najedzie na ten element kursorem, włącza się shader, który zwiększa wierzchołki tworzące prostokąt reprezentujący ten element i za pomocą krótkiej animacji doprowadza go do stanu przypominającego zgniecioną kartkę.



Rysunek 6.13. Zgniecenie kartki za pomocą filtru własnego CSS3 (<http://alteredqualia.com/css-shaders/crumple.html>)

Najważniejszą cechą filtrów CSS jest to, że zawartość elementu DOM jest zwykłym kodem HTML: kilka fragmentów tekstu ze stylami plus obraz. Filtry CSS umożliwiają tworzenie atrakcyjnych, interaktywnych efektów bez potrzeby poznawania nowych technologii.

Filtrów własne CSS to podzbiór języka GLSL (GLSL ES). Są bardzo podobne do wersji używanej w WebGL, ale różnią się od niej pod kilkoma względami. Ze względów bezpieczeństwa filtr własny CSS nie może mieć bezpośredniego dostępu do kolorów pikseli żadnego elementu na stronie. Musi generować kolor mieszany, który zostaje połączony z pikselem docelowym elementu w celu utworzenia ostatecznego koloru. Ponadto przeglądarka dostarcza kilka gotowych wartości w postaci zmiennych typu uniform, np. macierz przekształceń trójwymiarowych zdefiniowaną w standardzie (wcześniej znajdziesz tekst na ten temat). Inną ważną różnicą jest to, że używanie filtrów własnych CSS nie jest obowiązkowe, natomiast do wyrenderowania treści przy użyciu WebGL shader jest *niezbędny*.



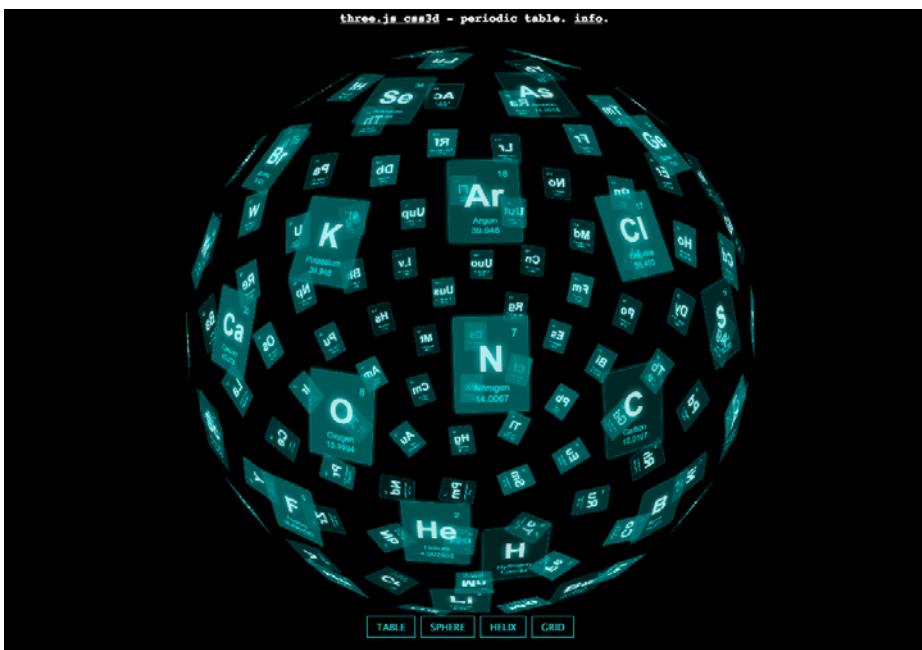
Filtrów własne CSS są na razie funkcją eksperymentalną i obsługują je tylko niektóre przeglądarki. Ponadto nie wiadomo, czy technologia ta w ogóle nie zostanie porzucona na rzecz włączenia elementów DOM do specyfikacji WebGL, aby można ich było używać jako tekstur. Wszystko rozstrzygnie się za jakiś czas. A na razie, jeśli chcesz poeksperymentować, możesz włączyć filtry w przeglądarce Chrome, wpisując w pasku adresu chrome://flags/#enable-experimental-web-platform-features i klikając odnośnik *Włącz* pod tą opcją. Później jeszcze tylko wyłącz i włącz przeglądarkę, i możesz się bawić.

Renderowanie trójwymiarowe w CSS przy użyciu Three.js

Nawet obecnie, w 2014 roku, istnieją przeglądarki, które nie obsługują WebGL. Do grupy tej zalicza się np. mobilna wersja przeglądarki Safari dla systemu iOS. Dlatego czasami trzeba też użyć innej technologii jako wyjścia awaryjnego. Jak się przekonaliśmy, jednym z dostępnych rozwiązań jest CSS3, chociaż tworzenie trójwymiarowych efektów w tej technologii może być bardzo pracochłonne. Aby opracować niewielki zbiór obiektów trójwymiarowych, trzeba napisać dziesiątki klas i elementów HTML.

Niedawno Mr.doob wpadł na pomysł utworzenia opartego na CSS systemu renderowania dla biblioteki Three.js. Jedną z zalet tej biblioteki jest właśnie to, że może renderować przy użyciu różnych technologii przeglądarczych. Program ten jest zbudowany tak, że można do niego dodawać w postaci wtyczek renderery dla WebGL, 2D Canvas, SVG, a teraz też CSS.

Renderer CSS biblioteki Three.js przesuwa, obraca i skaluje obiekty przy użyciu przekształceń trójwymiarowych, a więc doskonale nadaje się do mapowania interaktywnych elementów stron w przestrzeni trójwymiarowej. Spójrz na rysunek 6.14, na którym widać interaktywny układ okresowy pierwiastków. Każda pozycja jest elementem <div>, który można formatować za pomocą CSS i w którym można umieścić dowolną treść HTML. Renderer CSS jest doskonałym wyborem do tworzenia innowacyjnych układów prostokątnych obiektów zawierających dużo tekstu.



Rysunek 6.14. Interaktywny układ okresowy pierwiastków utworzony przy użyciu biblioteki Three.js i wyrenderowany za pomocą trójwymiarowych przekształceń CSS (http://mrdoob.github.io/three.js/examples/css3d_periodictable.html).

Podsumowanie

W rozdziale tym poznaleś wbudowane funkcje CSS3 przeglądarek do tworzenia efektów trójwymiarowych, takie jak przekształcenia, przejścia oraz animacje CSS. Za pomocą przekształceń można przesuwać, obracać i skalować elementy, renderować je z perspektywą i bez niej, rozprzestrzeniać przekształcenia w dół hierarchii DOM oraz kontrolować renderowanie tylnej ściany elementów. Korzystając z przejść, utworzyliśmy proste efekty animacyjne, a przy użyciu animacji uzyskaliśmy nieco bardziej złożone konstrukcje.

W CSS3 dostępny jest bogaty zestaw narzędzi do tworzenia trójwymiarowych elementów interfejsu użytkownika i przejść, ale technologia ta nie jest odpowiednia do trójwymiarowego renderowania na potrzeby gier komputerowych i innych aplikacji opartych na grafice 3D. Z drugiej strony, dostępne efekty tworzy się w bardzo łatwy sposób i do ich opracowania wystarczą tylko CSS i JavaScript. Działają jednolicie we wszystkich przeglądarkach we wszystkich urządzeniach, a co najważniejsze, są wbudowane w przeglądarki, więc do ich uruchomienia nie trzeba żadnych dodatkowych bibliotek. W rzadkich przypadkach konieczne jest wyjście poza granice możliwości CSS3 i wówczas można napisać własny kod JavaScript albo użyć jakiejś biblioteki renderującej do CSS3, np. Three.js.

Kanwa dwuwymiarowa

Ostatecznie nawet grafika trójwymiarowa jest renderowana na dwuwymiarowej powierzchni ekranu monitora, tabletu albo telefonu. Złudzenie trójwymiarowości uzyskuje się poprzez zastosowanie głębi i perspektywy, tzn. niektóre obiekty wydają się bliższe, a inne dalsze. Jeśli dodatkowo nasza grafika ma być interaktywna, renderowanie musi się odbywać bardzo szybko, aby zmiany były wprowadzane bez widocznego opóźnienia. Powinny następować przynajmniej 30, a najlepiej 60 razy na sekundę.

Technologie WebGL i CSS3 umożliwiają renderowanie trójwymiarowe na bieżąco przy użyciu GPU, czyli specjalnego procesora do przetwarzania grafiki, który jest obecny we wszystkich nowoczesnych komputerach i innych urządzeniach. Akcelerator grafiki trójwymiarowej, mimo że jest bardzo pomocny, nie jest jednak absolutnie niezbędny. Całkiem dobre efekty można też uzyskać programowo. W aplikacjach sieciowych do renderowania programowego używa się dwuwymiarowego kontekstu kanwy — uniwersalnego API do rysowania grafiki dwuwymiarowej w oknie przeglądarki.

W kilku sytuacjach powinno się rozważyć możliwość użycia kanwy dwuwymiarowej zamiast WebGL. Po pierwsze, kanwa jest wszechobecna, a biblioteka WebGL nie jest obsługiwana przez niektóre platformy mobilne, wśród których najważniejsza jest przeglądarka Mobile Safari dla systemu iOS. Na tych platformach kanwa dwuwymiarowa może być używana jako rozwiązanie awaryjne. Wprawdzie nie zapewni ono tak wysokiej wydajności lub dobrej jakości grafiki jak WebGL, ale przynajmniej będzie działać. Po drugie, aplikacje mogą być przeznaczone dla urządzeń o ograniczonych zasobach energii. Przykładowo procesory GPU niektórych smartfonów bardzo szybko wyczerpują akumulator i w nich lepiej zastosować rozwiązanie programowe, aby wydłużyć czas działania urządzenia. Po trzecie, czasami potrzebne są proste efekty, do utworzenia których użycie biblioteki WebGL wydaje się przesadą, a CSS3 ma za małe możliwości. W każdej z tych sytuacji warto rozważyć możliwość użycia kanwy dwuwymiarowej zamiast API WebGL.

W rozdziale tym dowiesz się, jak używać API Canvas 2D do renderowania grafiki trójwymiarowej, oraz ile stracisz na wydajności i funkcjonalności w porównaniu z tym, gdybyś użył biblioteki WebGL. Ponadto poznasz otwarte biblioteki ułatwiające wykonywanie obliczeń i renderowanie, za pomocą których będziesz mógł skoncentrować się na budowaniu aplikacji.

Kanwa — podstawowe wiadomości

API Canvas wprowadziła firma Apple w 2004 r., aby ułatwić programistom tworzenie zaawansowanych interfejsów w widżetach Pulpitu i przeglądarce Safari. Pomyśl był taki, aby zaoferować ogólną powierzchnię rysunkową. Później API to zostało zaadaptowane w silniku Gecko Mozilla i innych przeglądarkach opartych na WebKit, np. Google Chrome, a w końcu we wszystkich przeglądarkach i platformach obsługujących HTML5.

W odróżnieniu od elementów UI DOM i SVG, które były wcześniejszym standardem do rysowania dwuwymiarowej grafiki wektorowej, grafika na kanwie nie jest ograniczona do ustalonego zbioru kształtów definiowanych za pomocą znaczników. Na kanwie programista JavaScript może rysować i wypełniać dowolne kształty, takie jak linie, krzywe, wielokąty i tekst. Ponadto w kanwie, również w odróżnieniu od DOM i SVG, wykorzystywany jest niskopoziomowy proceduralny model, podobny do WebGL. Przeglądarka nie zachowuje wizualnej treści elementów kanwy w grafie sceny. Aplikacja musi sama utrzymywać własne obiekty i wywoływać funkcje rysujące w miarę potrzeby (np. podczas animacji).

Pełny opis API kanwy nie jest tematem tej książki. Jednak poniżej znajduje się objaśnienie podstawowych kwestii, których znajomość umożliwi zrozumienie jego powiązania z grafiką trójwymiarową.

Element kanwy i dwuwymiarowy kontekst rysunkowy

W HTML5 zdefiniowano nowy element o nazwie <canvas>, definiujący na stronie obszar o określonej szerokości i wysokości, na którym można rysować grafikę. Jest on podobny do elementu obrazu, tzn. można go utworzyć za pomocą specjalnego znacznika albo przy użyciu API DOM, np. z wykorzystaniem funkcji `document.createElement()`. Utworzony element kanwy można formatować za pomocą CSS, np. zdefiniować mu obramowanie i marginesy, ustawić go w odpowiednim miejscu, a nawet animować za pomocą przejęć.

Kanwa definiuje obszar na stronie, w którym można rysować grafikę. Aby rozpocząć rysowanie, należy uzyskać dostęp do **kontekstu**, który jest obiektem udostępniającym API rysunkowe. W odróżnieniu od trójwymiarowego kontekstu używanego w WebGL, kontekst kanwy jest dwuwymiarowy.

Na listingu 7.1 znajduje się przykład ilustrujący sposób utworzenia kanwy i narysowania na niej białego kwadratu. W arkuszu stylów został ustawiony czarny kolor tła. W kodzie HTML powstał element <canvas> o szerokości i wysokości określonych w pikselach. W funkcji dotyczącej ładowania strony pobieramy ten element kanwy przy użyciu jego identyfikatora i uzyskujemy dostęp do jego dwuwymiarowego kontekstu rysunkowego, wywołując `canvas.getContext("2d")`. Kiedy mamy kontekst, możemy rysować. Ustawiamy jego własność `fillStyle` na kolor biały za pomocą składni podobnej do używanej w CSS. Następnie rysujemy wypełniony prostokąt przy użyciu funkcji `context.drawRectangle()`, której przekazujemy współrzędne *x* i *y* lewego górnego rogu oraz szerokość i wysokość figury.

Listing 7.1. Prosty przykład rysowania na kanwie

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Programming 3D Applications in HTML5 and WebGL – Basic Canvas Example</title>
```

```
</head>

<style>
    #basicCanvas {
        background-color: Black;
    }
</style>
<body>

<canvas id="basicCanvas" width=500 height=500></canvas>

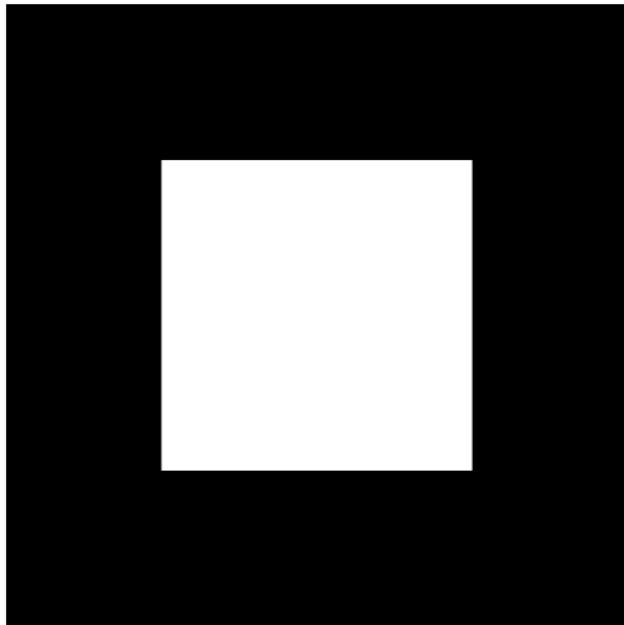
</body>
<script src="../libs/jquery-1.9.1/jquery-1.9.1.js"></script>
<script type="text/javascript">

$(document).ready(
    function() {

        var canvas = document.getElementById("basicCanvas");
        var context = canvas.getContext("2d");
        context.fillStyle = '#ffffff';
        context.fillRect(125, 125, 250, 250);
    }
);

</script>
</html>
```

Efekt powinien być podobny do widocznego na rysunku 7.1.



Rysunek 7.1. Prostokąt narysowany przy użyciu API Canvas

Przykład ten jest bardzo prosty i przypomina przykłady z rozdziałów 2. i 3., chociaż użyto w nim tylko kilku wierszy kodu JavaScript. (Wcale nie żartowalem, gdy napisałem, że są prostsze sposoby rysowania grafiki dwuwymiarowej na stronach internetowych). Kod z tego listingu znajduje się w pliku *r07/canvasbasic.html*.

Właściwości API Canvas

Kontekst Canvas 2D udostępnia API rastrowe, czyli takie, na którym rysowanie odbywa się w pikselach (w odróżnieniu od wektorów używanych np. w SVG). Jeśli aplikacja musi przeskakować grafikę w odniesieniu do rozmiaru okna, trzeba to zrobić ręcznie. Funkcje dwuwymiarowego API kanwy można z grubsza podzielić na następujące kategorie.

Funkcje do rysowania kształtów

Można rysować prostokąty, wielokąty i krzywe zarówno wypełnione kolorem, jak i puste.

Funkcje do rysowania linii i ścieżkowe

Można rysować odcinki, łuki i krzywe Béziera.

Funkcje do rysowania obrazów

Można rysować dane bitmapowe z innych źródeł, np. elementów obrazów lub innych kanw.

Funkcje do rysowania tekstu

Można rysować tekst wypełniony lub tylko obrysły liter oraz definiować właściwości tekstu za pomocą atrybutów podobnych do tych z CSS.

Style wypełnienia i obrysu

Można używać stylów i gradientów CSS do definiowania wypełnień i obrysów.

Przekształcenia

Można stosować przekształcenia dwuwymiarowe, takie jak przesunięcie, obrót, skalowanie i przy użyciu dowolnej macierzy 3×3 .

Składanie

Można kontrolować sposób mieszanego nowych kształtów z istniejącą treścią kanwy.

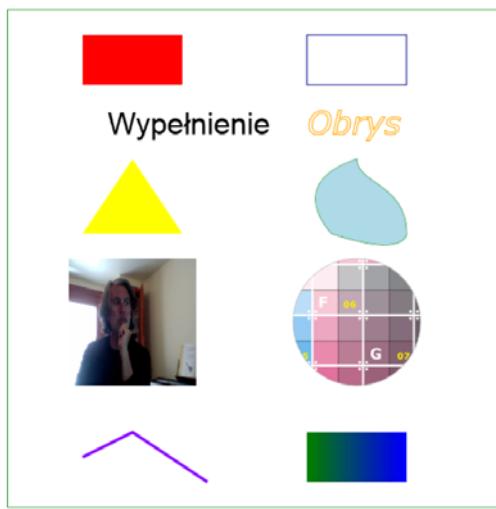
Na rysunku 7.2 można zobaczyć kanwę zawierającą kilka kształtów narysowanych za pomocą różnych wywołań rysunkowych API. Widać na nim dwa prostokąty, jeden wypełniony i jeden pusty, wypełniony i pusty tekst, wypełniony wielokąt (trójkąt), wypełnioną krzywą Béziera z obrysem, obraz bitmapowy, koło wypełnione mapą bitową, krzywą oraz prostokąt wypełniony gradientem.

Na listingu 7.2 przedstawiony został kod JavaScript do tego przykładu (z pliku *r07/canvas-features.html*). Nieistotny kod HTML został opuszczony dla uproszczenia.

Listing 7.2. Szczegółowy przykład rysowania na kanwie

```
function init()
{
    image1 = new Image();
    image1.src = '../images/parisi1.jpg';
```

Funkcje kanwy



Rysunek 7.2. Efekt działania funkcji API Canvas

```
image2 = new Image();
image2.onload = function()
{
    imagepattern = context.createPattern(image2, "repeat");
}

image2.src = '../images/ash_uvgrid01.jpg';

gradient = context.createLinearGradient(250,0,350,0);
gradient.addColorStop(0,"green");
gradient.addColorStop(1,"blue");
}

function run()
{
    requestAnimationFrame(run);
    draw(canvas, context);
}

$(document).ready(
    function() {

        canvas = document.getElementById("features");
        context = canvas.getContext("2d");
        init();
        run();
    }
);
```

Najpierw funkcja dotycząca wczytywania strony znajduje element kanwy i tworzy kontekst dwuwymiarowy, a następnie wywołuje funkcje `init()` i `run()` (która implementuje pętlę wykonawczą bazującą na funkcji `requestAnimationFrame()`). W odróżnieniu od poprzedniego przykładu, w którym kanwa była rysowana jednorazowo, tym razem rysowanie będzie wykonywane

wielokrotnie. Zrobiono tak z dwóch powodów: po pierwsze, jest to typowa struktura aplikacji opartej na kanwie, która zawiera animowaną lub reagującą na działania użytkownika treść, a warto od samego początku stosować poprawne techniki programowania; po drugie, w tym przypadku i tak potrzebujemy przynajmniej kilku klatek, ponieważ musimy sprawdzić, czy obrazy zostały załadowane. Nie powinno się podejmować próby rysowania obrazów, dopóki ich treść nie jest gotowa. Temat ten rozwinę za chwilę.

Funkcja `init()` tworzy dwa elementy obrazu, po jednym dla treści mapy bitowej i gradientu służącego jako wypełnienie prostokąta znajdującej się w prawym dolnym rogu. Ponadto buduje ona wypełnienie na bazie drugiej mapy bitowej poprzez dodanie procedury obsługi zdarzeń `onload` do obrazu przed jego załadowaniem. Procedura obsługi zdarzeń `onload` tworzy wypełnienie za pomocą metody kontekstu `createPattern()`. Metoda ta wymaga poprawnych danych bitmapowych, więc trzeba poczekać na załadowanie obrazu.

Funkcja `run()` implementuje pętlę wykonawczą. Najpierw żąda nowej klatki animacji, aby została wywołana ponownie w następnym cyklu aktualizacji przeglądarki. Następnie wywołuje odpowiedzialną za rysowanie funkcję `draw()`, której kod źródłowy znajduje się poniżej.

```
function draw(canvas, context)
{
    context.clearRect(0, 0, canvas.width, canvas.height);

    context.save();
    context.translate(50, 0);

    // Mały prostokąt wypełniony kolorem czerwonym.
    context.save();
    context.fillStyle = '#ff0000';
    context.fillRect(25, 25, 100, 50);
    context.restore();

    // Mały prostokąt wypełniony kolorem ciemnoniebieskim.
    context.save();
    context.strokeStyle = 'DarkBlue';
    context.strokeRect(250, 25, 100, 50);
    context.restore();

    // Wypełniony tekst.
    context.save();
    context.lineWidth = 1;
    context.fillStyle = 'Black';
    context.font = '30px sans-serif';
    context.fillText('Wypełnienie', 50, 125);
    context.restore();

    // Obrys tekstu.
    context.save();
    context.lineWidth = 1;
    context.strokeStyle = 'Orange';
    context.font = 'italic 2em Verdana';
    context.strokeText('Obrys', 250, 125);
    context.restore();

    // Trójkąt.
    context.save();
    context.beginPath();
    context.fillStyle = 'Yellow';
    context.moveTo(75, 150);
    context.lineTo(25, 225);
```

```

        context.lineTo(125, 225);
        context.lineTo(75, 150);
        context.closePath();
        context.fill();
        context.restore();

        // Wypełniona krzywa Béziera.
        context.save();
        context.beginPath();
        context.strokeStyle = 'Green';
        context.fillStyle = 'LightBlue';
        context.moveTo(300,150);
        context.bezierCurveTo(225,175,275,225,275,225);
        context.bezierCurveTo(350,250,350,225,350,225);
        context.bezierCurveTo(350,175,300,175,300,150);
        context.closePath();
        context.stroke();
        context.fill();
        context.restore();

        // Mapa bitowa.
        if (image1.width)
        {
            context.save();
            context.drawImage(image1, 11, 250, 128, 128);
            context.restore();
        }

        // Kolo wypełnione mapą bitową.
        if (image2.width)
        {
            context.save();
            context.strokeStyle = 'DarkGray';
            context.fillStyle = imagepattern;
            context.beginPath();
            context.arc(300, 314, 64, 0, 2 * Math.PI, false);
            context.scale(.5, .5);
            context.closePath();
            context.fill();
            context.stroke();
            context.restore();
        }

        // Lamana.
        context.save();
        context.strokeStyle = "rgb(128, 0, 255)";
        context.beginPath();
        context.lineWidth = 3;
        context.moveTo(25, 450);
        context.lineTo(75, 425);
        context.lineTo(150, 475);
        context.stroke();
        context.closePath();
        context.restore();

        // Wypełnienie gradientowe.
        context.save();
        context.fillStyle = gradient;
        context.fillRect(250, 425, 100, 50);
        context.restore();

        context.restore();
    }

```

Funkcja `draw()` jest demonstracją sposobu użycia wielu składników API Canvas 2D. Poniżej znajduje się opis kilku z nich.

- `context.clearRect()`: metoda kasująca zawartość kanwy. Gdyby jej nie wywoływano, nowa grafika byłaby dodawana na wierzchu starej.
- `context.translate()`: metoda służąca do zmieniania położenia wszystkich obiektów rysowanych po kolej. Przekazywane do niej wartości są dodawane do pozycji wszystkich innych operacji rysowania.
- Należy zwrócić uwagę na liberalne używanie metod `context.save()` i `context.restore()`. Metody te pozwalają na wykonanie zdjęcia stanu grafiki przed zmianami i przywrócenie tego stanu po zakończeniu rysowania. Zapisany stan zawiera przekształcenia, style obrysu i wypełnienia, czcionki, szerokości linii itd. Przeglądarka utrzymuje stan na stosie, a więc wywołania tych metod można zagnieździć. Jest to bardzo przydatne, gdy trzeba rysować obiekty hierarchiczne. Ogólnie rzecz biorąc, metod tych używa się w celu zapobieżenia „wykłekowi” stanu z jednej operacji rysowania do innej. Trzeba jednak mieć świadomość, że kosztem ich używania jest obniżenie wydajności, więc zanim się z nich skorzysta, należy dokładnie wszystko przemyśleć.
- Metody kontekstowe `beginPath()` i `closePath()` służą do tworzenia ścieżek linii łamanych i krzywych. Na kanwie dostępny jest wirtualny ołówek, którym steruje się za pomocą metod `moveTo()`, `lineTo()` i `bezierCurveTo()`. Metoda `beginPath()` zeruje stan ołówka, `closePath()` łączy punkt bieżącej pozycji ołówka z punktem jego pierwotnej pozycji zdefiniowanej przez pierwsze wywołanie funkcji `moveTo()`.
- Do rysowania obrazów służy metoda `context.drawImage()`. Zanim zacznie się rysować obraz na kanwie, należy poczekać na jego załadowanie. Aby dowiedzieć się, czy obraz został załadowany, sprawdza się, czy jego szerokość jest większa od zera. Metoda `drawImage()` może rysować obrazy w ich normalnych wymiarach lub je skalować, jeśli przekaże się szerokość i wysokość w czwartym i piątym argumencie. Obrazów można także używać jako wzorów do wypełniania obiektów. Skorzystaliśmy z tej możliwości, wywołując metodę `context.createPattern()` po załadowaniu obrazu `image2`. Otrzymany wzór został zapisany w zmiennej `imagepattern` i użyty do wypełnienia koła.

Analizując ten przykład, tylko powierzchownie poznaliśmy techniki rysowania grafiki dwuwymiarowej przy użyciu API Canvas, które jest bogatym systemem o dużych możliwościach. Ponadto renderując przy użyciu tego API, należy uwzględnić wiele problemów wydajnościowych i dotyczących stosowania najlepszych praktyk. Nie są to jednak tematy tej książki. W dodatku znajduje się lista źródeł informacji na temat API Canvas.

Renderowanie obiektów trójwymiarowych przy użyciu API Canvas

Znasz już podstawowe techniki rysowania na kanwie obiektów dwuwymiarowych, więc mogę przejść do omówienia kwestii dotyczących wykorzystania kanwy jako systemu renderowania rysunków trójwymiarowych. Podejść jest wiele, ale większość programowych implementacji imituje sprzętowy proces renderowania polegający na rysowaniu cieniowanych trójkątów, linii i punktów w przestrzeni ekranowej po uprzednim przekształceniu ich z przestrzeni modelowej (obiektowej).

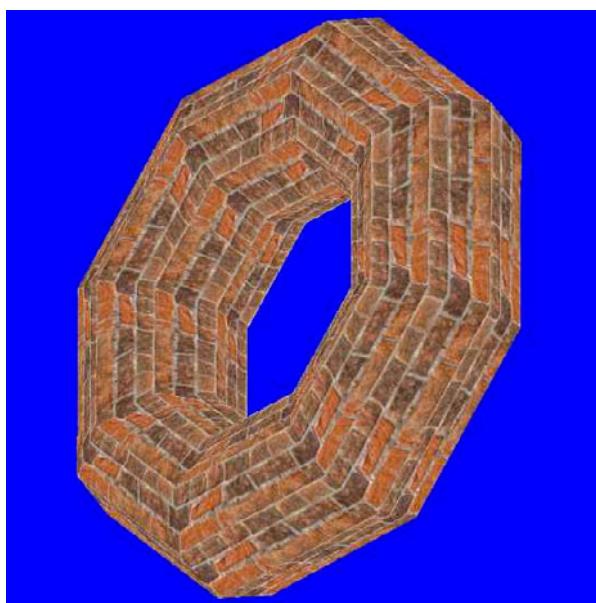
Gdy wykorzystywany jest sprzęt do generowania grafiki trójwymiarowej, większość obliczeń wykonuje się w shaderach napisanych w języku GLSL przy użyciu bardzo wydajnych wbudowanych elementów podstawowych kompilowanych do niskopoziomowego kodu maszynowego wykonywanego przez GPU. Nie mając dostępu do sprzętu, musimy wszystko robić w Java-Scripcie przed wywołaniem metod API kanwy dwuwymiarowej renderujących cieniowane i przekształcone wersje obiektów na ekranie. Obliczenia dotyczące przetwarzania geometrii trójwymiarowej, przekształceń, oświetlenia i cieniowania oraz rzutowania trójwymiarowych obiektów na dwuwymiarowy obszar widzenia są tak intensywne, że mogą spowolnić nawet najszybszy komputer, nie wspominając już nawet o spowolnieniu programisty.

Typowy renderer programowy musi wykonywać następujące zadania.

- **Przekształcanie trójkątów** z przestrzeni obiektoowej do ekranowej. W tym celu konieczne jest pomnożenie kilku macierzy, których liczba jest uzależniona od złożoności grafu sceny. Absolutnym minimum jest przekształcenie trójkątów obiektu z przestrzeni świata (przy założeniu, że nie ma żadnych dodatkowych przekształceń) na przestrzeń kamery, a potem na przestrzeń dwuwymiarowego ekranu w ramach rzutowania perspektywy.
- **Cieniowanie trójkątów** na podstawie materiałów. Jeśli obraz zawiera oświetlenie, należy włączyć normalne wierzchołków i światła. Aby wygenerować efekty świetlne przy użyciu dwuwymiarowego API Canvas, należy dynamicznie generować tekstury lub gradienty, a to może wymagać wykonania bardzo wielu obliczeń. Jeżeli materiał ma tekstury, to tekstury te również muszą być filtrowane, poprawione pod względem perspektywy oraz przetworzone na inne sposoby, aby zapewnić im gładkość i realistyczny wygląd. Szczególnie trudne jest poprawianie perspektywy i filtrowanie tekstur na bieżąco. Jak odkryjesz w kolejnych przykładach, obliczenia dotyczące tekstur mogą się bardzo różnić w zależności od aplikacji.
- **Sortowanie trójkątów** na podstawie ich odległości od obszaru widzenia. Aby scena wyglądała poprawnie, trójkąty znajdujące się bliżej obszaru widzenia powinny być renderowane na trójkątach znajdujących się dalej od niego, tzn. powinny je zasłaniać. W systemach sprzętowych do zarządzania odlegością narysowanych pikseli od kamery używany jest **bufor głębi**, zwany również **buforem z**. Bufor głębi jest tablicą równoległą do bufora kolorów. Dla każdej współrzędnej odpowiadającej buforowi kolorów bufor głębi przechowuje wartość określającą odległość, którą renderer sprawdza przed narysowaniem piksela w buforze kolorów. Jeśli dany piksel znajduje się bliżej obszaru widzenia niż jakikolwiek wcześniej narysowany piksel zarejestrowany w buforze głębi, zostanie narysowany. A jeśli znajduje się dalej, zostanie odrzucony. W programowych systemach renderowania bardzo rzadko spotyka się bufory głębi, ponieważ obliczanie wartości głębi pochłania zbyt dużo pamięci i zasobów procesora. Zamiast tego stosuje się sortowanie trójkątów na podstawie lokalizacji punktu na trójkącie. Często wybiera się do tego celu geometryczny środek figury, chociaż może to być też najbliższa lub najdalsza wartość z. Nie istnieje żaden ogólnie przyjęty standard. Sortowanie trójkątów to jedna z najbardziej obciążających systemów czynności w renderowaniu programowym, przez co czasami liczba trójkątów w aplikacji może stanowić największy problem wydajnościowy.

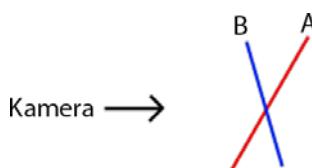
Nawet wysokiej jakości programowa implementacja renderingu nie jest pozbawiona wad. Wyglądzanie na bieżąco poszarpanych krawędzi obiektów (antialiasing) jest bardzo trudne do zrealizowania programowo, ponieważ wymaga wykonania kilku przebiegów w renderowaniu obiektu lub całej sceny. Filtrowanie tekstur przy użyciu takich technik jak mipmapping i filtrowanie bilinearne może wymagać tak intensywnych obliczeń, że będzie niemożliwe do przeprowadzenia.

W efekcie programowo generowane tekstury często wyglądają surowo i ziarniste (rysunek 7.3). Poza tym prędkość wypełniania tekstur jest w programie znacznie wolniejsza niż w sprzęcie.



Rysunek 7.3. Efekt programowego mapowania tekstur (<http://bit.ly/nomone-mapping-test>); reprodukcja za pozwoleniem

Ponadto sortowanie trójkątów w niektórych przypadkach całkiem dobrze zastępuje bufor głębi, a w innych kompletnie się rozsypuje. Przykładowo nie ma dobrego sposobu na posortowanie dwóch trójkątów, które pokrywają się częściowo. Spójrz na rysunek 7.4: z punktu widzenia kamery trójkąt B znajduje się zarówno za, jak i przed trójkątem A. Algorytm sortujący musiałby wybrać, który z nich narysować na wierzchu, a który całkowicie zasłonić. W efekcie podczas ruchu obiektów względem kamery w niektórych miejscach sceny trójkąty mogą „pojawiać się i znikać”. Gdy używany jest sprzętowy bufor głębi, coś takiego nigdy nie ma miejsca.

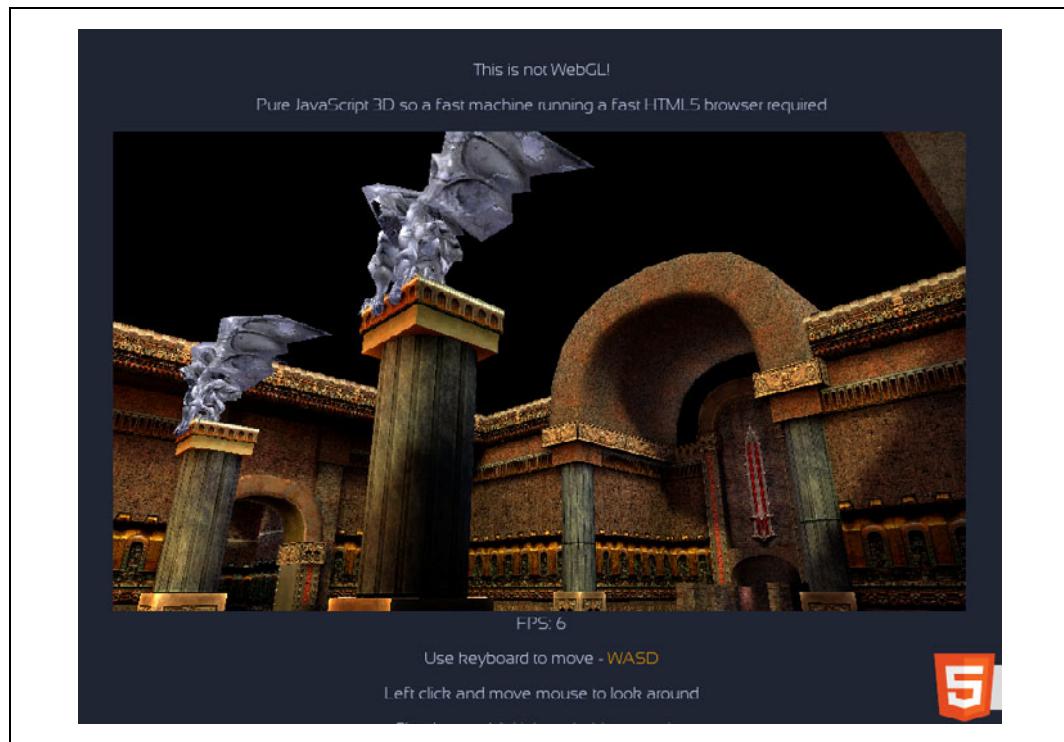


Rysunek 7.4. Sortowanie trójkątów: część trójkąta A jest bliżej kamery niż trójkąt B, ale część trójkąta B również jest bliżej kamery niż trójkąt A, więc nie da się ich dobrze posortować (obraz pochodzi z artykułu opublikowanego w portalu MSDN na temat sortowania głębokościowego — <http://bit.ly/depth-sorting-alpha-blended-objects>; reprodukcja za pozwoleniem)

Jak widać, renderowanie programowe nie jest optymalnym rozwiązaniem, gdy można skorzystać z renderowania sprzętowego. Uzyskanie takiej samej jakości obrazu i wydajności za pomocą renderera programowanego jak przy użyciu renderera sprzętowego jest niemożliwe.

Jednak, pomimo tych ograniczeń, podjęto kilka zdumiewających prób opracowania wysokowydajnego systemu renderującego grafikę trójwymiarową z wykorzystaniem API Canvas 2D.

Kilka lat temu Jean dArc z Wielkiej Brytanii utworzył opartą na dwuwymiarowej kanwie przeglądarkę map gry *Quake 3*. Na rysunku 7.5 pokazano zrzut ekranu z tej aplikacji, którą można wypróbować na stronie <http://www.zynaps.com/site/experiments/quake.html>. Na dobrym laptopie wydajność jest całkiem zadowalająca, a tekstury, mimo że trochę ziarniste z powodu braku filtrowania, wyglądają nieźle. Aplikacja ta powstała jako eksperyment do pokazania możliwości kanwy dwuwymiarowej w przeglądarce Chrome. Utworzono ją w czasie, gdy biblioteka WebGL była jeszcze mało znana. Obecnie wynik tego eksperymentu nie ma większego znaczenia, ale ukazuje, jakie możliwości daje API Canvas 2D w połączeniu z dobrymi technikami programistycznymi.



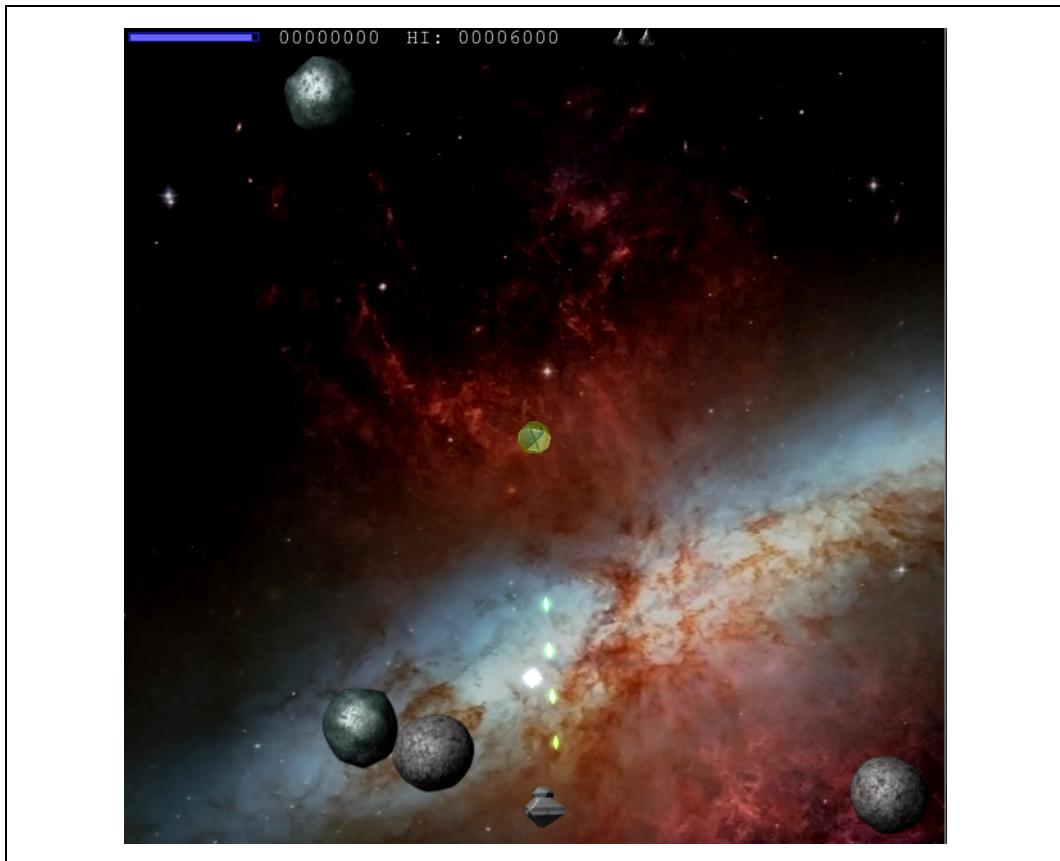
Rysunek 7.5. Przeglądarka map gry *Quake 3* renderująca programowo przy użyciu API Canvas 2D (<http://www.zynaps.com/site/experiments/quake.html>)

Trójwymiarowe biblioteki oparte na kanwie

Jak już pisałem, aby renderowanie grafiki trójwymiarowej na kanwie było możliwe, konieczne jest przezwyciężenie pewnych poważnych problemów technicznych. Powstało kilka bibliotek mających rozwiązać te problemy, np. K3D (<https://launchpad.net/canvask3d>), Cango3D (<http://bit.ly/cango3d>), Nihilogic (<http://www.nihilogic.dk/labs/canvas3d/>) oraz oczywiście Three.js (<https://github.com/mrdoob/three.js/>). Przyjrzymy się dwóm z nich, czyli K3D i Three.js.

K3D

K3D to owoc pracy Kevina Roasta z Wielkiej Brytanii (<http://www.kevs3d.co.uk/dev/>; Twitter: @kevinroast). Roast jest programistą interfejsów użytkownika i pasjonuje się grafiką. Jego biblioteka, mimo że jest jeszcze w początkowej fazie rozwoju i nie dorównuje bogactwem funkcji Three.js, robi duże wrażenie. Jest szybka i bardzo dobrze radzi sobie z cieniowaniem oraz teksturami. Na rysunku 7.6 widać zrzut ekranu z *Asteroids [Reloaded]*, implementacji klasycznej gry napisanej przez Roasta przy użyciu biblioteki K3D. Warto zwrócić uwagę na płynne cieniowanie, oświetlenie oraz szczegółowość tekstur na skałach.



Rysunek 7.6. *Asteroids [Reloaded]*, trójwymiarowa gra utworzona przy użyciu biblioteki K3D i renderowana za pomocą API Canvas 2D (<http://www.kevs3d.co.uk/dev/asteroids>)

Bazując na doświadczeniu zdobytym przy budowie K3D, Roast pracuje teraz nad biblioteką *Phoria* (<http://www.kevs3d.co.uk/dev/phoria>), czyli na nowo napisaną wersją K3D. Według jego zapewnień, *Phoria* ma mieć większe możliwości i być bardziej ogólna, ale na razie jest w fazie raczkowania, więc wersje demo utworzone przy użyciu K3D są o wiele ciekawsze.

Renderer biblioteki Three.js rysujący na kanwie

Jako że biblioteki Three.js używaliśmy we wcześniejszych przykładach, wydaje się sensownym rozwiązaniem wykorzystanie jej też do programowego renderowania grafiki trójwymiarowej zwłaszcza wtedy, gdy głównym celem jest opracowanie wyjścia awaryjnego dla platform nieobsługujących WebGL. Przy użyciu Three.js można renderować do WebGL, gdy jest to możliwe, oraz przełączać się na kanwę w pozostałych przypadkach, przy czym różnice w kodzie są bardzo nieznaczne. Oczywiście przełączanie się między rendererem trójwymiarowym a dwuwymiarowym nie jest całkowicie bezproblemowe — trzeba wprowadzić kilka zmian w kodzie, aby w pełni wykorzystać możliwości kanwy — ale problemy te nie są trudne do przezwyciężenia.

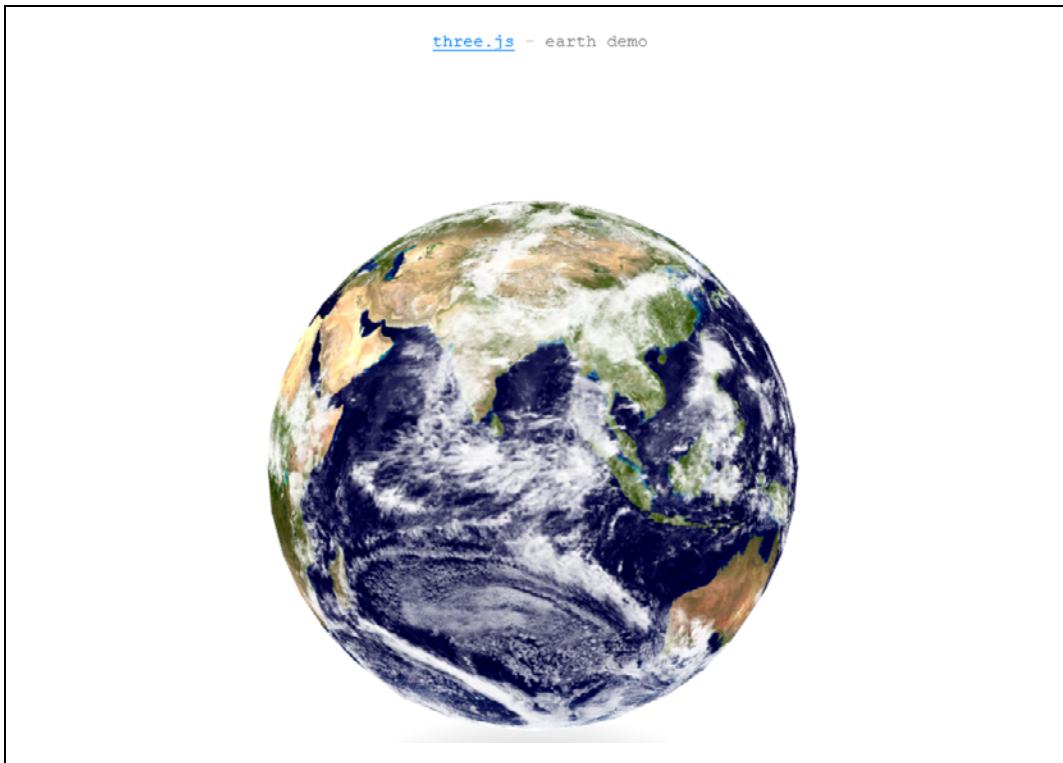


W bibliotece Three.js renderery można dodawać jako wtyczki, a renderer bazujący na dwuwymiarowym API Canvas jest gotowy do użytku od razu. Nie ma w tym nic dziwnego, gdy weźmie się pod uwagę pochodzenie tej biblioteki, która powstała na bazie wcześniejszej pracy Mr.dooba, która służyła do renderowania przy użyciu dwuwymiarowych elementów Flasha. Renderer kanwy HTML5 był więc naturalną kontynuacją tego projektu. Naprawdę został on zaimplementowany przed trójwymiarowym rendererem WebGL.

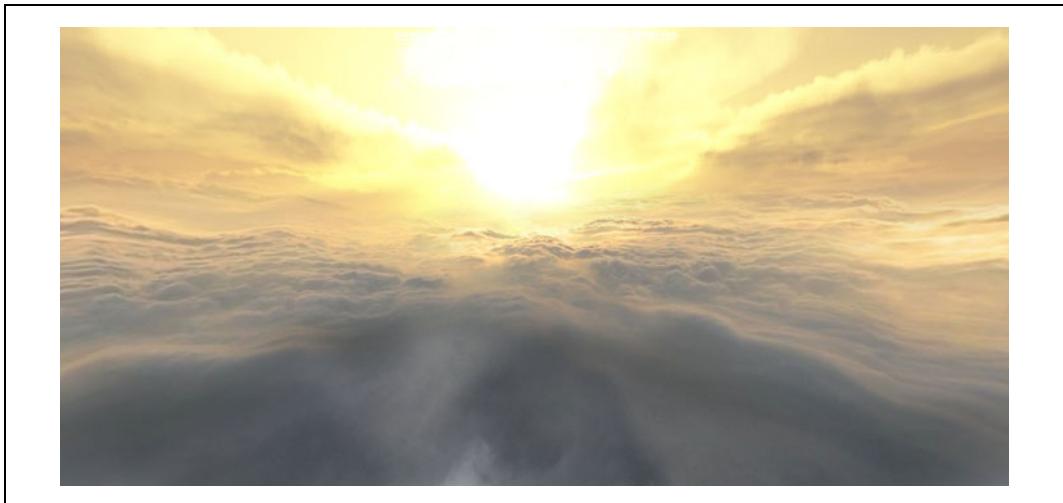
W bibliotece Three.js znajdziemy wiele przykładów programów opartych na kanwie, ale — niestety — niewiele z nich jest ciekawych. Otwórz plik `examples/canvas_geometry_earth.html`, aby wyświetlić stronę widoczną na rysunku 7.7. Jest to obracająca się Ziemia. Nie dorównuje ona jakości swojemu odpowiednikowi utworzonemu przy użyciu WebGL, ale i tak nie jest najgorsza. Najbardziej może się rzucić w oczy niewysoka teselação, tzn. kula jest zbudowana ze stosunkowo niewielu liczb trójkątów. Widać jej trójkątne krawędzie, gdy się obraca. Może nie wygląda jeszcze jak piłka golfowa, ale mogłaby być zdecydowanie bardziej gładka. Niedogodność ta jest spowodowana przez sortowanie głębokościowe trójkątów. Liczbę trójkątów należy ograniczać do minimum, ponieważ złożoność operacji ich sortowania w najlepszym przypadku wynosi $O(N \log N)$, przez co im więcej tych figur geometrycznych, tym dłużej trwa ich sortowanie.

Renderer kanowy bardzo dobrze spisuje się przy rysowaniu prostych trójwymiarowych panoram. W tym przypadku renderowanych jest 12 trójkątów (tzn. boków poewnętrznej stronie kostki), a więc ich liczba nie stanowi problemu. Otwórz plik `examples/canvas_geometry_panoramas.html`, aby wyświetlić trójwymiarową panoramę widoczną na rysunku 7.8 (można ją obracać za pomocą myszy). Nawigacja jest płynna, a tekstura wygląda świetnie.

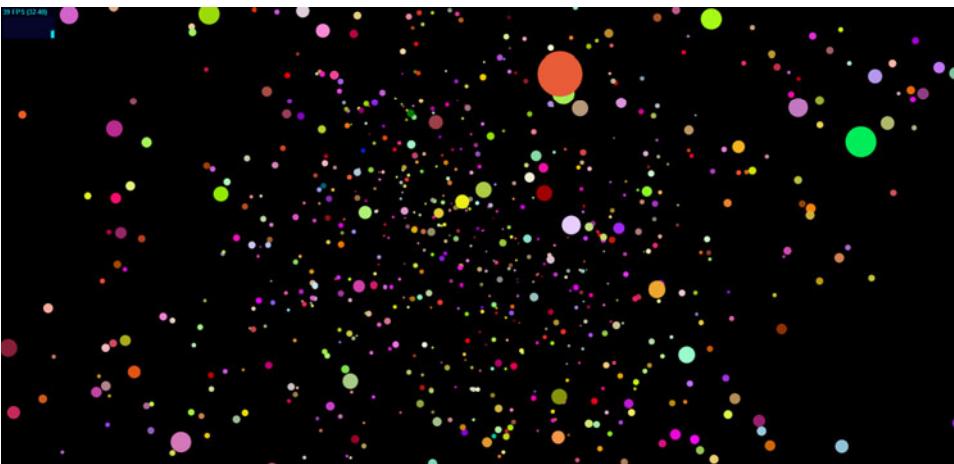
Ponadto renderer kanowy biblioteki Three.js świetnie radzi sobie z rysowaniem dużych ilości prostych kształtów, np. płaskich dwuwymiarowych wielokątów rozmiieszczonych w trójwymiarowej przestrzeni. Można to wykorzystać do tworzenia na stronie fantastycznych efektów, takich jak animowane częsteczki. Na rysunku 7.9 widać przykład (z pliku `examples/canvas_particles_random.html`) animacji tysiąca losowo rozmiieszczonych częsteczek. Figury są płaskie, ale poruszają się w trójwymiarowej przestrzeni, gdy przesunie się kursem po ekranie. Wyobraź sobie teraz, że masz to zrealizować za pomocą przekształceń trójwymiarowych CSS3. Tysiąc ruchomych elementów z pewnością bardzo obciążałby DOM przeglądarki. Natomiast implementacja kanwy i biblioteka Three.js sprawiają, że zadanie to jest proste.



Rysunek 7.7. Pokryta teksturą ziemia wyrenderowana przy użyciu renderera rysującego na kanwie



Rysunek 7.8. Panorama wygenerowana na kanwie przy użyciu biblioteki Three.js



Rysunek 7.9. Tysiąc animowanych cząsteczek wygenerowanych przy użyciu renderera kanowego biblioteki Three.js

Sposób użycia renderera kanowego biblioteki Three.js

Użycie kanwy w bibliotece Three.js sprawdza się do utworzenia określonego typu obiektu renderera, chociaż trzeba pamiętać o kilku szczegółach. Sprawdzimy, jak to się robi, na prostym przykładzie, a przy okazji zobaczymy, jakie są różnice wizualne i wydajnościowe między kanwą a WebGL. W końcu obejrzymy też przykład zbliżony do realnej sytuacji. Wszystkie przedstawione do tej pory przykłady są wyssane z palca — to zwykłe wersje demo prezentujące możliwości technologii. Teraz spróbowujemy czegoś bardziej realistycznego, czyli grafiki gry z wielokątowymi modelami i teksturami.

Na rysunku 7.10 widać zrzut ekranu z eksperymentalnej aplikacji mającej na celu sprawdzenie możliwości renderera kanowego biblioteki Three.js w zakresie budowy gier. Jest to prosta przeglądarka pozwalająca na ocenę wizualną jakości produktu i szybkości generowania klatek. Otwórz plik *r07/threescanvasasmodel.html* w przeglądarce internetowej. Na ekranie pojawią się trzy statki kosmiczne dryfujące powoli na prostym gwiazdostym niebie. Za pomocą myszy można obracać scenę, a przy użyciu kółka można ją pomniejszać i powiększać. Animację można zatrzymać i uruchomić, klikając pole wyboru *Animacja*. Dodatkowo istnieje możliwość przełączania między rendererami WebGL a kanowym, aby porównać efekt. Najpierw przyjrzymy się wersji z kanwą.

Spójrz na znajdujący się w lewym górnym rogu licznik klatek. Jego wartość wahą się w przedziale od 20 do 23, ale jeśli powiększysz scenę tak, aby jeden ze statków kosmicznych znalazł się poza ekranem, liczba ta powinna podskoczyć do około 30. Jeśli jeszcze bardziej powiększysz scenę, aby został na niej tylko jeden statek, możesz uzyskać nawet 50 do 60 klatek na sekundę. Jest to doskonała ilustracja opisanego wcześniej problemu dotyczącego sortowania trójkątów. Renderer kanowy biblioteki Three.js nie ma bufora głębi, więc biblioteka stosuje sortowanie trójkątów, a im więcej trójkątów, tym wolniejsze sortowanie. Gdy powiększysz scenę tak, aby był widoczny tylko jeden statek kosmiczny, biblioteka ignoruje niewidoczne trójkąty i ich nie sortuje. Modele statków kosmicznych w tym przykładzie są raczej proste —

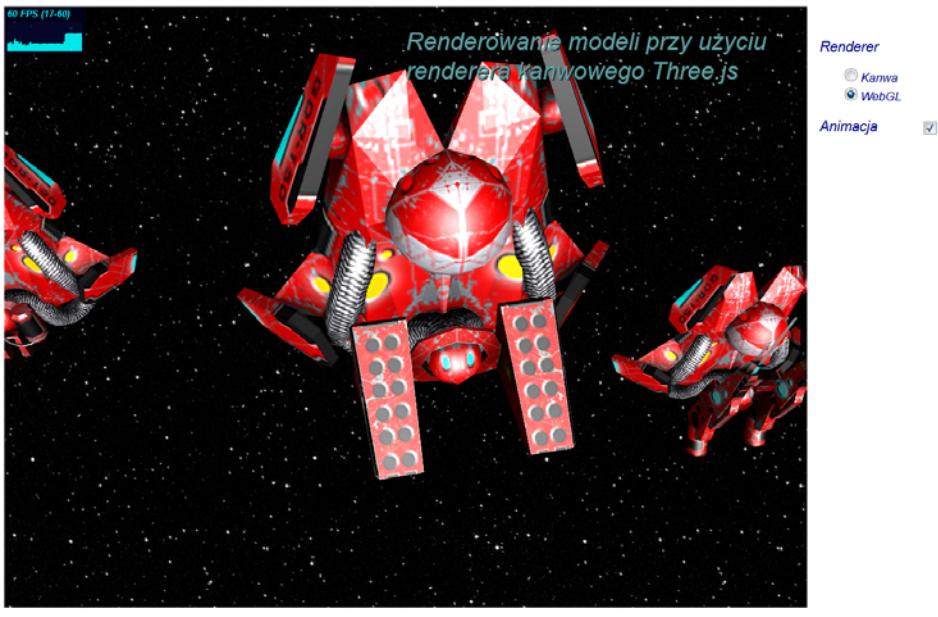


Rysunek 7.10. Renderowanie modeli przy użyciu renderera kanwowego biblioteki Three.js (modele statków kosmicznych utworzył gentlemank — <http://bit.ly/fYP2an> — który udostępnił je w serwisie Turbosquid — <http://bit.ly/1cju38z>)

każdy składa się z około 1200 trójkątów. Nie jest to duża liczba w dzisiejszych standardach dla gier, co pokazuje, jak bardzo trzeba ograniczać ilość wielokątów, gdy renderuje się na kanwie dwuwymiarowej. Przyjrzyjmy się też materiałom. Statki są oświetlone i na scenie obecne jest światło kierunkowe, które powinno oświetlać różne części geometrii statków, ale tego efektu nie widać. Biblioteka Three.js zastosowała trochę oświetlenia, ale są to proste efekty. Brak płynnego cieniowania na bokach. Obejrzyj inne przykłady dotyczące kanwy w bibliotece Three.js, aby zobaczyć, co można zrobić z materiałami i oświetleniem.

Porównanie renderera kanwowego z WebGL

Teraz zmienimy renderer, aby zobaczyć różnicę. Kliknij przycisk radiowy WebGL i włącz renderowanie sceny przy użyciu biblioteki WebGL. Spójrz na rysunek 7.11. Różnice od razu rzucają się w oczy. W wersji WebGL tekstury są gładzsze zwłaszcza wtedy, gdy obiekt znajduje się w głębi sceny, a w wersji renderowanej na kanwie wyglądają dość ziarniste. Także wygładzanie krawędzi w WebGL jest znacznie lepsze, chociaż w wersji na kanwie też zostało zastosowane. Jednak największa różnica jest w oświetleniu. W obrazie wygenerowanym przez WebGL wyraźnie widać refleksy światła kierunkowego, których na kanwie w ogóle nie było. Jeśli chodzi o wydajność, wystarczy spojrzeć na licznik klatek, który w przypadku wersji WebGL oscyluje w okolicy liczby 60 i to bez ignorowania niewidocznych obiektów. Oczywiście nie ma w tym nic dziwnego, bo programowo biblioteka Three.js ma tu niewiele pracy. Na scenie znajduje się zaledwie kilka obiektów zbudowanych z niewielkiej liczby wielokątów i pokrytych prostą tekstonią. Prawie wszystkie obliczenia wykonywane są przez sprzęt (tzn. kod GLSL shadera wbudowany w renderer WebGL biblioteki Three.js).



Rysunek 7.11. Scena ze statkami kosmicznymi wyrenderowana przy użyciu renderera WebGL biblioteki Three.js

Wynik tego porównania może sugerować, że nie warto używać kanwy do renderowania trójwymiarowych gier. Uzyskaliśmy niezbyt wysoką liczbę klatek na sekundę, a i tak musielibyśmy pogodzić się ze sporymi stratami pod względem wizualnym. Jednak takie ocenianie jest jak mówienie, że szklanka jest w połowie pusta, zamiast twierdzić, że jest w połowie pełna. Z drugiej strony, przecież udało się nam rozruszać na stronie tysiące pokrytych teksturą trójkątów przy użyciu JavaScriptu. Jeśli będziesz tworzyć proste gry i nauczysz się przygotowywać grafiki z uwzględnieniem ograniczeń swojego medium (np. złożone z niewielu wielokątów i od razu z oświetleniem), będziesz w stanie osiągnąć fantastyczne efekty.

Aby użyć renderera kanowego biblioteki Three.js, wystarczy napisać tylko kilka wierszy kodu. Kod źródłowy opisywanego przykładu znajduje się w pliku *r07/threescanvasmodel.html*. Na listingu 7.3 przedstawiono kod odpowiedzialny za utworzenie renderera. Zwróć szczególną uwagę na pogrubiony wiersz. Zamiast tworzyć renderer WebGL, utworzyliśmy obiekt typu THREE.CanvasRenderer.

Listing 7.3. Tworzenie renderera kanowego w bibliotece Three.js

```
function createRenderer(container, useCanvas)
{
    if (useCanvas) {
        renderer = new THREE.CanvasRenderer( { } );
    }
    else {
        renderer = new THREE.WebGLRenderer( { antialias: true } );
    }
    container.appendChild( renderer.domElement );

    // Ustawia rozmiar obszaru widzenia.
    renderer.setSize(container.offsetWidth, container.offsetHeight);
}
```

Kiedy już utworzony został renderer kanwowy, można w nim renderować w taki sam sposób jak w WebGL.

```
// Renderowanie sceny.  
renderer.render( scene.scene, scene.camera );
```

W prostych przypadkach jest to jedyna potrzebna zmiana w kodzie, ale w tym przykładzie musimy zrobić coś jeszcze. W bibliotece Three.js istnieje możliwość zastosowania prostego wygła- dzania krawędzi poprzez „nadrysowanie” trójkątów, tzn. narysowanie wszystkiego w rozmiarze o jeden piksel większym niż się powinno, aby ukryć łączenia. Niestety, nie wystarczy w tym celu ustawić znacznika `antialias` w rendererze (tak byśmy to zrobili w WebGL), tylko trzeba zastosować odpowiednie ustawienie dla każdego materiału osobno. W tym celu należy przejrzeć materiały po załadowaniu modelu i ustawić ich własność `overdraw` na `true`. Na listingu 7.4 znajduje się kod metody zwrotnej dotyczącej ładowania modeli. Metoda ta przegląda dany model za pomocą metody `traverse()`, która przegląda każdego potomka w grafie sceny. Funkcja pomocnicza `processNodes()` sprawdza, czy dany obiekt jest siatką. Jeśli tak, ustawia własność `overdraw` na materiał siatki. Ta dodatkowa praca jest trochę niewygodna, ale ogólnie cały kod i tak jest prosty. Te dwie zmiany stanowią jedyne różnice między programem bazującym na kanwie a programem opartym na bibliotece WebGL.

Listing 7.4. Iteracja przez materiały na scenie

```
function processNodes(n)  
{  
    if (n instanceof THREE.Mesh)  
    {  
        n.material.overdraw = true;  
    }  
}  
  
function handleSceneLoaded(data, parent)  
{  
    // Dodaje siatkę do grupy.  
    parent.add( data.scene );  
    data.scene.traverse(function(n) { processNodes(n) });  
}
```

Podsumowanie

W rozdziale tym poznaleś programowe techniki renderowania grafiki trójwymiarowej przy użyciu API Canvas 2D obsługiwanej przez wszystkie przeglądarki zawierające HTML5. Po krótkim wprowadzeniu do podstawowych technik rysowania na kanwie dwuwymiarowej dowiedziałeś się, jakie problemy stwarza renderowanie programowe w kwestii przekształceń, cieniowania i sortowania głębokościowego. W trakcie przeglądu bibliotek do trójwymiarowego renderowania na kanwie dowiedziałeś się, jak używać renderera kanwowego biblioteki Three.js zamiast renderera WebGL. Mimo że trzeba pogodzić się z widoczną stratą jakości i wydajności, kanwa jest dobrą alternatywą dla biblioteki WebGL w prostych aplikacjach; może też służyć jako wyjście awaryjne.

Techniki tworzenia aplikacji

Proces powstawania treści trójwymiarowej

Na początku istnienia internetu każdy, kto wiedział, jak tworzyć znaczniki, był twórcą treści. Nie było wizualnych edytorów typu Dreamweaver ani programów do obróbki grafiki, takich jak Photoshop. Większość zadań spoczywała na barkach programistów — i to było widać. Z czasem zaczęły się pojawiać profesjonalne narzędzia do tworzenia treści przeznaczonej na strony internetowe. Potem obowiązek przygotowywania treści przejęli artyści oraz projektanci i wówczas internet przekształcił się w produkt konsumencki.

Rozwój biblioteki WebGL przypomina wczesną ewolucję internetu. Przez kilka pierwszych lat istnienia tej technologii treść tworzyli ręcznie programiści, którzy wpisywali kod do edytorów tekstu albo korzystali z wszelkich trójwymiarowych formatów, dla których udało im się znaleźć konwerter. A jeśli brakowało jakiegoś konwertera, pisali go samodzielnie.

Na szczęście, ten stan rzeczy szybko ulega zmianie. Biblioteki, takie jak Three.js, coraz lepiej importują treść utworzoną przy użyciu profesjonalnych narzędzi. Ponadto przedstawiciele branży wspólnie tworzą nowy, trójwymiarowy, standardowy format plików przeznaczony specjalnie do użytku w internecie. Oczywiście tworzenie treści to wciąż grząski teren, ale przynajmniej nie jesteśmy już w epoce kamienia łupanego i bliżej nam do epoki brązu.

W rozdziale tym opiszę proces tworzenia treści trójwymiarowej dla aplikacji sieciowych. Najpierw przedstawię ogólny obraz problemu. Wiadomości te będą szczególnie przydatne dla tych, którzy nigdy wcześniej nie mieli do czynienia z tworzeniem grafiki trójwymiarowej. Następnie omówię popularne narzędzia do modelowania i budowania animacji w projektach WebGL oraz szczegółowo opiszę trójwymiarowe formaty plików, które najlepiej nadają się do użytku w sieci. Na końcu wyjaśnię, jak wczytywać te pliki do aplikacji za pomocą narzędzi biblioteki Three.js. Wiedza ta przyda się w następnych rozdziałach.

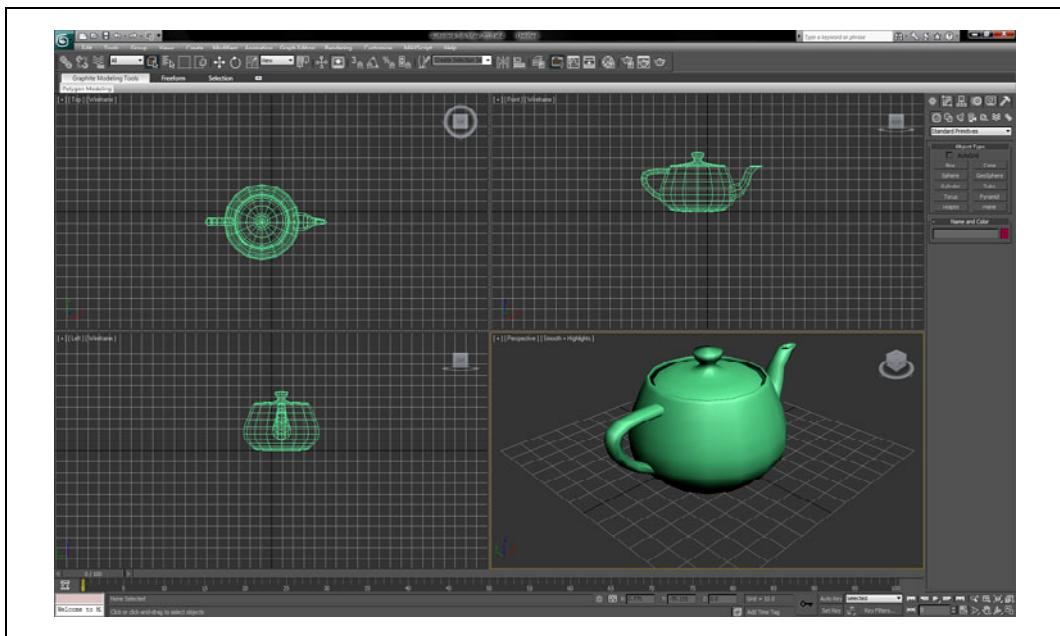
Proces tworzenia grafiki trójwymiarowej

Tworzenie grafiki trójwymiarowej to proces wymagający połączenia kilku specjalistycznych dziedzin. Profesjonalny grafik musi być bardzo dobrze wyszkolony, powinien bardzo dobrze obsługiwać skomplikowane programy i świetnie znać ogólny proces twórczy. Często jest tak, że jeden artysta robi wszystko, tzn. tworzy modele, tekstury i animacje. Jednak czasami, zwłaszcza w dużych projektach, ludzie specjalizują się w wybranych czynnościach.

Tworzenie treści trójwymiarowej pod wieloma względami przypomina budowanie dwuwymiarowych obrazów w takich programach jak Photoshop czy Illustrator, ale oczywiście między tymi dziedzinami występują też pewne ważne różnice. Jeśli nawet ktoś jest obeznaný z komputerami, przed rozpoczęciem pracy nad projektem trójwymiarowym powinien dowiedzieć się, co trzeba mieć i umieć, aby rozpocząć pracę. Sprawdźmy zatem, jakie są wymagania.

Modelowanie

Tworzenie modelu trójwymiarowego zazwyczaj rozpoczyna się od narysowania szkicu przez rysownika. Później szkic ten przekształca się za pomocą programu do modelowania w cyfrowy trójwymiarowy obraz. Modele najczęściej mają postać trójwymiarowych siatek wielokątów, które następnie pokrywa się materiałami. Czynności te nazywa się **modelowaniem trójwymiarowym**, a osobę, która je wykonuje zawodowo — **modelarzem**. Na rysunku 8.1 można zobaczyć prosty model czajnika utworzony w programie Autodesk 3ds Max. Widoczne są cztery rzuty modelu: z góry, od lewej, z przodu oraz perspektywiczny.

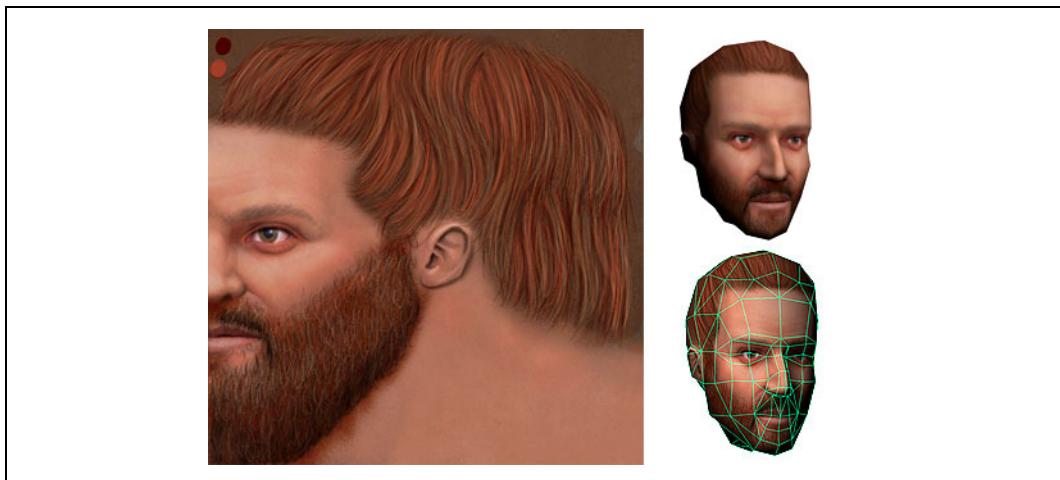


Rysunek 8.1. Model trójwymiarowy w programie 3ds Max zawierający rzuty z góry, z lewej, z przodu oraz perspektywiczny (obraz autorstwa firmy ©Autodesk pobrany z artykułu w Wikipedii na temat programu 3ds Max — http://en.wikipedia.org/wiki/File:3dsmax_2010_800px.png)

Teksturowanie

Teksturowanie (albo **mapowanie tekstur** lub **mapowanie UV**) polega na tworzeniu dwuwymiarowych obrazów i nakładaniu ich na powierzchnię trójwymiarowych obiektów. Modelarze często sami wykonują tekstury, chociaż w większych projektach obowiązek ten może spoczywać na **specjalistach zajmujących się wyłącznie teksturowaniem**. Mąpanie tekstur najczęściej wykonuje się przy użyciu wizualnych narzędzi dostępnych w programie do modelowania.

Narzędzia te umożliwiają powiązanie wierzchołków siatki z pozycjami na dwuwymiarowej teksturze, przy jednoczesnym obserwowaniu efektu na ekranie. Na rysunku 8.2 pokazano proces teksturowania. Po lewej stronie znajduje się tekstura, po prawej na górze widać ostateczny wynik, a na dole widoczna jest siatka z nałożoną tekstrzą. Warto zwrócić uwagę na dość dziwny wygląd obrazu po lewej, na którym widać tylko pół twarzy. Jest tak dlatego, ponieważ w tym przypadku lewa część twarzy jest lustrzanym odbiciem prawej. Dzięki temu artysta może upakować więcej danych w mniejszej przestrzeni.

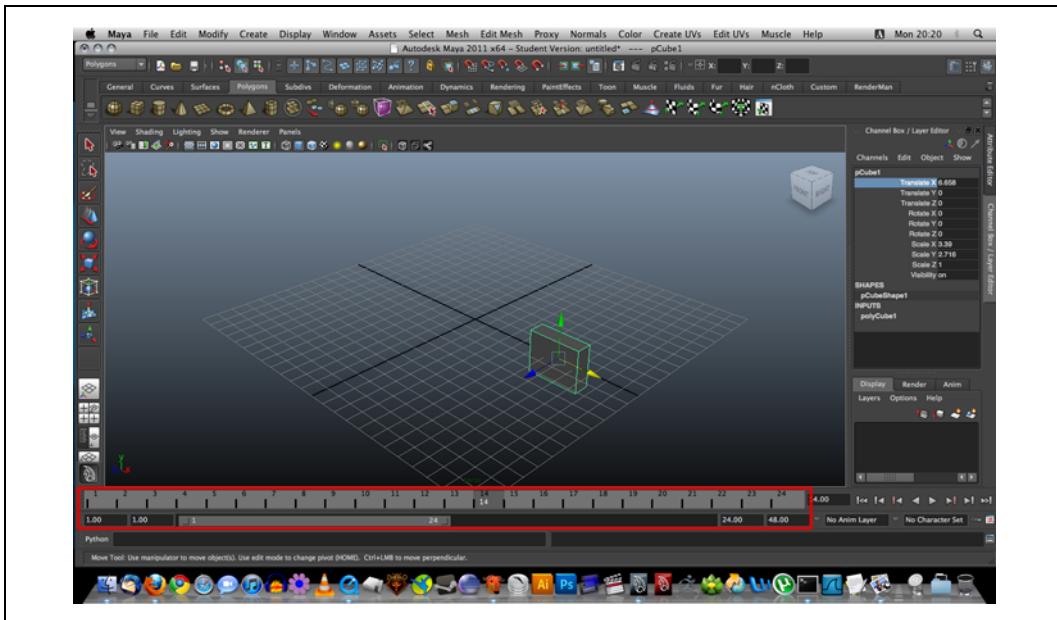


Rysunek 8.2. Mapowanie tekstuury: dwuwymiarowy obraz jest nakładany na powierzchnię trójwymiarowego obiektu (obraz opublikowany dzięki uprzejmości Simona Wottge'a — <http://www.simonwottge.com/?cat=13>)

Animowanie

Tworzenie animacji trójwymiarowej może być zarówno banalnie łatwe, jak i koszmarnie trudne. Wszystko zależy od tego, co chce się otrzymać. Animacje oparte na klatkach kluczowych z reguły są proste, przynajmniej koncepcyjnie. Interfejsy mogą być skomplikowane w obsłudze i za-gracone. Edytor klatek kluczowych, taki jak widoczny na rysunku 8.3 z programu Autodesk Amaya, zawiera zestaw przycisków do obsługi osi czasu (zaznaczony czerwonym prostokątem na dole okna), za pomocą których artysta, nazywany też **animatorem**, może przesuwać i zmieniać na inne sposoby przedmioty znajdujące się na widoku, aby później zdefiniować klatki kluczowe. Klatki kluczowe mogą służyć do zmiany położenia, kąta obrotu, skali, a nawet właściwości oświetlenia i materiałów. Gdy animator chce uwzględnić w klatce kluczowej więcej niż jeden atrybut, dodaje do osi czasu nową ścieżkę. Ścieżki w interfejsie są ułożone jedna na drugiej, przez co interfejs programu może być bardzo nieczytelny.

Animowanie postaci przy użyciu techniki skinningu jest znacznie bardziej skomplikowane. Najpierw należy utworzyć zestaw kości, czyli **rusztowanie** (ang. *rig*). Rusztowanie decyduje o różnych aspektach ruchu skóry w odpowiedzi na ruchy kości. Tworzenie rusztowania to ścisłe specjalistyczna dziedzina, którą często zajmują się specjaliści inni niż animatorzy.

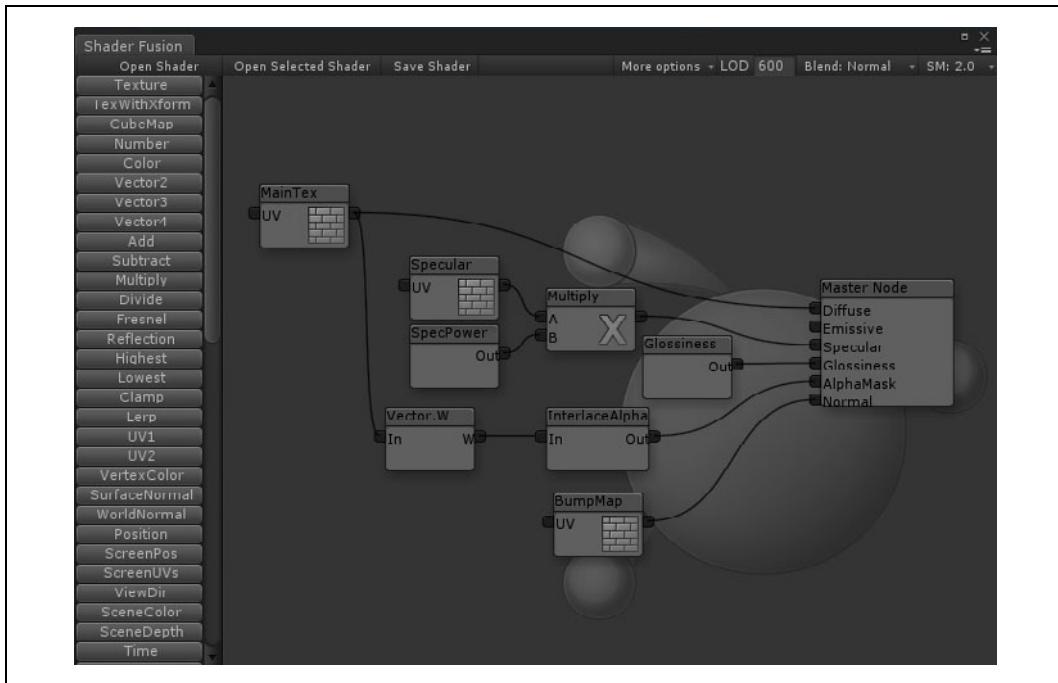


Rysunek 8.3. Narzędzie do obsługi osi czasu w programie Maya zawierające przyciski do obsługi klatek kluczowych animujących położenie, kąt obrotu, skalę i inne właściwości (obraz opublikowany dzięki uprzejmości UCBUGG Open Course Ware — <http://ucbugg.github.io/learn.ucbugg/introduction-to-maya/>)

Sztuka techniczna

Programowania nie należy traktować jako czynności polegającej na tworzeniu treści, ale w dziedzinie grafiki trójwymiarowej często tak się robi. Tworzenie za pomocą shaderów i technik przetwarzania końcowego skomplikowanych efektów specjalnych może wymagać umiejętności, jakie posiada tylko doświadczony programista. W studiach produkujących gry i animacje obowiązki te należą do pracownika o nazwie **artysta techniczny** (ang. *technical artist*) lub **dyrektor techniczny** (ang. *technical director*). Zakres obowiązków na tych stanowiskach nie jest precyzyjnie określony i trudno też wskazać, co naprawdę je różni. Chociaż zgodnie z nazwą można wnioskować, że dyrektor techniczny to zapewne starsza i bardziej doświadczona osoba. Dyrektorzy techniczni piszą skrypty, tworzą postaci, piszą konwertery formatów plików, implementują efekty specjalne, tworzą shadery i — ogólnie rzecz biorąc — robią wszystko to, z czym nie radzi sobie artysta grafik. Umiejętności dyrektora technicznego są wysoko cenione i dla wielu producentów dobry dyrektor techniczny jest wart worka złota.

Jeśli dyrektor techniczny zarabia programowaniem na życie, jego podstawowym narzędziem pracy jest edytor tekstu, chociaż dostępne są już ciekawe wizualne narzędzia do tworzenia shaderów i efektów specjalnych. Jednym z nich jest ShaderFusion, nowe wizualne narzędzie przeznaczone do użytku z silnikiem gier **Unity**. Przy jego użyciu można tworzyć shadery, definiując przepływy danych między wyjściami jednego obiektu (np. czas lub położenie) a wejściami innego (np. kolor i refrakcja). Interfejs tego programu pokazano na rysunku 8.4.



Rysunek 8.4. *ShaderFusion* (<http://www.shaderfusionblog.com/>) — wizualny edytor shaderów dla silnika Unity3D

Narzędzia do tworzenia trójwymiarowych modeli i animacji

W podrozdziale tym przeczytasz o wielu narzędziach, których artyści używają do tworzenia grafiki trójwymiarowej. Znajdują się wśród nich klasyczne programy komputerowe przeznaczone dla użytkowników o dowolnym poziomie umiejętności i wiedzy. Pojawił się też nowy i obiecujący rodzaj narzędzi działających jako oparte na HTML5 usługi w chmurze. Niektóre z nich są darmowe, a za korzystanie z innych trzeba uiszczać miesięczną opłatę. Ponadto artyści mogą też opierać się na pracach innych artystów pobranych z różnych stron internetowych.

Klasyczne programy komputerowe

Większość grafiki trójwymiarowej powstaje w specjalnych programach zwanych **narzędziami do tworzenia cyfrowej treści** (ang. *digital content creation tools* — DCC). Wywodzą się one z branży filmowej i inżynierii, a obecnie są powszechnie wykorzystywane w architekturze, tworzeniu gier itp. Są to jakby odpowiedniki Photoshopa przeznaczone dla grafiki trójwymiarowej. Wszystkie stanowią to samo ognisko w łańcuchu procesu tworzenia — są źródłem oryginalnych grafik, które później trzeba przekonwertować, zoptymalizować i wstawić na stronę internetową.

Trójwymiarowe narzędzia DCC najczęściej występują jako klasyczne programy komputerowe, „pudełkowe”, jak zwykło się mawiać, chociaż obecnie najczęściej nie kupuje się ich z półki

sklepowej, tylko pobiera ze strony internetowej producenta. Aby z nich korzystać, trzeba posiadać specjalistyczną wiedzę oraz opanować obsługę skomplikowanego interfejsu użytkownika. Na szczęście, coraz więcej grafików cyfrowych uczy się posługiwać się tym narzędziami już na etapie edukacji i na kursach dla profesjonalistów. Podobnie jak specjaliści Photoshopa, doświadczeni artyści tworzący grafikę trójwymiarową wkrótce będą nieodzowni w projektach internetowych.

Ceny narzędzi do tworzenia cyfrowej treści są zróżnicowane, a ich rozpiętość sięga od całkiem darmowych programów, np. Blender, do produktów firmy Autodesk 3ds Max i Maya, które kosztują po kilka tysięcy dolarów. Większość tych programów zawiera kompletny zestaw funkcji do modelowania, teksturowania i animowania, chociaż niektóre są wyspecjalizowane w jednym kierunku. Większość narzędzi ma wbudowane moduły do importowania i eksportowania standardowych formatów plików, o których będzie mowa dalej w tym rozdziale. Ponadto niektóre można rozszerzać na różne sposoby, np. przy użyciu macierzystego (napisaneego w języku C++) SDK albo wysokopoziomowego języka skryptowego służącego do pisania wtyczek wzbogacających interfejs użytkownika, dodających własne funkcje renderujące, umożliwiających eksportowanie danych do nowych formatów plików itd.

Poniżej znajduje się przegląd najczęściej używanych programów do modelowania i animacji, które można spotkać podczas pracy nad projektami związanymi z biblioteką WebGL. Dalej w tym rozdziale dowiesz się, jak niektóre z nich włączyć do procesu tworzenia treści WebGL.

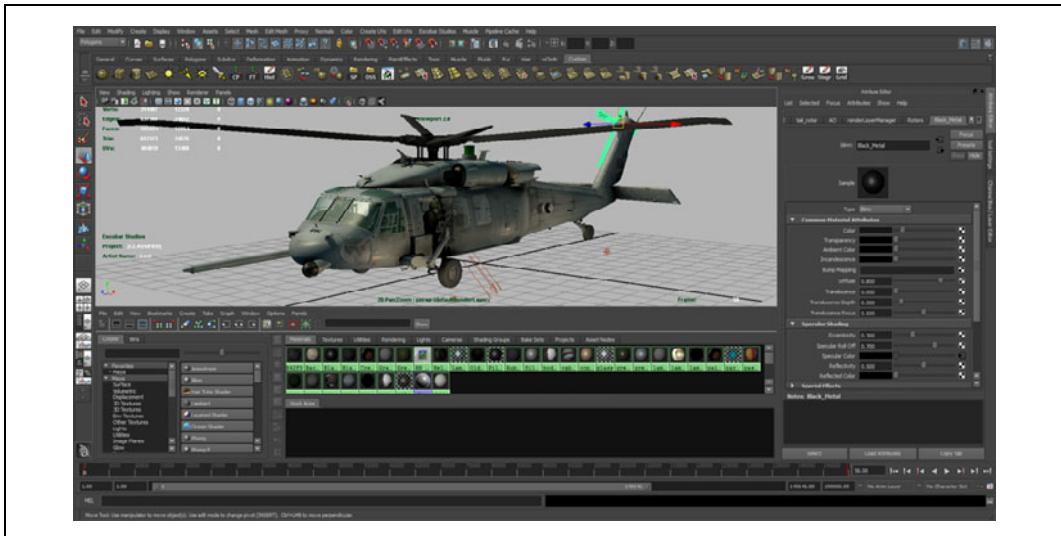
Autodesk 3ds Max, Maya oraz MotionBuilder

Firma Autodesk z San Rafael w Kalifornii jest producentem trzech najpopularniejszych na świecie programów do tworzenia trójwymiarowych modeli i animacji, są to 3ds Max, Maya oraz MotionBuilder. Ostatni z wymienionych programów jest wyspecjalizowany w animowaniu postaci, natomiast 3ds i Maya to pełne pakiety do projektowania grafiki trójwymiarowej. Pod względem funkcjonalności aplikacje te są podobne, przez co początkujący projektant może mieć problem z wyborem. Bardziej doświadczeni użytkownicy twierdzą, że wybór zależy od osobistych preferencji dotyczących sposobu pracy, gustu itd. Jedną z największych różnic jest to, że Maya działa w systemach Windows i Max OS, a 3ds Max tylko w systemie Windows. Wszystkie trzy programy zapisują pliki we wspólnym formacie FBX firmy Autodesk.



Po co firma Autodesk tworzy tak podobne do siebie programy? Około dziesięć lat temu firma zaszała i wykupiła produkty konkurencji — programy Maya firmy Alias Systems Corporation i MotionBuilder firmy Kaydara. Podczas gdy MotionBuilder to narzędzie przeznaczone do animowania postaci, dwa pozostałe produkty Autodesk są bardzo podobne pod względem funkcjonalności. W portalu Tom's Hardware (<http://www.tomshardware.com/forum/247220-49-maya>) znajduje się wartościowy artykuł zawierający porównanie programów 3ds Max i Maya.

Programy firmy Autodesk mają skomplikowane interfejsy użytkownika zawierające mnóstwo przycisków, widoków, kart właściwości i wyskakujących okienek. Są to kompletne „kombajny” do tworzenia grafiki trójwymiarowej. Początkowo interfejs zawiera cztery okna widoku, co widać na przedstawionym na rysunku 8.1 zrzucie ekranu z programu 3ds Max. Można je złożyć w pojedynczy widok sceny, co można zobaczyć na przedstawionym na rysunku 8.5 zrzucie ekranu z programu Maya. Oba te programy zawierają edytor materiałów, paski narzędzi do tworzenia obiektów (np. sfer, szescianów itp.), narzędzia do rysowania i edytowania siatek o dowolnym kształcie, narzędzia do zarządzania osią czasu, wtyczki do renderowania, edytory shaderów itd. itd.



Rysunek 8.5. Program Maya firmy Autodesk to kompletny pakiet do tworzenia trójwymiarowych modeli i animacji (prawa autorskie do obrazu posiada firma Autodesk, obraz pobrano z Wikipedii — <http://bit.ly/1hndkGA>)

Produkty firmy Autodesk są przeznaczone dla profesjonalistów, o czym świadczą także ich ceny — około 15 000 złotych za jeden program. Firma oferuje też roczne subskrypcje, a także edycje edukacyjne dla studentów.

Blender

Blender (<http://www.blender.org/>) to darmowy, otwarty i wieloplatformowy pakiet narzędzi do tworzenia grafiki trójwymiarowej. Ten działający we wszystkich najważniejszych systemach operacyjnych program jest dostępny na licencji GNU General Public License (GPL). Jego twórcą jest holenderski programista Ton Roosendaal, a utrzymaniem zajmuje się holenderska organizacja non profit Blender Foundation. Program ten jest niezwykle popularny, a fundacja szacuje, że ma około dwóch milionów użytkowników. Używają go zarówno artyści, jak i inżynierowie, studenci i profesjonalisci.

Podobnie jak 3ds Max i Maya, Blender ma skomplikowany interfejs użytkownika, zawierający wiele różnych widoków i pasków narzędzi z przyciskami. I podobnie, jego nauka jest trudna i długotrwała. Zatem, mimo że cena jest „odpowiednia”, program ten i tak przeznaczony jest tylko dla wytrwałych. Jednak Blender to atrakcyjny wybór dla programistów sieciowych i to z kilku powodów.

- Jest darmowy.
- Jest otwarty.
- Można go rozszerzać za pomocą języka Python.
- Umożliwia import i eksport wielu formatów plików, włącznie z 3ds Max, OBJ, COLLADA oraz FBX. Utworzono nawet eksporter z formatu Blenera do formatu JSON biblioteki Three.js (będzie o nim mowa dalej w tym rozdziale).

Trimble SketchUp

Produktem ze średniej półki w branży programów do tworzenia grafiki trójwymiarowej jest SketchUp (oficjalnie Trimble SketchUp). To łatwy w użyciu program do modelowania trójwymiarowego, z którego korzystają architekci, inżynierowie i w mniejszym stopniu twórcy gier.

Historia programu SketchUp jest bardzo ciekawa. Powstał w firmie @Last Software w 1999 r., ale wkrótce przyciągnął uwagę zespołu Google Earth, bo firma @Last Software opracowała wtyczkę dla ich systemu. W efekcie w 2006 r. firma Google kupiła firmę @Last Software. Przez wiele lat program SketchUp reklamowano jako narzędzie do tworzenia przez użytkowników trójwymiarowych modeli budynków i innych ważnych elementów architektury w Google Earth. Uzupełnieniem programu SketchUp była usługa 3D Warehouse, czyli internetowe repozytorium, w którym ochotnicy mogli publikować swoje trójwymiarowe modele. W 2012 r. firma Google zrezygnowała z udziału w tej branży i sprzedała program SketchUp produkującej systemy GPS firmie Trimble Navigation z Kalifornii. Firma ta nadal dystrybuuje program SketchUp i utrzymuje usługę 3D Warehouse, ale nie ma ona już nic wspólnego z Google Earth.

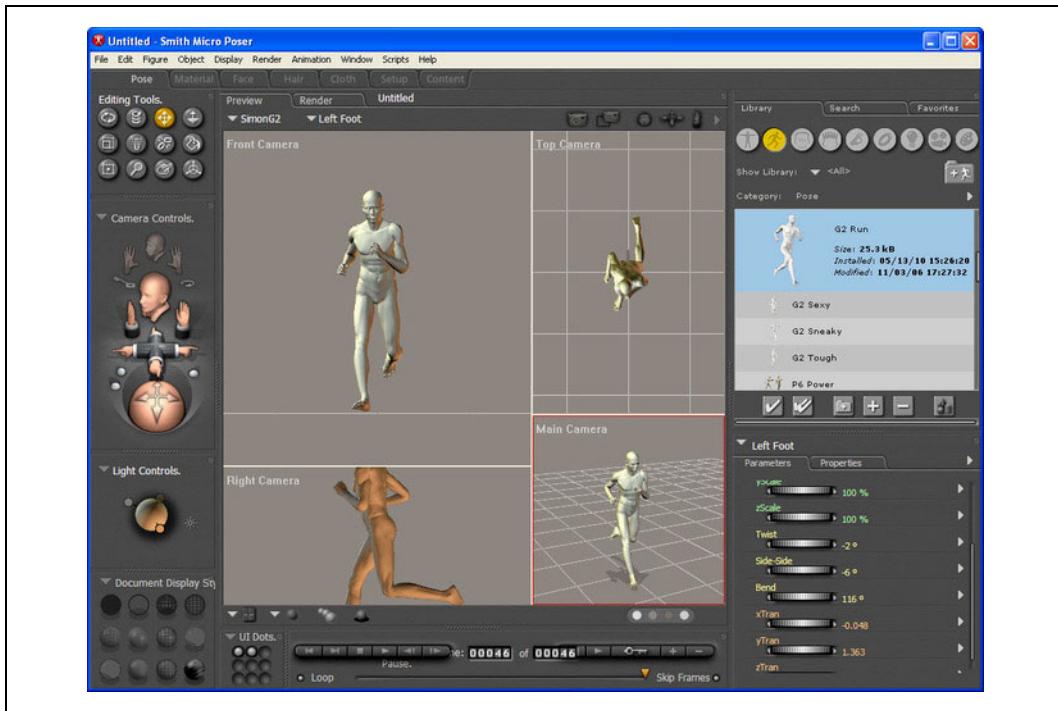
Program SketchUp działa na wszystkich platformach i ma przystępna cenę wynoszącą około 500 dolarów za wersję Pro. Dostępna jest też darmowa wersja dla sporadycznych użytkowników. Program SketchUp słynie z łatwości obsługi oraz metody modelowania opartej na rysowaniu linii, która doskonale sprawdza się w architekturze i inżynierii. Zawiera też doskonały eksporter do formatu COLLADA (podrozdział „COLLADA — format do wymiany zasobów cyfrowych”), a więc może być wykorzystywany przez programistów używających WebGL. Program SketchUp można pobrać z oficjalnej strony pod adresem <http://www.sketchup.com/>.

Poser

Program Poser firmy Smith Micro to też narzędzie ze średniej półki do trójwymiarowej animacji postaci. Podobnie jak SketchUp, ma atrakcyjną cenę i jest przeznaczone dla użytkowników, którzy tworzą coś od czasu do czasu. Program ma intuicyjny interfejs użytkownika do budowania i animowania postaci oraz zawiera bogatą bibliotekę gotowych, w pełni oteksturowanych postaci ludzkich i zwierząt, jak również tel scen, rekwizytów, pojazdów, kamer oraz konfiguracji światel. Za pomocą tego programu można tworzyć realistyczne fotografie, nieruchome obrazy i animacje. Na rysunku 8.6 przedstawiono interfejs użytkownika tego programu.

Dodatkowo program Poser wyróżnia się spośród opisanych do tej pory programów tym, że tworzący go programiści są również silnie zaangażowani w opracowywanie formatu plików COLLADA od samego początku jego istnienia oraz są aktywni w grupie Khronos, w której zajmują się pracami nad nowym standardem, glTF, o którym będzie mowa dalej w tym rozdziale. Pracownicy firmy Poser uważają, że standardowe formaty są najlepszą drogą do tego, aby rozpowszechnić treść trójwymiarową, zwłaszcza w sieci. Starszy dyrektor firmy Smith Micro Uli Klumpp w następujący sposób wyraził się na temat zastosowania programu Poser w połączeniu z WebGL:

Aplikacje obsługujące WebGL niczym nie różnią się od innych mediów; często trzeba przedstawiać postaci ludzkie (albo całkiem nieludzkie). Projektanci sieciowi używają programu Poser do tworzenia ilustracji od lat 90. ubiegłego wieku. Obecnie mają w końcu do dyspozycji trzy wymiary w każdym miejscu, a bogactwo treści programu Poser czeka, aż zacząć z niego czerpać.



Rysunek 8.6. Interfejs użytkownika programu Poser (<http://poser.smithmicro.com/>); obraz opublikowany dzięki uprzejmości firmy Smith Micro Software, Inc.

Przeglądarkowe środowiska zintegrowane

Pojawienie się technologii HTML5 i tanich usług chmurowych umożliwiło powstanie nowego typu narzędzi do tworzenia grafiki trójwymiarowej, czyli przeglądarkowych środowisk zintegrowanych. Wprawdzie do modelowania i animacji nadal używa się klasycznych narzędzi, takich jak opisane wcześniej, ale do tworzenia układów scen, programowania interakcji i publikowania w sieci coraz częściej używane są aplikacje przeglądarkowe.

Przeglądarkowe środowiska zintegrowane mają kilka ważnych cech, których brakuje ich klasycznym odpowiednikom. Po pierwsze oczywiście, nie trzeba ich pobierać na komputer. Po drugie, są tworzone przy użyciu WebGL, więc podgląd w nich jest generowany z wykorzystaniem tej samej technologii, w której opracowywana jest dana aplikacja. Ceny takich usług są atrakcyjne i często początkowo darmowe, a opłaty pobierane są dopiero wówczas, gdy użytkownik wykona coś w celach komercyjnych, np. utworzy projekt zespołowy albo wyczerpie darmową przestrzeń na pliki. W niektórych narzędziach obowiązują ścisłe zasady dotyczące zasad używania utworzonej przy ich użyciu treści, niektóre wymagają wykupienia hostingu lub publikowania dzieł poprzez ich serwery. Jest to nowa i dynamiczna dziedzina, więc można się spodziewać powstawania coraz większej liczby modeli biznesowych opartych na internecie.

Verold

Verold Studio to platforma do publikowania interaktywnej treści trójwymiarowej firmy Verold Inc. z Toronto. Jest to rozszerzalny bezwtyczkowo system zaopatrzony w proste API JavaScript, dzięki któremu hobbisi, studenci, nauczyciele i specjalisci od komunikacji wizualnej oraz marketingu mogą wstawać animowaną treść trójwymiarową na swoje strony internetowe.

Najczęściej praca z programem Verold wygląda tak, że artysta wysyła potrzebne zasoby (modele trójwymiarowe, animacje, tekstury) do projektu. Za pomocą narzędzi do współpracy można komentować te zasoby, a przy użyciu narzędzi do edycji można konfigurować materiały i shadery oraz rozmieszczać elementy na scenach lub poziomach. Po zakończeniu tych prac projektant stron internetowych eksportuje kod, który następnie umieszcza na docelowej stronie internetowej. Artysta może pracować razem w jednym pomieszczeniu z programistą albo w całkiem innym miejscu. Ten sposób pracy jest podobny do sytuacji, gdy zamawia się grafiki od innej firmy, zamiast tworzyć je samodzielnie. Na rysunku 8.7 przedstawiony jest interfejs użytkownika programu Verold. Warto zwrócić uwagę na przejrzysty układ, który wyraźnie różni się od przeładowanych oknami i okienkami interfejsów klasycznych programów.



Rysunek 8.7. Verold Studio (<http://www.verold.com>)

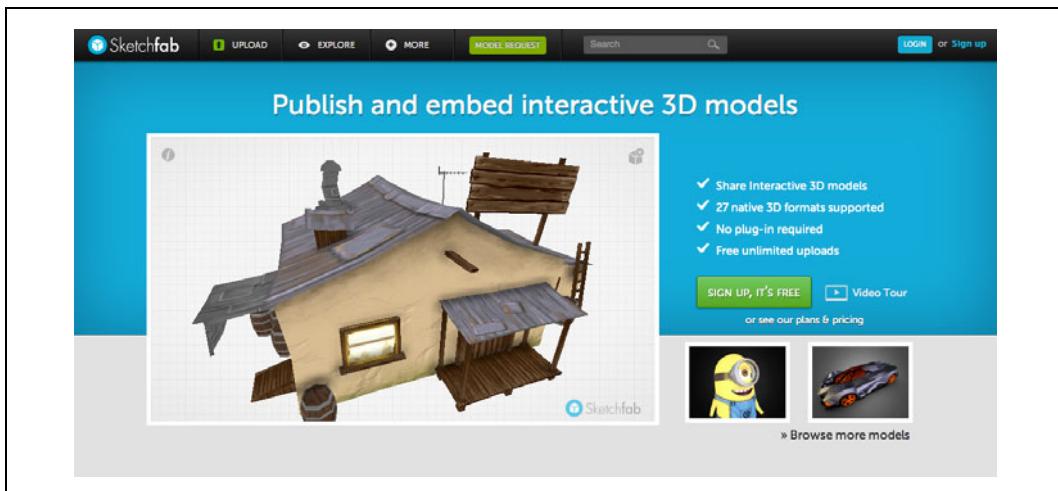
Funkcje programu Verold umożliwiające współpracę w czasie rzeczywistym, publikowanie dzieł w internecie oraz udostępnianie treści pozwalają na realizację nowatorskich projektów. Oto słowa założyciela i dyrektora technicznego firmy Rossa McKegneya:

Świetnym studium przypadku dotyczącym używania Verold Studio jest Swappz Interactive. Firma ta tworzy figurki Żółwi Ninja, Smerfów, postaci z Power Rangers i innych. Zabawki te wyróżniają się tym, że dziecko może je zeskanować i przenieść do wcześniej pobranej na urządzenie przenośne gry. Przy tworzeniu tych gier wykorzystano technologię Verold. Użyto jej do kontaktowania pracujących zdalnie artystów z lokalnymi animatorami, do przedstawiania postępów prac jednostce macierzystej i w celu uzyskania zatwierdzenia grafik przez Nickelodeon, a po zakończeniu prac zgromadzone zasoby wykorzystano na stronach promocyjnych gier.

Sketchfab

Kolejnym typem internetowych narzędzi trójwymiarowych są usługi do wysyłania i udostępniania dzieł. Artysta może wysyłać na serwer swoje dzieła w kilku formatach, aby następnie je przeglądać i udostępniać innym przy użyciu WebGL. Najbardziej rozwiniętą tego rodzaju usługą jest Sketchfab (<http://sketchfab.com/>). Jest to wspólne przedsięwzięcie mieszkającego w Paryżu Cédrica Pinsona oraz Albana Denoyela i Pierre'a-Antoine'a Passeta. Sketchfab to usługa sieciowa umożliwiająca publikowanie i udostępnianie w internecie interaktywnych modeli trójwymiarowych w czasie rzeczywistym i bez użycia jakichkolwiek wtyczek. Wystarczy kilka kliknięć, aby model znalazł się na stronie internetowej. Potem można pobrać specjalny kod HTML służący do osadzania tej treści w wybranym miejscu.

Sketchfab obsługuje kilka formatów grafiki trójwymiarowej oraz większość standardowych shaderów: map normalnych, odbiciowych, nierównościowych, rozmyciowych itd. Ponadto narzędzie zawiera edytor materiałów, przy użyciu którego można dostosować shadery i renderingi w oknie przeglądarki. Dodatkowo producent dostarczył narzędzi eksportowe dla najważniejszych klasycznych programów do tworzenia grafiki trójwymiarowej, dzięki czemu modele utworzone w tych programach, np. Maya, można wysłać na serwer bezpośrednio z tych programów. Na rysunku 8.8 można zobaczyć stronę główną usługi Sketchfab. Zajmująca większość ekranu grafika to wygenerowany przy użyciu WebGL widok na żywo jednego z modeli dostępnych w galerii Sketchfab.

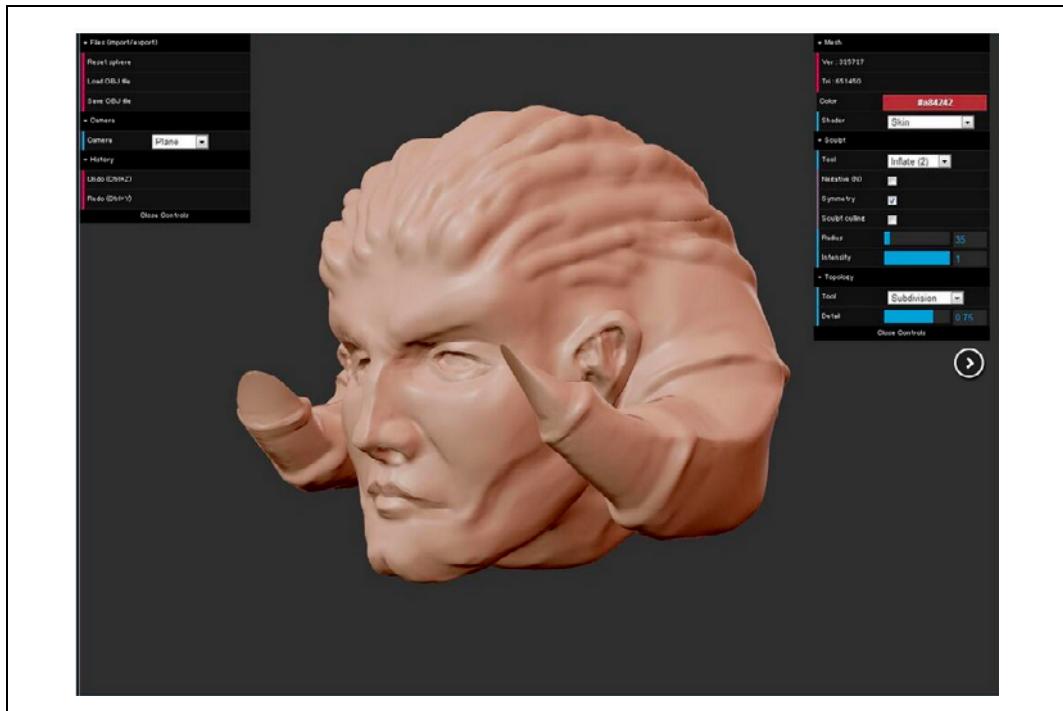


Rysunek 8.8. Strona internetowa Sketchfab (<http://sketchfab.com/>) zawiera narzędzia umożliwiające twórcom wysyłanie i udostępnianie trójwymiarowych modeli

SculptGL

Jeszcze kilka lat temu ograniczenia dotyczące renderowania trójwymiarowego i interfejsów użytkownika uniemożliwiały stworzenie przeglądarkowego narzędzia do modelowania. Obecnie dzięki HTML5 i WebGL pomysł ten nie wydaje się już tak trudny w realizacji. SculptGL to łatwe w użyciu sieciowe narzędzie do tworzenia trójwymiarowych modeli rzeźb autorstwa Stephane'a Giniera. Program ten jest darmowy i otwarty (dostępny w serwisie GitHub pod

adresem <https://github.com/stephomi/sculptgl>). Zawiera funkcje eksportu do różnych formatów oraz ma możliwość bezpośredniego eksportu do Verold i Sketchfab. Na rysunku 8.9 pokazano zrzut ekranu z przykładową rzeźbą.



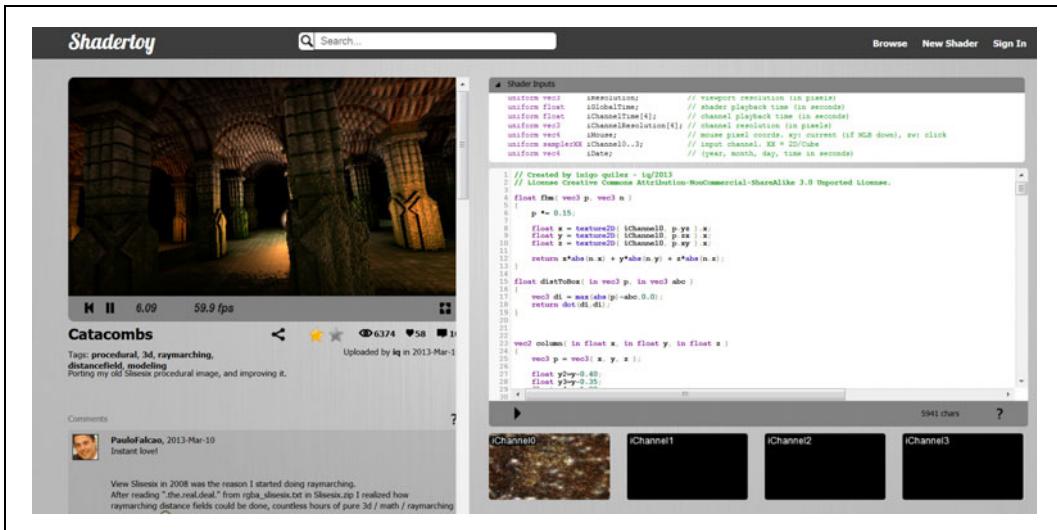
Rysunek 8.9. SculptGL (<http://stepheneginier.com/sculptgl/>) — otwarte przeglądarkowe narzędzie do modelowania trójwymiarowego

Shadertoy

Rosnąca popularność internetowych programów do „bawienia się kodem”, takich jak JSFiddle (<http://jsfiddle.net>), przy użyciu których programiści mogą eksperymentować z kodem źródłowym w oknie przeglądarki, sprawiła, że tylko kwestią czasu było pojawienie się analogicznych narzędzi dla WebGL. Jednym z nich jest Shadertoy (<https://www.shadertoy.com/>), czyli przeglądarkowe narzędzie do pisania i testowania shaderów w języku GLSL. Każdy, kto napisze i przetestuje jakiś shader, może go udostępnić innym użytkownikom portalu. Naśladowanie prac innych osób to bardzo skuteczna metoda nauki programowania. Skończony shader można udostępnić innym do wglądu albo skopiować jego kod GLSL do swojej aplikacji. Na rysunku 8.10 widać interfejs aplikacji Shadertoy zawierający podgląd na żywo siatki, okno z kodem źródłowym oraz interaktywne ikony służące do wyboru źródeł wejściowych shadera.

Repozytoria 3D i darmowe zdjęcia

Nie każdy ma talent do tworzenia grafiki trójwymiarowej, a posiadane środki finansowe i terminy nie zawsze pozwalają na wynajecie odpowiedniej osoby. Wówczas z pomocą mogą przyjść internetowe zasoby treści trójwymiarowej. Cenne są bardzo zróżnicowane. Niektóre dzieła są



Rysunek 8.10. Catacombs, czyli utworzony w Shadertoy eksperyment zawierający tekstury proceduralne (<https://www.shadertoy.com/view/lzf3zr>)

darmowe, a za inne trzeba zapłacić nawet kilkaset albo kilka tysięcy dolarów za model lub paczkę. Także jakość bywa różna. Niektórzy twórcy nie nakładają żadnych ograniczeń dotyczących sposobu użycia ich dzieł, a inni tak. Dlatego przed zakupem zawsze należy dokładnie przeczytać warunki, zwłaszcza gdy planuje się używać dzieła w aplikacji dostępnej w internecie.

Do utworzenia trójwymiarowej treści dla tej książki użyłem modeli z różnych miejsc w sieci. Oto one.

3D Warehouse firmy Trimble (<http://sketchup.google.com/3dwarehouse/>)

Serwis 3D Warehouse początkowo był własnością firmy Google i miał służyć jako platforma dla amatorów i hobbistów do publikowania wiernych modeli budynków i innych elementów architektonicznych utworzonych w programie SketchUp oraz do ich geolokalizacji przy użyciu usługi Google Earth. Gdy firma Trimble odkupiła od Google program SketchUp, usługa 3D Warehouse przestała być powiązana z Google Earth, ale nadal jest bogatym źródłem ładnych modeli budynków i innych obiektów.

Turbosquid (<http://www.turbosquid.com/>)

Założony w 2000 r. serwis Turbosquid to jedno z najpopularniejszych źródeł modeli do użytku w animacjach, grach i projektach architektonicznych. Wiele modeli składa się z małej lub średniej liczby wielokątów, dzięki czemu nadają się do użytku w sieci i aplikacjach czasu rzeczywistego.

Renderosity (<http://renderosity.com/>)

Renderosity to założona w 1998 r. bardzo zróżnicowana społeczność profesjonalistów zajmujących się grafiką dwu- i trójwymiarową. W serwisie znajduje się obszerny katalog modeli i tekstur. Wiele z nich składa się z dużej liczby wielokątów przydatnych do użytku w renderowanych z góry nieruchomych obrazach, na modelach złożonych z mniejszej liczby wielokątów.

3DRT.com (<http://3drt.com>)

3DRT to internetowy sklep z wysokiej jakości treścią trójwymiarową do użytku w grach i internecie. Organizacja witryny umożliwia szybkie znajdowanie postaci, rekwizytów i widoków. Modele nie są tanie, ale odznaczają się wysoką jakością.

Trójwymiarowe formaty plików

Istnieje tak wiele trójwymiarowych formatów plików, że wymienienie ich wszystkich jest niemożliwe. Niektóre z nich służą tylko do przechowywania danych jednego konkretnego programu, inne natomiast mają umożliwiać wymianę danych między różnymi programami. Niektóre formaty są zastrzeżone, tzn. znajdują się pod pełną kontrolą jakiejś firmy, podczas gdy inne są otwartymi standardami zdefiniowanymi przez jakąś grupę branżową. Niektóre formaty są w pełni tekstowe, a więc mogą być odczytywane także przez człowieka, a inne mają reprezentację binarną, aby oszczędzić miejsce.

Trójwymiarowe formaty plików można podzielić na trzy kategorie: formaty modelowe służące do reprezentowania pojedynczych obiektów, formaty animacyjne do animowania klatek kluczowych i postaci oraz w pełni funkcjonalne formaty, w których można zapisywać całe sceny zawierające wiele modeli, hierarchię przekształceń, kamery, światła i animacje. Przyjrzymy się każdemu z tych trzech rodzajów i zwrócimy szczególną uwagę na te, które najlepiej nadają się do użytku w aplikacjach sieciowych.

Formaty modelowe

Trójwymiarowe formaty do zapisywania pojedynczych modeli są powszechnie wykorzystywane do przenoszenia dzieł między różnymi programami. Przykładowo większość programów obsługuje format OBJ (następny podrozdział). Ze względu na prostą składnię i niewielką liczbę funkcji implementacja obsługi tych formatów jest bardzo łatwa, a co za tym idzie, są bardzo popularne. Jednak ceną za tę prostotę są poważne ograniczenia funkcjonalności.

Format Wavefront OBJ

Będący produktem firmy Wavefront Technologies, format plików OBJ jest jednym z najstarszych w branży i najlepiej obsługiwanych formatów do przechowywania pojedynczych modeli. Jest niezwykle prosty, ponieważ obsługuje tylko geometrię (z wierzchołkami, normalnymi i współrzędnymi teksturowymi). Ponadto firma Wavefront wprowadziła format uzupełniający MTL (ang. *Material Template Library*) służący do nakładania materiałów na geometrię.

Na listingu 8.1 pokazany jest kod pliku OBJ z klasycznego modelu krzesła kulowego, który później będziemy wczytywać za pomocą biblioteki Three.js (jest pokazany na rysunku 8.12). Plik, o którym mowa, to *models/ball_chair/ball_chair.obj* w paczce z przykładowymi plikami. Przyjrzymy się jego składni. Znak # oznacza komentarz. Ogólnie plik składa się z szeregu deklaracji. Pierwsza z nich zawiera odwołanie do biblioteki materiałów znajdującej się w powiązanym pliku MTL. Dalej znajdują się definicje kilku obiektów geometrycznych. Na omawianym listingu pokazano część definicji obiektu shell reprezentującego zewnętrzną powłokę fotela. Definicja ta składa się z danych określających pozycję wierzchołków i normalne oraz współrzędnych teksturowych. Dalej znajdują się dane boków. Definicja każdego wierzchołka boku składa się z trzech elementów v, vt, vn, gdzie v to indeks poprzedniej pozycji wierzchołka, vt to indeks współrzędnej teksturowej, a vn to indeks normalnej wierzchołka.

Listing 8.1. Model w formacie Wavefront OBJ

```
# 3ds Max Wavefront OBJ Exporter v0.97b - (c)2007 guruware
# File Created: 20.08.2013 13:29:52

mtllib ball_chair.mtl
#
# object shell
#
v -15.693047 49.273174 -15.297686
v -8.895294 50.974277 -18.244076
v -0.243294 51.662109 -19.435429
... dane pozycji kolejnych wierzchołków
vn -0.537169 0.350554 -0.767177
vn -0.462792 0.358374 -0.810797
vn -0.480322 0.274014 -0.833191
... normalne kolejnych wierzchołków
vt 0.368635 0.102796 0.000000
vt 0.348531 0.101201 0.000000
vt 0.349342 0.122852 0.000000
... kolejne współrzędne tekstury
g shell
usemtl shell
s 1
f 313/1/1 600/2/2 58/3/3 597/4/4
f 598/5/5 313/1/1 597/4/4 109/6/6
f 313/1/1 598/5/5 1/7/7 599/8/8
f 600/2/2 313/1/1 599/8/8 106/9/9
f 314/10/10 603/11/11 58/3/3 600/2/2
... kolejne definicje boków
```

Twarzyszące definicje materiałów znajdują się w pliku MTL *models/ball_chair/ball_chair.mtl*. Jak widać na listingu 8.2, jego składnia również jest prosta. Materiał deklaruje się za pomocą instrukcji newmtl, która zawiera zestaw parametrów dotyczących cieniowania Phonga obiektu: kolory odbicia i współczynniki (słowa kluczowe Ks, Ns i Ni), kolor rozmycia (Kd), kolor otoczenia (Ka), kolor emitowany (Ke) oraz tekstury (map_Ka i map_Kd). Model tekstur w formacie MTL przeszedł szereg zmian, w wyniku których dodano mapy nierówności, mapy przesunięć, mapy środowiskowe i inne typy tekstur. W tym przykładzie zdefiniowane są tylko tekstury rozproszenia i otoczenia dla materiału shell.

Listing 8.2. Definicje materiałów dla formatu Wavefront OBJ

```
newmtl shell
Ns 77.000000
Ni 1.500000
Tf 1.000000 1.000000 1.000000
illum 2
Ka 0.000000 0.000000 0.000000
Kd 0.588000 0.588000 0.588000
Ks 0.720000 0.720000 0.720000
Ke 0.000000 0.000000 0.000000
map_Ka maps\shell_color.jpg
map_Kd maps\shell_color.jpg
...
...
```

Format STL

Kolejnym prostym tekstowym formatem plików do przechowywania pojedynczych modeli jest opracowany przez firmę 3D Systems STL (ang. *StereoLithography*). Format ten służy do szybkiego tworzenia prototypów, w produkcji oraz druku trójwymiarowym. Pliki w tym formacie są nawet jeszcze prostsze niż w OBJ, ponieważ obsługuje on wyłącznie geometrię, a więc żadnych normalnych, współrzędnych teksturowych czy materiałów. Na listingu 8.3 widać fragment jednego z przykładowych plików STL z biblioteki Three.js (*examples/models/stl/pr2_head_pan.stl*). Aby zobaczyć efekt jego użycia, otwórz plik *examples/webgl_loader_stl.html*. STL to doskonały format trójwymiarowy do budowy potencjalnych internetowych aplikacji do drukowania trójwymiarowego przy użyciu WebGL, ponieważ pliki w tym formacie można wysyłać bezpośrednio do drukarki. Ponadto łatwo się go wczytuje i renderuje.

Listing 8.3. Format plików STL

```
solid MYSOLID created by IVCON, original data in binary/pr2_head_pan.stl
facet normal -0.761249 0.041314 -0.647143
    outer loop
        vertex -0.075633 -0.095256 -0.057711
        vertex -0.078756 -0.079398 -0.053025
        vertex -0.074338 -0.088143 -0.058780
    endloop
    endfacet
...
endsolid MYSOLID
```



Format STL jest tak prosty i popularny, że w serwisie GitHub dodano nawet narzędzie do jego przeglądania (<https://github.com/blog/1465-stl-file-viewing>). Przeglądarkę tę zbudowano przy użyciu WebGL i dobrze już znanej biblioteki Three.js.

Szczegółowe informacje techniczne dotyczące formatu STL można znaleźć w Wikipedii pod adresem http://en.wikipedia.org/wiki/STL_%28file_format%29.

Formaty animacyjne

Formaty opisane w poprzednim podrozdziale służą do przechowywania danych tylko pojedynczych modeli. Jednak w aplikacjach trójwymiarowych wiele elementów rusza się po scenie (tzn. są animowane). Do reprezentowania takich ruchomych modeli powstało kilka specjalnych formatów plików. Niektóre z nich są tekstowe — a więc nadające się do użytku w internecie — np. MD2, MD5 oraz BVH.

Formaty animacyjne firmy id Software: MD2 i MD5

Od czasu do czasu w sieci można spotkać dwa formaty animacyjne opracowane przez firmę id Software dla gier *Doom* i *Quake*. Są to formaty MD2 i jego następca MD5 służące do przechowywania animacji postaci. Ogólnie rzecz biorąc, ich właścicielem jest firma id Software, ale specyfikacje zostały opublikowane już dawno i od tamtej pory powstało wiele narzędzi do ich importowania.

Utworzony dla gry *Quake II* format MD2 jest binarny i obsługuje tylko animacje wierzchołkowe realizowane techniką morfingu. Format MD5 (którego nie należy mylić z algorytmem szyfrowania danych Message Digest) został utworzony dla gry *Quake III*. Wprowadzono w nim animację szkieletową oraz zmieniono go na format tekstowy, a więc czytelny dla człowieka.

W internecie można znaleźć dokumentację zarówno formatu MD2 (<http://tfc.duke.free.fr/coding/md2-specs-en.html>), jak i MD5 (<http://tfc.duke.free.fr/coding/md5-specs-en.html>).

Aby używać tych formatów w aplikacjach WebGL, można napisać program wczytujący je bezpośrednio albo, jeśli używa się biblioteki Three.js, można użyć konwertera. Gdy przekonwertuje się plik MD2 na format JSON, to wygląda tak, jak w przykładzie przedstawionym na rysunku 5.11, w rozdziale 5. Dla odświeżenia pamięci otwórz plik z przykładów do biblioteki Three.js *examples/webgl_morphtargets_md2_control.htm* i zajrzyj do jego kodu źródłowego. Wczytywanie i interpretacja kodu MD2 wymagają sporo pracy.

Biblioteka Three.js nie zawiera wśród przykładów programu do wczytywania plików w formacie MD5, ale w internecie dostępny jest fantastyczny konwerter z MD5 na JSON tej biblioteki. Jego twórcą jest Klas (*OutsideOfSociety*) ze szwedzkiej agencji North Kingdom (twórcy produkcji *Znajdź drogę do OZ*). Na ekranie powinien pojawić się dość szczegółowy model potwora, obok którego znajdują się przyciski do włączania animacji różnych ruchów.

Aby przepuścić własne pliki MD5 przez konwerter, można wejść na stronę http://oos.moxiecode.com/js_webgl/md5_converter/ i przeciągnąć pliki do okna przeglądarki. W efekcie program zwróci kod JSON.

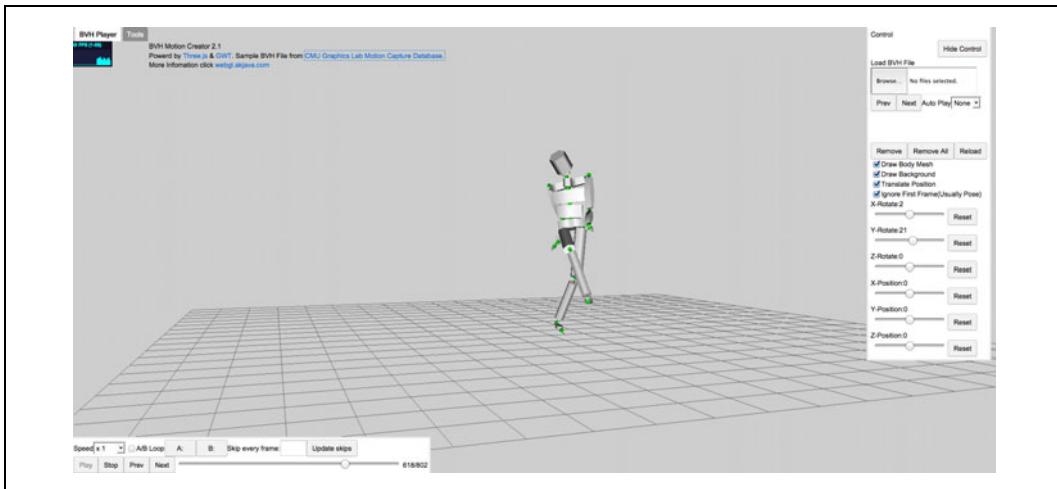
BVH: format danych z przechwytywania ruchu

Przechwytywanie ruchu (ang. *motion capture*), czyli technika polegająca na nagrywaniu ruchu obiektów, jest powszechnie wykorzystywane do tworzenia animacji, zwłaszcza ludzi. Technikę stosuje się w filmie, wojsku oraz aplikacjach sportowych. Jest obsługiwana przez otwarte formaty, do których zalicza się również Biovision Hierarchical Data, czyli BVH. Format BVH jest produktem zajmującej się przechwytywaniem ruchu firmy Biovision. Służy do reprezentowania ruchów w animacjach ludzkich postaci. Jest to bardzo popularny format tekstowy używany do importu i eksportu danych przez wiele programów.

Programista Aki Miyazaki zbudował eksperymentalną aplikację importującą dane w formacie BVH do aplikacji WebGL. Na rysunku 8.11 widać jego napisane przy użyciu biblioteki Three.js internetowe narzędzie o nazwie BVH Motion Creator, służące do podglądania plików BVH. Do programu można wysyłać plik w formacie BVH, aby na ekranie obejrzeć podgląd zapisanej w nim animacji postaci.

Formaty do zapisywania całych scen

Istnieją też formaty plików do przechowywania całych scen trójwymiarowych, zawierających wiele obiektów, hierarchie przekształceń, światła, kamery i animacje — w zasadzie wszystko, co może utworzyć artysta w programie typu 3ds Max, Maya lub Blender. Ogólnie rzecz biorąc, przechowywanie takich danych to bardzo trudne zadanie, przez co w powszechnym użyciu jest bardzo niewielka liczba formatów. Jednak wkrótce może się to zmienić, gdy w WebGL zostaną wprowadzone nowe wymagania dotyczące wielokrotnego wykorzystania treści i współpracy między aplikacjami. W tym punkcie omówię kilka potencjalnych formatów do użytku z WebGL.



Rysunek 8.11. Narzędzie BVH Motion Creator (<http://www.akjava.com/demo/bvhplayer>) — program do podglądzania zawartości plików BVH

VRML i X3D

Język Virtual Reality Markup Language (VRML) to standardowy format tekstowy do reprezentowania treści trójwymiarowej w internecie. Został opracowany w 1994 r. przez grupę, w skład której wchodzili wynalazca i teoretyk Mark Pesce, członkowie zespołu Silicon Graphics Open Inventor oraz ja. Od lat 90. ubiegłego wieku język VRML przeszedł wiele zmian, zyskał szerokie poparcie w branży oraz wsparcie konsorcjum standaryzacyjnego non profit. Na początku XXI wieku powstał oparty na języku XML następca języka VRML o nazwie X3D. Podczas gdy standardy te nie są już powszechnie używane w aplikacjach sieciowych, nadal większość narzędzi do modelowania obsługuje je jako formaty eksportu i importu.

Formaty VRML i X3D służą do definiowania całych scen, animacji (opartych na klatkach kluczowych, morfingu oraz szkieletowych), materiałów, światła, a nawet interaktywnych obiektów z określonymi zachowaniami. Na listingu 8.4 pokazano przykład kodu w formacie X3D tworzący scenę zawierającą czerwony sześcian wykonujący w reakcji na kliknięcie pełny obrót wokół osi y w ciągu dwóch sekund. Geometria, zachowania obiektu i animacje są zapisane w tym jednym pliku o intuicyjnej, czytelnej dla człowieka składni. Do dziś nie powstał żaden inny otwarty trójwymiarowy format plików, za pomocą którego można by było w tak prosty i elegancki sposób przedstawić tego rodzaju dane.

Listing 8.4. Przykład składni formatu X3D: czerwony sześcian obracający się w reakcji na kliknięcie

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE X3D PUBLIC "ISO//Web3D//DTD X3D 3.0//EN"
  "http://www.web3d.org/specifications/x3d-3.0.dtd">
<X3D profile='Interactive' version='3.0'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema-instance'
  xsd:noNamespaceSchemaLocation =
    'http://www.web3d.org/specifications/x3d-3.0.xsd '>
<head>
  ... <!-- Tu znajdują się metadane XML dla pliku X3D -->
</head>
<!--
```

```

Index for DEF nodes: Animation, Clicker, TimeSource, XForm
-->
<Scene>
<!-- XForm ROUTE: [from Animation.value_changed to rotation] -->
<Transform DEF='XForm'>
<Shape>
<Box/>
<Appearance>
<Material diffuseColor='1.0 0.0 0.0' />
</Appearance>
</Shape>
<!-- Clicker ROUTE: [from touchTime to TimeSource.startTime] -->
<TouchSensor DEF='Clicker' description='click to animate' />
<!-- TimeSource ROUTEs:
[from Clicker.touchTime to startTime] [from fraction_changed to
Animation.set_fraction] -->
<TimeSensor DEF='TimeSource' cycleInterval='2.0' />
<!-- Animation ROUTEs:
[from TimeSource.fraction_changed to set_fraction]
[from value_changed to XForm.rotation] -->
<OrientationInterpolator DEF='Animation' key='0.0 0.33 0.66 1.0'
keyValue='0.0 1.0 0.0 0.0 0.0 1.0 0.0 2.1 0.0 1.0 0.0 4.2 0.0 1.0 0.0 0.0' />
</Transform>
<ROUTE fromNode='Clicker' fromField='touchTime' toNode='TimeSource'
toField='startTime' />
<ROUTE fromNode='TimeSource' fromField='fraction_changed'
toNode='Animation' toField='set_fraction' />
<ROUTE fromNode='Animation' fromField='value_changed' toNode='XForm'
toField='rotation' />
</Scene>
</X3D>

```

Projekt formatu VRML uwzględnia wiele kluczowych koncepcji dotyczących interaktywnej grafiki trójwymiarowej i dlatego można oczekwać, że dobrze nadaje się do użytku z WebGL. Niestety, format ten powstał w erze przed pojawieniem się JavaScriptu i DOM, a także zanim powstało wiele przyspieszanych sprzętowo funkcji graficznych. Dlatego — moim zdaniem — VRML/X3D to przestarzałe formaty, które dziś już nie nadają się do użytku. Jednocześnie zastosowane w nich pomysły można wykorzystać w WebGL.

Przez lata powstało mnóstwo treści, która została zapisana w formatach VRML i X3D. Specjalisci z niemieckiego instytutu Fraunhofer Institute nie porzucili jednak formatu X3D i pracują nad X3DOM, biblioteką do przeglądania treści X3D przy użyciu WebGL, bez stosowania wtyczek. Więcej informacji na ten temat znajduje się na stronie <http://www.x3dom.org/>.

W internecie można znaleźć specyfikacje formatów VRML (<http://bit.ly/web3d-vrml>) i X3D (<http://bit.ly/web3d-x3d>).

COLLADA — format do wymiany zasobów cyfrowych

W połowie pierwszej dekady XXI wieku, w czasie gdy format VRML powoli robił się przestarzały, grupa firm, włącznie z Sony Computer Entertainment, Alias Systems Corporation i Avid Technology, połączyła siły w celu utworzenia nowego formatu do wymiany trójwymiarowych cyfrowych zasobów między grami i interaktywnymi aplikacjami trójwymiarowymi. Kierownictwo nad projektem powierzono Rémiemu Arnaudowi i Markowi C. Barnesowi z Sony, a opracowywany format nazwano COLLADA (ang. *COLLABorative Design Activity*). Po przeprowadzeniu

wstępnych prac nad specyfikacją i otrzymaniu wsparcia różnych firm pracy nad standardem przekazano do Khronos Group, tej samej organizacji non profit, która opiekuje się standardami WebGL, OpenGL oraz innymi standardami dotyczącymi sprzętu graficznego i API programistycznych.

COLLADA, podobnie jak X3D, to kompletny format XML, za pomocą którego można reprezentować całe sceny zawierające geometrię, materiały, animacje i różne typy oświetlenia. W odróżnieniu od X3D, format COLLADA nie ma służyć do przechowywania kompletnych gotowych do użytku produktów, z zachowaniami i semantyką czasu wykonywania. W istocie wyraźnie podkreślono, że to nie są cele tej technologii. COLLADA ma umożliwiać przechowywanie wszystkich informacji, które można wyeksportować z programu do tworzenia grafiki trójwymiarowej, aby następnie można było ich używać w innym programie albo zimportować do silnika gier lub środowiska programistycznego. Ogólnie chodziło o to, by twórcy programów do obróbki grafiki trójwymiarowej nie musieli pisać coraz nowych eksporterów do różnych własnych formatów, tylko mogli korzystać z powszechnie przyjętego formatu.

Na listingu 8.5 pokazano fragment reprezentującego scenę kodu COLLADA, który później wczytamy za pomocą biblioteki Three.js. Można w nim zauważyć kilka charakterystycznych cech strukturalnych. Po pierwsze, wszystkie konstrukcje są zorganizowane w **bibliotekach**, czyli zbiorach typów, np. obrazów, shaderów i materiałów. Biblioteki te najczęściej znajdują się na początku, bo później są używane przez inne konstrukcje (np. obrazy są używane w definicjach materiałów). Po drugie, w kodzie znajdują się deklaracje przypominające funkcje wbudowane, np. cieniowanie Blinna. W formacie COLLADA brakuje jednakże żadnych założeń na temat cieniowania i modeli renderingu. Jest to jedynie format do przechowywania danych, które można wykorzystać w jakimś programie. Dalej znajdują się dane wierzchołków siatki zapisane w elementach `float_array`. Na końcu siatka jest składana w scenę przy użyciu wcześniej zdefiniowanych geometrii i materiałów (elementy `instance_geometry`, `bind_material` oraz `instance_material`).

Listing 8.5. Struktura pliku w formacie COLLADA, przykładowe biblioteki, geometria i scena

```
<?xml version="1.0"?>
<COLLADA xmlns="http://www.collada.org/2005/11/COLLADASchema"
version="1.4.1">
  <asset>
    <contributor>
      <authoring_tool>CINEMA4D 12.043 COLLADA Exporter
      </authoring_tool>
    </contributor>
    <created>2012-04-25T16:44:59Z</created>
    <modified>2012-04-25T16:44:59Z</modified>
    <unit meter="0.01" name="centimeter"/>
    <up_axis>Y_UP</up_axis>
  </asset>
  <library_images>
    <image id="ID5">
      <init_from>tex/Buss.jpg</init_from>
    </image>
    ... <!-- kolejne definicje obrazów -->
  </library_images>
  <library_effects>
    <effect id="ID2">
      <profile_COMMON>
        <technique sid="COMMON">
          <blinn>
```

```

<diffuse>
    <color>0.8 0.8 0.8 1</color>
</diffuse>
<specular>
    <color>0.2 0.2 0.2 1</color>
</specular>
<shininess>
    <float>0.5</float>
</shininess>
</blinn>
</technique>
</profile_COMMON>
</effect>
...
... <!-- kolejne definicje efektów -->
<library_geometries>
    <geometry id="ID56">
        <mesh>
            <source id="ID57">
                <float_array id="ID58" count="22812">36.2471
9.43441 -6.14603 36.2471 11.6191 -6.14603 36.2471 9.43441 -9.04828
36.2471 11.6191 -9.04828 33.356 9.43441 -9.04828 33.356 11.6191
-9.04828 33.356 9.43441
...
... <!-- pozostała część definicji siatki -->
...
... <!-- definicja sceny jako hierarchii węzłów -->
<library_visual_scenes>
    <visual_scene id="ID53">
        <node id="ID55" name="Buss">
            <translate sid="translate">5.08833 -0.496439
-0.240191</translate>
            <rotate sid="rotateY">0 1 0 0</rotate>
            <rotate sid="rotateX">1 0 0 0</rotate>
            <rotate sid="rotateZ">0 0 1 0</rotate>
            <scale sid="scale">1 1 1</scale>
            <instance_geometry url="#ID56">
                <bind_material>
                    <technique_common>
                        <instance_material
                            symbol="Material1" target="#ID3">
                            <bind_vertex_input
                                semantic="UVSET0"
                                input_semantic="TEXCOORD"
                                input_set="0"/>
                        </instance_material>
                    </technique_common>
                </bind_material>
            </instance_geometry>
        </node>
...
... <!-- pozostała część definicji sceny -->

```

Po początkowym szale i przyjęciu standardu przez wiele firm format COLLADA zaczął tracić na popularności. Około 2010 r. praktycznie całkowicie ustąpiły wszelkie prace nad wtyczkami eksportowymi do popularnych programów do tworzenia grafiki trójwymiarowej. Jednak od niedawno znowu zainteresowaniem formatem COLLADA wzrasta i ma to związek głównie z tym, że coraz lepiej obsługiwana jest technologia WebGL, oraz z tym, że brakuje wbudowanego formatu plików dla WebGL (jeszcze rozwinę ten temat). Powstał nowy projekt open source o nazwie OpenCOLLADA (<http://bit.ly/open-collada>), w ramach którego zaktualizowano eksportery dla programów 3ds Max i Maya od wersji 2010 do aktualnych. Eksportery te tworzą czysty standardowy kod COLLADA.

Mimo że poprawiona obsługa formatu COLLADA jest niewątpliwie świetną wiadomością dla twórców grafiki trójwymiarowej, to jednak nie wszystko jest takie różowe. Jak widać w przedstawionym przykładzie, format ten jest bardzo obszerny, ponieważ został zaprojektowany do przechowywania danych, a nie w celu umożliwienia szybkiego pobierania i przetwarzania plików. Dlatego właśnie grupa Khronos podjęła nową inicjatywę, w ramach której stara się przenieść to, co najlepsze ze starego formatu — pełną reprezentację animowanych scen trójwymiarowych — do nowego formatu o nazwie glTF.

glTF — nowy format dla aplikacji WebGL, Open GL ES oraz OpenGL

Wzrost popularności biblioteki WebGL spowodował pojawienie się nowego problemu: potrzebna jest metoda dostarczania treści scen z programów do tworzenia grafiki trójwymiarowej do działających aplikacji WebGL. Formaty tekstowe, takie jak OBJ, nadają się do reprezentowania tylko pojedynczych obiektów i nie da się w nich zapisać struktury grafu sceny, oświetlenia, kamer ani animacji. COLLADA to format o dość dużych możliwościach, ale — jak przekonaliśmy się w poprzednim podrozdziale — jest dość obszerny. Ponadto bazuje na składni XML, przez co przetworzenie go na struktury odpowiednie do renderowania w WebGL wymaga intensywnej pracy CPU. Dlatego potrzebny był kompaktowy format nadający się do użycia w internecie, który nie wymaga intensywnego przetwarzania przed renderowaniem. Potrzebne było coś podobnego do formatu JPEG, tylko dla grafiki trójwymiarowej.

Latem 2012 r. Fabrice Robinet, inżynier zatrudniony w Motoroli i prezes grupy roboczej ds. formatu COLLADA w grupie Khronos, rozpoczął prace nad trójwymiarowym formatem plików o właściwościach graficznych zbliżonych do COLLADA, ale z bardziej zwięzłą i przyjazną dla WebGL reprezentacją. Początkowo projekt nazywał się COLLADA2JSON, ponieważ z założenia chodziło o przełożenie skomplikowanej składni XML na bardziej przejrzystą składnię JSON. Jednak produkt zaczął żyć własnym życiem. Do Robineta dołączyli inni członkowie grupy roboczej, m.in. ja, twórca formatu COLLADA Remi Arnaud oraz Patrick Cozzi, inżynier z firmy AGI zajmującej się tworzeniem oprogramowania obronnego. Polecono nam, abyśmy rozszerzyli zakres działalności i, zamiast tworzyć zoptymalizowaną wersję formatu COLLADA, opracowali nowy format do użytku w aplikacjach opartych na OpenGL dla internetu i urządzeń przenośnych. W ten sposób powstał format glTF (ang. *Graphics Library Transmission Format*).

Format glTF jest podobny do COLLADA pod względem funkcjonalności, ale to całkiem nowy format. Podczas prac zespołu wzorujemy się na poprzedniku w określaniu zestawu funkcji graficznych, ale szczegółowa ich implementacja wygląda całkiem inaczej. W formacie glTF używane są pliki JSON do opisu struktury grafu sceny i informacji wysokiego poziomu (np. kamer i światła) oraz pliki binarne do opisu danych dotyczących wierzchołków, normalnych, kolorów i animacji. Format binarny glTF jest tak skonstruowany, że może być wczytywany bezpośrednio do buforów WebGL (tablic typowanych, takich jak `Int32Array` i `FloatArray`). Dzięki temu proces wczytywania pliku glTF jest bardzo prosty i odbywa się następująco.

1. Wczytanie niewielkiego pliku opakowującego JSON.
2. Załadowanie zewnętrznej biblioteki przy użyciu Ajaksa.
3. Utworzenie kilku tablic typowanych.
4. Wywołanie metod kontekstu rysowania WebGL w celu wyrenderowania obrazu.

Oczywiście w praktyce sprawa jest nieco bardziej skomplikowana, ale czynności, które należy wykonać, są znacznie prostsze niż pobieranie i przetwarzanie pliku XML, a następnie konwertowanie tablic wartości typu JavaScript Number na tablice animowane. Format glTF może zapewnić znaczne korzyści zarówno pod względem rozmiaru plików, jak i szybkości wczytywania treści — a oba te czynniki mają krytyczne znaczenie przy budowie wysokowydajnych aplikacji internetowych i przeznaczonych do użytku w urządzeniach przenośnych.

Na listingu 8.6 pokazano składnię JSON typowej sceny glTF, czyli słynny model kaczki COLLADA. Należy zwrócić uwagę na podobieństwa do starszego formatu: najpierw zdefiniowane są biblioteki, a struktura grafu sceny jest zdefiniowana na końcu przy użyciu tych bibliotek. Na tym kończą się podobieństwa. W glTF nie ma jakichkolwiek zbędnych informacji, a struktury są zdefiniowane w taki sposób, by można je było szybko załadować do WebGL i OpenGL ES. Bardzo szczegółowo zdefiniowane są atrybuty (pozycje wierzchołków, normalne, kolory, współrzędne teksturowe itd.) wykorzystywane do renderowania obiektów przy użyciu programowalnych shaderów. Dzięki temu aplikacja obsługująca format glTF może wiernie wyrenderować każdą siatkę, nawet jeśli nie dysponuje własnym rozbudowanym systemem materiałów.

Listing 8.6. Przykład danych w formacie JSON glTF

```
{  
    "animations": {},  
    "asset": {  
        "generator": "collada2gltf 0.1.0"  
    },  
    "attributes": {  
        "attribute_22": {  
            "bufferView": "bufferView_28",  
            "byteOffset": 0,  
            "byteStride": 12,  
            "count": 2399,  
            "max": [  
                96.1799,  
                163.97,  
                53.9252  
            ],  
            "min": [  
                -69.2985,  
                9.92937,  
                -61.3282  
            ],  
            "type": "FLOAT_VEC3"  
        },  
        ...dalejs atrybuty wierzchołków  
        "bufferViews": {  
            "bufferView_28": {  
                "buffer": "duck.bin",  
                "byteLength": 76768,  
                "byteOffset": 0,  
                "target": "ARRAY_BUFFER"  
            },  
            "bufferView_29": {  
                "buffer": "duck.bin",  
                "byteLength": 25272,  
                "byteOffset": 76768,  
                "target": "ELEMENT_ARRAY_BUFFER"  
            }  
        },  
    },  
}
```

```

"buffers": {
    "duck.bin": {
        "byteLength": 102040,
        "path": "duck.bin"
    }
},
"cameras": {
    "camera_0": {
        "aspect_ratio": 1.5,
        "projection": "perspective",
        "yfov": 37.8492,
        "zfar": 10000,
        "znear": 1
    }
},
... inne obiekty wysokopoziomowe, np. materiały i światła
... na końcu jest graf sceny
"nodes": {
    "LOD3sp": {
        "children": [],
        "matrix": [
            ... matrix data here
        ],
        "meshes": [
            "LOD3spShape-1ib"
        ],
        "name": "LOD3sp"
    },
}

```

W kodzie glTF, oprócz pliku JSON, znajduje się odwołanie do przynajmniej jednego pliku binarnego (z rozszerzeniem *bin*) zawierającego m.in. dane wierzchołków siatek i animacji zapisane w strukturach zwanych **buforami** i **widokami buforów**. W ten sposób możemy strumieniować, pobierać po kawałku lub wczytywać całą treść glTF za jednym razem, zależnie od potrzeby.

Głównym celem projektantów glTF było stworzenie zwięzłego i efektywnego formatu do reprezentacji danych OpenGL, ale zachowano także możliwość zapisywania ważnych informacji trójwymiarowych opracowanych w narzędziach do budowania grafiki trójwymiarowej, takich jak animacje, kamery i oświetlenie. Oto zestawienie możliwości aktualnej (1.0) wersji formatu glTF.

Siatki

Siatki wielokątów mogą składać się z jednego lub większej liczby rodzajów podstawowych figur geometrycznych. Definicja siatki znajduje się w pliku JSON zawierającym odwołania do plików binarnych z zapisanymi danymi wierzchołków.

Materiały i shadery

Materiały mogą być zdefiniowane jako konstrukcje wysokopoziomowe (Blinn, Phong, Lambert) lub zaimplementowane przy użyciu shaderów wierzchołków i fragmentów w języku GLSL oraz dołączane jako pliki zewnętrzne do pliku JSON glTF.

Światła

Typowe rodzaje światel (kierunkowe, punktowe, reflektorowe i otaczające) są reprezentowane w pliku JSON jako konstrukcje wysokopoziomowe.

Kamery

W glTF zdefiniowane są typowe rodzaje kamer, np. perspektywiczna i ortogonalna.

Struktura grafu sceny

Scenę reprezentuje hierarchiczny graf węzłów (tzn. siatek, kamer i światel).

Hierarchia przekształceń

Z każdym węzłem grafu sceny jest związana macierz przekształceń. Każdy węzeł może mieć węzły podrzędne, które dziedziczą przekształcenia po węźle nadzewnętrznym.

Animacje

W glTF zdefiniowane są struktury danych do przechowywania animacji klatkowych, szkieletowych oraz morfingowych.

Media zewnętrzne

Obrazy i filmy wykorzystywane jako tekstury dołączają się za pomocą adresów URL.

Projekt glTF działa wprawdzie pod auspicjami grupy Khronos, ale jest otwarty dla wszystkich, którzy chcą pomóc w jego rozwijaniu. W serwisie GitHub znajduje się magazyn kodu, w którym można znaleźć działające przeglądarki i przykładową treść, a także specyfikację formatu. Podczas prac przyjęto zasadę, że standaryzacji będą poddawane tylko te funkcje, których przydatność udało się dowieść w kodzie źródłowym, i w efekcie opracowano już cztery niezależne przeglądarki glTF, z których jedna jest dla biblioteki Three.js (wkrótce przyjrzymy się jej dokładnie). Więcej informacji na ten temat znajduje się na stronie projektu glTF (<http://gltf.g1/>).

Autodesk FBX

Jest jeszcze jeden format do przechowywania całych scen, o którym warto przynajmniej nadmienić. Mowa o formacie FBX firmy Autodesk, który został opracowany przez firmę Kaydara dla programu MotionBuilder. Po przejęciu Kaydary firma Autodesk zaczęła używać formatu FBX także w kilku innych produktach, dzięki czemu stał się standardowym formatem wymiany danych między produktami firmy Autodesk (3ds Max, Maya oraz MotionBuilder).

FBX to rozbudowany format umożliwiający przechowywanie wielu trójwymiarowych i ruchomych typów danych. W odróżnieniu od pozostałych opisanych w tym rozdziale formatów, FBX jest własnością i pozostaje pod kontrolą firmy Autodesk. Firma ta napisała jego dokumentację i udostępniła pakiety SDK do odczytu i zapisu danych w formacie FBX w językach C++ i Python. Jednak do ich używania potrzebna jest licencja, której cena dla niektórych użytkowników jest zaporowa. Udało się też napisać importery i eksportery formatu FBX bez użycia SDK, np. dla programu Blender, ale nie ma pewności, czy ich używanie jest legalne.

Biorąc pod uwagę to, że format FBX jest własnością jednej firmy oraz niejasności związane z jego licencją, lepiej trzymać się od niego z daleka. Z drugiej strony jednak, technologia ta stwarza bardzo duże możliwości i jest wykorzystywana w najlepszych narzędziach w branży. Może więc warto się nią zainteresować. Więcej informacji na temat formatu FBX znajduje się na stronie <http://www.autodesk.com/products/fbx/overview>.

Wczytywanie treści do aplikacji WebGL

Przypomnę, że WebGL to biblioteka do rysowania. Brakuje w niej pojęć wielokąta, siatki, materiału, oświetlenia oraz jakichkolwiek innych wysokopoziomowych konstrukcji, których programiści używają do koncepcyjnego modelowania grafiki trójwymiarowej. W WebGL znane są tylko trójkąty i arytmetyka, więc pewnie dla nikogo nie będzie zaskoczeniem to, że biblioteka

ta nie ma żadnego własnego formatu plików ani wbudowanych narzędzi do obsługi którychkolwiek z wcześniej opisanych formatów. Aby wczytać plik z danymi trójwymiarowymi do aplikacji sieciowej, należy napisać własny kod albo użyć biblioteki pomocniczej.

Na szczęście, w bibliotece Three.js dostępne są przykłady kodu do wczytywania plików w wielu różnych popularnych formatach, np. OBJ, STL, VRML czy COLLADA. Jednak nie ma róży bez kolców, bo jak napisałem, są to tylko przykłady kodu i niektóre z nich są bardziej zaawansowane, a inne mniej. Ponadto biblioteka Three.js ma własne specjalne formaty plików: tekstowy, oparty na składni JSON, oraz podobny do glTF format binarny zapewniający mały rozmiar i szybkie wczytywanie plików. Jest nawet oparty na składni JSON format umożliwiający zapisywanie całych scen zawierających wiele obiektów, ale format ten jest jeszcze w fazie eksperymentalnej i wg mnie nie nadaje się na razie do użytku w środowisku produkcyjnym.

Krótko mówiąc, proces powstawania treści dla WebGL należy traktować jak wielką przygodę. Wprawdzie kiedyś w końcu dotrzemy do celu, ale z pewnością po drodze czeka nas wiele zwrotów akcji i niespodzianek. Czas zatem wziąć byka za rogi. Do końca tego rozdziału będziemy dryfować po wzburzonym, pełnym raf morzu technik wczytywania treści do aplikacji WebGL przy użyciu biblioteki Three.js.

Format JSON biblioteki Three.js

W rdzeniu biblioteki Three.js jest zdefiniowany format plików do wczytywania siatek przypominający format OBJ. Jednak w odróżnieniu od niego jest oparty na składni JSON, a więc po przetworzeniu może być używany przez bibliotekę Three.js.

Aktualnie liczba narzędzi eksportujących do formatu JSON biblioteki Three.js jest niewielka, chociaż twórcy biblioteki napisali eksporter dla Blendera, więc można zwrócić się w tę stronę. W ogóle Blender może być przydatny do przenoszenia danych do biblioteki Three.js, ponieważ program ten importuje treść z wielu różnych rodzajów źródeł i formatów plików. A jeśli nie używasz tego programu, możesz konwertować pliki OBJ, dla których biblioteka Three.js ma konwerter napisany w języku Ruby. Użyjemy go w następnym przykładzie.

Otwórz w przeglądarce plik *r08/pipelinetreejsmodel.html*, aby wyświetlić model krzesła w kształcie kuli z poduszką do siedzenia, takiego w stylu z mniej więcej połowy ubiegłego wieku (rysunek 8.12). Kliknij lewym klawiszem, aby obracać model, albo pokręć kółkiem, by go powiększyć lub zmniejszyć.

Cienie i oświetlenie zostały ręcznie zakodowane, co pozwoliło uzyskać ładny efekt, ale cały model jest w formacie OBJ. Po jego pobraniu z serwisu Turbosquid przekonwertowałem go za pomocą konwertera na format JSON rozpoznawany przez bibliotekę Three.js.

Konwerter ten znajduje się w podfolderze *utils* biblioteki Three.js. Aby przekonwertować opisywany model, wykonaj następujące polecenie:

```
python <ścieżka-do-three.js>/utils/exporters/convert_obj_three.py -i ball_chair.obj -o ball_chair.js
```

W efekcie powstanie plik *ball_chair.js* zawierający kod w formacie JSON, którego fragment pokazano na listingu 8.7. Na początku znajdują się metadane dotyczące numerów wersji itp., a dopiero po nich umieszczono właściwą treść. Pierwszy fragment zawiera definicje materiałów. Powinny one wyglądać znajomo, ponieważ są to przekonwertowane materiały z pliku



Rysunek 8.12. Efekt konwersji pliku w formacie Wavefront OBJ na format JSON biblioteki Three.js oraz załadowania danych za pomocą narzędzia THREE.JSONLoader. Jest to klasyczny model krzesła z Turbosquid (<http://bit.ly/1dNri0m>), którego twórca jest Luxxeon (<http://luxxeon.deviantart.com/>)

OBJ MTL przedstawionego na listingu 8.2. Następna jest definicja siatki, która zajmuje najwięcej miejsca w całym pliku. Definicja ta składa się ze zbioru tablic JSON określających pozycje wierzchołków, normalne, współrzędne teksturowe oraz boki. Mając wszystkie te dane w formacie JSON, biblioteka Three.js może bez problemu zbudować widoczne na ekranie siatki.

Listing 8.7. Przykład formatu JSON biblioteki Three.js

```
{  
  "metadata": {  
    "formatVersion": 3.1,  
    "sourceFile": "ball_chair(blender).obj",  
    "generatedBy": "OBJConverter",  
    "vertices": 12740,  
    "faces": 12480,  
    "normals": 13082,  
    "colors": 0,  
    "uvs": 15521,  
    "materials": 4  
  },  
  "scale": 1.000000,  
  "materials": [ {  
    "DbgColor": 15658734,  
    "DbgIndex": 0,  
    "DbgName": "shell",  
    "colorAmbient": [0.0, 0.0, 0.0],  
    "colorDiffuse": [1.0, 0.0, 0.0],  
    "colorSpecular": [0.0, 0.0, 0.0],  
    "normalMap": "shell_normal",  
    "normalScale": 1.0  
  } ]  
}
```

```

    "colorDiffuse" : [0.588, 0.588, 0.588],
    "colorSpecular" : [0.72, 0.72, 0.72],
    "illumination" : 2,
    "mapAmbient" : "shell_color.jpg",
    "mapDiffuse" : "shell_color.jpg",
    "opticalDensity" : 1.5,
    "specularCoef" : 77.0
  },
  ...dalej część definicji materiałów
  "vertices": [-1.569305,4.927318,-1.529769,-0.889529,
  ...pozostałe dane dotyczące wierzchołków
  "morphTargets": [],
  "morphColors": [],
  "normals": [-0.53717,0.35055,-0.76718,-0.46279,0.35837,
  ...pozostałe dane dotyczące normalnych, kolorów oraz współrzędnych teksturowych
  "faces": [43,312,599,57,596,0,0,1,2,3,0,1,2,3,43,597
  ...pozostałe dane dotyczące boków
}

```

Teraz spójrzmy na kod wczytujący model. Biblioteka Three.js nie zawiera wbudowanej przeglądarki modeli, więc musimy ją napisać samodzielnie. Nie jest to trudne, przynajmniej dopóki nie potrzebujemy niczego zaawansowanego. Podzielę ten przykład na dwa listingi. Na pierwszym znajdzie się kod dotyczący tworzenia sceny i ładowania modelu, a na drugim kod dotyczący konfiguracji środowiska widokowego z oświetleniem, cieniem i przyciskami do sterowania kamerą. Na listingu 8.8 przedstawiony jest kod tworzenia sceny i wczytywania modelu.

Listing 8.8. Kod wczytujący model w formacie JSON biblioteki Three.js

```

function loadModel() {
  // Krzesło w kształcie kuli wg Luxxeona
  // http://www.turbosquid.com/FullPreview/Index.cfm/ID/761919
  // http://www.turbosquid.com/Search/Artists/luxxeon
  // http://luxxeon.deviantart.com/
  var url = "../models/ball_chair/ball_chair.json";
  // Krzesło w kształcie jaja wg Luxxeona
  // http://www.turbosquid.com/FullPreview/Index.cfm/ID/738230
  // http://www.turbosquid.com/Search/Artists/luxxeon
  // http://luxxeon.deviantart.com/
  // var url = "../models/egg_chair/eggchair.json";
  var loader = new THREE.JSONLoader();
  loader.load( url, function( geometry, materials ) {
    handleModelLoaded(geometry, materials) } );
}

function handleModelLoaded(geometry, materials) {
  // Tworzy nową siatkę przy użyciu materiałów dla poszczególnych boków.
  var material = new THREE.MeshFaceMaterial(materials);
  var mesh = new THREE.Mesh( geometry, material );
}

```

```

// Włącza cienie.
mesh.castShadow = true;

// Przesuwa przedmiot do początku, jeśli nie jest wyśrodkowany.
geometry.computeBoundingBox();
center = new THREE.Vector3().addVectors(geometry.boundingBox.max,
    geometry.boundingBox.min).multiplyScalar(0.5);
mesh.position.set(-center.x, 0, -center.z);
scene.add( mesh );

// Znajduje odpowiednią pozycję kamery na podstawie rozmiaru geometrii.
var front = geometry.boundingBox.max.clone().sub(center);
 $\text{//camera.position.set}(0, \text{geometry.boundingBox.max.y / 2},$ 
    geometry.boundingBox.max.z * 8);
camera.position.set(0, front.y, front.z * 5);

if (orbitControls)
    orbitControls.center.copy(center);
}

function createScene(container) {

// Tworzy nową scenę Three.js.
scene = new THREE.Scene();

// Dodaje kamerę, aby można było oglądać scenę.
camera = new THREE.PerspectiveCamera( 45, container.offsetWidth /
    container.offsetHeight, 1, 4000 );
camera.position.z = 10;
scene.add(camera);

// Światła.
createLights();

// Podłożę.
if (addEnvironment)
    createEnvironment();

// Model.
loadModel();
}

```

Najpierw za pomocą funkcji `createScene()` budujemy pustą scenę Three.js, a z wykorzystaniem wywoływanych w niej funkcji pomocniczych tworzymy kamerę, oświetlenie oraz cień. Przypominam, że te formaty obsługujące pojedyncze modele nie przechowują danych kamer ani światel, więc trzeba te elementy dodać samodzielnie.

Następnie wywołujemy funkcję `loadModel()`, aby wczytać dane. W funkcji tej wykorzystujemy narzędzie `THREE.JSONLoader`, które konwertuje przetworzony kod JSON na nadającą się do użycia przez Three.js geometrię. Wywołujemy metodę `load()` programu wczytującego, przekazując jej adres URL modelu i funkcję zwrotną o nazwie `handleModelLoaded()`. Po przetworzeniu kodu JSON biblioteka Three.js tworzy obiekt `geometryczny` i wywołuje naszą funkcję zwrotną. Utworzenie materiałów jest naszym zadaniem (co wg mnie jest dość dziwne), z którego wywiązujemy się, używając specjalnego rodzaju materiału o nazwie `THREE.MeshFaceMaterial`. Jest to kontener na listę kilku materiałów: format JSON obsługuje geometrię, w której każdy bok ma inny materiał. Tworzymy nowy obiekt `MeshFaceMaterial`, używając listy materiałów przekazanej w drugim argumencie funkcji zwrotnej.

Mamy już gotową do renderowania siatkę, którą dodajemy do sceny. A oprócz tego wykonujemy jeszcze kilka drobnych czynności. Chcemy wyświetlić cień, więc ustawiamy własność castShadow siatki na true. Chcemy ustawić siatkę w dogodnym położeniu do oglądania za pomocą kamery orbitującej, więc umieszczamy ją centralnie na początku. Środek znajdujemy za pomocą metody Three.js o nazwie getBoundingBox(). Ponadto wyniku tej metody używamy także do określenia odpowiedniej pozycji dla kamery, którą umieszczamy na górze i nieco z przodu.

Na listingu 8.9 przedstawiony jest kod dotyczący tworzenia przeglądarki modeli. Pętla renderująca zawiera pewien ważny szczegół: obrotowe światło przednie (zwykłe światło kierunkowe) zawsze skierowane od aktualnej pozycji kamery do środka sceny. Dzięki temu geometria jest zawsze widoczna, bez względu na to, którą część modelu oglądamy.

Listing 8.9. Cień i oświetlenie sceny dla przeglądarki modeli w formacie JSON

```
function run() {  
    requestAnimationFrame(function() { run(); });  
  
    // Aktualizuje kontrolera kamery.  
    orbitControls.update();  
  
    // Zmienia pozycję światła przedniego tak, aby świeciło na model.  
    headlight.position.copy(camera.position);  
  
    // Renderuje scenę.  
    renderer.render( scene, camera );  
}  
  
var shadows = true;  
var addEnvironment = true;  
var SHADOW_MAP_WIDTH = 2048, SHADOW_MAP_HEIGHT = 2048;  
  
function createRenderer(container) {  
    // Tworzy renderer Three.js i wiąże go z naszą kanwą.  
    renderer = new THREE.WebGLRenderer( { antialias: true } );  
  
    // Włącza cienie.  
    if (shadows) {  
        renderer.shadowMapEnabled = true;  
        renderer.shadowMapType = THREE.PCFSoftShadowMap;  
    }  
  
    // Ustawia rozmiar obszaru widoku.  
    renderer.setSize(container.offsetWidth, container.offsetHeight);  
  
    container.appendChild(renderer.domElement);  
}  
  
function createLights() {  
  
    // Konfiguracja oświetlenia.  
    headlight = new THREE.DirectionalLight;  
    headlight.position.set(0, 0, 1);  
    scene.add(headlight);  
  
    var ambient = new THREE.AmbientLight(0xffffff);  
    scene.add(ambient);  
  
    if (shadows) {  
        var spot1 = new THREE.SpotLight(0xaaaaaa);  
        spot1.position.set(0, 150, 200);  
        scene.add(spot1);  
    }  
}
```

```

        spot1.shadowCameraNear = 1;
        spot1.shadowCameraFar = 1024;
        spot1.castShadow = true;
        spot1.shadowDarkness = 0.3;
        spot1.shadowBias = 0.0001;
        spot1.shadowMapWidth = SHADOW_MAP_WIDTH;
        spot1.shadowMapHeight = SHADOW_MAP_HEIGHT;
    }
}

function createEnvironment() {
    //Podłoże.
    var floorMaterial = new THREE.MeshPhongMaterial({
        color: 0xffffffff,
        ambient: 0x555555,
        shading: THREE.SmoothShading,
    });
    var floor = new THREE.Mesh( new THREE.PlaneGeometry(1024, 1024), floorMaterial);

    if (shadows) {
        floor.receiveShadow = true;
    }

    floor.rotation.x = -Math.PI / 2;
    scene.add(floor);
}

```

Chcemy uatrakcyjnić nasz obraz ładnym cieniem, więc ustawiamy właściwości cieni podczas tworzenie renderera i światel — spójrz odpowiednio na funkcje `createRenderer()` i `createLights()`. W końcu potrzebujemy też podłożą, na które można będzie rzucać cień. Tworzymy je w funkcji `createEnvironment()`.

Przedstawiony kod służący do wyświetlenia modelu krzesła jest bardzo typowy: tworzy cień, oświetlenie i kamerę, wczytuje model oraz kieruje oświetleniem podczas ruchu kamery. Można go więc użyć do oglądania praktycznie każdego prostego modelu.

Jednak struktura utrudnia jego wykorzystanie w różnych aplikacjach. Naprawimy to w następującym rozdziale, w którym utworzymy zestaw ogólnych klas przeglądarki modeli. Na razie nasz cel jest jasny: umożliwienie wczytywania plików zawierających pojedyncze modele, które uprzednio były zapisane w formacie OBJ. Przy użyciu biblioteki Three.js nie jest to trudne.

Format binarny biblioteki Three.js

W bibliotece Three.js dostępny jest też bardziej zwięzły i lepiej zoptymalizowany format do wczytywania siatek — binarny odpowiednik formatu JSON. Dane w tym formacie składają się z dwóch plików: niewielkiego opakowania JSON opisującego wysokopoziomowe aspekty siatki (np. listę materiałów) oraz pliku binarnego (`.bin`) zawierającego dane wierzchołków i boków.

Za pomocą konwertera plików OBJ biblioteki Three.js można utworzyć pliki binarne, używając przełącznika `-t` wiersza poleceń:

```
python <ściezka-do-three.js>/utils/exporters/convert_obj_three.py -i ball_chair.obj -o ball_chair_bin.js -t binary
```

Wykonaj powyższe polecenie, aby utworzyć plik *ball_chair_bin.js*. Jeśli do niego zanjrzysz, zauważysz, że zawiera kod JSON wyglądający mniej więcej tak samo jak wersja tekstopowa, z tym wyjątkiem, iż dane dotyczące siatki zostały przeniesione do pliku binarnego wskazanego we właściwości buffers:

```
"buffers": "ball_chair_bin.bin"
```

Należy zwrócić uwagę na różnicę w rozmiarze plików. Format binarny (pliki JSON i *.bin*) jest mniej więcej o połowę mniejszy od czystej wersji JSON. Aby zobaczyć jak działa format binarny, otwórz plik *r08/pipelinethreejsmodelbinary.html*. Wygląd modelu się nie zmienił i jest taki sam jak na rysunku 8.12. Aby wczytywać pliki binarne biblioteki Three.js, wystarczy tylko zmienić nazwę klasy THREE.JSONLoader na THREE.BinaryLoader, co pokazano na listingu 8.10.

Listing 8.10. Wczytywanie modeli przy użyciu formatu binarnego biblioteki Three.js

```
function loadModel() {  
    // Krzesło w kształcie kuli wg Luxxeona  
    // http://www.turbosquid.com/FullPreview/Index.cfm/ID/761919  
    // http://www.turbosquid.com/Search/Artists/luxxon  
    // http://luxxon.deviantart.com/  
  
    var url = "../models/ball_chair/ball_chair_bin.json";  
  
    // Krzesło w kształcie jaja wg Luxxeona  
    // http://www.turbosquid.com/FullPreview/Index.cfm/ID/738230  
    // http://www.turbosquid.com/Search/Artists/luxxon  
    // http://luxxon.deviantart.com/  
    // var url = "../models/egg_chair/eggchair.json";  
  
    var loader = new THREE.BinaryLoader();  
    loader.load( url, function( geometry, materials ) {  
        handleModelLoaded(geometry, materials) } );  
}
```

Wczytywanie sceny w formacie COLLADA przy użyciu biblioteki Three.js

Twórcy biblioteki Three.js włożyli dużo wysiłku w to, aby można było wczytywać wysokiej jakości modele w formatach typu JSON i ich własnym JSON. Świeśnie, ale w wielu przypadkach to za mało. Jeśli chcemy wczytać scenę zawierającą wiele obiektów i zachować hierarchię przekształceń oraz inne właściwości, takie jak kamery, światła i animacje, musimy użyć innego formatu, ponieważ wymienione formaty do tego się nie nadają. W przeciwnym razie musielibyśmy importować modele pojedynczo, a następnie ręcznie je poustawić, oświetlić i animować. (Niestety, ciągle się to zdarza w branży WebGL, ale sytuacja powoli zmienia się na lepsze).

Jak napisałem wcześniej, COLLADA to format, który doskonale nadaje się do reprezentowania kompletnych scen. Ma wszystkie potrzebne właściwości i można do niego eksportować dane w kilku programach. Przy użyciu formatu COLLADA artysta może wymodelować, obłożyć teksturomi i animować skomplikowaną scenę, a następnie wyeksportować ją w celu użycia w WebGL *bez pomocy programisty*. To właśnie jest naszym najważniejszym celem: pozostawienie zadań wymagających zdolności artystycznych do wykonania artystom. Oczywiście format COLLADA jest oparty na XML, więc jego wadami są rozwlekłość i powolność. Jednak na nasze potrzeby wystarczy. Na jego przykładzie przedstawię zagadnienia dotyczące wczytywania i przeglądania całych scen.

Otwórz w przeglądarce plik *r08/pipelinethreejsdaescene.html*, aby wyświetlić scenę rodem z gry przygodowej, zawierającą opuszczone budynki i stare samochody — rysunek 8.13.



Rysunek 8.13. Tło gry zawierające hierarchię i materiały wczytane z formatu COLLADA za pomocą narzędzia THREE.ColladaLoader; grafika pochodzi z serwisu Turbosquid (<http://bit.ly/1eQEq6V>), a jej autorem jest ERLHN (<http://www.turbosquid.com/Search/Artists/ERLHN>)

Program ten wczytuje scenę w formacie COLLADA zawierającą hierarchię kilku obiektów. Dane wczytywane są za pomocą jednego wywołania, ponieważ narzędzie do wczytywania danych w formacie COLLADA biblioteki Three.js może utworzyć całą hierarchię obiektów, włącznie z kamerami, animacjami, światłami itd. bez pomocy programisty. Funkcja zwrotna ładowania dodatkowo szuka kamer oraz świateł i jeśli ich nie znajdzie, włącza ustawienia domyślne. I to wszystko. Może się wydawać podejrzany brak jakichkolwiek ręcznych ustawień pozycji, orientacji i skali w celu rozmieszczenia poszczególnych obiektów. Porównaj to z typową przykładową sceną dołączaną do projektu Three.js — zazwyczaj jest to mieszanina ręcznie wpisywanych liczb. Przyjemne odprężenie.

Przeanalizujmy pokazany na listingu 8.11 kod wczytujący scenę w formacie COLLADA. Ukażane zostały tylko fragmenty dotyczące ładowania sceny i odpowiedniej funkcji zwrotnej.

Listing 8.11. Wczytywanie sceny w formacie COLLADA przy użyciu biblioteki Three.js

```
function loadScene() {
    // ruiny wg ERLHN-a
    // http://www.turbosquid.com/FullPreview/Index.cfm/ID/668298
    // http://www.turbosquid.com/Search/Artists/ERLHN
    var url = "../models/ruins/Ruins_dae.dae";

    var loader = new THREE.ColladaLoader();

    loader.load( url, function( data ) {
        handleSceneLoaded(data) } );
}

function handleSceneLoaded(data) {
    // Dodaje obiekty do sceny.
    scene.add(data.scene);

    // Szuka kamery i oświetlenia.
    var result = {};
    data.scene.traverse(function (n) { traverseScene(n, result); });
}
```

```

        if (result.cameras && result.cameras.length)
            camera = result.cameras[0];
        else {
            // Znajduje odpowiednią pozycję dla kamery na podstawie rozmiaru sceny.
            createDefaultCamera();
            var boundingBox = computeBoundingBox(data.scene);
            var front = boundingBox.max;
            camera.position.set(front.x, front.y, front.z);
        }

        if (result.lights && result.lights.length) {
        }
        else
            createDefaultLights();

        // Tworzy kontrolera.
        initControls();
    }

    function traverseScene(n, result)
    {
        // Szuka kamer.
        if (n instanceof THREE.Camera) {
            if (!result.cameras)
                result.cameras = [];

            result.cameras.push(n);
        }

        // Szuka światel.
        if (n instanceof THREE.Light) {
            if (!result.lights)
                result.lights = [];

            result.lights.push(n);
        }
    }
}

```

Funkcja `loadScene()` wczytuje ruiny za pomocą klasy `THREE.ColladaLoader`. Funkcji zwrotnej wczytywania `handleSceneLoaded()` przekazywany jest argument `data` zawierający obiekt JSON z kilkoma własnościami, które zostały zapisane podczas przetwarzania pliku COLLADA. Nas interesuje własność `data.scene` będąca obiektem klasy `THREE.Object` i zawierająca całą hierarchię załadowanej sceny. Dodajemy ją do naszej sceny najwyższego poziomu, aby została wyrenderowana przez bibliotekę `Three.js`.

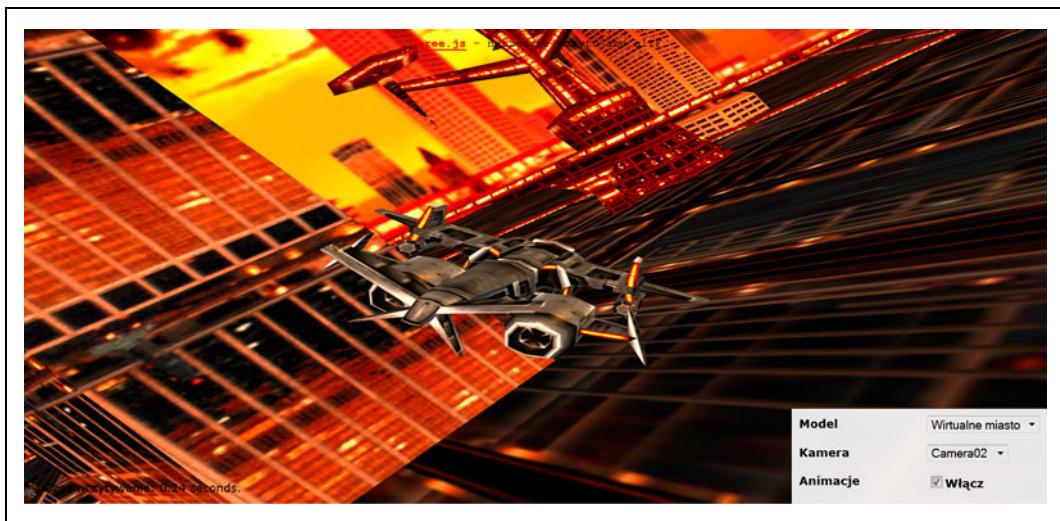
Teraz jesteśmy już praktycznie gotowi do obejrzenia sceny, ale wprowadzimy jeszcze kilka drobnych poprawek, aby poprawić jej wygląd. Najpierw przejrzymy zawartość wczytanej sceny w celu znalezienia kamer i światel. Jeśli znajdziemy jakieś kamery, pierwszej z nich użyjemy jako początkowej. Jeśli nie znajdziemy żadnej kamery, utworzymy domyślną. Jeżeli na scenie znajdują się światła, zostaną użyte. W przeciwnym przypadku utworzymy domyślne oświetlenie. Do przeglądania sceny używamy metody `traverse()` obiektu, która rekurencyjnie przegląda obiekt i jego potomków, wywołując podaną funkcję zwrotną. Funkcja ta w tym przypadku ma nazwę `traverseScene()` i szuka kamery oraz światel, sprawdzając, czy obiekty są typów `THREE.Camera` lub `THREE.Light` za pomocą operatora `instanceof`. Wszystko, co znajdzie, wstawią do tablic `result.cameras` i `result.lights`.

Jeśli scena nie zawiera ani jednej kamery, tworzymy własną kamerę, którą należy odpowiednio umieścić w zależności od rozmiaru sceny. Rozmiar sceny obliczamy za pomocą funkcji pomocniczej `computeBoundingBox()`, która oblicza rozmiar pola sceny za pomocą rekurencji. Gdy natknie się na obiekt geometryczny, oblicza za pomocą wbudowanej metody `Three.js` jego pole, które następnie dołącza do ogólnego pola całej sceny. Kod tej funkcji jest dość długi, więc nie został wydrukowany.

Ładowanie sceny glTF przy użyciu biblioteki Three.js

Format glTF reprezentuje nowatorskie podejście do konstrukcji plików z danymi trójwymiarowymi. Jest specjalnie przystosowany do użytku w aplikacjach mobilnych i opartych na OpenGL, zawierających wiele elementów graficznych reprezentowanych jako macierzyste bufore i inne gotowe do renderowania struktury. Jednocześnie glTF zawiera wiele powszechnie używanych konstrukcji trójwymiarowych, które nie mają bezpośredniej reprezentacji w OpenGL ES, takich jak materiały, kamery i światła. Celem jego twórców było utworzenie zwięzłego formatu, o właściwościach umożliwiających jego łatwe wczytywanie w aplikacjach sieciowych i mobilnych, a zarazem pozwalających na reprezentowanie praktycznych danych trójwymiarowych.

Trwają prace nad kilkoma projektami dotyczącymi implementacji obsługi formatu glTF w bibliotekach i aplikacjach graficznych. Zalicza się do nich też pisane przeze mnie narzędzie wczytujące pliki glTF dla biblioteki Three.js. Aby zobaczyć, jak działa, otwórz w przeglądarce plik `r08/pipelinethreejsglfscene.html`. Na ekranie powinien pojawić się widoczny na rysunku 8.14 obraz przedstawiający futurystyczne miasto, nad którym krążą kilka statków kosmicznych.



Rysunek 8.14. Efekt wczytania sceny glTF — z animacją, hierarchią grafu sceny, materiałami, kamerami i światłami — przy użyciu eksperymentalnej klasy `THREE.gltfLoader`. Kod źródłowy tej klasy znajduje się na stronie projektu glTF w serwisie GitHub (<https://github.com/KhronosGroup/glTF>). Scena wirtualnego miasta została użyta dzięki uprzejmości 3DRT (<http://3drt.com/store/free-downloads/33-sci-fi-skyscrapers-collection.html>)

Do renderowania sceny zastosowano mapy środowiskowe oraz cieniowanie Blinna i efekt jest naprawdę dobry. Zastosowano kilka animacji, wśród których jest też poruszająca kamerami. Znajdujące się w dolnym prawym rogu listy rozwijane służą do zmieniań kamer i wczytywania różnych scen, a pole wyboru *Animacje* umożliwia włączenie lub wyłączenie animacji. Przedstawiona scena została utworzona w programie 3ds Max. Fabrice Robinet pobrał plik tego programu z serwisu 3DRT.com, wyeksportował go do formatu COLLADA, a następnie przekonwertował na glTF.

Projekt swojego narzędzia do wczytywania plików w formacie glTF oparłem na dostępnych w bibliotece Three.js przykładowych narzędziach dla innych formatów. Klasa THREE.gltfLoader dziedziczy po bazowej klasie narzędzi wczytujących THREE.Loader. Jej metoda `load()` przetwarza plik JSON formatu glTF. Wczytuje ona zewnętrzne zasoby, takie jak bufore binarne, tekstury i shadery, oraz zwraca wynik za pomocą funkcji zwrotnej. Funkcja ta ma dostęp do utworzonej przez narzędzie wczytującą hierarchii obiektów Three.js, zatem może ją łatwo wczytać do sceny i rozpoczęć renderowanie.

Wstępne efekty wczytywania danych w formacie glTF są bardzo obiecujące, przynajmniej w porównaniu z formatem COLLADA. Pliki są mniej więcej o połowę mniejsze, a czas wczytywania został zredukowany w niektórych przypadkach nawet o 80%. Osiągnięcia są częściowo zasługą użycia nowego typu Three.js BufferGeometry, który umożliwia tworzenie geometrii bezpośrednio z załadowanej tablicy typowanej, np. `Int32Array` lub `FloatArray`, zamiast zwykłych tablic liczb JavaScript (które i tak przed renderowaniem przez WebGL muszą zostać wewnętrznie przekonwertowane na tablice typowane).

Podsumowanie

W tym rozdziale powierzchniowo poznaleś techniki tworzenia treści trójwymiarowej do użytku w WebGL. Najpierw zrobiliśmy przegląd procesu twórczego, a następnie przejrzaliśmy profesjonalne i amatorskie narzędzia do tworzenia treści trójwymiarowej. Niektóre z nich są klasycznymi instalowanymi programami, a inne działają w środowisku przeglądarki internetowej.

W dalszej części rozdziału poznaleś używane obecnie w różnych aplikacjach trójwymiarowe formaty plików, które mogą być używane w internecie w połączeniu z WebGL. Przeczytałeś zarówno o starych i wysłużonych produktach, jak i nowym formacie o nazwie glTF zaprojektowanym specjalnie do użytku w aplikacjach sieciowych i mobilnych. Na końcu przestudiowałeś kilka przykładów wczytywania do aplikacji WebGL plików w różnych formatach, służących zarówno do przechowywania pojedynczych modeli, jak i kompletnych scen.

Nie istnieje jedyny, poprawny sposób wczytywania treści trójwymiarowej do aplikacji sieciowych, a proces tworzenia treści WebGL wciąż jest niedojrzały i ewoluje, ale przynajmniej mamy kilka możliwości do wyboru.

Trójwymiarowe silniki i systemy szkieletowe

Three.js to fantastyczna biblioteka, z pomocą której złożone zadanie, jakim jest renderowanie skomplikowanej treści w WebGL, staje się czynnością dostępną zwykłym śmiertelnikom. Bez niej lub innej podobnej biblioteki programista musiałby pracować kilka miesięcy, aby w odpowiedni sposób wyświetlić obrazy na ekranie. Jednak, mimo wszystkich zalet, biblioteka Three.js ma też ograniczenia. Pomaga tylko w rysowaniu, a wszystkie inne czynności trzeba wykonać samodzielnie.

Powiedzmy, że chcemy utworzyć aplikację do sprzedaży samochodów z możliwością konfiguracji auta przez kupującego. Na stronie internetowej wyświetlany jest trójwymiarowy model samochodu, a użytkownik może kliknięciami myszą zmieniać kolory i style lub nacisnąć przycisk, by z wykorzystaniem płynnej animacji przejść do wnętrza samochodu. Przy użyciu samej biblioteki Three.js utworzenie takiej aplikacji wymagałoby napisania wielu setek linii kodu źródłowego. Mówiąc krótko, mamy skrzynkę z narzędziami, ale nie zestaw gotowych komponentów. Biblioteka Three.js jest narzędziem do tworzenia grafu sceny i renderowania, ale aplikacje trójwymiarowe nie kończą się na rysowaniu obrązków.

Przykład konfiguratora samochodów przedstawia typowy scenariusz programowania grafiki trójwymiarowej: wczytanie modelu, pobranie indywidualnych części modelu wg nazwy lub identyfikatora, uruchomienie odpowiedniej procedury, gdy jedna z części zostanie kliknięta, oraz zmiana widoków kamery. Ten sposób projektowania jest powszechnie stosowany przy tworzeniu gier, wirtualnych światów, pokazów architektury, kursów edukacyjnych oraz symulatorów treningowych — w zasadzie wszystkich aplikacji trójwymiarowych. Jeśli jesteś profesjonalistą i nie chcesz ciągle od nowa rozwiązywać starych problemów, powinieneś użyć jednego z wysokopoziomowych silników lub systemów szkieletowych.

W tym rozdziale dowiesz się, jak są zbudowane systemy szkieletowe do tworzenia aplikacji trójwymiarowych, oraz poznasz rozwiązania oparte na WebGL. Wiele tych systemów bazuje na bibliotece Three.js, więc jeśli poświęciłeś trochę czasu na naukę jej obsługi, teraz będzie łatwiej. Dalej w tym rozdziale poznasz Vizi, czyli szkielet mojego autorstwa, przy użyciu którego zostały utworzone przykłady opisane w następnych rozdziałach. Koncepcje opisywane w tych szkieletach mają charakter ogólny, tzn. większość z nich ma zastosowanie w wielu systemach i może być wykorzystana do budowy własnego szkieletu.

Koncepcje szkieletów trójwymiarowych

Systemy szkieletowe zawierają gotowe funkcje i solidne implementacje typowych wzorców projektowych. Dzięki nim można zaoszczędzić dużo czasu i podnieść jakość swoich aplikacji — przynajmniej teoretycznie. Dobry system szkieletowy chroni przed „ponownym wynajdowaniem koła”, ponieważ daje możliwość skorzystania z wiedzy doświadczonych programistów oraz pozwala skupić się na jednym zadaniu.

Czym jest system szkieletowy?

Nie istnieje jedna powszechnie obowiązująca definicja systemu szkieletowego, a — co gorsza — często nawet trudno powiedzieć, co różni taki system od biblioteki. Zarówno biblioteka, jak i szkielet oferują gotowe do użycia składniki, dzięki którym powinniśmy oszczędzić czas poświęcony na programowanie, oraz udostępniają wysokopoziomowy interfejs do niskopoziomowych usług podstawowego systemu. Mimo to, istnieje kilka ważnych cech, po których można poznać, czy ma się do czynienia z biblioteką, czy szkieletem¹. Oto one.

Poziom abstrakcji

Systemy szkieletowe operują na wyższym poziomie abstrakcji niż biblioteki. Przykładowo biblioteka trójwymiarowa może pomagać w tworzeniu powleczonych skórą siatek do animacji postaci, podczas gdy system szkieletowy raczej dostarczy gotową taką siatkę wraz z zestawem animowanych gestów, pod nazwą **avatar**. W szkielecie taki awatar będzie poruszany automatycznie w reakcji na działania użytkownika i będziemy o tym powiadamiani poprzez wywołania zwrotne.

Zachowania domyślne

Szkielety dostarczają zachowania domyślne. Gdy np. tworzona jest scena, wewnętrz niej zostaje umieszczona domyślna kamera o znany położeniu i kierunku widzenia. W dobrych szkieletach dodatkowo programista może zmienić ustawienia domyślne.

Rozszerzalność

W systemach szkieletowych szczególny nacisk kładzie się na rozszerzalność, aby można było używać zewnętrznych dodatków. W najlepszych systemach osiągnięto doskonałą równowagę między gotowymi komponentami a metodami rozszerzania lub zastępowania wybranych części systemu.

Odwrotny przepływ sterowania

Chyba najbardziej charakterystyczną cechą systemów szkieletowych jest to, że to one, a nie programista, kontrolują przepływ sterowania. Programista dostarcza tylko funkcje zwrotne lub przesłania metody, aby uzyskać efekt żądany w danej aplikacji. Pomyśl o budowie typowej strony dla aplikacji WebGL: tworzona jest scena, następuje inicjacja renderera, a następnie zostaje uruchomiona pętla wykonawcza. Programista używający systemu szkieletowego WebGL musiałby dostarczyć tylko kod tworzenia sceny, a szkielet zrobiłby za niego resztę.

Ponadto istnieje jeszcze jedna, nietechniczna różnica między szkieletem a biblioteką: szkielety są zazwyczaj bardziej polaryzujące. Należy je traktować jak broń obosieczną albo pakt z diabłem,

¹ Opis oparty na wyczerpującym artykule na temat systemów szkieletowych z Wikipedii (http://en.wikipedia.org/wiki/Software_framework).

w ramach którego za cenę szybkiego wejścia na rynek sprzedajemy swoją duszę. Szkielet może zawierać do 90% potrzebnych nam składników, co daje fałszywe poczucie pewności na samym początku pracy oraz wywołuje potężne rozczarowanie, gdy odkrywany, jak trudno zaimplementować pozostałe 10%. Ponadto w szkieletach trudno diagnostykuje się błędy oraz je optymalizuje, ponieważ zawierają kod napisany przez kogoś obcego. Ból ten zna każdy, kto kiedykolwiek używał systemów typu Zend albo Rails. Z tych powodów wielu programistów unika korzystania z wszelkich systemów szkieletów. Natomiast nienachalna biblioteka, taka jak jQuery, może znacznie ułatwić pracę programisty.



Programiści to jak wszyscy ludzie emocjonalne stworzenia i nic nie denerwuje ich bardziej niż odkurzanie dobrego starego szkieletu. Jeśli zobaczyś programistę przy takiej pracy, lepiej trzymaj się od niego z daleka. Sam mam również zdecydowane poglądy na temat używania szkieletów. Można je streścić w postaci następującej myśli:

Kocham systemy szkieletowe... pod warunkiem, że są moje.

Jeśli, niezależnie od Twojego zdania na temat systemów szkieletowych, planujesz utworzyć trójwymiarową aplikację o dowolnym rozmiarze, z pewnością natkniesz się na problemy opisane w tym rozdziale. Ponadto staniesz przed następującym wyborem: utworzyć własny system szkieletowy, użyć istniejącego, czy przygotować się na pisanie mnóstwa dodatkowego kodu.

Wymagania dotyczące systemów szkieletowych dla WebGL

Przeglądarkę internetową można traktować jak dwuwymiarowy szkielet do tworzenia aplikacji. DOM i CSS udostępniają gotowy zestaw obiektów wizualnych, które są renderowane przez przeglądarkę. Tworzenie aplikacji polega na dostarczeniu funkcji zwrotnych wywoływanych w reakcji na jakieś „interesujące” zdarzenia, takie jak kliknięcie przycisku, wczytanie strony itd. Gdy aplikacja chce zmienić wygląd lub zawartość strony, ustawia odpowiednio własności, a przeglądarka automatycznie aktualizuje prezentację na ekranie.

Niestety, zestaw gotowych obiektów trójwymiarowych w przeglądarkach nie jest bogaty, ponieważ ogranicza się do trójwymiarowych przekształceń CSS3. Wraz z pojawiением się technologii HTML5 twórcy przeglądarek internetowych przestali skupiać się na dostarczaniu gotowych obiektów wizualnych (tekstu, pasków przewijania, przycisków itd.), a skoncentrowali się na technikach kontrolowania sposobu renderowania i innych funkcjach na poziomie systemowym. Biblioteka WebGL i kanwa umożliwiają *narysowanie* wszystkiego, co się chce..., ale po drodze trzeba się dużo naprawać. Gdy wejdziemy już do świata elementu kanwy, okazuje się, że musimy samodzielnie zbudować graf sceny, utworzyć model zdarzeń, dostarczyć zestaw interakcji i zachowań oraz sporządzić animacje i zastosować przekształcenia. Oczywiście lepiej użyć gotowego systemu szkieletowego, który wszystkie te zadania wykona za nas.

Aplikacje WebGL reprezentują typowe problemy dotyczące projektowania systemów szkieletowych. Oprócz klasycznych trudności związanych z grafiką trójwymiarową, trzeba poradzić sobie z dodatkowymi kwestiami charakterystycznymi dla pracy na platformie przeglądarkowej. System szkieletowy WebGL powinien mieć większość z niżej opisanych cech.

Konfiguracja środowiska

System szkieletowy sprawdza, czy w danym środowisku obsługiwana jest technologia WebGL, i tworzy kontekst rysunkowy wraz z wszelkimi obiektami potrzebnymi do renderowania. Ponadto dodaje procedury obsługi zdarzeń DOM dla zmiany rozmiaru okna,

zdarzeń wywoływanych przez mysz i klawiaturę, utraty kontekstu WebGL i innych zdarzeń na stronie oraz przekazuje informacje do aplikacji, jeśli trzeba.

Sprawdzanie funkcjonalności i wyjścia awaryjne

System szkieletowy sprawdza, czy przeglądarka ma zaimplementowane różne funkcje, i ewentualnie uzupełnia braki lub stosuje rozwiązania awaryjne, takie jak np. dwuwymiarowe rysowanie na kanwie, gdy brakuje obsługi biblioteki WebGL.

Tworzenie domyślnej sceny

Szkielet tworzy pustą scenę, zazwyczaj zawierającą domyślne oświetlenie i kamery.

Pętla symulacyjna/wykonawcza

System szkieletowy dostarcza pętlę wykonawczą, podczas gdy aplikacja zawiera funkcje zwrotne dla zdarzeń oraz przesłania metody w celu implementacji potrzebnych w niej funkcji. System może dodatkowo zawierać definicję zegara lub modelu czasowego, którego należy się ścisłe trzymać w aplikacji.

Grafika i renderowanie

System szkieletowy dostarcza obiekty do renderowania grafiki. W przypadku szkieletów opartych na bibliotece Three.js może to po prostu oznaczać udostępnienie obiektów tej biblioteki.

Modele obiektów i zdarzeń

System szkieletowy określa spójny model własności obiektów, relacji między obiekttami w hierarchii lub grafie oraz współpracy obiektów przy użyciu zdarzeń, wywołań zwrotnych i metod dostępowych.

Interakcje

Szkielet automatycznie mapuje dane wejściowe z myszy i innych źródeł na obiekty na scenie i informuje aplikację o kliknięciu obiektu, jego przeciagnięciu itd.

Modele nawigacji/przeglądania

System szkieletowy może oferować jeden lub więcej modeli nawigacyjnych, czyli wysoko-kopoziomowych trybów poruszania kamerą w obrębie sceny (np. gra typu FPS), obsługi kolizji i poruszania się po terenie lub obracania kamery w celu zwrócenia się w kierunku wybranych obiektów. Ponadto system może zawierać wbudowaną logikę przełączania kamer i przechodzenia między scenami. Pierwszoosobowe modele nawigacyjne mogą także zawierać definicję awataru do reprezentacji użytkowników w środowiskach, w których występuje wielu użytkowników.

Zachowania i animacje

System szkieletowy zawiera gotowe zdefiniowane zachowania, od prostych rotacji i przesunięć stosowanych do obiektów w określonym czasie, po skomplikowane animacje wywołane za pomocą interakcji.

Fizyka

Niektóre systemy szkieletowe zawierają bogate modele fizyczne umożliwiające animowanie ciał w określonym kierunku i z określoną prędkością, uwzględnianie grawitacji, wykrywanie kolizji obiektów itd.

Wczytywanie zasobów

System szkieletowy automatycznie wczytuje modele, tekstury, filmy oraz dźwięki i informuje aplikację o załadowaniu wszystkich zasobów. Wysokiej jakości szkielety mogą

nawet zawierać serwerowe mechanizmy strumieniowego wczytywania danych trójwymiarowych i stopniowo dostarczające szczegółowy szatek o wysokiej rozdzielczości.

Narzędzia sceniczne

Wiele systemów szkieletowych zawiera mnóstwo narzędzi do szperania w grafie sceny. API zapytaniowe umożliwia znalezienie wszystkich obiektów wybranego typu albo mających identyfikatory pasujące do wzorca zdefiniowanego w wyrażeniu regularnym lub selektorze i wykonanie różnych działań, takich jak zmiana właściwości materiału, zastosowanie przekształceń trójwymiarowych, dodanie lub usunięcie elementów podrzędnych bądź też ukrycie lub odkrycie obiektów.

Zarządzanie pamięcią

Mimo że w aplikacjach JavaScript nieusuwanie przez programistę obiekty są usuwane automatycznie, w aplikacjach wzbogaconych należy zwracać szczególną uwagę na sposób i czas alokacji pamięci. W przeciwnym przypadku proces oczyszczania może być uruchamiany w nieodpowiednim czasie, powodując zmniejszenie liczby klatek i pogarszając walory użytkowe aplikacji. W niektórych szkieletach, aby rozwiązać ten problem, zaimplementowano „inteligentne” usługi zarządzania pamięcią. (Więcej na ich temat dowiesz się w rozdziale 12.).

Wydajność i elegancka redukcja funkcjonalności

System szkieletowy może automatycznie dostosowywać rozdzielczość lub jakość renderingu na podstawie liczby generowanych klatek lub zużycia zasobów w celu zapewnienia użytkownikowi jak najlepszych warunków pracy.

Mechanizm rozszerzalności

Dobry system szkieletowy nie zmusza programisty do posługiwania się wyłącznie komponentami z określonego zestawu. Dopuszcza również rozszerzanie funkcjonalności. W przypadku szkieletów WebGL oznacza to udostępnienie uchwytów do zachowań, umożliwienie nadpisywania interakcji oraz, co najważniejsze, zezwolenie na stosowanie własnych metod renderowania.

Lista jest dłuża, ale utworzenie wysokiej jakości aplikacji trójwymiarowej wymaga dużo pracy, przy której systemy szkieletowe mogą być bardzo pomocne. Istnieje kilka dobrych systemów do pracy z WebGL. Przyjrzymy się im w następnym podrozdziale.

Przegląd systemów szkieletowych dla WebGL

Systemy szkieletowe dla WebGL można podzielić na dwie kategorie: silniki gier i szkielety prezentacyjne. Silniki gier mają z reguły większe możliwości, ale są też trudniejsze w obsłudze. Natomiast szkielety prezentacyjne lepiej nadają się do tworzenia prostych aplikacji, takich jak osadzone na stronach modele obsługujące podstawowe formy interakcji. W tej części rozdziału przyjrzymy się systemom szkieletowym, które w czasie pisania tej książki były wciąż w fazie rozwojowej.

Silniki gier

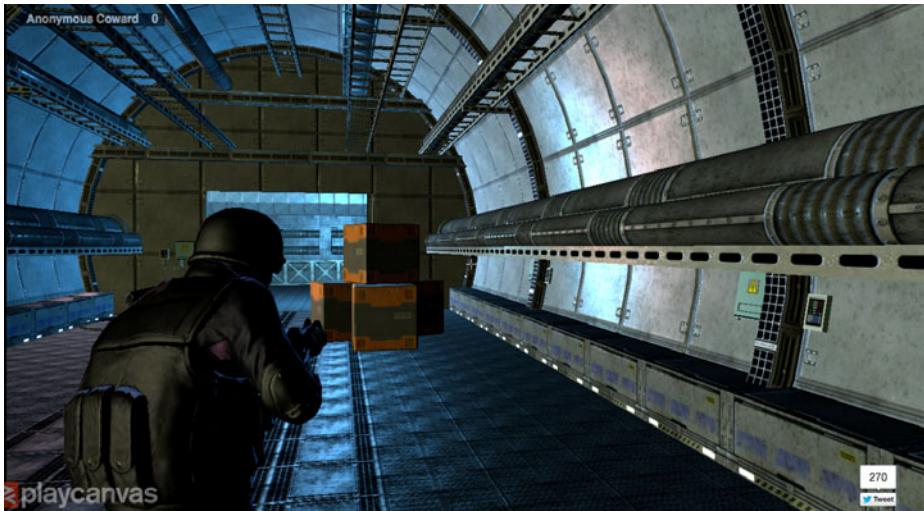
Jeśli chcesz przy użyciu WebGL utworzyć pierwszorzędną grę, możesz skorzystać z jednego z kilku silników gier, które pojawiły się w czasie kilku ostatnich lat. Różnica między silnikiem gier a systemem szkieletowym jest ulotna. Silnik jest zazwyczaj bardziej rozbudowany niż

przecienny szkielet. Z drugiej strony, silniki tworzy się z myślą o bardziej zaawansowanych programistach. Tworzenie gier to dziedzina łącząca kilka trudnych dyscyplin technicznych, a używane w niej silniki gier stanowią tego potwierdzenie.

Do wyboru mamy kilka silników gier WebGL. Mogą one różnić się zarówno pod względem funkcjonalności, jak i wymaganego do ich używania zasobu wiedzy. Niektóre silniki są otwartymi projektami. Niektóre są darmowe, a za inne trzeba uiścić zapłatę za licencję, hosting albo coś innego, np. za publikację pracy w kanałach właściciela. W żadnym z opisanych dalej silników gier nie używa się do renderowania biblioteki Three.js, ponieważ wszystkie w pełni kontrolują cały proces twórczy. Jest to jedna z kwestii, które należy rozważyć przy ocenianiu przydatności danego silnika dla własnego projektu.

PlayCanvas (<http://www.playcanvas.com/>)

PlayCanvas to bogato wyposażony silnik i narzędzie do publikacji efektów pracy w chmurze opracowane przez programistów z Londynu. Narzędzie umożliwia zespołową pracę nad sceną w czasie rzeczywistym, jest zintegrowane z usługami GitHub i Bitbucket oraz pozwala na publikowanie prac w mediach społecznościowych za pomocą jednego kliknięcia. Na rysunku 9.1 można zobaczyć zrzut ekranu z gry utworzonej na bazie tej platformy.



Rysunek 9.1. Strzelanka utworzona przy użyciu PlayCanvas

Turbulenz (<http://biz.turbulenz.com/developers/>)

Niezwykle rozbudowany, otwarty i całkowicie darmowy silnik gier w postaci pakietu SDK do pobrania ze strony internetowej. Właściciel pobiera opłaty jedynie za publikację prac w jego sieci (<http://biz.turbulenz.com/developers/>). Turbulenz jest najbardziej skomplikowane ze wszystkich opisywanych API, zawiera ogromną liczbę klas i wymaga poświęcenia dużej ilości czasu na naukę obsługi. Produkt przeznaczony zdecydowanie dla doświadczonych programistów gier.

Goo Engine (<http://www.gootechnologies.com/>)

Obecnie silnik ten jest w testowej fazie alfa. Na stronie internetowej wypisano listę typowych funkcji silników gier oraz pochwalono się przenośnością między platformami poprzez

WebGL. W witrynie brakuje jednak szczegółowych informacji technicznych i dotyczących licencji, ale zamieszczone wersje demo wyglądają pięknie — rysunek 9.2.



Rysunek 9.2. Zrzut ekranu z Peral Boy, gry przygodowej, której akcja rozgrywa się pod wodą, a utworzonej przy użyciu silnika Goo Engine (<http://www.gootechnologies.com/>)

Babylon.js (<http://www.babylonjs.com/>)

Niedawno do branży WebGL wkroczyła także firma Microsoft i od razu wniosła znaczący wkład. Babylon.js to łatwy w obsłudze silnik, który pod względem funkcjonalności i łatwości używania plasuje się gdzieś między biblioteką Three.js a najbardziej rozbudowanymi systemami. Na stronie demonstracyjnej zaprezentowano wiele przykładowych aplikacji, wśród których można znaleźć zarówno kosmiczne strzelanki, jak i przechadzki po budynkach o ciekawej architekturze.

KickJS (<http://www.kickjs.org>)

Silnik gier i biblioteka renderingu o otwartym kodzie źródłowym, będące owocem akademickiej pracy Mortena Nobla-Jørgensena. Jako że projekt wydaje się jeszcze niezbyt rozwinięty i raczej trudno znaleźć pomoc na jego temat, lepiej pomyśleć, zanim się go użyje. Dołączyłem go do listy, ponieważ z wszystkich opisanych silników gier w budowie tego najlepiej zastosowano nowoczesne techniki projektowania. (Więcej na ten temat piszę w opisie systemu Vizi). Poza tym silnik ten jest doskonałym materiałem referencyjnym dla tych, którzy chcą zaprojektować własny szkielet.

Jak widać, do wyboru jest wiele silników gier WebGL, których można używać nie tylko do tworzenia gier. Trzeba tylko pamiętać, że nauka ich obsługi jest trudna, więc lepiej się upewnić, czy wybrany produkt jest na pewno odpowiedni w danym przypadku. Do budowania prostszych aplikacji wizualnych można użyć jednego ze skromniejszych systemów szkieletowych, np. któregoś z opisanych w następnym punkcie.

Prezentacyjne systemy szkieletowe

Gry to tylko niewielki ułamek całego świata aplikacji trójwymiarowych, jakie można utworzyć przy użyciu WebGL. Używanie silników gier do budowania prostszych aplikacji, takich jak grafiki na strony internetowe, prezentacje produktów w sklepach internetowych czy wizualizacje danych naukowych, jest przesadą. Aplikacja prezentacyjna z reguły tylko wczytuje na stronę internetową prostą scenę, odtwarza kilka animacji oraz reaguje na działania użytkownika zmienianiem paru własności. Jak już wspomniałem, nawet te podstawowe czynności wymagają napisania dużej ilości dodatkowego kodu w Three.js i dlatego powinniśmy szukać pomocy wśród systemów szkieletowych. Oto kilka wertych uwag trójwymiarowych systemów prezentacyjnych.

tQuery

System tQuery (<http://jeromeetienne.github.io/tquery/>) to dzieło Jerome'a Etienne'a, właściciela popularnego bloga na temat nauki posługiwania się biblioteką Three.js (<http://learningthreejs.com/>). W serwisie tym można znaleźć wiele cennych wskazówek i sztuczek.

Wzorowana na jQuery biblioteka tQuery ma z założenia łączyć „wielkie możliwości Three.js z walorami użytkowymi API jQuery”. Innymi słowy, jest to bardzo proste API do grafu sceny Three.js. Można w nim tworzyć łańcuchy wywołań funkcji oraz używać wysokopoziomowych interaktywnych zachowań poprzez wywołania zwrotne. Nazywanie tQuery systemem szkieletowym to raczej przesada, ponieważ bliżej mu do biblioteki w stylu jQuery. Jeśli używasz biblioteki Three.js i chcesz oszczędzić sobie trochę pisania, warto zainteresować się tym projektem.

Na listingu 9.1 przedstawiono krótki przykład ilustrujący sposób wstawiania torusa na stronę internetową przy użyciu tQuery. Porównaj to z przykładami Three.js przedstawionymi w poprzednich rozdziałach, a zrozumiesz, jak dzięki systemom szkieletowym tworzenie prostej grafiki trójwymiarowej staje się igraszką.

Listing 9.1. Tworzenie prostej sceny przy użyciu biblioteki tQuery

```
<!doctype html><title>Minimalistyczna strona z kodem tQuery</title>
<script src="tquery-bundle.js"></script>
<body><script>
    var world = tQuery.createWorld().boilerplate().start();
    var object = tQuery.createTorus().addTo(world);
</script></body>
```

Podejście Etienne'a do projektowania można podsumować następującym zdaniem: „Niech tworzenie grafiki trójwymiarowej będzie jak najbardziej podobne do tworzenia grafiki dwuwymiarowej”. Programiści stron internetowych znają już jQuery, więc wystarczy dostarczyć im podobne API do 3D, aby szybko zwiększyli produktywność. Trudno się nie zgodzić.

Voodoo.js

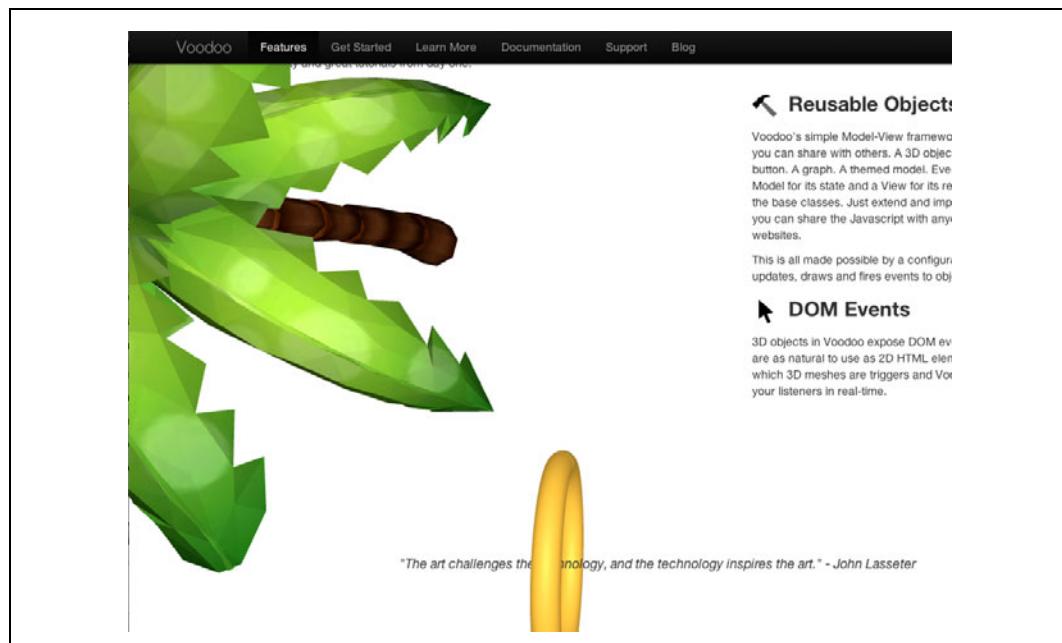
Misji dostarczenia grafiki trójwymiarowej pod strzechy podjął się programista z Seattle Brent Gunning, który zafascynował się możliwościami, jakie daje biblioteka WebGL, a jednocześnie rozczarował się tym, jak trudno się przy jej użyciu programuje. W efekcie powstał system szkieletowy Voodoo.js (<http://www.voodoojs.com/>). Celem, jaki Gunning chciał osiągnąć, było ułatwienie tworzenia i umieszczania na stronach internetowych treści trójwymiarowej. Zresztą sam programista złożył na swoim blogu następującą deklarację:

Obecnie grafika trójwymiarowa w internecie to tylko zwykła zabawka. Aby cokolwiek stworzyć, trzeba wykonać masę pracy, a i tak nie uzyska się niczego praktycznego. Co gorsza, sceny trójwymiarowe osadzamy na ograniczonych kanwach ściśle oddzielonych od treści dwuwymiarowej tylko dlatego, że mają dodatkowy wymiar. To projektowy koszmar i niesprawiedliwość. Postanowilem coś na to poradzić i w efekcie z dumą prezentuję pierwszą publicznie dostępną wersję biblioteki Voodoo 0.8.0 beta.

Plany Gunninga obejmują nie tylko łatwe programowanie za pomocą przeciągania i upuszczania obiektów, ale również przewidują cały system gotowych obiektów, składników, stylów wizualnych i motywów. Szkielet Voodoo.js zawiera niewielki zestaw klas o funkcjonalności obejmującej wczytywanie i przeglądanie modeli, interakcje z myszą oraz kilka konfigurowalnych opcji. System jest zbudowany na bazie biblioteki Three.js, więc teoretycznie powinno się dać go łatwo rozszerzać i dostosowywać za pomocą nowych typów obiektów. Na listingu 9.2 znajduje się przykład kodu ze strony głównej Voodoo.js, który tworzy trójwymiarowy obiekt i umieszcza go na stronie w elemencie `example2` przy użyciu tylko jednego wywołania funkcji. Łatwiej już chyba być nie może. Wynik można zobaczyć na rysunku 9.3.

Listing 9.2. Wstawianie trójwymiarowego obiektu na stronę przy użyciu Voodoo.js

```
new VoodooJsonModel({  
    elementId: 'example2',  
    jsonFile: '3d/tree.json',  
    offsetWidthMultiplier: 2.0 / 3.0,  
    scale: 50,  
    rotationX: Math.PI / 2.0,  
    rotationY: Math.PI / 2.0  
});
```



Rysunek 9.3. Strona główna Voodoo.js (<http://voodoojs.com/>) zawierająca kilka osadzonych obiektów trójwymiarowych

PhiloGL

PhiloGL (<http://www.senchalabs.org/philocgl/>) to eksperymentalny pakiet autorstwa specjalisty pracującego nad wizualizacją danych Nicolasa Garcii Belmonte'a opracowany w laboratoriach Sencha Inc. Celem PhiloGL jest „sprawienie, by programowanie przy użyciu WebGL było łatwe i przyjemne jak praca z każdym innym popularnym systemem szkieletowym”. Swoje poglądy na temat projektowania Garcia opisał we wpisie na blogu (<http://bit.ly/JYTRk7>). Mimo że szkielet ten jest dopiero w fazie eksperymentalnej, warto zwrócić na niego uwagę. Firma Sencha Inc. tworzy światowej jakości systemy szkieletowe do budowy interfejsów użytkownika, więc wie co nieco na temat tworzenia efektywnych interfejsów przy użyciu HTML5. Na listingu 9.3 pokazano przykład utworzenia prostej sceny przy użyciu PhiloGL. Aby zbudować scenę zawierającą kulę pokrytą tekstonią, wystarczy zdefiniować kilka obiektów JavaScript. Na stronie internetowej systemu znajduje się kilka działających przykładów, wśród których jest też przeniesiony cały zestaw kursów z witryny Learning WebGL (<http://www.learningwebgl.com/>).

Listing 9.3. Tworzenie prostej sceny trójwymiarowej przy użyciu PhiloGL

```
// Tworzy aplikację.
PhiloGL('canvasId', {
    camera: {
        position: {
            x: 0, y: 0, z: -7
        }
    },
    scene: {
        lights: {
            enable: true,
            ambient: { r: 0.5, g: 0.5, b: 0.5 },
            directional: {
                color: { r: 0.7, g: 0.7, b: 0.9 },
                direction: { x: 1, y: 1, z: 1 }
            }
        }
    },
    textures: {
        src: ['moon.gif']
    },
    events: {
        onClick: function(e) {
            /* Tu powinna znajdować się procedura obsługi zdarzeń. */
        }
    },
    onError: function() {
        alert("Wystąpił błąd podczas tworzenia aplikacji.");
    },
    onLoad: function(app) {
        // Robimy różne rzeczy z aplikacją...
        // Dodaje obiekt do sceny.
        scene.add(moon)
        // Animuje.
        setInterval(draw, 1000/60);
        // Rysuje scenę.
        function draw() {
            // Renderuje księżyc.
            scene.render();
        }
    }
});
```

Vizi — komponentowy system do tworzenia wizualnych aplikacji sieciowych

Czas bliżej przyjrzeć się technikom tworzenia aplikacji trójwymiarowych na bazie systemu szkieletowego. Chcemy mieć szerokie pole manewru, więc powinniśmy użyć odpowiednio ogólnego szkieletu. Oczywiście nie istnieje jeden system nadający się do wszystkiego, ale w różnych aplikacjach można znaleźć wiele cech wspólnych. Dlatego właśnie utworzyłem Vizi, czyli bazujący na WebGL system szkieletowy, przy użyciu którego powstały przedstawione w pozostałych rozdziałach przykłady. W tym podrozdziale znajduje się wprowadzenie do tego systemu.

Tło i metody projektowania

Tak jak twórcom tQuery, Voodoo.js i PhiloGL, nie podobał mi się sposób tworzenia aplikacji WebGL. Zaliczam się do największych fanów Mr.dooba, ale mimo to uważam, że sama biblioteka Three.js nie wystarczy do pisania aplikacji o wystarczająco wysokiej jakości, by używać ich w środowisku produkcyjnym. Większość opisywanych tu problemów została już rozwiązana lata temu, we wcześniejszych systemach i silnikach gier trójwymiarowych. Oczywiście podstawowe platformy zmieniły się, ale problemy w dużej mierze pozostały te same: po pierwsze, ładowanie treści sceny; po drugie, ustawnie kamery; po trzecie, narysowanie obiektów; po czwarte, przesuwanie obiektów wg czasomierzy i w reakcji na działania użytkownika; po piąte, oczyszczenie pola i powtórka czynności.

W czasie kilku ostatnich lat *zmieniło się* tylko jedno: sposób projektowania silników gier. Branża gier przez dwie dekady rozwinęła się do tego stopnia, że stała się katalizatorem do powstania chyba największych innowacji w historii komputerów. Zalicza się do nich sposób projektowania silników programowych. Mówiąc konkretnie, podejście polegające na wykorzystaniu klas i dziedziczenia zamieniono na architekturę opartą na komponentach i agregacji. (Może się wydawać, że różnica jest niewielka, ale wkrótce się przekonasz, że jej implikacje mają fundamentalne znaczenie). Wyposażony w doświadczenie zdobyte podczas pracy w różnych projektach i świeże spojrzenie bazujące na najlepszych w tym czasie praktykach programowania silników gier postanowiłem rozpocząć nową przygodę — projekt Vizi.

Celem, jaki mi przyświecał podczas tworzenia Vizi, było umożliwienie łatwego i szybkiego budowania *ciekawych* aplikacji trójwymiarowych. Jeśli chodzi o funkcjonalność, mój system mieści się między silnikiem gier typu PlayCanvas a systemem prezentacyjnym typu Voodoo.js. Typowym zastosowaniem Vizi może być aplikacja do konfiguracji produktów opisana na początku tego rozdziału: scena zawierająca kilka interaktywnych obiektów, dynamiczne aktualizacje na podstawie działań użytkownika, modele wczytywane na żądanie oraz zaawansowana nawigacja oparta na kamerze. Wydaje mi się, że taka funkcjonalność jest podstawą programowania przy użyciu WebGL i na niej się skupiłem.

Na rysunku 9.4 pokazano prototyp koncepcyjnego sklepu internetowego utworzony przy użyciu Vizi. Jest to wirtualna wystawa samochodowa. Panele z obrazami o wysokiej rozdzielcości wolno obracają się wokół środka sceny. Każdy z nich rzuca cień na otaczającą kratkę, która została subtelnie podkreślona za pomocą słabo widocznych linii siatki. Kilka sekund po załadunku strony na środek sceny wjeżdża szczegółowy model samochodu. Na powierzchni karoserii



Rysunek 9.4. Koncepcyjna wystawa samochodowa utworzona przy użyciu Vizi; twórcą modelu samochodu jest *be fast* (<http://www.turbosquid.com/Search/Artists/be-fast>), projekt wizualny i otoczenia jest autorstwa TC Changa (<http://www.tcchang.com/>)

odbija się wzór otoczenia. Kliknięcie panelu powoduje przeniesienie go do przodu i odtworzenie reklamy wideo. Na brzegach sceny znajdują się dwuwymiarowe elementy interfejsu użytkownika, dające dostęp do dodatkowych informacji i innych miejsc witryny. Jest to tylko koncepcyjny produkt, ale dobrze ilustruje główną cechę Vizi: połączenie treści dwuwymiarowej z treścią trójwymiarową, aby można było zastosować nowe rodzaje interakcji w sklepach internetowych i innych aplikacjach sieciowych.

Architektura systemu Vizi

Architektura systemu Vizi jest inspirowana nowoczesnymi zasadami projektowania silników gier. Wprawdzie gry trójwymiarowe są bardziej wymagające niż inne wizualne aplikacje, ale mają też z nimi wiele wspólnego. Wiele aplikacji także wymaga różnych funkcji dostępnych w silnikach gier. Przykładowo symulacja edukacyjna może wymagać kolizji, fizyki oraz awatarów, mimo że nic w niej nie dzieje się z dużą prędkością i nikt nikogo nie zabija.

Jedną z najważniejszych cech architektury systemu Vizi jest **model obiektowy oparty na komponentach**. Aktualnie odchodzi się od klasycznych projektów opartych na dziedziczeniu na rzecz agregacji komponentów. Istnieje podstawowy typ obiektów `Vizi.Object`, który stanowi praktycznie tylko kontener na komponenty, w których zaimplementowana jest większość funkcjonalności — np. komponent `Visual` zawiera geometrię i materiały, `Picker` obsługuje przekazywanie zdarzeń myszy dla poszczególnych obiektów, a `Camera` służy do oglądania. Systemy komponentowe są spójne i można je elastycznie implementować, mają też wysoki potencjał w zakresie wielokrotnego użytku składników. Ponadto są kluczowe dla rozszerzalności.

Oto pozostałe najważniejsze cechy systemu Vizi.

Obiekt aplikacji

Singletonowy obiekt aplikacji tworzy kontekst WebGL, procedury obsługi zdarzeń DOM oraz inicjuje bibliotekę Three.js. Obiekt ten ma zaimplementowaną pętlę wykonawczą.

Inne obiekty tylko dodaje się do aplikacji i można wykonać aktualizację w każdej klatce animacji.

Model simulacji i zdarzeń

Wszystkie obiekty korzystają z jednej standardowej bazy czasowej. Zdarzenia są wyzwalane w ścisłe określonym czasie i zgodne z pewnymi regułami. Obiekty publikują zdarzenia, które inne obiekty mogą subskrybować. Obiekty mogą subskrybować zdarzenia za pomocą procedur nasłuchowych lub bezpośrednio „połączyć” się ze zdarzeniami innych obiektów w łańcuchu zachowań. To umożliwia tworzenie zachowań i interakcji w bardzo zwięzły sposób.

Architektura usługowa

Wszystkie podsystemy są usługami w postaci czarnych skrzynek. Podczas inicjacji i wykonywania aplikacja odsyła do usług, takich jak Time, Events, Graphics i Input, bez zważania na to, co dokładnie te usługi robią. To ułatwia dodawanie nowych usług, których brakuje w rdzennej części systemu.

Grafika

Do rysowania grafiki zawsze jest używana biblioteka Three.js. Nie ukryłem jej, została zintegrowana z systemem Vizi, który opakowuje jej obiekty w taki sposób, by obiekty Vizi mogły łatwo się z nią komunikować.

Interakcje

Vizi obsługuje zdarzenia myszy dla poszczególnych obiektów i do obsługi wykrywania zdarzeń wykorzystuje klasę Projector z biblioteki Three.js. W ten sposób powstał znacznie bardziej przejrzysty interfejs do obsługi myszy i urządzeń dotykowych. Ponadto Vizi zawiera gotowe obiekty interakcji implementujące różne rodzaje przeciągania (np. na powierzchni kuli).

Zachowania

System Vizi zawiera wiele gotowych zachowań, które automatycznie obracają, przesuwają, odbijają, podświetlają i modyfikują obiekty na inne sposoby.

Wysokopoziomowy model widoku

W systemie Vizi istnieje możliwość zdefiniowania wielu kamer i łatwego przełączania się między nimi. Ponadto system ten udostępnia tryby nawigacyjne do różnych celów, np. oglądania obiektów, rozgrywek pierwszoosobowych, pokazów architektury itd.

Łatwość dostosowywania

Dostosowane do indywidualnych potrzeb składniki mogą implementować nowe zachowania, interakcje i kontrolery kamery — praktycznie wszystko, co może być potrzebne. Komponenty są obiektami JavaScript dziedziczącymi po obiekcie `Vizi.Component`. Można łatwo utworzyć nowy typ komponentu i dodać do niego własną funkcjonalność poprzez przesłonięcie metod `realize()` i `update()`. Komponenty można nieinwazyjnie dodawać do istniejących obiektów, aby wzbogacić je w nowe funkcje, których twórca systemu nawet sobie nie wyobrażała.

Prefabrykaty

Programista używający Vizi może tworzyć typy wielokrotnego użytku zawierające zbiory obiektów. Jako że system Vizi ma strukturę komponentową, typów nie buduje się w nim przy użyciu klas JavaScript, tylko jako kolekcje obiektów zwane **prefabrykatami** (ang. *prefabs*). Prefabrykaty zazwyczaj zawierają hierarchie obiektów gry i ich komponenty, przy najmniej jednego subskrybenta zdarzeń lub połączeń oraz skrypt negocjujący interakcje między wszystkimi składnikami.



Komponentowa architektura systemu Vizi powstała pod dużym wpływem prac przedstawionych w nowatorskiej książce Jasona Gregory'ego pt. *Game Engine Architecture* (<http://www.gameenginebook.com/>). Jest to obowiązkowa pozycja dla każdego profesjonalnego projektanta silników i systemów szkieletowych. Wachlarz opisywanych w niej zagadnień jest szeroki, a w naszym kontekście najważniejsze są części dotyczące architektur obiektowych. Gregory jest gorącym zwolennikiem odejścia od klasycznego dziedziczenia klasowego do systemów komponentowych. Systemy takie są — ogólnie rzecz biorąc — bardziej elastyczne i rozszerzalne oraz pozwalały uniknąć wielu problemów typowych dla systemów klasowych, zwłaszcza tych bardziej rozbudowanych.

Ponadto prace nad Vizi były inspirowane budową systemu Unity (<http://unity3d.com/>), czyli najpopularniejszego komercyjnego silnika gier używanego przez niezależnych twórców i małe studia. Unity reprezentuje bardzo udaną realizację zasad projektowania komponentowych silników gier opisanych w książce Gregory'ego. Aktualnie Unity nie obsługuje WebGL. System ten powstał długo przed pojawiением się technologii HTML5 i dlatego wykorzystuje własny język skryptowy oraz system renderowania. Gdyby Unity obsługiwał HTML5 i WebGL, być może nie trzeba by było tworzyć systemu Vizi.

Podstawy obsługi systemu Vizi

Aby użyć systemu Vizi, należy pobrać jego najnowszą wersję z repozytorium GitHub (<https://github.com/tparisi/Vizi>). W folderze *engine/build/* znajduje się kilka plików. Skopiuj plik *vizi.js* (niezminimalizowana wersja przeznaczona do celów testowych) lub *vizi.min.js* (wersja zminalizowana) i wklej go w drzewie swojego projektu.

Następnie dołącz skrypt Vizi do strony internetowej. To wszystko, aby rozpocząć pracę:

```
<script src="../<ścieżka_do_vizi>/vizi.js"></script>
```



Dostępnych jest kilka rodzajów systemu Vizi. Do wymienionych plików dołączone są dodatkowo biblioteki pomocnicze, takie jak Three.js, Tween.js, RequestAnimationFrame.js i kilka wspomagających obiektów bazujące na Three.js. Jeśli nie chcesz dołączać tych wszystkich zależności, możesz użyć wersji *nodeps*, a potrzebne dodatki dołączyć w innym miejscu na stronie. Oczywiście musisz uważać na wersje bibliotek. Szczegółowe informacje na ten temat znajdują się w pliku *README* i uwagach. Natomiast w dodatku znajdziesz wskazówki, jak przygotować własną wersję systemu Vizi.

Prosta aplikacja Vizi

Spójrzmy na konkretny przykład ilustrujący możliwości systemu Vizi. Otwórz w przeglądarce internetowej plik *r9/vizicube.html*. Na ekranie powinno pojawić się coś znajomego. Jest to pokryta tekstrurą kostka z rozdziałów 2. i 3., tylko tym razem napisana przy użyciu Vizi. Porównaj kod widoczny na listingu 9.4, za pomocą którego przy użyciu Vizi powstaje i uruchamia się trójwymiarowa scena, z analogicznym kodem Three.js przedstawionym na listingu 3.1 w rozdziale 3.

Listing 9.4. Prosta aplikacja Vizi: obracająca się kostka

```
<script type="text/javascript">  
$(document).ready(function() {  
    // Tworzy obiekt aplikacji Vizi.
```

```

var container = document.getElementById("container");
var app = new Vizi.Application({ container : container });

// Tworzy kostkę z cieniowaniem Phonga i teksturą.
var cube = new Vizi.Object;
var visual = new Vizi.Visual(
    { geometry: new THREE.CubeGeometry(2, 2, 2),
      material: new THREE.MeshPhongMaterial(
          {map:THREE.ImageUtils.loadTexture(
              "../images/webgl-logo-256.jpg")})
    });
cube.addComponent(visual);

// Dodaje rotację, aby nieco ożywić kostkę.
var rotator = new Vizi.RotateBehavior({autoStart:true});
cube.addComponent(rotator);

// Obraca kostkę w kierunku użytkownika, aby pokazać trójwymiarowość.
cube.transform.rotation.x = Math.PI / 5;

// Dodaje światło, aby było widoczne cieniowanie.
var light = new Vizi.Object;
light.addComponent(new ViziDirectionalLight);

// Dodaje kostkę i światło do sceny.
app.addObject(cube);
app.addObject(light);

// Uruchamia aplikację.
app.run();
}
);
</script>

```

Do utworzenia obracającej się i pokrytej teksturą kostki przy użyciu systemu Vizi wystarczy około 40 wierszy kodu, zamiast 80, jak w przypadku używania biblioteki Three.js. Jednak ilość kodu to nie wszystko, o czym wkrótce się przekonasz. Przeanalizujmy nasz przykład. Najpierw tworzymy nowy obiekt aplikacji typu `Vizi.Application` i przekazujemy mu element kontenera. Ta prosta czynność powoduje, że w systemie zaczyna działać się wiele rzeczy: zostają utworzone obiekt renderujący Three.js oraz pusta scena Three.js z domyślną kamerą, procedurami obsługi zdarzeń dotyczących zmiany rozmiaru strony, myszy itd. Wszystko to trzeba by dodać ręcznie za pomocą wywołań API DOM lub funkcji Three.js, ale Vizi robi to za nas. Jeśli chcesz się dowiedzieć, jak dokładnie wszystko się odbywa, zajrzyj do plików `core/application.js` i `graphics/graphicsThreeJS.js` w projekcie Vizi. Jest na co popatrzeć.

Następnie dodajemy do sceny obiekty. W tym momencie do gry wchodzi komponentowy model obiektowy systemu Vizi. Każdy obiekt na scenie powstaje jako obiekt typu `Vizi.Object`, a następnie dodaje się do niego różne komponenty. W przypadku kostki tworzymy obiekt `Vizi.Visual` przy użyciu geometrii Three.js i teksturowanego materiału Phonga. Zwróc uwagę, że system Vizi nie definiuje własnych obiektów graficznych, lecz używa tych z biblioteki Three.js. Jest to świadomy wybór projektowy. Nie ukryłem biblioteki Three.js, tylko postarałem się ją w pełni wykorzystać, aby ułatwić tworzenie wszystkich typów obiektów wizualnych.

Po utworzeniu komponentu wizualnego i dodaniu go do obiektu dokładamy zachowanie. W tym miejscu zaczyna się magia Vizi. System ten zawiera gotowy zestaw zachowań, które można zastosować do obiektu, dodając je jako komponenty. W tym przypadku dodajemy `Vizi.RotateBehavior` i ustawiamy jego flagę `autoStart` na `true`, dzięki czemu obiekt zacznie się obracać od razu po uruchomieniu aplikacji.

Kostkę pochylimy w stronę użytkownika, aby było widać, że jest trójwymiarowa. W systemie Vizi należy w tym celu zmienić właściwość obrotu składnika przekształcenia obiektu.

```
// Obraca kostkę w kierunku użytkownika, aby pokazać trójwymiarowość.  
cube.transform.rotation.x = Math.PI / 5;
```

Aby było wygodnie, dla każdego obiektu Vizi automatycznie tworzony jest domyślny składnik przekształcenia. Jest on odpowiedni do użytku w większości przypadków. Jeśli składnik przekształcenia jest niepotrzebny, można ustawić na false opcjonalną flagę autoCreateTransform konstruktora klasy `Vizi.Object`.

Aby ukazać cieniowanie Phonga na kostce, oświetlenie do sceny dodaliśmy jako osobny obiekt ze składnikiem `ViziDirectionalLight`. W kolejnych rozdziałach pokazuję, jak uniknąć konieczności bezpośredniego tworzenia światła poprzez użycie gotowego szablonu aplikacji zawierającego własną konfigurację oświetlenia. Na koniec można uruchomić aplikację, co robimy przy użyciu metody `run()`. To wystarczy. Nie trzeba pisać własnej funkcji `requestAnimationFrame()`, aby ręcznie aktualizować obrót kostki w każdym cyklu. Aplikacja po prostu działa.

Interakcja

Opisane w poprzednich rozdziałach przykłady nie były interaktywne. Częściowo ma to związek z tym, że jeszcze po prostu nie dotarliśmy do opisu odpowiednich technik, ale częściowo jest to spowodowane też tym, że w bibliotece Three.js implementacja interaktywności jest pracochłonna. Biblioteka ta udostępnia „projektor”, czyli specjalny obiekt pozwalający na sprawdzenie, nad którymi obiektami aktualnie znajduje się kursor. Brakuje w nim jednak interfejsu zdarzeniowego czy modelu do obsługi klikania i przeciągania. Lukę tę uzupełnia system Vizi, który zawiera implementację mechanizmu rozpoznawania tego, co znajduje się pod kursem oraz automatycznego przekazywania tych informacji.

Zaimplementujemy prostą interaktywność w poprzednim przykładzie. Sprawimy, że kostka, zamiast automatycznie obracać się po wczytaniu strony, będzie się obracała w reakcji na ruch kursem nad nią. Otwórz plik `r09/vizicubeinteractive.html` w przeglądarce internetowej. Kod źródłowy opisywanego przykładu jest widoczny na listingu 9.5. Pogrubione wiersze zawierają modyfikacje. Tym razem nie ustawiamy opcji `autoRotate` podczas tworzenia zachowania, więc rotacja nie nastąpi automatycznie po załadowaniu aplikacji. Następnie dodajemy do obiektu kostki nowy rodzaj składnika — `ViziPicker`. Składnik ten definiuje typowy zestaw zdarzeń myszy — `over`, `up`, `out`, `down` — które automatycznie przekazuje, gdy kursor znajduje się nad widocznym obszarem w obrębie obiektu zawierającego składnik `ViziPicker`. Pozostało już tylko dodać procedury nasłuchu zdarzeń, a następnie uruchomić i zatrzymać rotację w odpowiedzi na najechanie kursem na kostkę i jego zabranie nad kostką.

Listing 9.5. Implementacja interakcji z myszą przy użyciu składnika Vizi.Picker

```
<script type="text/javascript">  
  
$(document).ready(function() {  
  
    // Utworzenie obiektu aplikacji Vizi.  
    var container = document.getElementById("container");  
    var app = new Vizi.Application({ container : container });  
  
    // Utworzenie kostki pokrytej teksturą i z cieniowaniem Phonga.  
    var cube = new Vizi.Object;  
    var visual = new Vizi.Visual(
```

```

        { geometry: new THREE.CubeGeometry(2, 2, 2),
          material: new THREE.MeshPhongMaterial(
            {map:THREE.ImageUtils.loadTexture(
              "../images/webgl-logo-256.jpg")})
      });

      cube.addComponent(visual);

      // Dodanie rotacji, aby ożywić kostkę.
      var rotator = new Vizi.RotateBehavior;
      cube.addComponent(rotator);

      // Umożliwienie wyboru kostki.
      var picker = new Vizi.Picker;
      cube.addComponent(picker);

      // Połączenie wybieraka z rotatorem, obiekt jest obracany tylko wtedy, gdy najedziemy na niego kursorem.
      picker.addEventListener("mouseover", function() {
        rotator.start(); });
      picker.addEventListener("mouseout", function() {
        rotator.stop(); });

      // Obrócenie kostki w kierunku użytkownika, aby pokazać trójwymiarowość.
      cube.transform.rotation.x = Math.PI / 5;

      // Dodanie światła, aby pokazać cieniowanie.
      var light = new Vizi.Object;
      light.addComponent(new ViziDirectionalLight);

      // Dodanie kostki i oświetlenia do sceny.
      app.addObject(cube);
      app.addObject(light);

      // Uruchomienie aplikacji.
      app.run();
    }
  );
</script>

```

To było łatwe. Aby dowiedzieć się, jak naprawdę działa ten program, musimy sprawdzić, w jaki sposób trójwymiarowe obiekty wykrywane są pod myszą. W Vizi do implementacji tego mechanizmu wykorzystano klasę Three.js THREE.Projector. Jej kod jest skomplikowany. Na listingu 9.6 pokazano kod metody objectFromMouse() podsystemu graficznego Vizi. Metoda ta zwraca obiekt Vizi znajdujący się pod kursem, jeśli taki znajdzie. Proces ten składa się z kilku etapów.

1. Najpierw przekształcamy elementowe współrzędne kurSORA z właściwości elementX i elementY zdarzenia na współrzędne obszaru widoku w zakresie od -0,5 do 0,5 w każdym wymiarze, jednocześnie odwracającą współrzędną y, aby odpowiadała trójwymiarowemu układowi współrzędnych. (Właściwości elementX i elementY nie są standardowymi właściwościami DOM zdarzeń myszy — zostały obliczone w procedurze Vizi obsługi zdarzeń DOM przed przekazaniem danych do tej metody).
2. Otrzymane współrzędne obszaru widoku pozycji kurSORA musimy przekształcić na trójwymiarową pozycję bezpośrednio pod myszą, ale w połowie drogi — w objętość widokową. Dane te zapisujemy w zmiennej vector.
3. Następnie przekształcaną wyrażoną względem obszaru widoku pozycję kurSORA z **przestrzeni kamery na przestrzeń świata**. Dzięki temu poznajemy pozycję kurSORA myszy „w świecie”.

W tym celu wywołujemy metodę `unprojectVector()` naszego obiektu projekcyjnego. (Obiekt ten został utworzony podczas inicjacji systemu graficznego i jest typu `THREE.Projector`).

4. Znając pozycję kurSORA „w świecie”, możemy utworzyć **promień** biegący od pozycji kamery w przestrzeni świata do pozycji kurSORA. Wszystko, co przecina ten promień, znajduje się „pod myszą”. (Przepraszam za te cudzysłowy, ale dość luźno posługuję się tu pojęciami „wewnętrz” i „pod”). Sprawdzanie przecięć promienia jest wykonywane przez metodę `intersectObjects()` obiektu `THREE.Raycaster`. Przyjmuje ona listę obiektów i zwraca listę tych, które przecinają promień w kolejności od przodu.
5. Na koniec pobieramy z listy pierwszy widoczny element, który reprezentuje obiekt znajdujący się na wierzchu.

Listing 9.6. Implementacja Vizi mechanizmu wyszukiwania elementów znajdujących się pod kursorem, oparta na klasie THREE.Projector

```
Vizi.GraphicsThreeJS.prototype.objectFromMouse = function(event)
{
    var eltx = event.elementX, elty = event.elementY;

    // Tłumaczy współrzędne klienta na vpx i vpy.
    var vpx = ( eltx / this.container.offsetWidth ) * 2 - 1;
    var vpy = - ( elty / this.container.offsetHeight ) * 2 + 1;

    var vector = new THREE.Vector3( vpx, vpy, 0.5 );

    this.projector.unprojectVector( vector, this.camera );

    var pos = new THREE.Vector3;
    pos = pos.applyMatrix4(this.camera.matrixWorld);

    var raycaster = new THREE.Raycaster( pos, vector.sub( pos ).normalize() );

    var intersects = raycaster.intersectObjects( this.scene.children,true );

    if ( intersects.length > 0 ) {
        var i = 0;
        while(!intersects[i].object.visible)
        {
            i++;
        }

        var intersected = intersects[i];

        if ( i >= intersects.length )
        {
            return { object : null, point : null, normal : null };
        }

        return (this.findObjectFromIntersected(intersected.object,
                                                intersected.point, intersected.face.normal));
    }
    else
    {
        return { object : null, point : null, normal : null };
    }
}
```

Ostrzegam, że to nie jest proste. Jest to ten rodzaj kodu, który pisze się tylko raz, a dzięki takim systemom jak Vizi nie trzeba go w ogóle pisać.

Dodawanie większej liczby zachowań

W systemie Vizi można w łatwy sposób dodać do obiektu wiele zachowań. Wykorzystamy poprzedni przykład, aby dodać zachowania do kostki. Sprawimy, że najechanie kursorem na sześciian będzie powodować zmianę jego koloru na jasnoniebieski, a kliknięcie będzie powodować jego obracanie i schodzenie z widoku w podskokach. Ponowne kliknięcie zatrzyma ten ruch. Aby dokładnie zobaczyć, jak to działa, otwórz w przeglądarce plik [r9/vizicubebehaviors.html](#). Na listingu 9.7 przedstawiono odpowiedni kod źródłowy z pogrubionymi wierszami zawierającymi modyfikacje.

Listing 9.7. Dodawanie wielu zachowań

```
// Dodawanie kilku zachowań.  
var rotator = new Vizi.RotateBehavior;  
var bouncer = new Vizi.BounceBehavior({loop:true});  
var mover = new Vizi.MoveBehavior({loop:true, duration:2,  
    moveVector:new THREE.Vector3(0, 0, -2)});  
cube.addComponent(rotator);  
cube.addComponent(bouncer);  
cube.addComponent(mover);  
  
// Umożliwienie, aby dalo się wybierać kostkę.  
var picker = new Vizi.Picker;  
cube.addComponent(picker);  
  
// Dodanie koloru na zmianę po najechaniu kursorem.  
var highlight = new Vizi.HighlightBehavior(  
    {highlightColor:0x88eff});  
cube.addComponent(highlight);  
  
// Połączenie obiektu wybierającego z rotatorem.  
// Zmienia kolor po najechaniu kursorem, zmienia zachowania po kliknięciu.  
picker.addEventListener("mouseover", function() {  
    highlight.on(); });  
picker.addEventListener("mouseout", function() {  
    highlight.off(); });  
picker.addEventListener("mouseup", function() {  
    rotator.toggle();  
    bouncer.toggle();  
    mover.toggle(); });
```

Dodanie zachowań do tego przykładu jest tak samo łatwe jak dodanie kolejnych składników. Dodaliśmy też procedurę obsługi zdarzenia zabrania kurSORA znad obiektu, która wywołuje metodę `toggle()` na każdym z zachowań i je włącza lub wyłącza w reakcji na kliknięciu myszą. To wszystko. Na rysunku 9.5 widać dobrze znaną kostkę pokrytą teksturą z logo WebGL. Figura ta obraca się, kręci, skacze i ucieka w dal.

Ten prosty przykład stanowi ilustrację możliwości, jakie dają systemy typu Vizi. System ten poznamy bardziej szczegółowo w kolejnych rozdziałach, w których przestudiujemy różne aplikacje i techniki.



System Vizi nie jest jeszcze ukończony. Aktualnie jest to biblioteka w wersji 0.6 lub 0.7, tzn. zawiera już wiele funkcji, ale wciąż wiele jest do zrobienia. Pracuję nad nim w wolnym czasie oraz gdy uda mi się go użyć w jakimś projekcie WebGL. Mam nadzieję, że zanim ta książka ukaże się w druku, zdązę opublikować wersję 1.0.



Rysunek 9.5. Pokryta teksturową kostka z obsługą zdarzeń i interakcją z myszą utworzona przy użyciu systemu Vizi

Podsumowanie

W tym rozdziale poznaleś różne silniki i systemy szkieletowe służące do tworzenia aplikacji trójwymiarowych. W bibliotece Three.js, mimo że jest bardzo przydatna, brakuje wielu konstrukcji ułatwiających codzienną pracę programisty, takich jak wysokopoziomowe zachowania i interakcje, gotowe konfiguracje i wiele innych, które są potrzebne w prawie każdej aplikacji trójwymiarowej.

Dodatkowo miałeś okazję przyjrzeć się silnikom gier i systemom prezentacyjnym. Branża WebGL jest nowa i dynamicznie się rozwija, czego odzwierciedleniem jest stan używanych w niej systemów szkieletowych. Do wyboru mamy szeroki wachlarz rozwiązań, ale z każdym wiąże się jakiś kompromis, np. trzeba zapłacić, zrezygnować z pewnych funkcji albo pogodzić się z komplikacjami.

Na końcu pokazałem, jak pracuje się z systemem szkieletowym na przykładzie systemu Vizi. Jest to nowe rozwiązanie, które utworzyłem w celu przestudiowania kwestii związanych z budową systemów szkieletowych oraz aby mieć na czym oprzeć przykłady przedstawione w tej książce. Na ich podstawie pokazuję, jak system szkieletowy może wybawić programistę od konieczności wykonywania powtarzalnych czynności, dzięki czemu może on lepiej skupić się na pisaniu kodu i samej aplikacji. Vizi to tylko jeden z wielu systemów szkieletowych. Warto też zainteresować się innymi albo zbudować własny. Niezależnie od tego, czy kupisz system, czy utworzysz własny, jeśli będziesz pisać wysokiej jakości aplikacje trójwymiarowe, to wcześniej czy później na pewno natkniesz się na problemy opisane w tym rozdziale.

Budowa prostej aplikacji trójwymiarowej

Do tej pory koncentrowaliśmy się na podstawach: podstawowych API i architekturze HTML5, bibliotekach i systemach szkieletowych JavaScript oraz narzędziach do tworzenia treści trójwymiarowej. Teraz zdobyte wiadomości wykorzystamy w praktyce. W pozostałych rozdziałach nie będziemy już poznawać nowych API ani narzędzi, tylko skupimy się na praktycznych aspektach budowania aplikacji.

Zacznijmy od utworzenia jednego z najprostszych rodzajów aplikacji trójwymiarowych, będą to konfigurator i przeglądarka produktów. Aplikacja taka zazwyczaj zawiera interaktywny trójwymiarowy model jakiegoś produktu oraz bogato wyposażony interfejs użytkownika, za pomocą którego można dokładnie obejrzeć towar i zmienić jego właściwości. Internetowe konfiguratory produktów nie są niczym nowym. Pierwsze były dwuwymiarowe, które zostały zastąpione przez 2,5- i 3-wymiarowe rozwiązania we Flashu. Dopiero niedawno pojawiły się trójwymiarowe aplikacje oparte na kanwie i WebGL. Taki konfigurator produktów może być bardzo skutecznym narzędziem marketingowym (tzn. interaktywną reklamą) albo służyć do konfigurowania i zamawiania produktów w sklepie internetowym.

Na rysunku 10.1 pokazano koncepcyjną aplikację przedstawiającą „samochód przyszłości”, którą można wypróbować, otwierając plik *r10/futurgo.html*. Za pomocą myszy można obracać pojazd, a przy użyciu kółka myszy powiększać go i zmniejszać. Gdy nad któryś z części samochodu znajdzie się kurSOR, w chmurce wyświetlane są dodatkowe informacje o tej części. Po prawej stronie znajdują się przyciski umożliwiające zjazdzenie do środka samochodu i wyłączenie obracania kół. Można nawet zmienić kolor pojazdu na swój ulubiony. To „stylowe urządzenie transportowe” wyznacza nowy kierunek w motoryzacji osobowej. Po części skuter, po części wózek do golfa, po części inteligentny samochód, a wszystko uzupełnione najnowocześniejszą elektroniką — to jest Futurgo!

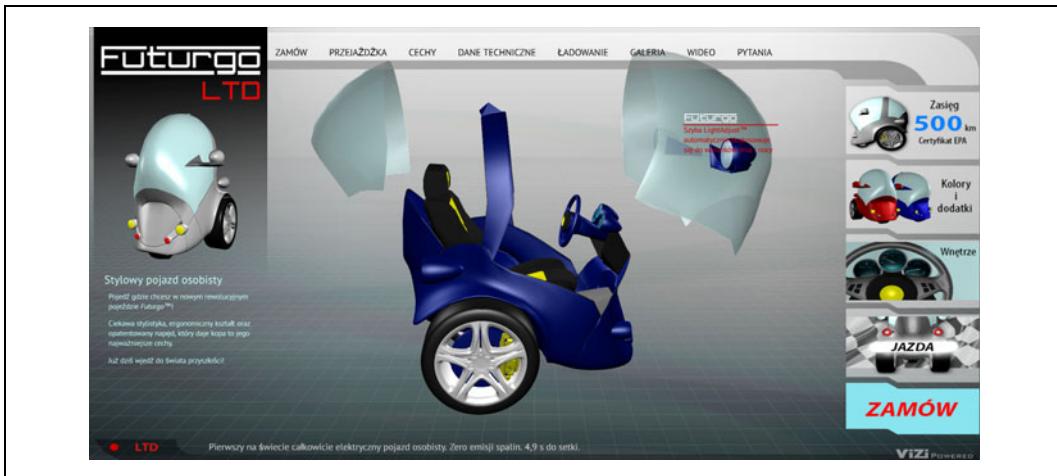
Ten zabawny, zmyślony przykład konfiguratora produktu ilustruje najważniejsze kwestie dotyczące tworzenia trójwymiarowej strony produktu. Oto one.

Projekt aplikacji

Konieczne jest zaprojektowanie wyglądu trójwymiarowego modelu i dwuwymiarowej treści strony oraz zaplanowanie sposobu interakcji z użytkownikiem.

Utworzenie treści trójwymiarowej

Za pomocą narzędzi typu Autodesk Maya należy utworzyć modele i animacje, które następnie trzeba przekonwertować na format nadający się do użytku w aplikacji internetowej.



Rysunek 10.1. Koncepcyjny samochód Futurgo: trójwymiarowa strona produktu

Podgląd i testowanie treści trójwymiarowej

Należy utworzyć zestaw narzędzi do sprawdzania, czy wyeksportowana treść trójwymiarowa będzie działać w aplikacji (np. czy poprawnie wygląda i czy animacje przebiegają w zaplanowany sposób).

Integracja treści trójwymiarowej z aplikacją

Konieczna jest integracja treści trójwymiarowej (która została już wcześniej sprawdzona) z dwuwymiarową treścią strony internetowej i pozostały kodem aplikacji.

Zaimplementowanie trójwymiarowych zachowań i interakcji

Aby ożywić treść trójwymiarową, należy zaimplementować kilka zachowań i funkcji interaktywnych, włącznie z efektem zanikania, karuzelą, efektami zmiany obrazu pod kursem, animacjami uruchamianymi przez działania użytkownika i interaktywnym zmienianiem kolorów.

Wszystkie wymienione czynności muszą być wykonywane w powtarzalnym procesie. Gdy zostanie wykryty jakiś błąd, zmiany, które będzie trzeba wprowadzić w wizualnej warstwie aplikacji (zwłaszcza w treści trójwymiarowej), nie powinny pociągać za sobą konieczności przepisywania kodu programu.

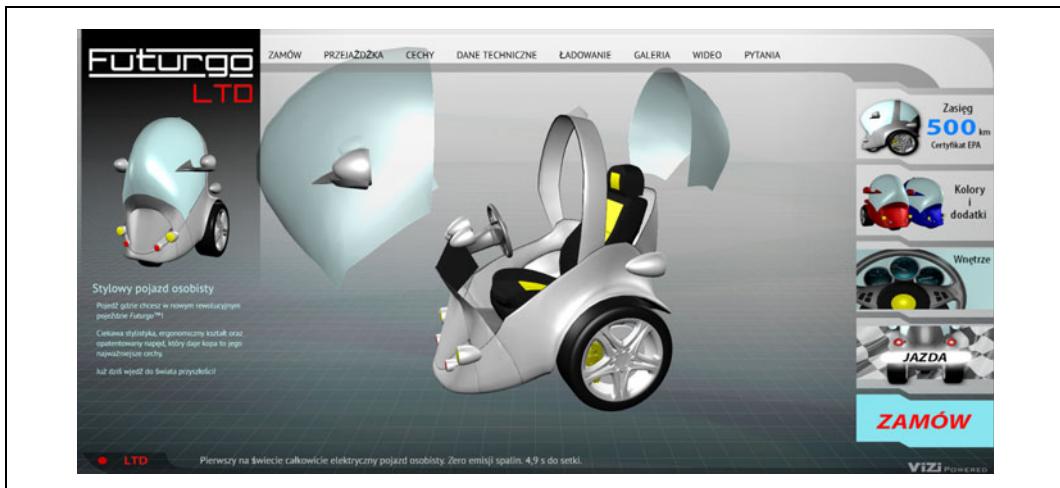
Do budowy strony pojazdu Futurgo i obsługujących ją narzędzi wykorzystamy opisany w rozdziale 9. system Vizi. Ten bazujący na bibliotece Three.js system oferuje gotowe zachowania i obiekty, dzięki którym praca programisty staje się łatwiejsza. Trzeba też napisać mniejszą ilość kodu źródłowego, aby uzyskać zamierzony efekt. Ponadto użyjemy otwartych eksporterów i konwerterów plików w celu przekonwertowania treści z programu Maya na format nadający się do użycia w internecie. Do dzieła.

Projektowanie aplikacji

Do współpracy nad aplikacją Futurgo zaprosiłem TC Changę, artystę zajmującego się grafiką trójwymiarową. Chcieliśmy stworzyć wizualizację produktu, która byłaby zarazem i zabawna, i futurystyczna. W efekcie powstał pojazd mający cechy osobistych urządzeń transportowych,

takich jak Segway, i samochodów, np. karoserię i przednią szybę. Nie mamy pojęcia, czy realizacja tego pomysłu byłaby w ogóle możliwa, nie wspominając nawet o kwestiach dopuszczenia tego czegoś do ruchu, ale świetnie się bawiliśmy podczas pracy nad tym projektem!

Po wstępnej burzy mózgów i narysowaniu kilku wstępnych szkiców TC opracował całą makietę modelu, którą widać na rysunku 10.2. Należy zwrócić uwagę, jak niewiele różni się od ostatecznej wersji¹. Wykorzystaliśmy wyeksportowane do formatu PNG zasoby Photoshopa, a treść trójwymiarowa pochodziła wprost z programu Maya, dzięki czemu wygląda identycznie z wersją renderowaną na bieżąco za pomocą biblioteki Three.js. Trochę czasu zajęło nam sprawienie, by wyeksportowana treść trójwymiarowa była zgodna z oryginalnym renderingiem, o czym szerzej piszę nieco dalej. Jednak efekt był wart zachodu. Cały ten rozdział jest o tym, jak osiągnąć tak wysoki stopień wierności odwzorowania oraz jak płynnie połączyć dzieło artystyczne z kodem w celu utworzenia profesjonalnej aplikacji.



Rysunek 10.2. Artystyczna koncepcja futurystycznego samochodu Futurgo; projekt i grafika opublikowane dzięki uprzejmości TC Chang (http://tcchang.com/)



TC Chang jest doświadczonym dyrektorem artystycznym o imponującym CV. Pracował dla takich firm jak Disney Interactive, Sony czy Electronic Arts oraz przy takich grach jak *The Godfather*, *James Bond* czy *Jet Li*. Ponadto TC głęboko wierzy w wielkie możliwości grafiki trójwymiarowej w internecie. Jest nawet założycielem nowego projektu w tej branży o nazwie Flatland. Jego prace można obejrzeć pod adresem <http://www.tcchang.com/>.

Tworzenie trójwymiarowej treści

Do utworzenia poszczególnych części modelu pojazdu i ich animowania TC użył programu Autodesk Maya 2013. Futurgo to wprawdzie koncepcyjnie jeden obiekt, ale składa się z kilku statek reprezentujących różne części samochodu: stalową karoserię, koła, wnętrze i pulpit, itd.

¹ Jedyna zauważalna różnica dotyczy czcionki. TC użył czcionki Myriad Pro, która nie należy do fontów sieciowych. Dlatego zastąpiliśmy ją później czcionką PT Sans z Google Fonts (<http://www.google.com/fonts/specimen/PT+Sans>).

Podzielenie modelu na części jest konieczne, by można było je osobno animować oraz zaimplementować dla nich różne rodzaje interakcji, takie jak wyświetlanie dodatkowych informacji po najechaniu kursorem na okno lub karoserię. Na rysunku 10.3 widać moment pracy nad modelem Futurgo w programie Maya. Na nakładce tekstowej znajdują się różne informacje statystyczne o modelu, np. liczby wierzchołków i trójkątów.



Rysunek 10.3. Modelowanie pojazdu Futurgo w programie Maya; obraz opublikowany dzięki uprzejmości TC Changa (<http://tcchang.com/>)

W tej fazie razem z TC szczegółowo zaplanowaliśmy aspekty modelu, takie jak skala — tzn. jednostkę długości (w tym przypadku metr) — oświetlenie oraz sposób utworzenia animacji. Zestaw narzędzi do animowania w programie Maya jest dość ograniczony. Zawiera tylko jedną oś czasu dla całego pliku, przez co wszystkie animacje muszą trwać tyle samo albo krótsze muszą zawierać „martwe miejsca”. Postanowiliśmy, że nasze animacje będą krótkie — jedna sekunda wystarczy na rozsunięcie okien w celu ukazania wnętrza pojazdu oraz pełny obrót kół.

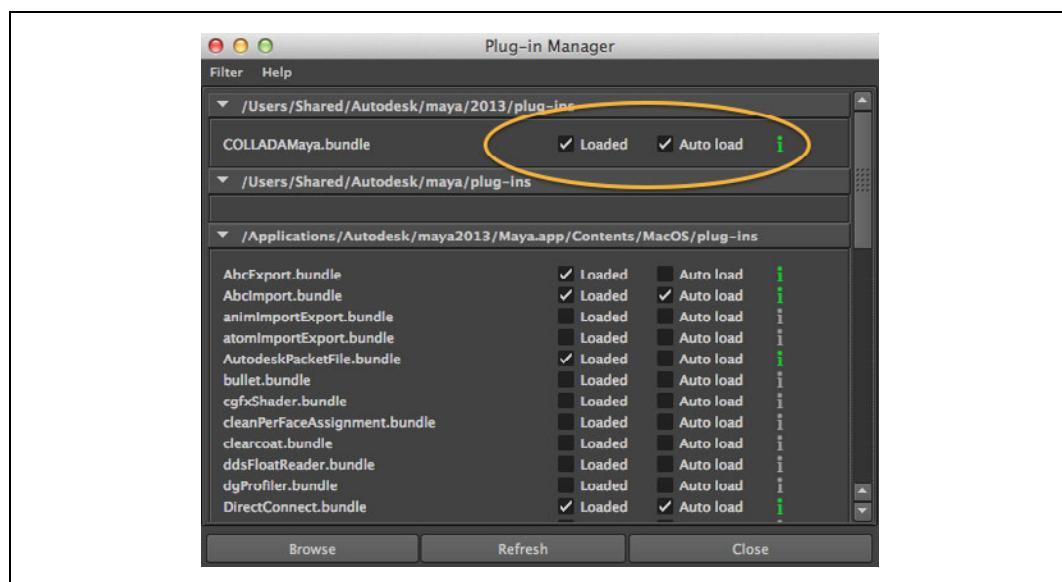
Razem z TC popracowaliśmy też nad optymalizacją wydajności aplikacji poprzez zminimalizowanie liczby wielokątów. Futurgo składa się z około 96 000 trójkątów. Liczba ta wystarcza, by uzyskać ładny rendering w Three.js, a jednocześnie nie jest zbyt duża, dzięki czemu nie spowalnia działania przeglądarki internetowej. Ponadto mniejsza liczba wielokątów pozwala zachować niewielki rozmiar plików. Jest to ważne w przypadku aplikacji sieciowej, którą trzeba szybko pobierać. Ostateczna wersja pliku po eksportie z programu Maya i konwersji na format glTF ma rozmiar około 6 MB. Nie jest to mało, ale przy nowoczesnych łączach internetowych i włączonej kompresji plików .bin jego pobieranie będzie trwało bardzo krótko (zaledwie parę sekund).

Eksportowanie sceny Maya do formatu COLLADA

Pliki programu Maya przed wyświetlaniem w przeglądarce internetowej muszą zostać przekonwertowane na format zgodny z WebGL. Postanowiliśmy użyć formatu glTF ze względu na jego niewielki rozmiar i szybkość pobierania. Jednak na razie w programie Maya nie ma opcji

bezpośredniego eksportu do formatu glTF. Dlatego musielibyśmy najpierw wyeksportować dane do formatu COLLADA, a dopiero potem przekonwertować je na docelowy format glTF².

Dostępny w programie Maya 2013 eksporter do formatu COLLADA zawiera błędy i jest przestępca, więc postanowiliśmy zamiast niego użyć narzędzia OpenCOLLADA (<http://opencollada.org/>). Jest to wydajny i otwarty eksporter służący do tworzenia wysokiej jakości danych w formacie COLLADA zgodnie ze specyfikacją tego formatu. Aktualnie OpenCOLLADA dla programu Maya (jest też wersja dla programu 3ds Max) działa bardzo dobrze i za jego pomocą bez problemu wyeksportowaliśmy model Futurgo. Na stronie internetowej narzędzia znajdują się odnośniki służące do pobrania programu dla programów Maya i 3ds Max w wersjach od 2010 do 2013. (Firma Autodesk co roku uaktualnia swoje SDK wtyczek, przez co eksportery również muszą być uaktualniane z taką samą częstotliwością. Koniecznie pobierz wersję eksportera odpowiednią dla swojego programu). Narzędzie eksportujące po zainstalowaniu należy włączyć w programie Maya w menedżerze wtyczek — *Window/Settings/Preferences/Plug-in Manager* (okno/ustawienia/preferencje/menedżer wtyczek). Spójrz na rysunek 10.4.



Rysunek 10.4. Włączanie eksportera OpenCOLLADA w menedżerze wtyczek programu Autodesk Maya 2013

Wyeksportowany plik COLLADA modelu Futurgo jest dołączony do przykładów kodu (*models/futurgo/futurgo.dae*).

Konwertowanie pliku COLLADA na glTF

Wyeksportowany z programu Maya do formatu COLLADA model Futurgo można przekonwertować na format glTF. Do tego celu można użyć narzędzia wiersza poleceń napisanego przez Fabrice'a Robineta, szefa grupy roboczej ds. formatu COLLADA i głównego projektanta formatu glTF.

² Opis formatów glTF (ang. *Graphics Library Transmission Format*) i COLLADA (oparty na składni XML standard wymiany grafiki) znajduje się w rozdziale 8.



OpenCOLLADA to projekt o otwartym kodzie źródłowym, będący efektem czynień pracy z zamiłowaniem, więc z jego używaniem wiążą się pewne pułapki. Nie ma gwarancji, że projekt będzie cały czas rozwijany i doskonalony, zwłaszcza pod względem uaktualnień SDK przez firmę Autodesk. Jednak należy pamiętać, że COLLADA to nie jedyny format, którego można używać w programach Maya i 3ds Max. Inną możliwością jest eksport danych do formatu FBX i następnie jego konwersja na glTF. Programy firmy Autodesk będą dobrze eksportowały dane do formatu FBX jeszcze przez jakiś czas. Obecnie kilka firm, m.in. firma Verold (<http://www.verold.com/>) opisana w rozdziale 8., pracuje nad konwersją FBX do glTF.

W moim komputerze MacBook Air z systemem Mac OS 10.8 polecenie to jest obsługiwane przez plik wykonywalny o nazwie *collada2gltf*. Aby przekonwertować model Futurgo, wykonalem poniższe polecenie (dane zwrócone przez program są wydrukowane pochyłym):

```
$ <path-to-converter>/collada2gltf -f futurgo.dae -d
```

```
[option] export pass details
converting:futurgo.dae ... as futurgo.json
[shader]: futurgo0VS.gls
[shader]: futurgo0FS.gls
[shader]: futurgo2VS.gls
[shader]: futurgo2FS.gls
[shader]: futurgo4VS.gls
[shader]: futurgo4FS.gls
[completed conversion]
```

Po zakończeniu konwersji w folderze pojawi się plik o nazwie *futurgo.json* oraz pomocnicze pliki źródłowe shaderów GLSL (z rozszerzeniem *.gls*). Otrzymane dane w formacie glTF możemy załadować do aplikacji za pomocą narzędzia do wczytywania danych w tym formacie, które napisałem dla biblioteki Three.js. Zajmiemy się tym w kolejnym podrozdziale. Dane glTF modelu Futurgo znajdują się wśród przykładów kodu w plikach *models/futurgo/futurgo.json* (główny plik JSON) i *models/futurgo/futurgo.bin* (dane binarne).



Na razie format glTF jest we wczesnej fazie rozwoju, co ma dwojakie konsekwencje. Po pierwsze, specyfikacja może się zmienić, a więc zanim się ustabilizuje, pliki dostarczone z tą książką mogą stać się przestarzałe. Wówczas trzeba będzie wykonać migrację lub aktualizację treści. Po drugie, narzędzia są bardzo świeże. Przykładowo program collada2gltf trzeba skompilować z plików źródłowych na platformie docelowej. Informacje na temat komplikacji tego konwertera znajdują się w repozytorium glTF w serwisie GitHub (<https://github.com/KhronosGroup/glTF>) i na stronie internetowej <http://gltf.g/>.

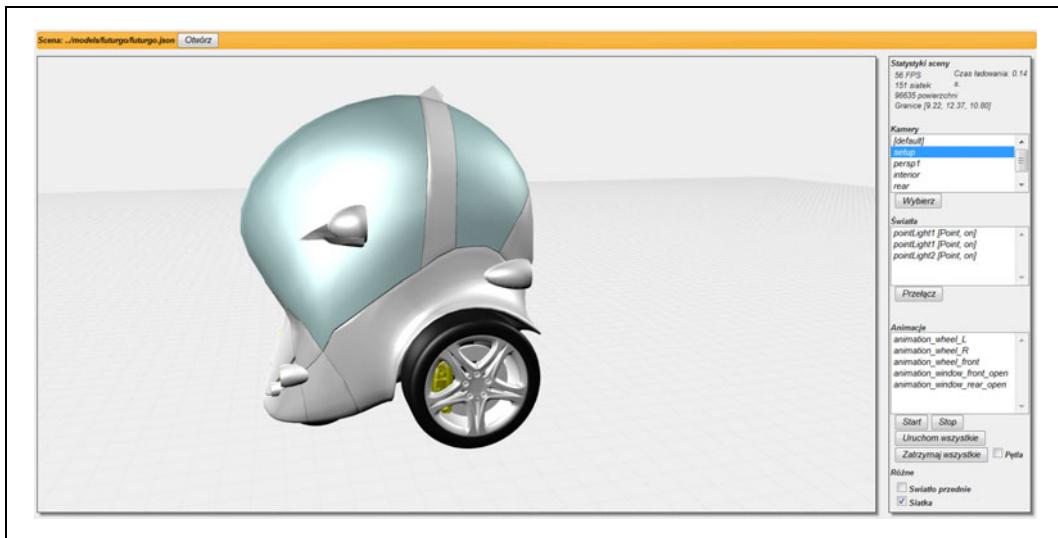
Podglądanie i testowanie treści trójwymiarowej

Po wyeksportowaniu treści z programu Maya musimy zmierzyć się z kolejnym problemem — wyświetleniem modelu na stronie internetowej. Plików w formacie glTF nie da się tak po prostu otworzyć, aby obejrzeć ich zawartość — przypomnij, że w WebGL nie istnieje pojęcie formatu plików. Modele i sceny należy ładować przy użyciu własnych bibliotek. A przecież zanim zaczniemy pracę nad właściwą aplikacją, przydałoby się sprawdzić, czy nasza treść trójwymiarowa jest w dobrym stanie, tzn. czy da się ją wyrenderować w WebGL bez naruszenia materiałów, tekstur, światła, kamer, przekształceń, animacji itd. W związku z tym zbudujemy narzędzie do podglądania i testowania treści trójwymiarowej.

Narzędzie do podglądu na bazie systemu Vizi

Narzędzie do podglądu treści trójwymiarowej utworzymy przy użyciu systemu Vizi mojego autorstwa, który przedstawiłem już w rozdziale 9. W systemie tym zastosowałem komponentowe podejście do budowania aplikacji trójwymiarowych, tzn. zautomatyzowałem powtarzalne zadania, takie jak inicjacja i porządkowanie, dostarczyłem pętlę wykonawczą i mechanizmy obsługi zdarzeń oraz udostępniłem zestaw gotowych zachowań i interakcji. Grafika jest nadal renderowana przy użyciu biblioteki Three.js — która jest w zasadzie standardem renderowania grafiki w WebGL — ale Vizi ułatwia jej używanie i przyspiesza pracę.

Na rysunku 10.5 można zobaczyć opisywane narzędzie z załadowanym plikiem glTF modelu Futurgo. Po lewej stronie znajduje się główny obszar treści do oglądania modeli i scen z siatką w tle. U góry znajduje się pasek menu zawierający przycisk *Open* (otwórz) oraz nazwę aktualnie oglądanego pliku. Po prawej stronie umieszczony został podzielony na kilka części panel sterowania. W okienku *Statystyki sceny* pokazane są: aktualna liczba klatek na sekundę, liczba siatek i wielokątów oraz czas ładowania sceny. W okienkach *Kamery*, *Światła* i *Animacje* znajdują się narzędzia do testowania kamer i światel oraz uruchamiania animacji. Na samym dole mamy jeszcze okienko *Różne*, w którym można włączyć *Światło przednie* dla modeli eksportowanych bez światel oraz włączyć lub wyłączyć siatkę w tle za pomocą specjalnego pola wyboru.



Rysunek 10.5. Podgląd modelu glTF przy użyciu systemu Vizi

Nasze narzędzie umożliwia nie tylko wizualną inspekcję i przetestowanie treści, ale również dostarcza ważne informacje o scenie, a konkretnie identyfikatory obiektów potrzebne podczas implementowania interakcji. Po wysłaniu wyeksportowanych plików COLLADA TC przysłał mi e-mail z listą nazw animacji. Nie jest to jednak najlepsza metoda określania identyfikatorów obiektów, które będą używane w kodzie. Dzięki narzędziu do podglądu mamy pewność dotyczącej identyfikatorów obiektów użytych w animacji w formacie COLLADA i przekonwertowanych plikach glTF (identyfikatory obiektów to nazwy znajdujące się w okienku *Animations* po prawej stronie okna programu).

Przedstawione narzędzie jest bardzo proste, ale bezcenne. Oczywiście daleko mu do kompletnego środowiska programistycznego, ale jest niezbędnym składnikiem do testowania treści przed przekazaniem jej do aplikacji. Podobnych narzędzi będziemy używać we wszystkich kolejnych projektach, więc warto dowiedzieć się, jak jest zbudowane.

Klasa Vizi.Viewer

Wiele aplikacji trójwymiarowych działa wg typowego schematu: inicjacja renderera, utworzenie pustej sceny, załadowanie modelu, dodanie interakcji, uruchomienie programu. Jest to tak typowy wzorzec, że postanowiłem go zaimplementować w specjalnej klasie w systemie Vizi. Klasa ta nazywa się `Vizi.Viewer` i jest wyprowadzona z klasy `Vizi.Application`. Jest to zatem specjalny typ aplikacji do przeglądania modeli i scen oraz pracy z nimi. Oglądane modele i sceny można obracać za pomocą myszy, przesuwać w czterech kierunkach oraz zmniejszać i powiększać.

Klasa `Vizi.Viewer` ma wiele zastosowań. Przy jej użyciu można budować proste przeglądarki z możliwością wczytywania modeli i ich obracania, bez żadnych dodatkowych opcji. Można też tworzyć takie przeglądarki jak opisana w tym podrozdziale, a nawet kompletnie strony wizualizacji produktów, takie jak strona pojazdu Futurgo (wróćmy do niej dalej w tym rozdziale).

Jak na prawdziwy system szkieletowy przystało, klasa `Vizi.Viewer` zawiera wiele gotowych funkcji, których implementacja przy użyciu biblioteki Three.js wymagałaby napisania setek wierszy kodu. Oto jej najważniejsze cechy.

Sterowanie widokiem modelu

Klasa `Vizi.Viewer` używa ulepszonej wersji obiektu `THREE.OrbitControls` dostarczanego z przykładami do biblioteki Three.js. Lewy przycisk myszy służy do obracania, prawy do przesuwania, a kółko i trackpad do zmieniań skali. Ponadto przeglądarka zawiera opcje umożliwiające zmianę tych ustawień.

Domyślne oświetlenie i kamera

Dla scen niezawierających kamery klasa `Vizi.Viewer` ma kamerę domyślną. Dla scen niezawierających oświetlenia przeglądarka może utworzyć domyślne oświetlenie automatycznie oświetlające modele i aktualizowane, gdy użytkownik zmienia położenie kamery.

Pomocnicze obiekty sceny

`Vizi.Viewer` opcjonalnie wyświetla płaszczyznę podłożę z prostokątną siatką oraz, jeśli trzeba, obrys modelu.

Statystyki sceny i renderowania

Przeglądarka może przekazywać zdarzenia raportujące o szybkości zmiany klatek oraz statystyki sceny, takie jak liczba siatek i wielokątów, wymiary otaczającej ramki i czas ładowania plików.

Przyciski do sterowania oświetleniem, kamerą i animacją

Klasa `Vizi.Viewer` zawiera metody pomocnicze umożliwiające włączanie i wyłączanie oświetlenia, przełączanie kamer oraz uruchamianie i zatrzymywanie animacji. Przeglądarka dostarcza aplikacji listę obiektów każdego z tych typów, dzięki czemu nie trzeba ich szukać w kodzie.

Operowanie jednym przyciskiem myszy

Obiekt `THREE.OrbitControls` został zmodyfikowany w taki sposób, aby obsługiwał tylko jeden przycisk myszy, tzn. prawy przycisk jest wyłączony albo przypisany do lewego przycisku myszy.

Aby zobaczyć klasę `Vizi.Viewer` w akcji, przeanalizujemy sposób jej użycia do implementacji przeglądarki. Przeglądarkę tę można uruchomić, otwierając w przeglądarce internetowej plik `r10/previewer.html`. Na listingu 10.1 znajduje się fragment jej kodu źródłowego, w którym tworzony jest obiekt klasy `Vizi.Viewer`. Jak zwykle, przekazujemy parametr `containter`, czyli element `<div>`, w którym biblioteka Three.js doda swój renderer WebGL (kanwę z kontekstem rysunkowym WebGL). Ponadto ustawiamy kilka opcji, za pomocą których nakazujemy przeglądarce wyświetlenie siatki oraz zastosowanie światła domyślnego, jeśli na scenie nie ma innego oświetlenia. Po utworzeniu przeglądarki dodajemy kilka procedur nasłuchu zdarzeń obserwujących zmiany częstotliwości klatek i innych aspektów sceny. Bardziej szczegółowo zajmiemy się nimi dalej w tym rozdziale. Następnie tworzymy listę plików do wyboru przy użyciu polecenia `Otwórz` na pasku menu i na końcu wywołujemy pętlę wykonawczą w celu uruchomienia aplikacji.

Listing 10.1. Tworzenie obiektu przeglądarki scen i modeli Vizi

```
var viewer = null;
$(document).ready(function() {

    var container = document.getElementById("container");
    var renderStats = document.getElementById("render_stats");
    var sceneStats = document.getElementById("scene_stats");

    viewer = new Vizi.Viewer({ container : container,
        showGrid : true, headlight : true,
        showBoundingBox : false });
    viewer.addEventListener("renderstats", function(stats) {
        onRenderStats(stats, renderStats); });
    viewer.addEventListener("scenestats", function(stats) {
        onSceneStats(stats, sceneStats); });

    buildFileList();

    viewer.run();
}
);
```

Po uruchomieniu przeglądarki na ekranie powinno być widoczne puste okno sceny. Na pomańczowym pasku menu u góry okna znajduje się interfejs, za pomocą którego można otworzyć jeden z kilku dostępnych plików trójwymiarowych.

Klasa wczytująca Vizi

Kliknięcie przycisku `Otwórz` powoduje wyświetlenie okna dialogowego wyboru pliku, w którym należy wskazać pozycję `./models/futurgo/futurgo.json`. W efekcie na ekranie powinien pojawić się model pojazdu Futurgo, co widać na rysunku 10.5. Można się nim pobawić za pomocą myszy i gładzika albo kółka myszy.

Przeglądarka wczytuje model do obiektu klasy `Vizi.Viewer` za pomocą innej klasy — `Vizi.Loader`, co pokazano na listingu 10.2.

Listing 10.2. Wczytywanie plików za pomocą obiektu klasy Vizi.Loader

```
function openfile()
{
    var select = document.getElementById("files");
    var index = select.selectedIndex;
```

```

if (index >= 0)
{
    var url = select.options[index].text;

    var loader = new Vizi.Loader;

    loader.addEventListener("loaded", function(data) {
        onLoadComplete(data, loadStartTime); });
    loader.addEventListener("progress", function(progress) {
        onLoadProgress(progress); });

    var fileViewingName = document.getElementById("fileViewingName");
    fileViewingName.innerHTML=url;

    var loadStartTime = Date.now();
    loader.loadScene(url);

    var loadStatus = document.getElementById("loadStatus");
    loadStatus.style.display = 'block';
}

$('#fileOpenDialog').dialog("close");
}

```

Klasa `Vizi.Loader` do przetwarzania i ładowania do pamięci danych w różnych formatach korzysta z narzędzi do wczytywania plików JSON, COLLADA i glTF biblioteki Three.js. Ponadto opakowuje utworzoną scenę Three.js w składniki Vizi, otrzymując w ten sposób scenę odpowiednią do użytku w aplikacjach opartych na Vizi. Na końcu, po pobraniu i przetworzeniu pliku wysyła zdarzenia `loaded` i `progress` do procedur nasłuchujących. Na listingu 10.3 pokazano kod źródłowy funkcji nasłuchującej zdarzeń `onLoadComplete()`, która wykrywa, kiedy plik jest całkowicie wczytany i gotowy do dodania do przeglądarki.

Listing 10.3. Procedura nasłuchu załadowania pliku w przeglądarce

```

function onLoadComplete(data, loadStartTime)
{
    // Ukrywa pasek ładowania.
    var loadStatus = document.getElementById("loadStatus");
    loadStatus.style.display = 'none';

    viewer.replaceScene(data);

    var loadTime = (Date.now() - loadStartTime) / 1000;
    var loadTimeStats = document.getElementById("load_time_stats");
    loadTimeStats.innerHTML = "Czas ładowania: " +
        loadTime.toFixed(2) + " s."

    updateCamerasList(viewer);
    updateLightsList(viewer);
    updateAnimationsList(viewer);
    updateMiscControls(viewer);

    if (viewer.cameraNames.length > 1) {
        selectCamera(1);
    }
}

```

Procedura ta wykonuje kilka czynności. Najpierw ukrywa element `<div>`, który został wyświetlony na początku ładowania sceny z napisem „Ładowanie sceny”. Jest to znak dla użytkownika, że scena została już załadowana. Następnie za pomocą wywołania metody `replaceScene()`

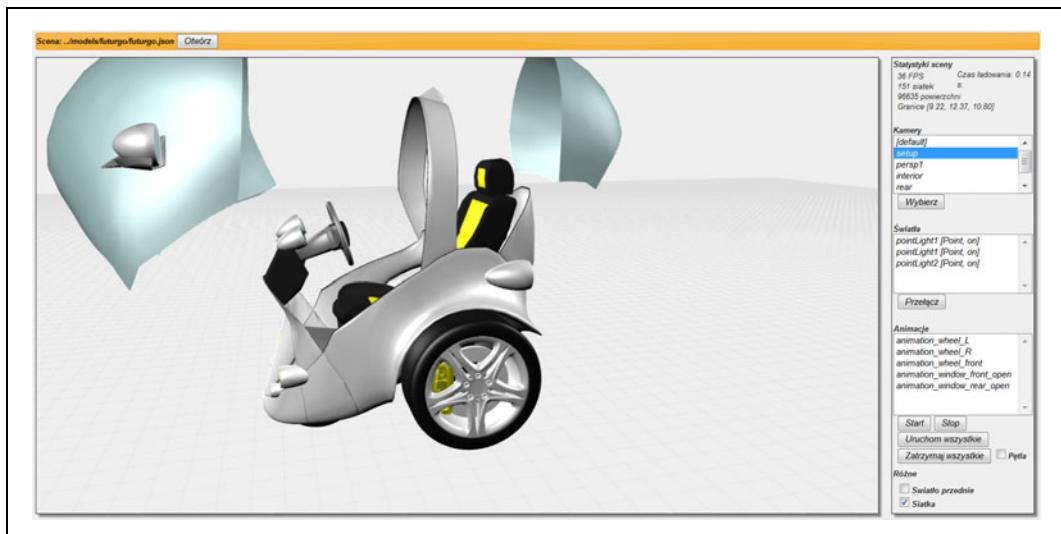
dodaje załadowaną treść do przeglądarki. To właśnie dzięki temu możemy oglądać pojazd Futurgo i bawić się nim w oknie sceny. Później procedura aktualizuje czas ładowania w okienku *Statystyki sceny*. Kolejną czynnością jest wywołanie kilku metod pomocniczych w celu zaktualizowania list w interfejsie użytkownika (np. kamery i światła) na podstawie tablic odpowiednich obiektów utrzymywanych w przeglądarce. Na koniec wywołuje funkcję `selectCamera()` w celu wybrania pierwszej kamery dostępnej w scenie (nie licząc domyślnej), jeśli jakakolwiek istnieje. Do przełączania kamer w funkcji `selectCamera()` użyto metody `useCamera()` przeglądarki:

```
function selectCamera(index)
{
    var select = document.getElementById("cameras_list");
    if (index === undefined) {
        index = select.selectedIndex;
    }
    else {
        select.selectedIndex = index;
    }

    if (index >= 0) {
        viewer.useCamera(index);
    }
}
```

Teraz przeglądarka jest już gotowa do oglądania i testowania modelu, który można obracać, przesuwać oraz zmniejszać i powiększać. Za pomocą znajdujących się po prawej stronie kontrolek można zmieniać kamery, włączać i wyłączać oświetlenie oraz odtwarzać animacje.

Możliwość testowania animacji jest jedną z najważniejszych cech przeglądarki, ponieważ podczas tworzenia animacji trójwymiarowych może wystąpić wiele różnych problemów. Dzięki przetestowaniu modelu zawsze oszczędzamy sobie konieczności zmierzenia się z trudnościami później. Na rysunku 10.6 widać pojazd Futurgo po odtworzeniu animacji o nazwie `animation_window_front_open` i `animation_window_rear_open`. Widać, że osiągnęliśmy zamierzony efekt: przednie i tylne okna zostały odłączone od pojazdu, dlatego widać jego wnętrze.



Rysunek 10.6. Odtwarzanie animacji w przeglądarce

Animacje nie są częścią grafu sceny Vizi jako takiego (ani grafu sceny Three.js). Są zapisane w osobnej tablicy obiektów w przeglądarce. W związku z tym, do włączania, zatrzymywania oraz zapętlania animacji musimy użyć metod pomocniczych przeglądarki. Na listingu 10.4 pokazano umieszczone w kodzie HTML funkcje wywołujące metody przeglądarki do włączania, zatrzymywania i zapętlania animacji, takie jak `viewer.playAnimation()`, `viewer.stopAnimation()`, `viewer.playAllAnimations()`, `viewer.stopAllAnimations()` oraz `viewer.setLoopAnimations()`.

Listing 10.4. Użycie metod Vizi przeglądarki do sterowania animacją

```
function selectAnimation()
{
    var select = document.getElementById("animations_list");
    var index = select.selectedIndex;
    if (index >= 0)
    {
        viewer.playAnimation(index, viewer.loopAnimations);
    }
}

function playAnimation()
{
    var select = document.getElementById("animations_list");
    var index = select.selectedIndex;
    if (index >= 0)
    {
        viewer.playAnimation(index, viewer.loopAnimations);
    }
}

function stopAnimation()
{
    var select = document.getElementById("animations_list");
    var index = select.selectedIndex;
    if (index >= 0)
    {
        viewer.stopAnimation(index);
    }
}

function playAllAnimations()
{
    viewer.playAllAnimations(viewer.loopAnimations);
}

function stopAllAnimations()
{
    viewer.stopAllAnimations();
}

function onLoopChecked(elt)
{
    viewer.setLoopAnimations(elt.checked);
}
```

Integrowanie treści trójwymiarowej z aplikacją

Kiedy już mamy pewność, że nasza treść trójwymiarowa da się wczytać, wyrenderować i animować w sposób zgodny z oczekiwaniami, możemy rozpocząć budowę właściwej aplikacji.

Pierwszą czynnością w tym zakresie będzie zintegrowanie treści trójwymiarowej ze stroną internetową programu. Jeszcze raz otwórz w przeglądarce internetowej plik *r10/futurgo.html*, aby wyświetlić stronę widoczną na rysunku 10.7.



Rysunek 10.7. Model pojazdu *Futurgo* na stronie HTML

Wszystkie elementy tej strony są płynnie zintegrowane, co jest zasługą mechanizmu składania przeglądarki internetowej. Każdy składnik tej strony jest elementem `<div>` lub kilkoma zagnieżdzonymi elementami `<div>` z odpowiednimi ustawieniami właściwości `z-index`. Trójwymiarowy widok znajduje się pod pozostałymi elementami strony, dzięki czemu interfejs użytkownika jest umieszczony na wierzchu. Niektóre elementy interfejsu użytkownika są przezroczyste, aby było widać więcej treści trójwymiarowej. Osoby oglądające scenę w kolorze zauważają także piękny fioletowoszary gradient w tle, który został przeniesiony wprost z projektu TC. Tło zostało ustawione przy użyciu właściwości CSS `background` elementu kontenera. Postanowiliśmy też pozostawić siatkę dostarczoną przez klasę `Vizi.Viewer` (skopiowaliśmy ją przypadkowo, ale spodobała się i ją zostawiliśmy).

Kod źródłowy tej aplikacji znajduje się w plikach *r10/futurgo.html*, *css/futurgo.css* oraz *r10/futurgo.js*. Zmieniliśmy parę rzeczy w porównaniu z przeglądarką. Kod HTML zawiera teraz prawie same elementy `<div>` i tylko niewielką ilość kodu JavaScript do obsługi ładowania strony i znajdujących się po prawej stronie kart interfejsu użytkownika oraz zmiany obrazów po najechaniu kursem.

Kod obsługujący wczytywanie strony, pokazany na listingu 10.5, tworzy nowy obiekt *Futurgo*, przekazując element kontenera i metody zwrotne dotyczące zakończenia ładowania i pozycji kurSORA. Następnie wywołuje metodę *Futurgo.go()*, która ładuje scenę trójwymiarową i uruchamia pętlę wykonawczą.

Listing 10.5. Kod ładowania strony *Futurgo*

```
<script>

var futurgo = null;
var overlay = null;
var overlayContents = null;
```

```

var loadStatus = null;
var part_materials = [];

$(document).ready(function() {

    initControls();
    overlay = document.getElementById("overlay");
    overlayContents = document.getElementById("overlayContents");
    loadStatus = document.getElementById("loadStatus");
    var container = document.getElementById("container");
    futurgo = new Futurgo({ container : container,
        loadCallback : onLoadComplete,
        loadProgressCallback : onLoadProgress,
        mouseOverCallback : onMouseOver,
        mouseOutCallback : onMouseOut,
    });
    loadStatus.style.display = 'block';
    futurgo.go();
}
);

```

Obiekt `Futurgo` rozwiązuje większość problemów dotyczących wczytywania. Jedynym zadaniem metody zwrotnej wczytywania jest ukrycie elementu `<div>` zawierającego napis „Ładowanie sceny”, co widać na listingu 10.6. Opis metod `onMouseOver()` i `onMouseOut()` dotyczących myszy znajduje się w następnym podrozdziale.

Listing 10.6. Ukrywanie informacji o ładowaniu sceny

```

function onLoadComplete(loadTime)
{
    // Ukrywa pasek ładowania.
    loadStatus.style.display = 'none';
}

```

Teraz zobaczymy, jak klasa `Futurgo` inicjuje przeglądarkę i ładuje scenę — listing 10.7 (plik źródłowy `r10/futurgo.js`).

Listing 10.7. Kod konfiguracji przeglądarki i ładowania plików aplikacji `Futurgo`

```

Futurgo = function(param) {

    this.container = param.container;
    this.loadCallback = param.loadCallback;
    this.loadProgressCallback = param.loadProgressCallback;
    this.mouseOverCallback = param.mouseOverCallback;
    this.mouseOutCallback = param.mouseOutCallback;
    this.part_materials = [];
    this.vehicleOpen = false;
    this.wheelsMoving = false;
}

Futurgo.prototype.go = function() {
    this.viewer = new Vizi.Viewer({ container : this.container,
        showGrid : true,
        allowPan: false, oneButton: true});
    this.loadURL(Futurgo.URL);
    this.viewer.run();
}

```

```

Futurgo.prototype.loadURL = function(url) {
    var that = this;
    var loader = new Vizi.Loader;
    loader.addEventListener("loaded", function(data) {
        that.onLoadComplete(data, loadStartTime); });
    loader.addEventListener("progress", function(progress) {
        that.onLoadProgress(progress); });

    var loadStartTime = Date.now();
    loader.loadScene(url);
}

```

Znaczna część tego kodu powinna już wyglądać znajomo. Podobnie jak w przeglądarce, utworzyliśmy obiekty `Vizi.Viewer` i `Vizi.Loader`, ale ustawiliśmy też parę opcji dotyczących tworzenia obiektu `Vizi.Viewer` (pogrubiony wiersz). Opcja `allowPan` określa, czy można przesuwać obiekt za pomocą prawego przycisku myszy. Ustawiliśmy ją na `false`, ponieważ chcemy, aby pojazd zawsze znajdował się na środku sceny. Opcja `oneButton` określa, czy prawy przycisk myszy również może być używany do obracania modelu. Jej ustawienie na `true` oznacza tak.

Przedstawiony do tej pory kod ładuje na stronę model `Futurgo` oraz nadaje mu ładny wygląd i umożliwia interakcję z nim. W następnym podrozdziale tchniemy w nasz model życie poprzez dodanie zachowań trójwymiarowych i obsługi interakcji.

Trójwymiarowe zachowania i interakcje

Już teraz nasza aplikacja jest całkiem ciekawa. Wyświetlony w niej trójwymiarowy model ma ciekawą oprawę wizualną i można przy nim szperać za pomocą myszy. Jednak to nie jest szczyt naszych możliwości. Aby wykorzystać cały potencjał generowanej na bieżąco internetowej grafiki trójwymiarowej, możemy zaimplementować trójwymiarowe zachowania i interakcje. Po załadowaniu modelu zastosujemy automatyczną animację, dodamy nakładki z informacjami o poszczególnych częściach pojazdu, które będą pojawiać się po najechaniu kursorem na te części, oraz włączymy dynamiczne zmienianie wyglądu obiektu trójwymiarowego po kliknięciu dwuwymiarowych elementów na stronie.

Metody API grafu sceny Vizi: `findNode()` i `map()`

Implementacja opisanych w tym podrozdziale zachowań wymaga przeglądania grafu sceny treści trójwymiarowej załadowanej przez obiekt klasy `Vizi.Loader`, aby można było dodać zachowania i interakcje z myszą do wybranych obiektów. Czasami obiekty można znaleźć po nazwach lub identyfikatorach, a czasami konieczne jest przejrzenie części lub całego grafu sceny. System Vizi zawiera zestaw służących do tego metod API grafu sceny. Metodom tym można przekazać identyfikator w postaci łańcucha, wyrażenie regularne JavaScript albo typ obiektu JavaScript (porównywany przy użyciu operatora `instanceof`). Metody `findNode()` i `findNodes()` zwracają znalezione obiekty, a metoda `map()` znajduje obiekty i wywołuje funkcję na wyniku.

`findNode(zapytanie)`

Metoda znajduje węzeł (egzemplarz klasy `Vizi.Object` lub `Vizi.Component`) na podstawie zapytania. Zapytanie to może być identyfikatorem (np. `"body2"`), typem obiektu (np. `Vizi.Visual`) lub wyrażeniem regularnym (np. `/windows_front|windows_rear/`). Jeśli w grafie sceny jest kilka pasujących węzłów, metoda zwraca pierwszy z nich.

```
findNodes(zapytanie)
```

Metoda znajduje wszystkie węzły (egzemplarze klasy `Vizi.Object` lub `Vizi.Component`) na podstawie zapytania. Zapytanie to może być identyfikatorem (np. "body2"), typem obiektu (np. `Vizi.Visual`) lub wyrażeniem regularnym (np. `/windows_front|windows_rear/`).

```
map(zapytanie, funkcja_zwrotna)
```

Metoda korzysta z metody `findNodes()` do znalezienia wszystkich węzłów odpowiadających zapytaniu, a następnie na każdym z nich wywołuje funkcję zwrotną.



Metody Vizi API grafu sceny pod względem działania można porównać z zapytaniami jQuery, chociaż ich budowa jest całkiem inna. W Vizi nie istnieje pojęcie selektora, zamiast którego używane są łańcuchy i typy danych JavaScript. Jest to celowy wybór podyktowany obiektywko-komponentową naturą systemu.

Teraz przyjrzymy się kodowi obsługującemu ładowanie sceny, aby zobaczyć, jak dodawane są zachowania. Spójrz na listing 10.8.

Listing 10.8. Dodawanie zachowań do aplikacji Futurgo po załadowaniu sceny

```
Futurgo.prototype.onLoadComplete = function(data, loadStartTime)
{
    var scene = data.scene;
    this.viewer.replaceScene(data);

    // Dodanie efektu zanikania koloru okien oraz obiektów Picker do wyświetlanego informacji po najechaniu kursorem
    // na wybraną część.
    var that = this;
    scene.map(/windows_front|windows_rear/, function(o) {
        var fader = new Vizi.FadeBehavior({duration:2, opacity:.8});
        o.addComponent(fader);
        setTimeout(function() {
            fader.start();
        }, 2000);

        var picker = new Vizi.Picker;
        picker.addEventListener("mouseover", function(event) {
            that.onMouseOver("glass", event); });
        picker.addEventListener("mouseout", function(event) {
            that.onMouseOut("glass", event); });
        o.addComponent(picker);
    });

    // Automatyczne obracanie sceny.
    var main = scene.findNode("vizi_mobile");
    var carousel = new Vizi.RotateBehavior({autoStart:true,
        duration:20});
    main.addComponent(carousel);

    // Zebranie materiałów części w jednym miejscu, aby można było zmieniać kolory.
    var frame_parts_exp =
        /rear_view_arm_L|rear_view_arm_R|rear_view_frame_L|rear_view_frame_R/;

    scene.map(frame_parts_exp, function(o) {
        o.map(Vizi.Visual, function(v) {
            that.part_materials.push(v.material);
        });
    });
}
```

```

// Dodanie obiektów Picker do wyświetlania informacji po najechaniu kursorem na wybranączęść.
scene.map(/body2|rear_view_arm_L|rear_view_arm_R/, function(o) {
    var picker = new Vizi.Picker;
    picker.addEventListener("mouseover", function(event) {
        that.onMouseOver("body", event); });
    picker.addEventListener("mouseout", function(event) {
        that.onMouseOut("body", event); });
    o.addComponent(picker);
});

scene.map("wheels", function(o) {

    var picker = new Vizi.Picker;
    picker.addEventListener("mouseover", function(event) {
        that.onMouseOver("wheels", event); });
    picker.addEventListener("mouseout", function(event) {
        that.onMouseOut("wheels", event); });
    o.addComponent(picker);
});

// Informuje stronę o zakończeniu ładowania.
if (this.loadCallback) {
    var loadTime = (Date.now() - loadStartTime) / 1000;
    this.loadCallback(loadTime);
}
}
}

```

Najpierw metoda `onLoadComplete()` dodaje załadowaną scenę Futurgo do przeglądarki za pomocą wywołania `this.viewer.replaceScene(data)`. Wewnętrznie przeglądarka nie tylko dodaje obiekty do swojego grafu sceny, ale dodatkowo rachuje światła, kamery i animacje (jak opisałem wcześniej), dzięki czemu mamy listę tych obiektów, przy użyciu której możemy — jeśli trzeba — przelać kamerę, odtwarzać animacje itd. Następnie dodajemy zachowania, z których niektóre są od razu uruchamiane. Każde z zachowań dodanych w tej metodzie jest opisane w osobnym punkcie.

Animowanie przezroczystości za pomocą klasy `Vizi.FadeBehavior`

Okna naszego pojazdu powinny być przezroczyste, aby było widać jego wnętrze. Wprawdzie moglibyśmy ustawić poziom przezroczystości od razu przy ładowaniu sceny, ale woleliśmy zastosować ciekawszy efekt przejścia powodujący zastosowanie odpowiedniej przezroczystości w określonym czasie — listing 10.9.

Listing 10.9. Dodawanie efektu zmiany przezroczystości do okien

```

var that = this;
scene.map(/windows_front|windows_rear/, function(o) {
    var fader = new Vizi.FadeBehavior({duration:2, opacity:.8});
    o.addComponent(fader);
    setTimeout(function() {
        fader.start();
    }, 2000);
}
)

```

Składnik `Vizi.FadeBehavior` powoduje zanikanie materiałów wszystkich widocznych elementów znajdujących się w obejmującym go obiekcie. Jako parametry przyjmuje liczbę sekund trwania efektu oraz wartość określającą docelowy poziom przezroczystości. W tym przykładzie zwiększamy przezroczystość do poziomu .8 (częściowa przezroczystość) w ciągu dwóch sekund. Ponadto zastosowaliśmy dwusekundowe opóźnienie rozpoczęcia zanikania. Do tego celu użyliśmy starej, dobrej funkcji `setTimeout()`.

Aby w pełni docenić klasę `Vizi.FadeBehavior`, zajrzymy do jej implementacji. Na listingu 10.10 pokazano fragment kodu źródłowego z pliku `src/behaviors/fadeBehavior.js`. Uruchomione zachowanie iteruje przez znajdujące się w obiekcie widoczne elementy i oblicza aktualną wartość przezroczystości. Wartość ta jest wykorzystywana w obiekcie klasy `Tween` biblioteki `Tween.js` (rozdział 5.), który jest uruchamiany i działa przez czas trwania zachowania. Metoda `evaluate()` zachowania, wywoływaną w każdym cyklu pętli wykonawczej (jeśli zachowanie jest aktywne), najpierw sprawdza warunek pętli `i`, jeśli trzeba, ponownie uruchamia zachowanie. Następnie przechodzi przez wszystkie widoczne elementy w obiekcie nadziednym i ustawia przezroczystość materiałów na nową wartość obliczoną przez bibliotekę `Tween.js`. Rozwiążanie takie daje bardzo duże możliwości. Dzięki dostarczeniu spójnego zestawu interfejsów do obiektów i ich składników łatwo możemy tworzyć takie zachowania jak `FadeBehavior`, które można stosować do dowolnych elementów widocznych na scenie.

Listing 10.10. Implementacja zachowania Vizi.FadeBehavior

```
Vizi.FadeBehavior.prototype.start = function()
{
    if (this.running)
        return;

    if (this._realized && this._object.visuals) {
        var visuals = this._object.visuals;
        var i, len = visuals.length;
        for (i = 0; i < len; i++) {
            this.savedOpacities.push(visuals[i].material.opacity);
            this.savedTransparencies.push(
                visuals[i].material.transparent);
            visuals[i].material.transparent = this.targetOpacity < 1 ?
                true : false;
        }
    }

    this.opacity = { opacity : this.savedOpacities[0] };
    this.opacityTarget = { opacity : this.targetOpacity };
    this.tween = new TWEEN.Tween(this.opacity).to(this.opacityTarget,
        this.duration * 1000)
        .easing(TWEEN.Easing.Quadratic.InOut)
        .repeat(0)
        .start();

    Vizi.Behavior.prototype.start.call(this);
}

Vizi.FadeBehavior.prototype.evaluate = function(t)
{
    if (t >= this.duration)
    {
        this.stop();
        if (this.loop)
            this.start();
    }

    if (this._object.visuals)
    {
        var visuals = this._object.visuals;
        var i, len = visuals.length;
        for (i = 0; i < len; i++) {
            visuals[i].material.opacity = this.opacity.opacity;
        }
    }
}
```

Automatyczne obracanie modelu za pomocą klasy Vizi.RotateBehavior

To świetnie, że możemy obracać za pomocą myszy widoczny na scenie model, ale byłoby jeszcze lepiej, gdybyśmy nieco ożywiali tę scenę nawet wtedy, gdy użytkownik nic na niej nie robi. Dlatego sprawimy, że scena po załadowaniu będzie się automatycznie obracać. Procedura nasłuchująca zdarzeń załadowania sceny znajduje za pomocą metody `findNode()` krożeń sceny Futurgo i dodaje komponent `RotateBehavior`. Zachowanie to jest uruchamiane automatycznie i wykonywane w 20-sekundowej pętli, co widać w kodzie na listingu 10.11.

Listing 10.11. Dodanie zachowania Vizi.RotateBehavior w celu włączenia automatycznego obracania sceny

```
// Automatyczne obracanie sceny.  
var main = scene.findNode("vizi_mobile");  
var carousel = new Vizi.RotateBehavior({autoStart:true,  
duration:20});  
main.addComponent(carousel);
```

Wyświetlanie informacji o częściach za pomocą klasy Vizi.Picker

Technika wyświetlania dodatkowych informacji po najechaniu kursorem na wybrane części jest doskonałym uzupełnieniem naszej aplikacji. Możemy ją zaimplementować przy użyciu poznanego w rozdziale 9. komponentu `Vizi.Picker`. Komponent ten służy do obsługi zdarzeń myszy rozsyłanych, gdy kursor znajdzie się nad wybranym obiektem.

Wróćmy w kodzie do miejsca, w którym dodaliśmy efekt zmiany przezroczystości szyb pojazdu. Przypominam, że w tym samym miejscu dodaliśmy obiekt klasy `Picker`, co widać w pogrubionym fragmencie kodu na listingu 10.12. W podobny sposób dodamy „pickery” do części karoserii i kół. Każda procedura nasłuchowa używa innego oznaczenia — `glass`, `body` i `wheels` — które zostanie przesłane do aplikacji w celu zidentyfikowania części, nad którą znajduje się kursor.

Listing 10.12. Dodanie komponentów Vizi.Picker dotyczących implementacji efektu rollover

```
// Dodanie efektu zanikania koloru okien oraz obiektów Picker do wyświetlania informacji po najechaniu kursem  
// na wybraną część.  
var that = this;  
scene.map(/windows_front|windows_rear/, function(o) {  
    var fader = new Vizi.FadeBehavior({duration:2, opacity:.8});  
    o.addComponent(fader);  
    setTimeout(function() {  
        fader.start();  
    }, 2000);  
  
    var picker = new Vizi.Picker;  
    picker.addEventListener("mouseover", function(event) {  
        that.onMouseOver("glass", event); });  
    picker.addEventListener("mouseout", function(event) {  
        that.onMouseOut("glass", event); });  
    o.addComponent(picker);  
});  
...  
// Dodanie obiektów Picker do wyświetlania informacji po najechaniu kursem na wybraną część.  
scene.map(/body2|rear_view_arm_L|rear_view_arm_R/, function(o) {  
    var picker = new Vizi.Picker;  
    picker.addEventListener("mouseover", function(event) {  
        that.onMouseOver("body", event); });
```

```

picker.addEventListener("mouseout", function(event) {
    that.onMouseOut("body", event); });
o.addComponent(picker);
});

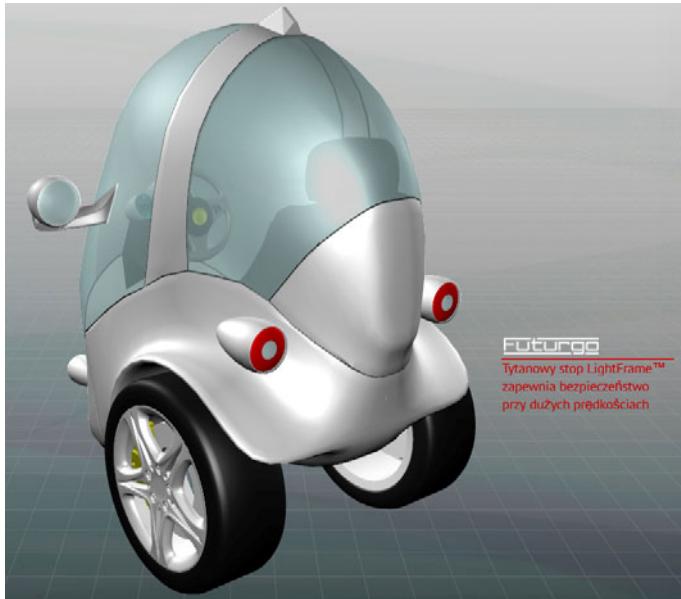
scene.map("wheels", function(o) {

    var picker = new Vizi.Picker;
    picker.addEventListener("mouseover", function(event) {
        that.onMouseOver("wheels", event); });
    picker.addEventListener("mouseout", function(event) {
        that.onMouseOut("wheels", event); });
    o.addComponent(picker);
});

```

Metody pomocnicze `Futurgo.onMouseOver()` i `Futurgo.onMouseOut()` przekazują sterowanie do metod zwrotnych `onMouseOver` i `onMouseOut` zarejestrowanych podczas tworzenia egzemplarza klasy `Futurgo` (listing 10.5).

Skutek działania efektu *rollover* widać na rysunku 10.8. Gdy kurSOR znajdzie się nad jednym z wyznaczonych obiektów, mniej więcej na jego wysokości po prawej stronie sceny zostaje wyświetlony odpowiedni element `<div>`.



Rysunek 10.8. Dodatkowe informacje wyświetlane po umieszczeniu kurSORa na części pojazdu

Sterowanie animacjami w interfejsie użytkownika

Do sterowania działaniem sceny trójwymiarowej można też używać elementów interfejsu użytkownika w postaci dwuwymiarowych elementów strony HTML. Przykładowo kliknięcie jednej ze znajdujących się po prawej stronie zakładek, np. *Wnętrze* albo *Jazda*, powoduje uruchomienie animacji. W kodzie HTML strony znajdują się procedury obsługi kliknięć, które

wywołują metody na obiekcie `Futurgo`, co pokazano na listingu 10.13. Metody te wywołują metody `playAnimation()` i `stopAnimation()`. Jednak należy zwrócić uwagę na jeden szczegół: pierwsze kliknięcie zakładki *Wnętrze* powinno powodować rozłożenie okien, a następne — ich powrót na miejsce. Jednak zamiast tworzyć dwie różne animacje do rozkładania i składania pojazdu, za drugim razem po prostu odtwarzamy animację w *odwrotnym* kierunku. Spójrz na kod metody `playCloseAnimations()`, przekazuje ona dodatkowe argumenty do przeglądarki. Drugi argument, `loop`, jest ustawiony na `false`, ale trzeci, `reverse`, jest ustawiony na `true`. Podobnie jak w `Tween.js`, silnik animacji Vizi zawiera wbudowane narzędzia do odtwarzania animacji w obie strony.

Listing 10.13. Sterowanie działaniem animacji za pomocą interfejsu użytkownika

```
Futurgo.prototype.playOpenAnimations = function() {
    this.playAnimation("animation_window_rear_open");
    this.playAnimation("animation_window_front_open");
}

Futurgo.prototype.playCloseAnimations = function() {
    this.playAnimation("animation_window_rear_open", false, true);
    this.playAnimation("animation_window_front_open", false, true);
}

Futurgo.prototype.toggleInterior = function() {
    this.vehicleOpen = !this.vehicleOpen;
    var that = this;
    if (this.vehicleOpen) {
        this.playOpenAnimations();
    }
    else {
        this.playCloseAnimations();
    }
}

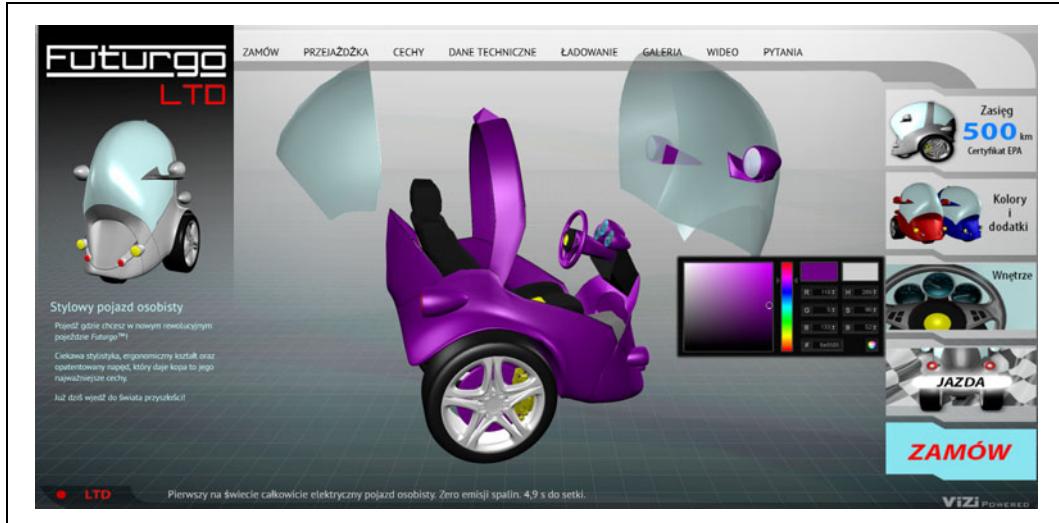
Futurgo.prototype.playWheelAnimations = function() {
    this.playAnimation("animation_wheel_L", true);
    this.playAnimation("animation_wheel_R", true);
    this.playAnimation("animation_wheel_front", true);
}

Futurgo.prototype.stopWheelAnimations = function() {
    this.stopAnimation("animation_wheel_L");
    this.stopAnimation("animation_wheel_R");
    this.stopAnimation("animation_wheel_front");
}

Futurgo.prototype.toggleWheelAnimations = function() {
    this.wheelsMoving = !this.wheelsMoving;
    if (this.wheelsMoving) {
        this.playWheelAnimations();
    }
    else {
        this.stopWheelAnimations();
    }
}
```

Zmienianie kolorów przy użyciu wybieraka

Każda szanująca się strona prezentacyjna produktu umożliwia zmienianie kolorów. Strona pojazdu Futurgo nie jest w tej kwestii wyjątkiem. Dlatego dodaliśmy do niej widżet jQuery umożliwiający obejrzenie naszego pojazdu w 16 milionach kolorów. Zmiana koloru w tym narzędziu powoduje natychmiastową zmianę koloru ramy urządzenia — rysunek 10.9.



Rysunek 10.9. Zmienianie kolorów przy użyciu widżetu jQuery

Przypomnimy sobie kod, przy użyciu którego powiązaliśmy części ramy z efektem *rollover*. Na lisingu 10.14 jeszcze raz przedstawiony jest fragment kodu metody `onLoadComplete()`. Warto zwrócić uwagę na rekurencyjne wywołanie metody `map()`: dla każdego węzła znalezionej przez wyrażenie regularne wyszukujemy wszystkie jego wizualne składniki oraz dodajemy znajdujący się w nich materiał Three.js do tablicy `part_materials`.

Listing 10.14. Kod zapisujący materiały użyte do budowy pojazdu Futurgo

```
var frame_parts_exp =
/rear_view_arm_L|rear_view_arm_R|rear_view_frame_L|rear_view_frame_R/;

scene.map(frame_parts_exp, function(o) {
    o.map(Vizi.Visual, function(v) {
        that.part_materials.push(v.material);
    });
});
```

Zapisanymi materiałami można rozporządzać za pomocą elementów interfejsu użytkownika. Obiekt klasy `Futurgo` zawiera dwie dodatkowe metody, do pobierania i ustawiania koloru karoserii, które są używane przez wybierak kolorów — listing 10.15.

Listing 10.15. Kod do pobierania i ustawiania kolorów karoserii pojazdu

```
Futurgo.prototype.getBodyColor = function() {
    var color = '#ffffff';
    if (this.part_materials.length) {
        var material = this.part_materials[0];
```

```

        if (material instanceof THREE.MeshFaceMaterial) {
            color = '#' + material.materials[0].color.getHexString();
        }
        else {
            color = '#' + material.color.getHexString();
        }
    }

    return color;
}

Futuro.prototype.setBodyColor = function(r, g, b) {

    // Konwertowanie z formatu szesnastkowego RGB na zmiennoprzecinkowy.
    r /= 255;
    g /= 255;
    b /= 255;

    var i, len = this.part_materials.length;
    for (i = 0; i < len; i++) {
        var material = this.part_materials[i];
        if (material instanceof THREE.MeshFaceMaterial) {
            var j, mlen = material.materials.length;
            for (j = 0; j < mlen; j++) {
                material.materials[j].color.setRGB(r, g, b);
            }
        }
        else {
            material.color.setRGB(r, g, b);
        }
    }
}

```

Metoda `getBodyColor()` zwraca bieżący kolor materiałów karoserii. Na liście jest kilka materiałów, ale wystarczy pobrać wartość tylko pierwszego z nich, ponieważ (teoretycznie) wszystkie są takie same. Zwracana jest wartość w formacie szesnastkowym CSS. Przy jej użyciu wybierak kolorów inicjuje próbnik i wartości wejściowe, a następnie wyświetla okno dialogowe.

W metodzie `setBodyColor()` przeglądamy iteracyjnie wszystkie znajdujące się w tablicy materiały i ustawiamy ich kolor. Przypominam, że w bibliotece Three.js niektóre obiekty mogą mieć materiał typu `THREE.MeshFaceMaterial`, który w istocie jest tablicą materiałów dla poszczególnych powierzchni danego obiektu. W omawianym kodzie zostało to uwzględnione. Przekazane do tej funkcji wartości RGB są szesnastkowymi składnikami RGB koloru (tzn. liczbami całkowitymi z przedziału od 0 do 255), podczas gdy biblioteka Three.js wymaga liczb zmiennoprzecinkowych z przedziału od 0 do 1. Dlatego wykonywana jest konwersja.

Podsumowanie

W tym rozdziale zostały szczegółowo opisane czynności, jakie należy wykonać, aby utworzyć prostą, ale w pełni funkcjonalną trójwymiarową aplikację sieciową. Jako przykładu użyłem strony prezentacyjnej produktu, ponieważ jest to dobry materiał do przedstawienia wszystkich najważniejszych koncepcji. Najpierw krótko prześledziliśmy proces projektowania warstwy wizualnej, potem przyjrzaliśmy się procesowi tworzenia treści w profesjonalnym narzędziu o nazwie Maya, a następnie otrzymany wynik przekształciliśmy na nadający się do użycia w internecie format glTF. Później przy użyciu systemu szkieletowego Vizi utworzyliśmy

proste narzędzie do podglądzania i testowania modeli trójwymiarowych. Następnie zintegrowaliśmy nasz model ze stroną internetową i na koniec dodaliśmy obsługę paru zachowań i interakcji, aby uatrakcyjnić naszą aplikację i podnieść jej walory użytkowe.

Proces tworzenia trójwymiarowych aplikacji sieciowych jest dość skomplikowany, ale jeśli ma się odpowiednie narzędzia i wiedzę, można zrobić wszystko. W kolejnym rozdziale poznasz nowe rodzaje zachowań i interakcji trójwymiarowych, ale ogólne techniki i procesy opisane w tym rozdziale będą miały zastosowanie w każdym projekcie.

Tworzenie trójwymiarowego środowiska

Technik opisanych w rozdziale 10. można użyć w wielu różnych sytuacjach. Z interaktywnych aplikacji opartych na trójwymiarowych modelach obiektów korzysta się w celach marketingowych, handlowych, informacyjnych i rozrywkowych, ale na tym świat się nie kończy. Aby utworzyć wciągającą grę, prezentację budynku albo interaktywny system treningowy, należy nauczyć się tworzenia trójwymiarowych **środowisk**, zawierających wiele obiektów i obsługujących bardziej złożone typy interakcji.

W tym rozdziale zbudujemy trójwymiarowe środowisko z realistyczną scenerią i ruchomymi obiekttami, w którym użytkownik będzie mógł poruszać się po scenie za pomocą sterowania kamerą. Bazując na tym, co przy naszym udziale powstało w poprzednim rozdziale, utworzymy wirtualne miasto i wybierzemy się na przejażdżkę pojazdem Futurgo. Planowany efekt pracy jest pokazany na rysunku 11.1.



Rysunek 11.1. Samochód koncepcyjny Futurgo w trójwymiarowym środowisku

Pojazd Futurgo LTD czeka na ulicy gotowy do jazdy próbnej. Na scenie widać kilka bloków oraz parę drapaczy chmur kontrastujących z ciemnym tłem nieba, które odbija się w oknach

biurowców. Klikając scenę i przeciagając myszą, można spojrzeć do góry i do dołu oraz w prawo i w lewo. Za pomocą klawiszy strzałek można poruszać się we wszystkich kierunkach. Po dejdz do pojazdu i kliknij go, aby wsiąść. Ten świat może wydawać się trochę złowieszczy, ale gdy wsiadziemy do naszego auta, od razu poczujemy się bezpiecznie!

Aby wypróbować aplikację, otwórz plik *r11/futurgoCity.html* w przeglądarce internetowej. Podczas analizy budowy tego programu omówię kilka kwestii programistycznych. Oto one.

Tworzenie warstwy wizualnej środowiska

Utworzmy realistyczną, trójwymiarową scenę z drogami, budynkami i parkami.

Podglądarka i testowanie

Rozwinieśmy przeglądarkę, którą utworzyliśmy w poprzednim rozdziale. Tym razem potrzebujemy narzędzia wczytującego wiele plików do jednej sceny, wyświetlającego strukturę grafu sceny oraz umożliwiającego przeglądanie właściwości różnych obiektów.

Tworzenie trójwymiarowego tła

Dodamy do sceny realistyczne tło za pomocą **pudła nieba** (ang. *skybox*), czyli wyłożonego teksturową sześciangu umieszczonego w nieskończonej oddali w tle. Ta sama tekstuura jest także używana jako sześcienna mapa środowiska odbijająca niebo na budynkach i pojazdzie.

Integracja treści trójwymiarowej z aplikacją

Wczytamy do jednej aplikacji kilka modeli oraz dostosujemy oświetlenie, położenie i inne właściwości pojazdu, tak aby pasowały do otoczenia.

Implementacja nawigacji z perspektywy pierwszej osoby

Dodamy możliwość rozglądzania się i poruszania po scenie za pomocą myszy i klawiatury. Zaimplementujemy obsługę kolizji, aby nie dało się przechodzić przez obiekty.

Używanie wielu kamer

Umożliwimy zmienianie kamer, dzięki czemu użytkownik będzie mógł obejrzeć środowisko z różnych perspektyw i zwiedzać je na wiele sposobów.

Tworzenie czasowych i animowanych przejść

Użyjemy czasomierzy i technik animacyjnych w celu odtworzenia sekwencji zdarzeń podczas wsiadania do samochodu.

Zachowania obiektów

Użyjemy systemu szkieletowego Vizi w celu utworzenia własnych składników do sterowania zachowaniem i wyglądem pojazdu Futurgo.

Dźwięki

Dodamy efekty dźwiękowe za pomocą elementów HTML5.

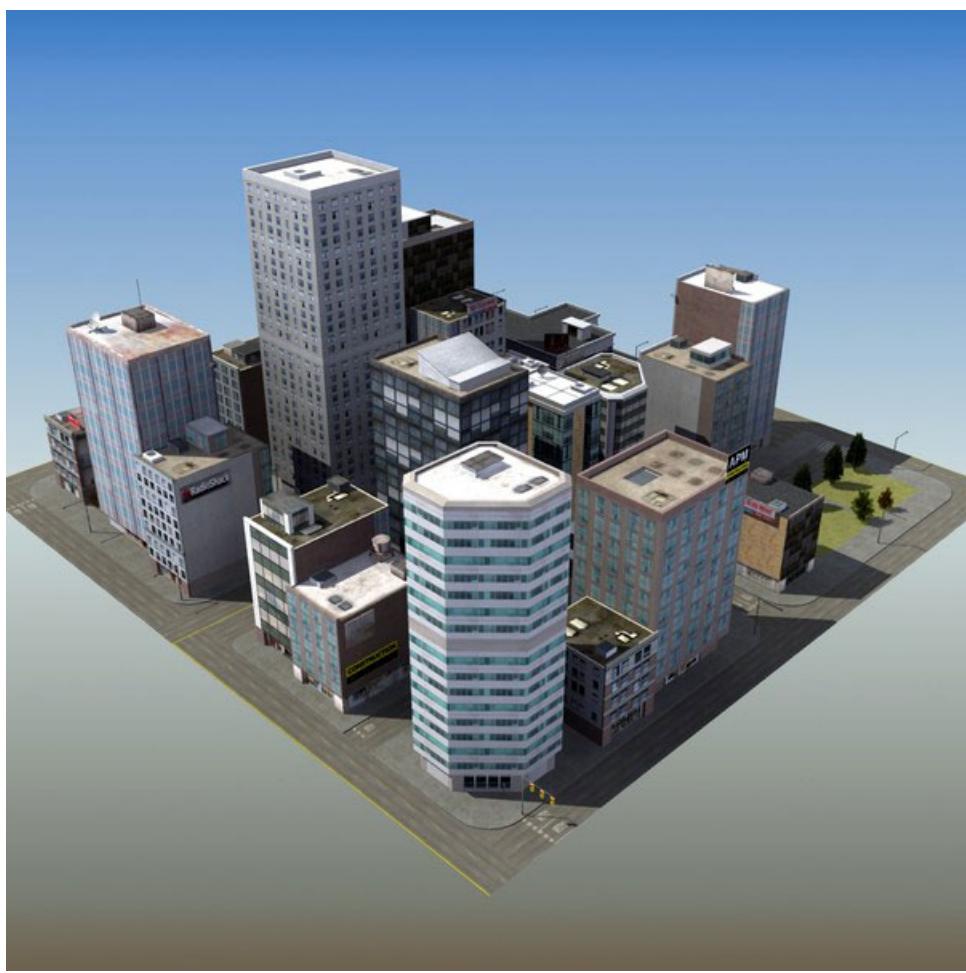
Renderowanie dynamicznych tekstur

Zaimplementujemy programowe aktualizowanie tekstur trójwymiarowych obiektów przy użyciu API Canvas 2D, aby na bieżąco wyświetlać informacje dla użytkownika.

Wirtualne środowisko, które utworzymy w tym rozdziale, będzie raczej proste. W typowej grze lub typowym środowisku trójwymiarowym znajduje się więcej obiektów i są zaimplementowane bardziej skomplikowane interakcje. Kiedy jednak poznasz opisane tu techniki, będziesz mógł budować także bardziej zaawansowane aplikacje.

Tworzenie warstwy wizualnej

Do współpracy nad trójwymiarowym środowiskiem ponownie zaprosiłem TC Changę. Jako że utworzenie tła reprezentującego miasto jest bardzo pracochłonne, postanowiliśmy poszukać gotowego modelu. Dokładnie to, czego szukaliśmy, udało się znaleźć w serwisie TurboSquid (rysunek 11.2).



Rysunek 11.2. Model miasta autorstwa ES3DStudios (<http://www.turbosquid.com/FullPreview/index.cfm?ID/652874>); obraz opublikowany dzięki uprzejmości serwisu TurboSquid

Użyty przez nas model miasta został utworzony w programie Lightwave (<https://www.lightwave3d.com/overview/>). Autor sam przekonwertował swoje dzieło na różne formaty, w tym na format programu Maya. Kupiliśmy i pobraliśmy model z internetu, a następnie TC otworzył go w programie Maya w celu przygotowania do użycia w naszej aplikacji. Do modelu dołączone zostały szczegółowe teksturyowane budynki, ale nie było światel. Dlatego TC dodał trzy źródła oświetlenia i po tym zabiegu produkt był praktycznie gotowy do użycia. Gdy wyeksportowaliśmy

model do formatu COLLADA, aby można było otworzyć go w naszej przeglądarce (opisanej w kolejnym podrozdziale), natknęliśmy się na niewielki problem z przezroczystością na niektórych teksturach. Jednak w zasadzie przygotowanie tego modelu do użytku wymagało bardzo niewiele pracy.

Podglądarka i testowanie środowiska

Do przetestowania tak skomplikowanego modelu jak scena miasta potrzebna jest przeglądarka, podobna do tej, którą utworzyliśmy w rozdziale 10., ale wzbogacona o pewne funkcje. Na rysunku 11.3 przedstawiono nową wersję przeglądarki, wyposażoną w następujące udoskonalenia.



Rysunek 11.3. Środowisko miasta wyświetcone w przeglądarce opartej na systemie Vizi

Kilka trybów widokowych

Istnieje możliwość oglądania treści jako pojedynczego modelu z kamerą skierowaną na jego środek lub jako sceny z kamerą skierowaną ku płaszczyźnie podstawowej.

Możliwość przeglądania grafu sceny

Wyświetlone jest drzewo grafu sceny przedstawiające nazwy obiektów i występujące między nimi relacje podzialeń.

Możliwość inspekcji obiektów

Można wyświetlić w wyskakującym okienku szczegółowe informacje o obiektach, takie jak sposób przekształcenia, statystki siatki, właściwości materiału oraz parametry kamery i oświetlenia.

Możliwość wyświetlania konturów

Można wyświetlić kontury wybranych obiektów oraz kontury wszystkich obiektów na scenie.

Możliwość podglądarki wielu obiektów

Mamy możliwość załadowania do jednego podglądu dodatkowych obiektów, aby obejrzeć i przetestować wynik ich połączenia.

Otwórz w przeglądarce internetowej plik `r11/previewer.html`. Kliknij przycisk *Otwórz*, aby wyświetlić okno dialogowe do otwierania plików i wybierz plik `../models/futurgo_city/futurgo_city.dae`. Na liście kamer wskaż kamerę domyślną (`[default]`), aby swobodnie poruszać się po scenie. Za pomocą kółka myszy lub gładzika możesz przybliżać i oddalać widok, aby dokładniej przyjrzeć się wybranym częściom modelu. (Możliwość swobodnej nawigacji zapewnia tylko kamera domyślna).

Podglądarkie sceny w trybie pierwszoosobowym

Podczas obracania oraz zmieniania rozmiaru modelu miasta można zauważyc, że kamera nigdy nie dociera do samego podłożu sceny (ulicy). Ma to związek z tym, że przeglądarka domyślnie traktuje model jako pojedynczy obiekt z kamerą skierowaną na geometryczny środek tego obiektu. Jednak w przypadku środowisk takie traktowanie sceny nie jest dobre, więc dodaliśmy jeszcze inny tryb widoku.

W prawym górnym rogu interfejsu przeglądarki znajdują się zgrupowane dwa przyciski radiowe (*Model* i *FPS*) służące do przełączania trybu widoku. Grupę tę nazwano *Kontroler*, aby odróżnić ją od *trybów kamery*. W naszej aplikacji miasta będzie używany tryb *FPS*, umożliwiający poruszanie się po środowisku, a nie tylko oglądanie go jako pojedynczego obiektu. (Nawigacja pierwszoosobowa jest szczegółowo opisana dalej w tym rozdziale). Kliknij przycisk *FPS*. Spowoduje to obniżenie kamery do poziomu ulicy i umożliwi zbliżenie się do jej powierzchni.

Warto zwrócić uwagę, że w trybie *FPS* nie jest używany pierwszoosobowy tryb nawigacji, tylko po prostu kamera jest ustawiana mniej więcej w tym samym miejscu, w którym zostały umieszczone, gdyby taki tryb był naprawdę włączony. Wewnętrznie w przeglądarce nadal zastosowany jest kontroler modelowy, dzięki czemu można szybko powiększać i pomniejszać cały obraz. Innymi słowy, czasami traktujemy scenę jak pojedynczy model, aby nim łatwiej manipulować, a czasami wolimy imitować rodzaje widoków potrzebne do eksplorowania środowiska w aplikacji.

Przycisk *FPS* reprezentuje prostą sztuczkę, która umożliwia skorzystanie z obu tych dobrodziejstw.

Przeglądanie grafu sceny

Kiedy sceny, z którymi pracujemy, stają się coraz bardziej skomplikowane, musimy przystosować naszą przeglądarkę tak, aby można było zobaczyć większą ilość szczegółów. Przykładowo scena miasta zawiera ponad 200 siatek, co można sprawdzić w okienku *Statystyki*. Aby zaprogramować elementy interaktywne, musimy znać nazwy, rozmiary i lokalizacje poszczególnych obiektów, a także inne ich właściwości, takie jak typ (np. siatka, kamera lub światło), oraz sposób ich pogrupowania w hierarchie. Jest to szczególnie przydatne podczas pracy z modelami pochodząymi od zewnętrznych dostawców, którzy nie tworzyli modeli w ścisłym porozumieniu z nami.

Jednym z prymitywnych sposobów przeglądania grafu sceny jest otwarcie pliku w formacie COLLADA lub glTF w edytorze tekstu i znalezienie łańcuchów tekstowych oznaczających typy. Jednak dla większości programistów to zbyt żmudne zadanie, które na dodatek wymaga dogłębnej znajomości budowy plików w tych formatach. (Znam te formaty bardzo dobrze, ale i tak nie mam cierpliwości do przeczytywania wielkich ilości tekstu). Znacznie lepszą metodą jest wyświetlenie odpowiednich informacji przez przeglądarkę.

W naszej ulepszonej wersji przeglądarki znajduje się nowe okienko o nazwie *Scena*, zawierające drzewo reprezentujące hierarchię grafu sceny. Aby je obejrzeć, przewin kawałek listy i kliknij znak plus lub minus, aby rozwinąć albo zwinąć wybrany poziom hierarchii. Na najwyższym poziomie znajduje się kilka światel, po których widać grupę o nazwie *MidTower_Block_01* i kilka kamer. Zwróć uwagę na znak plus obok nazwy grupy. Jego kliknięcie powoduje rozwinięcie grupy i wyświetlenie poziomu podległego, zawierającego takie nazwy jak *Tower_A_01*, *Roof_Detail_01* itd. Niektóre z tych grup także można rozwijać, aby obejrzeć dalsze poziomy hierarchii.

Kiedy mamy możliwość podglądania nazw węzłów i hierarchicznych relacji między obiektyami w obrębie sceny, możemy wyznaczyć obiekty, dla których w działającej aplikacji zaimplementujemy funkcje interaktywności i inne szczegóły. Przykładowo po załadowaniu sceny do aplikacji planujemy dodać mapy środowiskowe odbijające tło pudła nieba, ale tylko na budynkach, natomiast na ulicach i parkach już nie. Wystarczy rzut oka na hierarchię sceny, aby dowiedzieć się, że nazwy wszystkich budynków zaczynają się od słów **Tower** i **Office**. W związku z tym możemy użyć metody API grafu sceny systemu Vizi o nazwie *Vizi.Object.map()*, aby za pomocą wyrażenia regularnego znaleźć wszystkie interesujące nas obiekty i zmienić ich materiały. Odpowiedni kod źródłowy pokazano dalej w tym rozdziale.

Użyta w przeglądarce kontrolka widoku drzewa została zaimplementowana przy użyciu wtyczki do jQuery o nazwie dynatree (<http://code.google.com/p/dynatree/>). Na listingu 11.1 można zobaczyć kod inicjujący tę kontrolkę z różnymi opcjami oraz kod procedur obsługi zdarzeń kliknięcia i podwójnego kliknięcia elementów. Kod źródłowy całej przeglądarki znajduje się w pliku *r11/previewer.html*.

Listing 11.1. Inicjacja kontrolki widoku drzewa dynatree

```
function initSceneTree(viewer) {
    // Inicjacja drzewa w elemencie <div>.
    $("#scene_tree").dynatree({
        imagePath: "./images/previewer_skin/",
        title: "Scene Graph",
        minExpandLevel: 2,
        selectMode: 1,
        onDbClick: function(node) {
            openSceneNode(viewer, node);
        },
        onActivate: function(node) {
            selectSceneNode(viewer, node);
            if (infoPopupVisible) {
                openSceneNode(viewer, node);
            }
        },
        onDeactivate: function(node) {
        },
        onFocus: function(node) {
        },
        onBlur: function(node) {
        },
    });
}
```

Teraz prześledzimy proces zapelniania kontrolki drzewa treścią grafu sceny po załadowaniu pliku sceny. Najpierw przyjrzymy się w metodzie zwrotnej ładowania wywołania funkcji pomocniczej o nazwie *updateSceneTree()*:

```

function onLoadComplete(data, loadStartTime)
{
    // Ukrywa pasek ładowania.
    var loadStatus = document.getElementById("loadStatus");
    loadStatus.style.display = 'none';

    viewer.replaceScene(data);

    var loadTime = (Date.now() - loadStartTime) / 1000;
    var loadTimeStats = document.getElementById("load_time_stats");
    loadTimeStats.innerHTML = "Czas ładowania:<br>" + loadTime.toFixed(2) + " s"
    // Vizi.System.log("Załadowano w " + loadTime.toFixed(2) + " s.");

    updateSceneTree(viewer);
    updateCamerasList(viewer);
    updateLightsList(viewer);
    updateAnimationsList(viewer);
    updateMiscControls(viewer);

    if (viewer.cameraNames.length > 1) {
        selectCamera(1);
    }

    addRollovers(viewer, data.scene);
}

```

Funkcja `updateSceneTree()` wykonuje kilka czynności. Najpierw za pomocą wywołania `removeChildren()` na korzeniu widoku drzewa ponownie inicjuje widżet kontrolki drzewa, na wypadek gdyby został wcześniej użyty w innej scenie. Następnie wywołuje funkcję `buildSceneTree()`, która iteruje po grafie sceny i wstawia elementy do kontrolki drzewa. Należy zwrócić uwagę, że wywołanie to jest nieco opóźnione za pomocą funkcji `setTimeout()`. Opóźnienie to zostało zastosowane dla wygody użytkownika. Tworzenie drzewa przy użyciu dynatree zajmuje trochę czasu, a nie chcemy, żeby czynność ta opóźniła renderowanie sceny. Dlatego na początku umieściliśmy tekst, który później został usunięty.

```

function updateSceneTree(viewer) {

    // Przykład: dodanie hierarchicznej gałęzi przy użyciu kodu.
    // W ten sposób dodalibyśmy węzły drzewa programowo.
    var rootNode = $("#scene_tree").dynatree("getRoot");
    rootNode.removeChildren();
    var initMessage = rootNode.addChild({
        title: "Inicjowanie...",
        isFolder: false,
    });

    setTimeout(function() {
        rootNode.removeChild(initMessage);
        rootNode.expand(false);
        var i, len = viewer.scenes.length;
        for (i = 0; i < len; i++) {
            buildSceneTree(viewer.scenes[i], rootNode);
        }
    }, 1000);
}

```

Kod do zapełniania drzewa sceny jest dość prosty. Na listingu 11.2 pokazano kod źródłowy funkcji `buildSceneTree()`, który znajduje się też w pliku `r11/sceneTree.js`.

Listing 11.2. Funkcja zapełniająca drzewo sceny

```
sceneTreeMap = {};  
  
buildSceneTree = function(scene, tree) {  
  
    function build(object, node, level) {  
  
        var noname = level ? "[object]" : "Scene";  
  
        var childNode = node.addChild({  
            title: object.name ? object.name : noname,  
            expand: level <= 1,  
            activeVisible:true,  
            vizi:object,  
        });  
  
        sceneTreeMap[object._id] = childNode;  
  
        var i, len = object._children.length;  
        for (i = 0; i < len; i++) {  
            build(object._children[i], childNode, level+1);  
        }  
    }  
  
    build(scene, tree, 0);  
}  
}
```

Najpierw inicjujemy globalny obiekt o nazwie `sceneTreeMap`, przy użyciu którego powiążemy obiekty Vizi znajdujące się w grafie sceny Vizi z elementami w kontrolce widoku drzewa. Powiązań tych użyjemy do obsługi kliknięć obiektów na scenie oraz wyróżniania pozycji klikniętych w kontrolce.

W funkcji `buildSceneTree()` zdefiniowaliśmy zagnieżdżoną funkcję `build()`, która rekurencyjnie dodaje elementy do kontrolki drzewa. Funkcja ta dla każdego obiektu w grafie sceny Vizi tworzy nowy węzeł za pomocą wywołania `node.addChild()`. Metoda ta buduje nowy element o określonych parametrach.

Parametr `title` definiuje etykietę elementu. Parametr `expand` określa, czy dany element powinien być wstępnie rozwinięty. Z opcji tej korzystamy dla elementów najwyższego poziomu. Ustawienie `activeVisible` nakazuje kontrolce przewinięcie do elementu i zaznaczenie go, jeśli jest aktywny (tzn. wybrany na poziomie kodu, np. po kliknięciu na scenie związanego z nim obiektu). Ostatni parametr to `vizi`, czyli obiekt grafu sceny Vizi, który jest używany, gdy użytkownik kliknie element w kontrolce drzewa. Kliknięcie obiektu sprawia, że zostaje on otoczony żółtą ramką, a podwójne kliknięcie powoduje wyświetlenie okienka z jego właściwościami (następny podrozdział).

Po utworzeniu elementu kontrolki drzewa dodajemy go do obiektu `sceneTreeMap` w celu jego późniejszego użycia i rekurencyjnie wywołujemy metodę `build()`, aby dodać elementy kontrolki dla potomków obiektu, jeśli istnieją.



Zbudowanie dobrego drzewa przy użyciu elementów HTML wymaga sporo pracy. Na szczęście, twórcy widżetu dynatree zaoszczędzili nam tej męki. Widżet ten umożliwia szybkie tworzenie drzew z dowolnej hierarchicznej listy HTML. Ponadto jest wyposażony w bogaty interfejs API do tworzenia, modyfikowania i usuwania elementów oraz zawiera opcje do dostosowywania wyglądu drzewa. Widżet można pobrać pod adresem <http://code.google.com/p/dynatree/>.

Przeglądanie właściwości obiektów

W naszej przeglądarce istnieje możliwość przeglądania właściwości wszystkich obiektów. Wystarczy dwukrotnie kliknąć dowolny obiekt w drzewie, aby wyświetlić jego **kartę charakterystyki** w postaci okienka dialogowego jQuery — widać to na rysunku 11.4. Przedstawiona na zrzucie ekranu karta charakterystyki zawiera właściwości obiektu o nazwie Tower_D_01. Znajdują się na niej trzy karty z informacjami: o przekształceniach (pozycja, obrót, skala), o geometrii i otaczającej ramce obiektu (np. liczbie wierzchołków i powierzchni w siatce) oraz o materiale (włącznie z modelem cieniowania, kolorami i nazwą obrazu użytego w charakterze tekstury).



Rysunek 11.4. Przeglądanie właściwości obiektu w przeglądarce

Ponadto w przeglądarce można wyświetlić właściwości dowolnego obiektu, klikając go na scenie. Jednokrotne kliknięcie obiektu, gdy karta charakterystyki jest już wyświetlona, spowoduje zamianę jej zawartości na informacje dotyczące klikniętego obiektu. Jeśli karta nie jest wyświetlona, można spowodować jej wyświetlenie za pomocą dwukrotnego kliknięcia wybranego przedmiotu na scenie.

Na listingu 11.3 przedstawiony został kod obsługi kliknięć na scenie trójwymiarowej (z pliku *r11/previewer.html*), umożliwiający wybieranie poszczególnych obiektów. Funkcja *addRollovers()* korzysta z metody *map()* z API grafu sceny Vizi do znalezienia każdego obiektu na scenie i dodania obsługi myszy poprzez utworzenie obiektu *Vizi.Picker*. W kodzie tym zdefiniowane są procedury obsługi zdarzeń kliknięcia i puszczenia przycisku myszy, najechania kursorem na obiekt oraz podwójnego kliknięcia.

Listing 11.3. Implementacja procedury wybierania obiektów na scenie

```
function addRollover(viewer, o) {
    var picker = new Vizi.Picker;
    picker.addEventListener("mouseover", function(event) {
        onPickerMouseOver(viewer, o, event); });
    picker.addEventListener("mouseout", function(event) {
        onPickerMouseOut(viewer, o, event); });
    picker.addEventListener("mouseup", function(event) {
        onPickerMouseUp(viewer, o, event); });
```

```

    picker.addEventListener("dblclick", function(event) {
        onPickerMouseDoubleClick(viewer, o, event);
    });
}

function addRollovers(viewer, scene) {
    scene.map(Vizi.Object, function(o) {
        addRollover(viewer, o);
    });
}

```

Teraz przeanalizujemy kod procedur obsługi zdarzeniaпусzczenia прыціску мышы, używanej do wykrywania pojedynczych kliknięć, i obsługi zdarzenia dwukrotnego kliknięcia. Są one prawie identyczne. Jako że w naszej przeglądarce obiekty można wybierać tylko za pomocą lewego przycisku myszy, najpierw sprawdzamy kod przycisku biorącego udział w zdarzeniu. Jeśli wykryjemy, że został naciśnięty lewy przycisk, wywołujemy metodę `highlightObject()` obiektu `Vizi.Viewer`. Metoda ta rysuje żółtą ramkę wokół klikniętego obiektu. (Szczegółowy opis implementacji procedury rysującej tę ramkę znajduje się w następnym podrozdziale).

Następnie zaznaczamy powiązany z klikniętym obiektem element widoku drzewa. W tym celu używamy właściwości `_id` obiektu `Vizi`, która jest automatycznie generowana przez system `Vizi` w momencie utworzenia obiektu. Służy ona jako indeks umożliwiający znalezienie elementu do wyróżnienia. W końcu, jeśli użytkownik kliknął tylko raz, a karta charakterystyki była już wyświetlona (o czym świadczy odpowiednia wartość logiczna zmiennej `infoPopupVisible`), wywołujemy funkcję pomocniczą `openSceneNode()`, która ponownie wstawia do karty charakterystyki dane będące danymi nowo wybranego węzła. W przypadku dwukrotnego kliknięcia również wywoływana jest funkcja `openSceneNode()`, co powoduje wyświetlenie okna dialogowego, jeśli nie jest jeszcze widoczne.

```

function onPickerMouseUp(viewer, o, event) {
    if (event.button == 0) {
        viewer.highlightObject(o);
        node = selectSceneNodeFromId(viewer, o._id);
        if (node && infoPopupVisible) {
            openSceneNode(viewer, node);
        }
    }
}

function onPickerMouseDoubleClick(viewer, o, event) {
    if (event.button == 0) {
        viewer.highlightObject(o);
        node = selectSceneNodeFromId(viewer, o._id);
        openSceneNode(viewer, node);
    }
}

```

Wyświetlanie ramek obiektów

Ramki obiektów są używane w przeglądarce do dwóch celów: do zaznaczania wybranego obiektu oraz do wyświetlania obramowania wszystkich obiektów za pomocą specjalnej opcji interfejsu użytkownika.

Aby można było zaznaczyć wybrany obiekt, w obiekcie `Vizi.Viewer` umieszczono metodę `highlightObject()`, której implementacja znajduje się na listingu 11.4. Najpierw usuwane jest obramowanie poprzednio zaznaczonego obiektu, jeśli takie istnieje. Następnie przeglądarka

oblicza ramkę nowego obiektu i przy jej użyciu tworzy żółte obramowanie, które wyświetla na ekranie.

W kodzie tym należy zwrócić uwagę na pewne szczegóły, które są ukryte w pogrubionych wierszach. Tworzymy obiekt klasy `Vizi.Decoration` do przechowywania geometrii ramki. Klasa ta jest specjalną podklassą klasy `Vizi.Visual` używanej przez system do renderowania wiadoczej na ekranie treści, ale nie do obsługi interakcji. Nie ma ona wpływu na wybieranie obiektów pod myszą ani na kolizje obiektów. Następnie dodajemy tę dekorację do **nadrzędnego** obiektu. Obramowanie każdego obiektu jest obliczane w układzie współrzędnych obiektu nadrzędnego, a więc musimy je dodać do grafu sceny jako potomka obiektu nadrzędnego, by zapewnić jego poprawne przekształcenie.

Listing 11.4. Tworzenie obramowania dla wybranego obiektu

```
Vizi.Viewer.prototype.highlightObject = function(object) {  
  
    if (this.highlightedObject) {  
        this.highlightedObject._parent.removeComponent(  
            this.highlightDecoration);  
    }  
  
    if (object) {  
        var bbox = Vizi.SceneUtils.computeBoundingBox(object);  
  
        var geo = new THREE.CubeGeometry(bbox.max.x - bbox.min.x,  
            bbox.max.y - bbox.min.y,  
            bbox.max.z - bbox.min.z);  
  
        var mat = new THREE.MeshBasicMaterial({color:0xaaaa00,  
            transparent:false,  
            wireframe:true, opacity:1});  
  
        var mesh = new THREE.Mesh(geo, mat);  
        this.highlightDecoration = new Vizi.Decoration({object:mesh});  
        object._parent.addComponent(this.highlightDecoration);  
  
        var center = bbox.max.clone().add(bbox.min)  
            .multiplyScalar(0.5);  
        this.highlightDecoration.position.add(center);  
    }  
  
    this.highlightedObject = object;  
}
```

W przeglądarce można wyświetlić obramowanie wszystkich obiektów. Służy do tego opcja o nazwie *Kontury* znajdująca się w okienku *Różne*, w dolnej części interfejsu użytkownika. Zaznaczenie tej opcji powoduje wyświetlenie zielonych ramek wszystkich obiektów na scenie, co pokazano na rysunku 11.5.

Kod wyświetlający obramowanie wszystkich obiektów jest podobny do kodu wyświetlającego kontury wybranego obiektu. Różnica polega na tym, że stosujemy go do wszystkich obiektów w grafie sceny za pomocą metody `map()` z API grafu sceny `Vizi` — listing 11.5.



Rysunek 11.5. Scena z wyświetlonymi ramkami wszystkich obiektów

Listing 11.5. Tworzenie i renderowanie konturów wszystkich obiektów na scenie

```
this.sceneRoot.map(Vizi.Object, function(o) {
    if (o._parent) {
        var bbox = Vizi.SceneUtils.computeBoundingBox(o);

        var geo = new THREE.CubeGeometry(bbox.max.x - bbox.min.x,
            bbox.max.y - bbox.min.y,
            bbox.max.z - bbox.min.z);
        var mat = new THREE.MeshBasicMaterial(
            {color:0x00ff00, transparent:true,
             wireframe:true, opacity:.2});
        var mesh = new THREE.Mesh(geo, mat);
        var decoration = new Vizi.Decoration({object:mesh});
        o._parent.addComponent(decoration);

        var center = bbox.max.clone().add(bbox.min)
            .multiplyScalar(0.5);
        decoration.position.add(center);
        decoration.object.visible = this.showBoundingBoxes;
    }
});
```

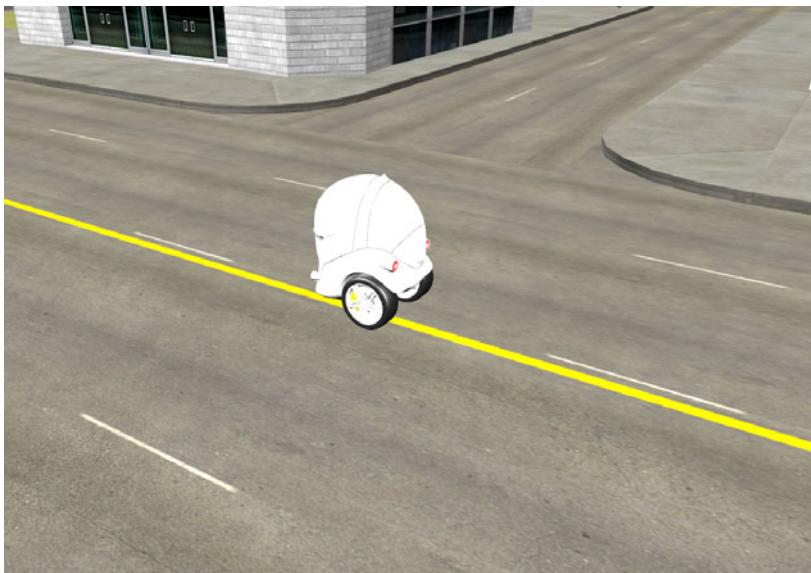
Teraz kliknięcie pola wyboru *Kontury* w celu włączenia lub wyłączenia omawianej opcji spowoduje wywołanie przez przeglądarkę metody `setBoundingBoxes()`. Metoda ta za pomocą funkcji `map()` znajduje wszystkie obiekty typu `Vizi.Decoration` i zmienia ich widoczność poprzez zmianę ustawienia właściwości `visible`.

```
Vizi.Viewer.prototype.setBoundingBoxesOn = function(on)
{
    this.showBoundingBoxes = !this.showBoundingBoxes;
    var that = this;
    this.sceneRoot.map(Vizi.Decoration, function(o) {
        if (!that.highlightedObject || (o != that.highlightDecoration)) {
            o.visible = that.showBoundingBoxes;
        }
    });
}
```

Oglądanie wielu obiektów

Podczas tworzenia środowiska zawierającego wiele obiektów kluczem do sukcesu jest możliwość podejrzenia i przetestowania ich wszystkich naraz. Trzeba sprawdzić, czy wszystkie obiekty są w tej samej skali, czy są prawidłowo położone względem siebie, czy są właściwie oświetlone itd. Sprawdzenie tego wszystkiego jest szczególnie ważne, gdy obiekty pochodzą z wielu źródeł, zostały utworzone przez różnych artystów lub pochodzą z różnych serwisów internetowych z modelami.

Wczytamy model samochodu Futurgo, aby sprawdzić, czy wszystko będzie w porządku. W tym celu kliknij przycisk *Dodaj* na górnym pasku narzędzi. W oknie dialogowym wyboru pliku wybierz plik *./models/futurgo_mobile/futurgo_mobile.json*. (Pamiętaj, aby włączyć kamerę domyślną i ustawić ją na główną ulicę znajdującą się pośrodku sceny — w tym miejscu powinien pojawić się pojazd). Gdy samochód znajdzie się na środku sceny, możesz go przybliżyć, aby dokładniej mu się przyjrzeć — rysunek 11.6.



Rysunek 11.6. Model Futurgo w scenie miasta

Kod obsługujący dodawanie kolejnych modeli do sceny jest prawie taki sam jak kod obsługujący ładowanie pierwszego modelu. Tworzymy obiekt klasy *Vizi.Loader* i dodajemy procedurę na słuchu zdarzeń załadowania modelu, w której dokładamy nowe obiekty sceny do przeglądarki. Jedyna różnica w porównaniu z poprzednim kodem polega na tym, że obiekty do przeglądarki **dodajemy**, a nie zastępujemy nimi poprzedniej treści. Omawiany kod został pokazany na listingu 11.6 (można go też znaleźć w pliku *r11/previewer.html*). Wywołujemy w nim metodę *viewer.addToScene()*, za pomocą której dodajemy obiekty do aktualnego grafu sceny (w tym przypadku jest to model samochodu Futurgo) oraz aktualizujemy struktury danych przeglądarki. Następnie aktualizujemy elementy interfejsu użytkownika, tak jak poprzednio, czyli drzewo widoku oraz listy światel, kamer i animacji.

Listing 11.6. Dodawanie modeli do sceny

```
function onAddComplete(data, loadStartTime)
{
    // Ukrywa pasek wczytywania.
    var loadStatus = document.getElementById("loadStatus");
    loadStatus.style.display = 'none';

    viewer.addToScene(data);

    var loadTime = (Date.now() - loadStartTime) / 1000;
    var loadTimeStats = document.getElementById("load_time_stats");
    loadTimeStats.innerHTML = "Load time<br>" + loadTime.toFixed(2) + "s"
    // Vizi.System.log("Czas ładowania: " + loadTime.toFixed(2) + "s.");

    updateSceneTree(viewer);
    updateCamerasList(viewer);
    updateLightsList(viewer);
    updateAnimationsList(viewer);
    updateMiscControls(viewer);

    addRollovers(viewer, data.scene);
}
```

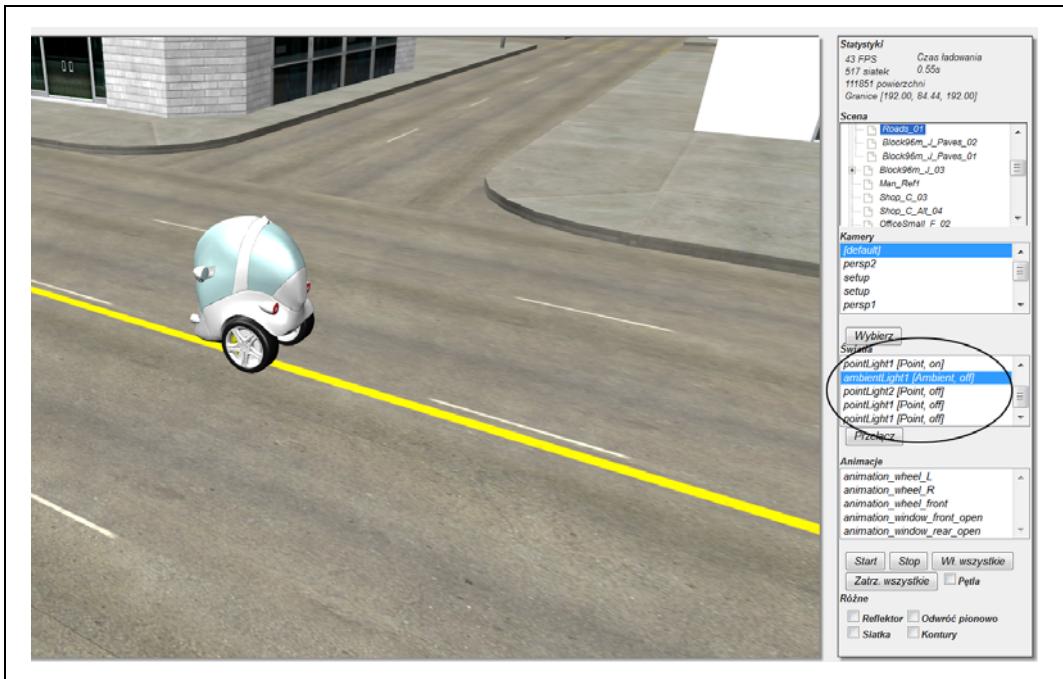
Zapewne zwróciłeś uwagę na to, że załadowany model pojazdu jest prawie całkowicie biały. Jest to spowodowane tym, że model ten zawiera własne oświetlenie, którego używaliśmy w aplikacji zbudowanej w rozdziale 10. Mogłem poprosić TC o utworzenie specjalnej wersji samochodu bez oświetlenia dla tej aplikacji, ale nie było takiej potrzeby. W przeglądarce można znaleźć sprawiające problemy światła i je wyłączyć. Warto też zapisać sobie ich nazwy, aby to samo zrobić we właściwej aplikacji.

Przeszkadzające światła wyłączymy za pomocą listy światel w przeglądarce. Miałem przeczucie (które mnie nie myliło), że będą to światła znajdujące się na końcu listy, ponieważ model Futurgo przecież został dodany do sceny jako ostatni. W związku z tym wyłączyłem trzy ostatnie światła punktowe. Jednak pojazd nadal był trochę wyblakły, więc wyłączyłem też światło otaczające. I to załatwiło sprawę. Na rysunku 11.7 widać efekt wykonania opisanych czynności. Zmiany wykonane w interfejsie użytkownika zostały zaznaczone pętelką.

Wyszukiwanie za pomocą przeglądarki innych problemów ze sceną

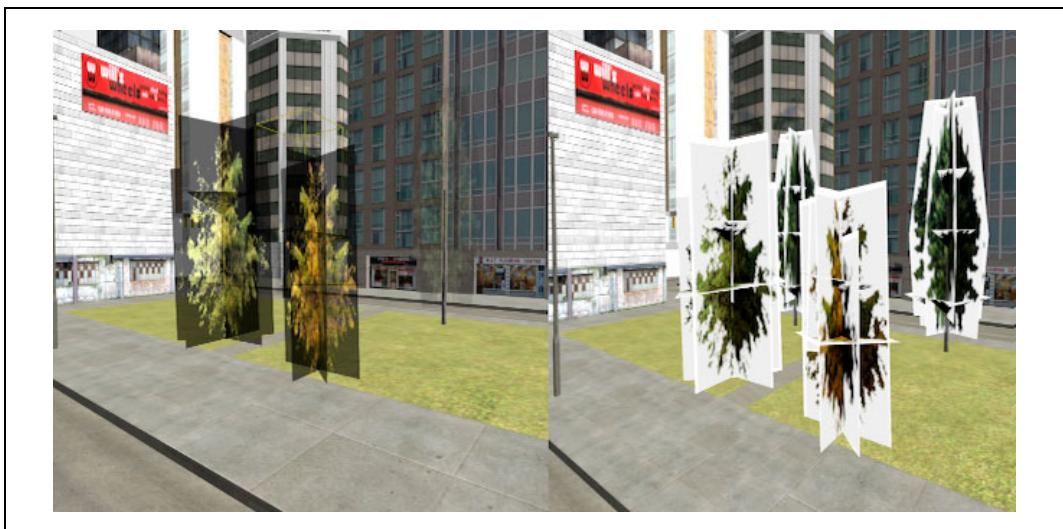
Dzięki przeglądarce udało się wykryć jeszcze jeden problem techniczny ze sceną, dotyczący drzew. Twórca sceny użył starej jak świat sztuczki do efektywnego renderowania drzew: skorzystał ze zbioru nakładających się płaskich wielokątów z teksturami drzewa widocznego pod różnymi kątami. Zazwyczaj krzyżuje się dwa pionowe wielokąty i przecina się je jednym lub większą liczbą poziomych wielokątów. Jest to jedna ze sztuczek używanych od lat przez twórców modeli, aby zminimalizować liczbę wielokątów. Wyobraź sobie, ile trójkątów trzeba by użyć, aby utworzyć realistycznie wyglądające drzewo.

Jedyny problem dla WebGL związany z drzewami w tej scenie dotyczy formatu grafik. Tekstury wszystkich wielokątów są parami plików w formacie Microsoft BMP. Jeden z nich zawiera informacje dotyczące kolorów, a drugi — maskę alfa. Nie da się tego łatwo wykorzystać w aplikacji opartej na systemach Vizi i Three.js. Technicznie jest to oczywiście wykonalne, ale aktualnie



Rysunek 11.7. Model Futurgo wczytany do sceny miasta z dostosowanym oświetleniem

silnik nie udostępnia takiej możliwości. Dlatego poprosiłem TC, aby przekonwertował pary plików BMP na pojedyncze pliki PNG z kanałem alfa. Gdy to zrobił, wprowadziłem odpowiednie zmiany w pliku programu Maya i ponownie wyeksportowałem. Na rysunku 11.8 pokazano porównanie efektu przed modyfikacją i po niej.



Rysunek 11.8. Porównanie drzew w przeglądarce przed konwersją z formatu BMP na PNG (po lewej) i po konwersji (po prawej). Białe otoczki drzew po prawej stronie są artefaktami przeglądarki i znikną w aplikacji po włączeniu przezroczystości w Three.js



Wprawdzie trudno się zgodzić, że obraz po prawej wygląda lepiej, ale w gotowej aplikacji będzie wyświetlony poprawnie. Widoczny artefakt ma związek z niedoskonałością naszej przeglądarki. Mimo że w pliku PNG znajdują się informacje dotyczące przezroczystości, ani Vizi, ani Three.js nie wykorzystują tego faktu, jeśli programista nie ustawia przezroczystości dla samego materiału. Jako że wartości te nie są określone w treści, w aplikacji trzeba będzie ustawić je ręcznie po załadowaniu sceny.

Tworzenie trójwymiarowego tła przy użyciu pudła nieba

Po obejrzeniu modelu miasta, usunięciu usterek i sprawdzeniu, jak komponuje się z nim pojazd Futurgo, można rozpocząć budowę właściwej aplikacji. Jednak wcześniej musimy zająć się jeszcze jedną kwestią. W tej chwili model miasta jest całkiem atrakcyjny, ale obejmuje tylko cztery bloki. Jeśli chcemy mieć realistyczną scenę, po której będzie można jeździć samochodem, musimy stworzyć iluzję dużego miasta. W tym celu wyrenderujemy tło w postaci pudła nieba.

Trójwymiarowe pudło nieba

W odróżnieniu od typowych obrazów tła używanych na stronach internetowych, do naszej aplikacji potrzebujemy tła trójwymiarowego, które będzie się zmieniać podczas obracania kamery. **Pudło nieba** (ang. *skybox*) to panoramiczny obraz składający się z sześciu tekstur nałożonych od wewnętrz na boki sześcianu. Sześcian ten jest renderowany ze stacjonarnej kamery, która obraca się wraz z kamerą i ukazuje różne części tła. Pudło nieba to bardzo prosta technika tworzenia realistycznych, trójwymiarowych tła.

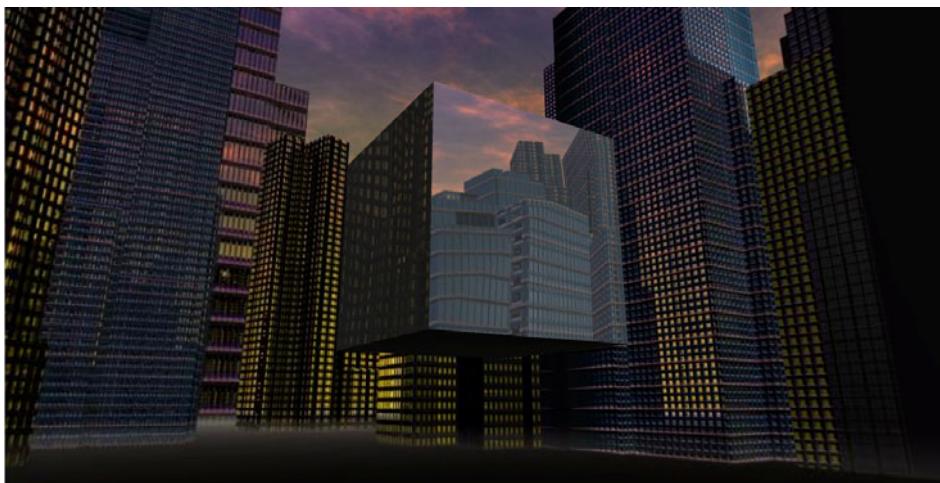
W zestawie przykładów do biblioteki Three.js znajduje się kilka demonstracji zastosowania pudła nieba, np. [webgl_materials_cubemap_balls_reflection.html](#). W tym przypadku tło wygląda doskonale, ale jest to prymitywna implementacja o poważnych ograniczeniach. Autorzy tych przykładów utworzyli na obrzeżach sceny bardzo duże sześciany, które wydają się bardzo oddalone. Gdyby można było poruszać się po scenie, użytkownik mógłby zbliżyć się do ich ścian, a to zepsułoby iluzję.



W grafice trójwymiarowej iluzja to podstawa. Pudła nieba stwarzają realistyczną iluzję nieskończonej sceny, ale tylko wtedy, gdy użytkownik nie zbliży się do ich ścian. Jeśli oglądasz genialny film *Truman Show* Petera Weira, może pamiętasz scenę, kiedy Truman uderza ręką w ścianę, która została narysowana w stworzonym dla niego przez reżysera sztucznym świecie. Gdy Truman wbiegł w tę ścianę, wszystko się wydało.

Obiekt Skybox systemu Vizi

Aby utworzyć przekonującą scenę miasta, należy użyć odpowiedniego pudła nieba. Na szczęście, w systemie Vizi już takie jest. Zanim dodamy pudło nieba do naszej aplikacji, przeanalizujemy prosty przykład, aby zobaczyć, jak w ogóle się to robi. Otwórz w przeglądarce internetowej plik [r11/skybox.html](#), aby zobaczyć scenę widoczną na rysunku 11.9. Za pomocą myszy możesz obrócić scenę, żeby obejrzeć całe jej tło. Przy użyciu kółka myszy lub gładzika możesz przybliżać i oddalać scenę. Sześcian przybliża się i oddala, ale tło cały czas pozostaje w oddali.



Rysunek 11.9. Pudło nieba w tle i sześciian na pierwszym planie — gdy użytkownik przemieszcza się do przodu lub do tyłu, przybliża się do sześciianu lub oddala się od niego, ale tło pozostaje cały czas w tej samej odległości. Na sześcianie widać odbicie tła uzyskane dzięki użyciu sześciennnej mapy środowiska z tą samą teksturą

Warto też zwrócić uwagę na to, że pudło nieba odbija się na powierzchni sześciianu. Efekt ten uzyskano dzięki użyciu sześciennnej mapy środowiska utworzonej za pomocą tej samej tekstury.

Panoramiczny obraz pudła nieba składa się z sześciu map bitowych o układzie pokazanym na rysunku 11.10. Mapy te zostały utworzone tak, aby po nałożeniu ich na wnętrzu sześciianu nie było widać jakichkolwiek łączeń.



Rysunek 11.10. Sześć tekstur tworzących sześcienną teksturę pudła nieba (tekstury pudła nieba zostały pobrane ze strony <http://3delyvisions.com/skf1.htm>)

Na listingu 11.7 przedstawiony został kod tworzący pudło nieba i dodający je do sceny. Najpierw za pomocą narzędzia Three.js THREE.ImageUtils.loadTextureCube() tworzymy sześcienną teksturę. Następnie wywołujemy funkcję Vizi.Prefabs.Skybox() w celu utworzenia gotowego obiektu (*prefabrykatu*) pudła nieba. Na koniec ustawiamy własność texture pudła na teksturę sześcienną i gotowe pudło dodajemy do aplikacji.

Listing 11.7. Tworzenie pudła nieba

```
var app = new Vizi.Application({ container : container });

// Pudło nieba ze strony http://www.3delyvisions.com/
// http://www.3delyvisions.com/skfl.htm
var path = "../images/sky35/";

var urls = [ path + "rightcity.jpg", path + "leftcity.jpg",
    path + "topcity.jpg", path + "botcity.jpg",
    path + "frontcity.jpg", path + "backcity.jpg" ];

var cubeTexture = THREE.ImageUtils.loadTextureCube( urls );

var skybox = Vizi.Prefabs.Skybox();
var skyboxScript = skybox.getComponent(Vizi.SkyboxScript);
skyboxScript.texture = cubeTexture;

app.addObject(skybox);
```



O co chodzi z tymi „prefabrykatami”?

W Vizi *prefabrykat* to gotowy zestaw obiektów i składników, które można tworzyć i wstawiać bezpośrednio do sceny. Obiekty tego typu są zaprojektowane w sposób typowy dla silników gier, takich jak Unity. Przypominam, że w nowoczesnych silnikach gier odchodzi się od tradycyjnego tworzenia klas w celu rozszerzania funkcjonalności i dąży do agregowania komponentów w celu tworzenia bogatszych struktur.

Pudło nieba w systemie Vizi jest sześcianem rysującym tło i śledzącym ruch kamery głównej, aby utrzymać poprawne położenie. Jeśli chcesz dowiedzieć się, jak zaimplementowane są obiekty gotowe w systemie Vizi, zajrzyj do pliku *objects/skybox.js* w plikach źródłowych tego systemu.

Widoczny w omawianym przykładzie sześcian odbija obraz tła. Efekt ten dało się łatwo uzyskać przy użyciu tej samej tekstury jako mapy środowiska na materiale sześcianu. Odpowiedni kod źródłowy jest pokazany na listingu 11.8.

Listing 11.8. Dodawanie sześciennej tekstury do obiektu pierwszego planu

```
var cube = new Vizi.Object();

var visual = new Vizi.Visual(
    { geometry: new THREE.CubeGeometry(2, 2, 2),
        material: new THREE.MeshPhongMaterial({
            color:0xffffffff,
            envMap:cubeTexture,
            reflectivity:0.8,
            refractionRatio:0.1
        })
    });

cube.addComponent(visual);
app.addObject(cube);
```

Dodawanie do aplikacji trójwymiarowej treści

Przy użyciu przeglądarki przejrzałem graf sceny, aby sprawdzić nazwy i właściwości obiektów, oraz wykonaliśmy inspekcję oświetlenia i innych wizualnych aspektów treści. Zanotowaliśmy też, co trzeba zrobić z modelami po załadowaniu ich do gotowej aplikacji. Ponadto dowiedzieliśmy się, jak utworzyć pudło nieba i jego odbicie na obiektach na scenie. Możemy więc w końcu złożyć całą aplikację z naszego środowiska trójwymiarowego.

Ładowanie i inicjowanie środowiska

Aplikacja testowa Futurgo znajduje się w pliku HTML *r11/futurgoCity.html*. Znajdujący się w tym pliku kod jest bardzo prosty. Tworzymy w nim tylko egzemplarz klasy *FuturgoCity*, która ładuje modele, składa scenę oraz uruchamia aplikację. Kod źródłowy tej klasy znajduje się w pliku *r11/futurgoCity.js*.

Kod aplikacji rozpoczyna się od załadowania modelu miasta. Metoda zwrotna dotycząca wczytywania plików o nazwie *onLoadComplete()* składa środowisko. Spójrz na listing 11.9. Po wywołaniu metody przeglądarki *replaceScene()* dodającej nowo załadowaną treść do sceny nakazujemy przeglądarce wyłączenie nawigacji pierwszoosobowej za pomocą wywołania *setController("FPS")*. (Szczegółowy opis kontrolerów kamery i nawigacji pierwszoosobowej znajduje się dalej w tym rozdziale). Następnie zapisujemy informacje o kontrolerze kamery przeglądarki i bieżącej kamerze, których będziemy potrzebować nieco później. Potem wywołujemy kilka metod pomocniczych w celu dodania pudła nieba i map środowiska oraz wykonujemy parę innych, ważnych czynności przygotowawczych.

Listing 11.9. Funkcja zwrotna wywoływana po załadowaniu środowiska

```
FuturgoCity.prototype.onLoadComplete = function(data, loadStartTime)
{
    var scene = data.scene;
    this.scene = data.scene;
    this.viewer.replaceScene(data);

    if (this.loadCallback) {
        var loadTime = (Date.now() - loadStartTime) / 1000;
        this.loadCallback(loadTime);
    }

    this.viewer.setController("FPS");
    this.cameraController = this.viewer.controllerScript;
    this.walkCamera = this.viewer.defaultCamera;

    this.addBackground();
    this.addCollisionBox();
    this.fixTrees();
    this.setupCamera();
    this.loadFuturgo();
}
```

Podobnie jak w przykładzie z poprzedniego podrozdziału, metoda *addBackground()* tworzy pudło nieba, a następnie dodaje mapy środowiska do budynków. Przypominam, że wcześniej za pomocą przeglądarki sprawdziliśmy, jak nazywają się największe budynki. Ich nazwy zaczynają się od słów „Tower” i „Office”. Trzeba zwrócić uwagę na pogrubiony wiersz kodu za-

wierający wyrażenie regularne. Za pomocą metody `map()` grafu sceny Vizi znajdujemy pasujące obiekty i dla każdego z nich ustawiamy mapę środowiskową na materiale Three.js.

```
this.scene.map(/Tower.*|Office.*/, function(o) {  
    var visuals = o.visuals;  
    if (visuals) {  
        for (var vi = 0; vi < visuals.length; vi++) {  
            var v = visuals[vi];  
            var material = v.material;  
            if (material) {  
                if (material instanceof THREE.MeshFaceMaterial) {  
                    var materials = material.materials;  
                    var mi, len = materials.length;  
                    for (mi = 0; mi < len; mi++) {  
                        addEnvMap(materials[mi]);  
                    }  
                }  
                else {  
                    addEnvMap(material);  
                }  
            }  
        }  
    }  
});
```

Teraz dodajemy pole kolizji. Dalej w tym rozdziale napiszę, jak zaimplementować kolizję w taki sposób, aby współgrała z poruszaniem się po scenie w trybie pierwszoosobowym. Na razie przedstawiam kod dotyczący tworzenia niewidocznego pola na granicach miasta, aby nie dało się z niego wyjść. Kod ten jest bardzo prosty: tworzymy nowy obiekt klasy `Vizi.Visual` zawierający sześcian o rozmiarze odpowiadającym rozmiarowi sceny i ustawiamy jego własność `opacity` na 0, aby był całkowicie przezroczysty. Ponadto kolizja musi następować z *wewnętrzna* stroną ściany sześcianu. W tym celu nakazujemy bibliotece Three.js renderowanie tylnych powierzchni geometrii sześcianu poprzez ustawienie jego właściwości `side` na wartość wyliczonową `THREE.DoubleSide` (oznaczającą renderowanie obu stron sześcianu). Odpowiedni kod źródłowy przedstawiony został na listingu 11.10.

Listing 11.10. Dodanie pola kolizji do sceny

```
FuturgoCity.prototype.addCollisionBox = function() {  
  
    var bbox = Vizi.SceneUtils.computeBoundingBox(this.scene);  
  
    var box = new Vizi.Object;  
    box.name = "_futurgoCollisionBox";  
  
    var geometry = new THREE.CubeGeometry(bbox.max.x - bbox.min.x,  
                                           bbox.max.y - bbox.min.y,  
                                           bbox.max.z - bbox.min.z);  
  
    var material = new THREE.MeshBasicMaterial({  
        transparent:true,  
        opacity:0,  
        side:THREE.DoubleSide  
    });  
  
    var visual = new Vizi.Visual({  
        geometry : geometry,  
        material : material});
```

```
        box.addComponent(visual);

        this.viewer.addObject(box);
    }
}
```

Musimy też naprawić usterkę związaną z przezroczystością drzew. W metodzie `fixTrees()` ponownie używamy funkcji `map()` w celu znalezienia wszystkich węzłów, których nazwa zaczyna się od słowa „Tree”, odszukujemy wszystkie znajdujące się w tych węzłach obiekty wizualne i ustawiamy własność `transparent` ich materiałów na `true`. Ustawienie to włącza mieszanie alfa w systemie renderowania Three.js. Gdybyśmy tego nie zrobili, drzewa byłyby renderowane bez przezroczystości, tak jak pokazano na rysunku 11.8.

```
this.scene.map(/^Tree.*/, function(o) {

    o.map(Vizi.Visual, function(v){
        var material = v.material;
        if (material instanceof THREE.MeshFaceMaterial) {
            var materials = material.materials;
            var i, len = materials.length;
            for (i = 0; i < len; i++) {
                material = materials[i];
                material.transparent = true;
            }
        }
        else {
            material.transparent = true;
        }
    });
});
```

Po umieszczeniu kamery w odpowiednim miejscu możemy przejść do kolejnego, ważnego etapu budowania aplikacji, czyli ładowania modelu samochodu.

Ładowanie i inicjowanie modelu samochodu

Ładowanie i przygotowywanie modelu samochodu to proces kilkuetapowy. Należy dodać załadowany model do sceny, dodać efekt zmiany przezroczystości okien, dodać mapy środowiska do okien i karoserii, aby odbijało się w nich pudło nieba, usunąć niepotrzebne światła, które widzieliśmy już w przeglądarce, oraz umieścić samochód w odpowiednim miejscu. Po tym wszystkim trzeba jeszcze utworzyć interaktywne obiekty do jeżdżenia samochodem i jego animowania, ale o tym będzie mowa w następnych podrozdziałach.

Funkcja zwrotna dotycząca załadowania pliku zaczyna się od wywołania metody `this.viewer.add_toScene()` w celu dodania obiektu do sceny. Następnie, podobnie jak w aplikacji opisanej w poprzednim rozdziale, dodajemy automatycznie włączany dwusekundowy efekt zmiany przezroczystości okien na półprzezroczystość. Ponadto obiekty dotyczące tego efektu zapisujemy we właściwości aplikacji `faders`, która jest tablicą; użyjemy jej później, aby jeszcze bardziej zwiększyć przezroczystość okien po wejściu użytkownika do samochodu oraz do przywrócenia poprzedniego stanu, gdy wysiądzie. Podczas iterowania przez materiały okien dodajemy także tę samą mapę środowiska, której użyliśmy na budynkach, czyli teksturę sześcienną przedstawiającą drapacze chmur w pudle nieba. Opisaną sekwencję wywołań zawiera kod przedstawiony na listingu 11.11.

Listing 11.11. Kod wywołania zwracającego do obsługi ładowania samochodu Futurgo

```
FuturgoCity.prototype.onFuturgoLoadComplete = function(data) {  
  
    // Dodaje model Futurgo do sceny.  
    this.viewer.addToScene(data);  
    var futurgoScene = data.scene;  
  
    // Dodaje interakcje i zachowania.  
    var that = this;  
  
    // Dodaje mapę środowiska i „fadery” do okien.  
    // Zmienia przezroczystość okien po uruchomieniu aplikacji.  
    this.faders = [];  
    futurgoScene.map(/windows_front|windows_rear/, function(o) {  
  
        var fader = new Vizi.FadeBehavior({duration:2,  
            opacity:FuturgoCity.OPACITY_SEMI_OPAQUE});  
        o.addComponent(fader);  
        fader.start();  
        that.faders.push(fader);  
  
        var visuals = o.visuals;  
        var i, len = visuals.length;  
        for (i = 0; i < len; i++) {  
            visuals[i].material.envMap = that.envMap;  
            visuals[i].material.reflectivity = 0.1;  
            visuals[i].material.refractionRatio = 0.1;  
        }  
    });  
};
```

Następnie dodajemy mapę środowiska do karoserii samochodu (metalowa rama i lusterka wsteczne).

```
// Dodaje mapę środowiska do karoserii.  
futurgoScene.map(/body2/, function(o) {  
    var visuals = o.visuals;  
    var i, len = visuals.length;  
    for (i = 0; i < len; i++) {  
        visuals[i].material.envMap = that.envMap;  
        visuals[i].material.reflectivity = 0.1;  
        visuals[i].material.refractionRatio = 0.1;  
    }  
});
```

Następnie iterujemy przez wszystkie części karoserii, aby dodać obiekty `Vizi.Picker`. Dzięki temu jazdę testową będzie można rozpoczęć kliknięciem dowolnego miejsca na powierzchni samochodu Futurgo. Obiekty te zapisujemy w tablicy obiektu `pickers`, ponieważ każdy z nich trzeba wyłączyć podczas wsiadania do samochodu i włączyć podczas wysiadania z niego.

Później zajmiemy się problemem ze światłami, który zauważyliszyśmy podczas podglądzania modelu w przeglądarce. Dodatkowe oświetlenie obecne w tym modelu w połączeniu ze światłami sceny powodowało, że model wydawał się wyblakły. Dlatego wyłączymy światła dostarczone wraz z modelem pojazdu oraz światło otaczające z modelem miasta.

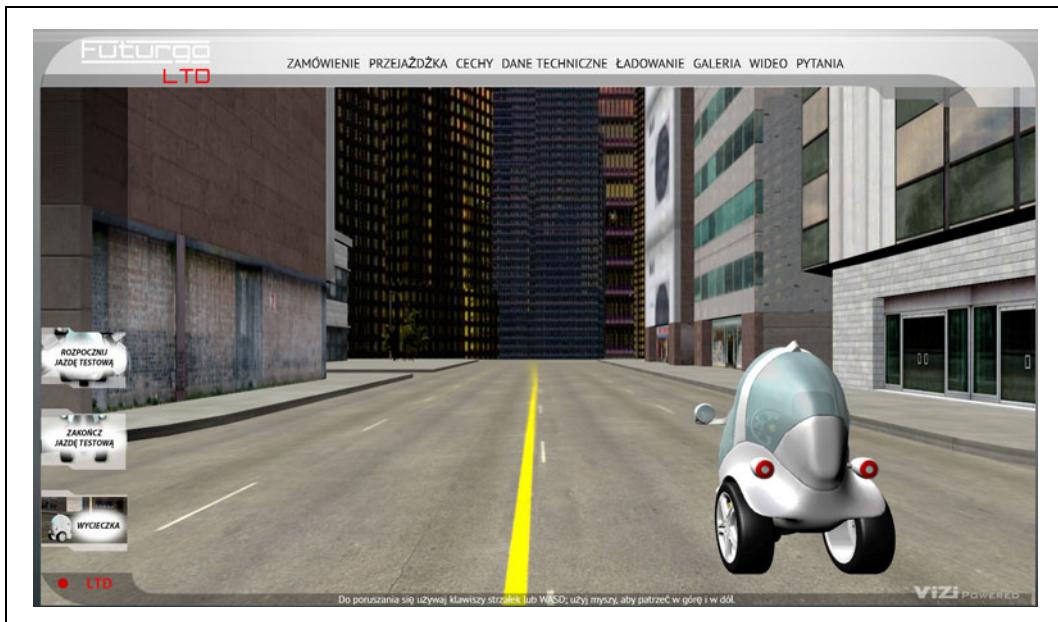
```
// Połączenie światel z dwóch scen sprawia, że samochód jest wyblakły.  
// Wyłącza światła dostarczone wraz z modelem samochodu.  
futurgoScene.map(Vizi.PointLight, function(light) {  
    light.intensity = 0;  
});
```

```
// Dodatkowo należy wyłączyć światło otaczające, dostarczone z modelem miasta.
this.scene.map(/ambient/, function(o) {
    o.light.color.set(0, 0, 0);
});
```

W końcu ustawiamy pojazd w dogodnym miejscu początkowym. Jako że wartości x i z położenia kamery są zerowe, samochód ustawiamy o kilka jednostek w prawo i w tył względem tej pozycji.

```
// Ustawia pojazd Futurgo w dogodnej pozycji początkowej.
var futurgo = futurgoScene.findNode("vizi_mobile");
futurgo.transform.position.set(2.33, 0, -6);
```

Wciąż jeszcze musimy dodać parę zachowań i interakcji do jazdy samochodem, ale tym zajmiemy się nieco później. Na razie mamy już kompletną scenę, z pudłem nieba w tle i odbiciami w postaci map środowiska. Samochód został ustawiony w odpowiednim miejscu, jego okna są częściowo przezroczyste, a mapy środowiska tła odbijają się na jego karoserii. Na rysunku 11.11 pokazano, jak wygląda nasza strona bezpośrednio po załadowaniu.



Rysunek 11.11. Początkowy stan aplikacji Futurgo

Rozejrzyj się po planszy. Poruszaj myszą, aby popatrzeć dookoła, naciśnij klawisze strzałek i przejdź się po scenie, obejrzyj piętrzące się nad głową budynki, w których odbija się niebo. Wszystko to stworzyliśmy podczas kilku dni pracy. Robi wrażenie.

Jednak koniec tego guzdrania. Czas wracać do pracy.

Implementowanie nawigacji pierwszoosobowej

Po załadowaniu środowiska chcielibyśmy mieć możliwość poruszania się po nim. Użytkownik powinien zwiedzać miasto pieszo lub naszym samochodem. W tym podrozdziale dowiesz się, jak zaimplementować rodzaj nawigacji, taki jak w niektórych grach, tzw. **nawigację pierwszoosobową** (ang. *first-person navigation*).

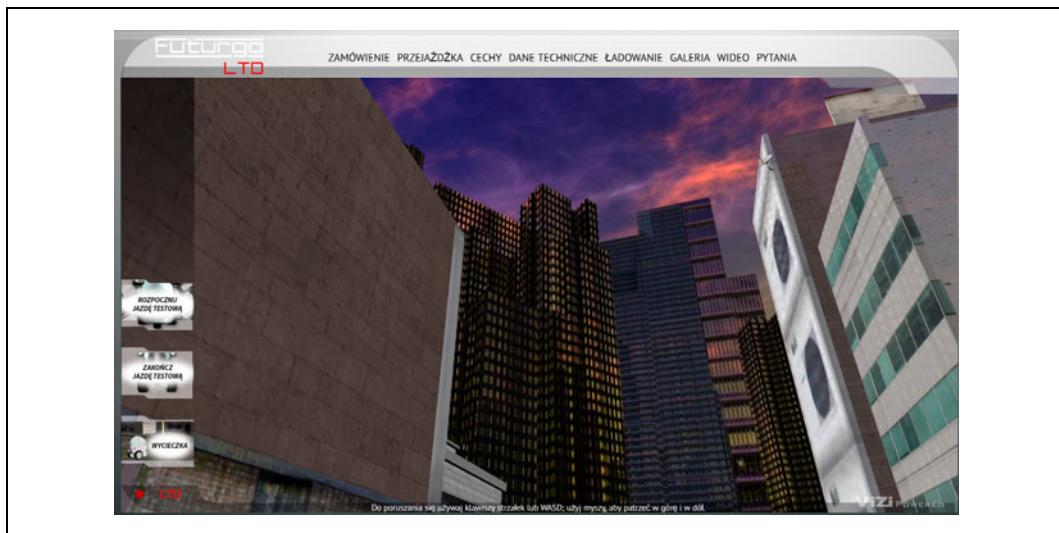
Pojęcia **pierwszoosobowy** (ang. *first-person*) lub **perspektywa pierwszej osoby** (ang. *first-person perspective*) oznaczają renderowanie trójwymiarowej sceny z punktu widzenia użytkownika, tak jakby kamera była umieszczona między jego oczami. **Nawigacja** pierwszoosobowa to tryb sterowania kamerą za pomocą myszy, klawiatury, dżojstika lub innego tego typu urządzenia. Nawigacja pierwszoosobowa jest powszechnie stosowana w grach komputerowych, zwłaszcza w strzelankach, czyli tzw. grach FPS (ang. *first-person shooter*).

W komputerach stacjonarnych przy nawigacji pierwszoosobowej najczęściej używa się myszy do sterowania kamerą i klawiatury do chodzenia po scenie lub obracania postaci. W tabeli 11.1 przedstawione są typowe zastosowania klawiszy w tego rodzaju nawigacjach. Strzałki służą do poruszania się w przód, tył, lewo i prawo. Takie same zastosowanie mają klawisze WASD, dzięki czemu lewą ręką można poruszać postacią w grze, a w prawej trzymać mysz, aby sterować kamerą albo strzelać do wrogów.

Tabela 11.1. Typowe zastosowania klawiszy w trybie pierwszoosobowym

Klawisz lub czynność wykonywaną myszą	Działanie
W, strzałka w góre	Ruch do przodu
A, strzałka w lewo	Ruch w lewo
S, strzałka w dół	Ruch do tyłu
D, strzałka w prawo	Ruch w prawo
Pociągnięcie myszą do góry	Obrócenie kamery do góry
Pociągnięcie myszą w dół	Obrócenie kamery w dół
Pociągnięcie myszą w lewo	Obrócenie kamery w lewo
Pociągnięcie myszą w prawo	Obrócenie kamery w prawo

Przejdź się po mieście za pomocą klawiszy WASD oraz spójrz w górę, w dół, w lewo i w prawo, klikając lewym przyciskiem myszy i przeciągając po scenie w odpowiednim kierunku — rysunek 11.12.



Rysunek 11.12. Zwiedzanie miasta w trybie pierwszoosobowym

Kontrolery kamery

Nawigację pierwszoosobową zaimplementujemy przy użyciu obiektu zwanego **kontrolerem kamery**. Obiekty te, zgodnie z nazwą, kontrolują ruch kamery na podstawie danych otrzymywanych od użytkownika. Klasa `Vizi.Viewer` obsługuje różne tryby kontrolowania kamery, takie jak modelowy i pierwszoosobowy. Dla każdego z tych trybów automatycznie tworzy obiekty kontrolujące. Wystarczy tylko wywołać metodę obiektu klasy `Vizi.Viewer` o nazwie `setController()`, która pobiera łańcuch wskazujący, którego kontrolera należy użyć. Możliwe wartości to "model" i "FPS".

W aplikacji Futurgo przedstawionej w rozdziale 10. użyto kontrolera modelowego, służącego do krążenia wokół oglądanego obiektu. W efekcie użytkownikowi wydaje się, że model się obraca, choć w rzeczywistości to kamera jest przesuwana za pomocą myszy lub innego urządzenia sterującego. Ten rodzaj nawigacji doskonale nadaje się do użytku w prezentacjach pojedynczych modeli. Jednak w aplikacjach przedstawiających miasto lepsza jest nawigacja pierwszoosobowa.

Kontroler pierwszoosobowy — obliczenia

Kluczem do implementacji kontrolera pierwszoosobowego jest przekształcenie zmian pozycji kurSORA na kąty obrotu kamery. Przykładowo przeciągnięcie myszą w lewo lub w prawo powinno powodować odpowiednie obrócenie kamery wokół osi *y*, a przeciągnięcie w dół i w górę — wokół osi *x*. Naciśnięcia wybranych klawiszy powinny wywoływać ruch w kierunku, w którym aktualnie zwrócona jest kamera, np. naciśnięcie strzałki w górę powinno spowodować ruch po prostej linii w kierunku, w którym zwrócona jest kamera.

Aby zobaczyć, jak wyglądają obliczenia dotyczące programowania kontrolera pierwszoosobowego, przeanalizujemy parę fragmentów kodu z implementacją `Vizi`. Metoda `update()` klasy `Vizi.FirstPersonControls`, która jest wywoływana w każdej iteracji pętli wykonawczej, oblicza kąt obrotu wokół osi *x* i *y* (listing 11.12).

Listing 11.12. Kod z klasy Vizi.FirstPersonControls

```
if (this.mouseDragOn || this.mouseLook) {  
  
    var deltax = this.lastMouseX - this.mouseX;  
    var dlon = deltax / this.viewHalfX * 900;  
    this.lon += dlon * this.lookSpeed;  
  
    var deltay = this.lastMouseY - this.mouseY;  
    var dlat = deltay / this.viewHalfY * 900;  
    this.lat += dlat * this.lookSpeed;  
  
    this.theta = THREE.Math.degToRad( this.lon );  
  
    this.lat = Math.max( - 85, Math.min( 85, this.lat ) );  
    this.phi = THREE.Math.degToRad( this.lat );  
  
    var targetPosition = this.target,  
        position = this.object.position;  
  
    targetPosition.x = position.x - Math.sin( this.theta );  
    targetPosition.y = position.y + Math.sin( this.phi );  
    targetPosition.z = position.z - Math.cos( this.theta );
```

```

        this.object.lookAt( targetPosition );

        this.lastMouseX = this.mouseX;
        this.lastMouseY = this.mouseY;
    }
}

```

Najpierw obliczamy zmianę współrzędnych x i y położenia kurSORA wzgórđem poprzedniego położenia. Następnie przekształcamy otrzymany wynik na deltę obrotu w stopniach dłucości i szerokości geograficznej. Lokalna zmienna `dlon` reprezentuje zmianę dłucości geograficznej w stopniach. Obliczamy ją wg następującego wzoru:

```
var dlon = deltax / this.viewHalfX * 900;
```

Zmianę położenia kurSORA na osi x dzielimy przez połowę szerokości ekranu w celu obliczenia, jaki procent szerokości ekranu przebył kurSOR w poziomie. Każde 10% szerokości ekranu jest równe 90 stopniom obrotu (dlatego mnożymy przez 900). Następnie dodajemy tę deltę do bieżącego obrotu poziomego:

```
this.lon += dlon * this.lookSpeed;
```

Później wartość obrotu poziomego w stopniach jest przekształcana na radiany, bo tego wymaga biblioteka Three.js, i zostaje zapisana we właściwości `this.theta`:

```
this.theta = THREE.Math.degToRad( this.lon );
```

W podobny sposób, przy użyciu zmian położenia kurSORa na osi y obliczany jest obrót pionowy. Obliczona wartość zostaje zapisana we właściwości `this.phi`. Przy użyciu nowych wartości można wykonać obrót widoku. W tym celu obliczamy „kierunek patrzenia” na sferze jednostkowej, której środek pokrywa się z punktem umiejscowienia kamery, i za pomocą metody `lookAt()` biblioteki Three.js kierujemy kamerę na to miejsce. Teraz kamera jest skierowana w nowym kierunku.

```

targetPosition.x = position.x - Math.sin( this.theta );
targetPosition.y = position.y + Math.sin( this.phi );
targetPosition.z = position.z - Math.cos( this.theta );
this.object.lookAt( targetPosition );

```

Ruch kamery odbywa się po linii patrzenia. Jeżeli użytkownik naciśnie jeden z klawiszy nawigacyjnych, oznaczamy ten fakt za pomocą odpowiedniego ustalenia właściwości logicznych `moveForward`, `moveBackward`, `moveLeft` oraz `moveRight` i sprawdzamy je w metodzie `update()`.

```

this.update = function( delta ) {

    this.startY = this.object.position.y;

    var actualMoveSpeed = delta * this.movementSpeed;

    if ( this.moveForward )
        this.object.translateZ( - actualMoveSpeed );
    if ( this.moveBackward )
        this.object.translateZ( actualMoveSpeed );

    if ( this.moveLeft )
        this.object.translateX( - actualMoveSpeed );
    if ( this.moveRight )
        this.object.translateX( actualMoveSpeed );

    this.object.position.y = this.startY;
}

```

Do obliczenia nowej pozycji kamery używamy biblioteki Three.js. Metody `translateZ()` i `translateX()` przesuwają kamerę wzdłuż odpowiednich osi. Jako że kierunek kamery może

być odchylony w góre lub w dół względem osi poziomej, ruch również może odbyć się w góre lub w dół, ale nie chcemy, żeby tak się stało. Wolimy cały czas być na ziemi. Dlatego nadpisujemy wszelkie zmiany w pozycji na osi *y*, przywracając zapisaną wcześniej wartość.

Wybieranie kierunku patrzenia za pomocą myszy

W omawianej aplikacji użytkownik może kliknąć na scenie myszą i przeciągnąć nią, aby obrócić kamerę. W grach zazwyczaj można zrobić to samo *bez klikania*. Tryb ten nazywa się „patrzeniem myszą” (ang. *mouse look*). To doskonałe rozwiązańe w grach FPS, ponieważ jest szybkie i bezproblemowe. Ponadto nie angażuje przycisku myszy, który można wykorzystać do strzelania albo innych celów.

Jednak w aplikacji sieciowej patrzenie myszą może być porażką, ponieważ użytkownik może zechcieć kliknąć coś na pasku narzędzi albo wybrać element w dwuwymiarowym interfejsie użytkownika na stronie. Gdy tylko poruszy myszą, od razu zmieni się też kąt patrzenia kamery na scenie trójwymiarowej. Każdy taki ruch będzie powodował zmianę widoku na scenie, co wcale nie jest zabawne. Jeśli chcesz zobaczyć, na czym to polega, w omawianej aplikacji ustaw własność kontrolera *mouseLook* na *true*. Moim zdaniem, patrzenie myszą to technika nadająca się do użycia tylko w aplikacjach pełnoekranowych.



W trybie pełnoekranowym często też ukrywa się kurSOR, co można zaobserwować w większości gier typu FPS. Nowsze przeglądarki również umożliwiają zastosowanie tej techniki zwanej **blokowaniem kurSora** (ang. *pointer lock*) lub **blokowaniem myszy** (ang. *mouse lock*). Na internetowej stronie [https://dvcs.w3.org/hg\(pointerlock/raw-file/tip/index.html](https://dvcs.w3.org/hg(pointerlock/raw-file/tip/index.html)) jest dostępna oficjalna rekomendacja W3C dotycząca tej funkcji. Ponadto na stronie <http://www.html5rocks.com/en/tutorials/pointerlock/intro/>) znajduje się ciekawy artykuł autorstwa Johna McCutcheona.

Proste wykrywanie kolizji

Przy tworzeniu złudzenia realistycznego środowiska ważnym czynnikiem jest **wykrywanie kolizji** (ang. *collision detection*), czyli sprawdzanie, czy pole widzenia użytkownika (albo innego obiektu) nie koliduje z geometrią sceny, oraz uniemożliwianie przechodzenia przez tę geometrię. Przecież miasto nie byłoby realistyczne, gdyby użytkownik mógł przechodzić przez ściany budynków.

W tym punkcie zaimplementujemy w mieście Futurogo bardzo proste wykrywanie kolizji. Wykorzystamy obiekty matematyczne z biblioteki Three.js do rzucania promienia z punktu oka i znajdowania obiektów leżących na linii widzenia. Jeśli na tej linii w określonej odległości od użytkownika zostaną znalezione obiekty, będzie to oznaczać kolizję i użytkownik nie będzie mógł dalej iść.

Klasa *Vizi.FirstPersonControllerScript* jest częścią prefabrykatu implementującego pierwszoosobowy system nawigacji systemu Vizi. Na listingu 11.13 pokazano część jej kodu źródłowego. Najpierw zapisujemy pierwotną pozycję kamery. Następnie *Vizi.FirstPersonControls* aktualizuje pozycję kamery na podstawie danych otrzymanych z myszy i klawiatury. Efektem może być zmiana pozycji kamery. Następnie wywołujemy metodę pomocniczą *testCollision()*, aby dowiedzieć się, czy ruch od pozycji zapisanej do nowej pozycji mógłby spowodować kolizję. Jeśli tak, przywracamy pierwotną pozycję kamery i wysyłamy zdarzenie *collide*, które mogą odebrać ewentualne procedury nasłuchujące. (Zapewne takie się znajdują, ale szerzej na ten temat piszę nieco dalej).

Listing 11.13. Kod wykrywania kolizji z kontrolera pierwszoosobowego

```
Vizi.FirstPersonControllerScript.prototype.update = function()
{
    this.saveCamera();
    this.controls.update(this.clock.getDelta());
    var collide = this.testCollision();
    if (collide && collide.object) {
        this.restoreCamera();
        this.dispatchEvent("collide", collide);
    }
}
```

Teraz przyjrzymy się metodzie `testCollision()`. Przypomnij sobie kod znajdowania obiektów pod kursorem z rozdziału 9. System graficzny Vizi używa techniki *ray castingu* biblioteki Three.js do znajdowania punktu przecięcia promienia biegącego z punktu oka z geometrią na scenie. Jeśli część promienia znajdująca się między minimalną odległością 1 a maksymalną odległość 2 przecina jakąś geometrię, zostaje zwrócony obiekt, który jest zapisywany w zmiennej `collide`.

```
Vizi.FirstPersonControllerScript.prototype.testCollision = function() {

    this.movementVector.copy(this._camera.position).sub(this.savedCameraPos);
    if (this.movementVector.length()) {

        var collide = Vizi.Graphics.instance.objectFromRay(null,
            this.savedCameraPos,
            this.movementVector, 1, 2);

        if (collide && collide.object) {
            var dist = this.savedCameraPos.distanceTo(collide.hitPointWorld);
        }

        return collide;
    }

    return null;
}
```

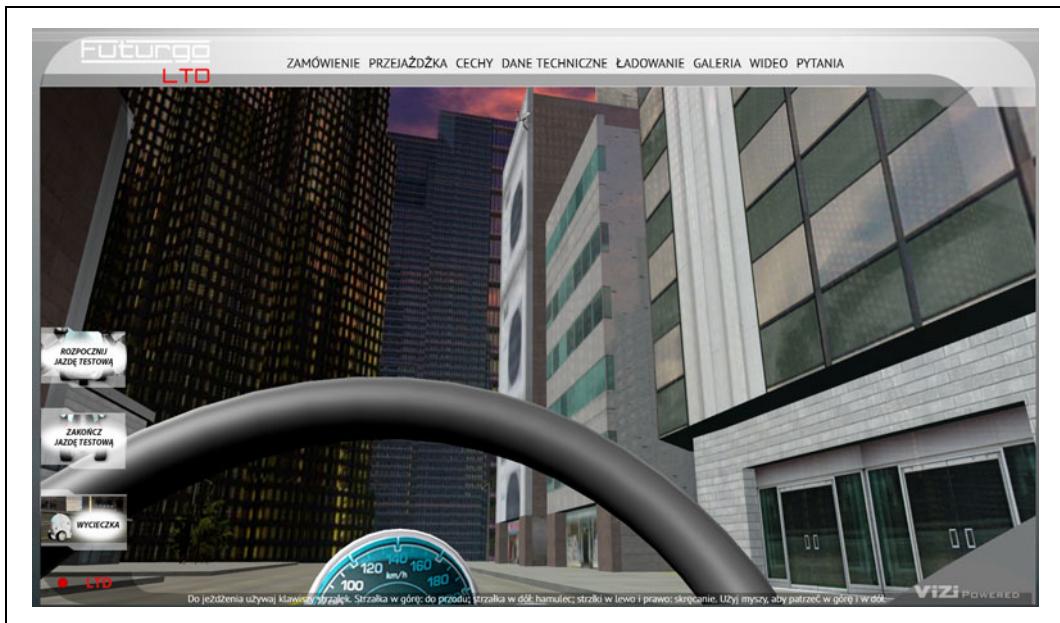


Przedstawiony algorytm to najprostsza możliwa implementacja wykrywania kolizji. Z pozycji kamery rzucamy promień w kierunku patrzenia. Promień nie ma objętości, tzn. jest nieskończoność cienki. Nie jest to bardzo realistyczne, bo prawdziwe awatary mają krzywizny, a przynajmniej objętość. W bardziej zaawansowanym algorytmie do kolizji użylibyśmy sfery, cylindra lub jakiegoś innego obiektu geometrycznego. Takie rozwiązania są stosowane w większości gier. Jednak nam wystarczy wykrywanie kolizji oparte na wysyłaniu promienia.

Posługiwanie się wieloma kamerami

Jedną z największych zalet scen trójwymiarowych jest możliwość używania różnych kamer, co pozwala na renderowanie sceny z odmiennych punktów widzenia oraz pod wieloma kątami i w różnych proporcjach. Wprawdzie wszystko to dałoby się też zrealizować za pomocą jednej kamery o dynamicznie zmienianych właściwościach, ale w bibliotece Three.js bardzo łatwo tworzy się dodatkowe kamery, które można zapisać w celu użycia w dogodnym momencie. W systemie Vizi kamery Three.js są opakowane w składniki. Można jełączyć oraz wykonywać wiele czynności w sposób niezauważalny, np. można automatycznie zmieniać proporcje obrazu w odpowiedzi na zmianę rozmiaru okna. Wykorzystamy te możliwości w naszej

aplikacji i utworzymy drugą kamerę umieszczoną wewnątrz samochodu. Na rysunku 11.13 pokazano właśnie widok z tej kamery.



Rysunek 11.13. Widok z kamery umieszczonej wewnątrz pojazdu Futurgo

Na listingu 11.14 znajduje się kod tworzący tę dodatkową kamerę. Najpierw budujemy nowy obiekt klasy `Vizi.Object` o nazwie `driveCam` do przechowywania kamery. Obiekt ten zostanie dodany jako potomek obiektu samochodu `Futurgo`. Dlaczego? Aby kamera poruszała się wraz z pojazdem, gdy ten ruszy (szerzej piszę o tym trochę dalej). Przypomnij sobie opis **hierarchii przekształceń** z wcześniejszego rozdziału: właściwości przekształceń obiektu (pozycja, obrót, skala) dotyczą także jego obiektów potomnych. Dlatego gdy samochód porusza się, wraz z nim porusza się kamera.

Listing 11.14. Tworzenie kamery wewnątrz pojazdu

```
// Wstawia kamerę do pojazdu.  
var driveCam = new Vizi.Object;  
var camera = new Vizi.PerspectiveCamera;  
camera.near = 0.01;  
driveCam.addComponent(camera);  
futurgo.addChild(driveCam);  
// Uwzględnia skalę w modelu, aby poprawnie umiejscowić kamerę.  
var scaley = futurgo.transform.scale.y;  
var scalez = futurgo.transform.scale.z;  
var camy = FuturgoCity.AVATAR_HEIGHT_SEATED / scaley;  
var camz = 0 / scalez;  
driveCam.transform.position.set(0, camy, camz);  
this.driveCamera = camera;
```

Następnie pozycjonujemy kamerę w pojeździe. Samo dodanie jej jako obiektu dziecka obiektu `Futurgo` powoduje domyślne ustawienie jej na początku tego obiektu, a więc na ziemi. Musimy to zmienić i w tym celu zrobimy coś niezwykłego: TC pozostawił w pojeździe wartość

określającą skalę. (Dowiedziałem się o tym po załadowaniu modelu w przeglądarce i sprawdzeniu wartości skalowania grupy najwyższego poziomu). Nie prosiłem TC, aby przeskalał model, tylko po prostu dostosowałem wartości pozycjonowania kamery, dzieląc żądane wartości przez skalę każdego wymiaru. W efekcie otrzymałem kamerę ustawioną na poziomie oczu siedzącego kierowcy, która wynosi około 1,8 m (rysunek 11.13).

W następnym podrozdziale dowiesz się, jak włączać tę kamerę w ramach sekwencji przejść dotyczących wchodzenia do pojazdu i wychodzenia z niego oraz włączania i wyłączania przejażdżki testowej.

Tworzenie animowanych i czasowych przejść

Niedługo będziemy mogli udać się na jazdę testową. Kliknięcie pojazdu Futurgo lub znajdującego się po lewej stronie przycisku *Rozpocznij jazdę testową* powoduje przeniesienie użytkownika do wnętrza samochodu. Aby trochę urozmaicić ten proces i wzbogacić wrażenia użytkownika, zaprogramujemy serię przejść i animacji. Do tego celu użyjemy składników Vizi i prostych czasomierzy opartych na metodzie `setTimeout()`.

Kod przedstawiony na listingu 11.15 i w dalszych przykładach wykonuje następujące czynności.

1. Wyłącza obsługę wyboru obiektów pod kursorem, aby przypadkowe kliknięcia nie powodowały uruchomienia jakichś animacji.
2. Otwiera samochód za pomocą animacji rozsuwającej okna na zewnątrz.
3. Po zakończeniu tej animacji użytkownik zostaje przeniesiony do samochodu.
4. Następnie okna zostają zamknięte i stopniowo stają się całkowicie przezroczyste. Także odgłosy miasta stają się przytumione. Na koniec zostają uruchomione skrypty obsługujące jazdę.

Listing 11.15. Animacje przejść dotyczących wsiadania do samochodu i rozpoczętania jazdy testowej

```
FuturgoCity.prototype.startTestDrive = function(event) {  
    if (this.testDriveRunning)  
        return;  
  
    this.testDriveRunning = true;  
  
    // Wyłącza „pickery”, gdy użytkownik znajduje się w samochodzie.  
    var i, len = this.pickers.length;  
    for (i = 0; i < len; i++) {  
        this.pickers[i].enabled = false;  
    }  
  
    // Otwiera okna samochodu.  
    this.playOpenAnimations();
```

Gdy wszystkie te czynności dobiegną końca, będziemy siedzieć w samochodzie gotowi do jazdy. Przeanalizujmy odpowiedni kod po kawałku.

Najpierw wyłączymy „pickery” i uruchomimy animacje dotyczące otwierania pojazdu.

Czekamy sekundę i przechodzimy do następnego kroku, czyli przejścia do widoku z kamery `driveCamera`. W tym celu ustawiamy właściwość `active` tej kamery na wartość `true` (która nakazuje

systemowi Vizi renderowanie właśnie przy użyciu tej kamery). Ponadto wyłączmy możliwość poruszania się kamery, ustawiając jej własność move na false. Oczywiście nadal chcemy mieć możliwość obracania kamerą, a więc korzystamy z kontrolera pierwszoosobowego. Dzięki temu, będąc w samochodzie, będziemy mogli poruszać kamerą za pomocą myszy.

```
// Po otwarciu samochodu zmieniamy kamerę na wewnętrzną i włączamy kontrolera jazdy testowej  
— robimy to z opóźnieniem.  
var that = this;  
setTimeout(function() {  
  
    // Przełączka na kamerę wewnętrzną pojazdu.  
    that.cameraController.camera = that.driveCamera;  
    // Włączamy możliwość przesuwania kamery, aby nie dalo się wyjść z samochodu.  
    that.cameraController.move = false;  
    that.driveCamera.rotation.set(0, 0, 0);  
    that.driveCamera.active = true;  
  
}, 1000);
```

Kiedy siedzimy już w samochodzie, po sekundzie uruchamiamy kolejną sekwencję działań mającą na celu zamknięcie pojazdu. Włączamy animacje zamykające okna oraz wyciszamy zewnętrzne odgłosy (kwestię obsługi dźwięku opisałem dalej w tym rozdziale). Ustawiamy prawie całkowitą przezroczystość okien, aby miasto było dobrze widoczne. Na koniec włączamy skrypty dotyczące jazdy samochodu i animacji kokpitu. Teraz możemy jeździć.

```
// Będąc wewnątrz samochodu, włączamy jego kontrolera.  
// Ponadto zamykamy okna i zwiększymy ich przezroczystość, aby dobrze było widać miasto.  
setTimeout(function() {  
  
    // Zamyka okna samochodu.  
    that.playCloseAnimations();  
  
    // Tłumi odgłosy miasta.  
    that.sound.interior();  
  
    // Zwiększa przezroczystość okien.  
    var i, len = that.faders.length;  
    for (i = 0; i < len; i++) {  
        var fader = that.faders[i];  
        fader.opacity = FuturoCity.OPACITY_MOSTLY_TRANSPARENT;  
        fader.start();  
    }  
  
    // Włącza skrypty obsługi samochodu — animacje kontrolera i kokpitu.  
    that.carController.enabled = true;  
    that.dashboardScript.enabled = true;  
  
}, 2000);
```

Kod zamykający tryb jazdy próbnej, który znajduje się w niepokazanej tutaj metodzie endTest →Drive(), zasadniczo wycofuje wcześniej wykonane czynności.

1. Wyłącza skrypty obsługi samochodu.
2. Otwiera okna.
3. Włącza możliwość wybierania elementów pod kursorem, przenosi kamerę na zewnątrz, włącza z powrotem tryb ruchu kontrolera kamery, przywraca normalną głośność odgłosów miasta oraz zmniejsza przezroczystość szyb.
4. Zamyka okna.

Implementacja zachowań obiektów

Czas poruszyć się samochodem. W tym celu napiszemy kontrolera podobnego do kontrolera pierwszoosobowego służącego do obsługi chodzenia po scenie. Jednak w tym przypadku przyciski klawiatury będą sterować samochodem, a nie kamerą. (Mysz nadal będzie służyła do zmiany kierunku patrzenia, a więc pozostaniemy przy już istniejącym kontrolerze kamery, tylko podłączymy do niego wewnętrzną kamerę `driveCamera`, jak napisałem wcześniej). Aby utworzyć kontrolera, zbudujemy własny składnik przy użyciu systemu szkieletowego Vizi.

Implementowanie własnych składników na bazie klasy `Vizi.Script`

Moc systemu Vizi wynika z kombinacji dwóch prostych pomysłów: zestawu fragmentów kodu do obsługi typowych wzorców programowania trójwymiarowego (np. włączania i wyłączania animacji, znajdowania obiektów pod kursorem, przełączania kamer itd.) oraz możliwości łączenia różnych mechanizmów, aby ze sobą współpracowały. Składniki systemu Vizi mogą współpracować z każdym obiektem, ponieważ obiekty są zorganizowane w spójny sposób wg paru prostych zasad. Przykładowo każdy egzemplarz klasy `Vizi.Object` zawiera składnik przekształcenia z własnościami `position`, `rotation` i `scale`, które mogą zmieniać inne składniki tego obiektu.

Opisane wcześniej w tym rozdziale prefabrykaty, np. `Vizi.Prefabs.Skybox()` i `Vizi.Prefabs.FirstPersonController()`, to funkcje tworzące hierarchie gotowych obiektów i zwracające obiekt klasy `Vizi.Object` jako ich korzeń. Obiekt ten może być prosty i zawierać jedynie np. sześcian, ale może też reprezentować skomplikowaną hierarchię zawierającą kilka obiektów i składników. Prefabrykaty zawierające cokolwiek innego niż tylko geometrię mogą zawierać *skrypty* implementujące ich logikę. Przykładowo prefabrykat dotyczący pudła nieba zawiera geometrię sześcianu oraz skrypt dopasowujący orientację pudła nieba do orientacji głównej kamery na scenie.

W naszej aplikacji potrzebujemy skryptu sterującego samochodem. Jeśli skrypt ten uczynimy składnikiem Vizi i dodamy go do obiektu samochodu `Futurgo`, system Vizi będzie automatycznie wywoływał jego metodę `update()` w każdej iteracji pętli wykonawczej, umożliwiając w ten sposób reagowanie na dane wprowadzane przez użytkownika, a więc sterowanie samochodem. Zobaczmy, jak jest zbudowany.

Kontroler samochodu

Przypomnij sobie kod użyty do inicjacji samochodu po załadowaniu modelu. Zostały w nim dodane różne składniki oraz mapy środowiska. Ponadto był w nim taki fragment:

```
// Dodaje kontrolera samochodu.  
this.carController = new FuturgoController({enabled:false,  
    scene: this.scene});  
futurgo.addComponent(this.carController);
```

`FuturgoController` to składnik, którego zadaniem jest obsługa sterowania samochodem za pomocą klawiatury. Strzałka w góre powoduje jazdę do przodu, strzałka w dół to hamulec, a strzałki w lewo i prawo służą do skręcania. Ponadto kontroler wykrywa kolizje, aby zapobiec wjechaniu przez samochód w ściany, oraz analizuje powierzchnię terenu, aby samochód wjeżdżała na kra-

węźniki i inne wznieśienia, zamiast w nie uderzać. A ponieważ kamerę `driveCamera` umieściliśmy w samochodzie, kamera ta porusza się wraz z samochodem, co jest zasługą hierarchii przekształceń.

Na listingu 11.16 pokazano kod źródłowy omawianego kontrolera. Znajduje się on w pliku źródłowym `r11/futurgoController.js`. Funkcja konstrukcyjna najpierw tworzy podklasę klasy `Vizi.Script`, która jest podstawowym typem wszystkich składników skryptowych w systemie. Następnie inicjuje kilka własności: stan klawiszy służących do sterowania, aktualną prędkość i wartość przyspieszenia, kilka dodatkowych zmiennych pomocnych przy wykrywaniu kolizji i analizie ukształtowania terenu oraz kilka znaczników czasu pomocnych w implementacji algorytmów pseudofizycznych, za pomocą których będziemy kontrolować prędkość samochodu.

Listing 11.16. Konstruktor składnika `FuturgoController`

```
FuturgoController = function(param)
{
    param = param || {};
    Vizi.Script.call(this, param);

    this.enabled = (param.enabled !== undefined) ? param.enabled : true;
    this.scene = param.scene || null;

    this.turnSpeed = Math.PI / 2; // 90 stopni na sekundę

    this.moveForward = false;
    this.moveBackward = false;
    this.turnLeft = false;
    this.turnRight = false;

    this.accelerate = false;
    this.brake = false;
    this.acceleration = 0;
    this.braking = 0;
    this.speed = 0;
    this.rpm = 0;

    this.eyePosition = new THREE.Vector3();
    this.downVector = new THREE.Vector3(0, -1, 0);
    this.groundY = 0;
    this.avatarHeight = FuturgoCity.AVATAR_HEIGHT_SEATED;

    this.savedPos = new THREE.Vector3();
    this.movementVector = new THREE.Vector3();

    this.lastUpdateTime = Date.now();
    this.accelerateStartTime = this.brakeStartTime =
        this.accelerateEndTime = this.brakeEndTime =
        this.lastUpdateTime;
}
```

Składniki systemu `Vizi` zazwyczaj implementują dwie metody, czyli `realize()` i `update()`. Metoda `realize()` jest wywoływana przez system, gdy trzeba utworzyć struktury danych potrzebne do renderowania, wprowadzania danych, obsługi sieci i realizacji różnych innych usług przeglądarki. W kontrolerze samochodu metoda `realize()` robi dwie rzeczy: zapisuje początkową pozycję pojazdu i tworzy efekt odbicia od przeszkody, który będzie uruchamiany, gdy

samochód z czymś się zderzy. Dostęp do samochodu jest możliwy przy użyciu własności `this._object`, którą Vizi ustawia automatycznie na składniku podczas jego dodawania do obiektu.

```
FuturgoController.prototype.realize = function()
{
    this.lastUpdateTime = Date.now();

    // Zapisuje pozycję podłożu.
    this.groundY = this._object.transform.position.y;

    // Dodaje efekt odbicia od przeszkode, który będzie uruchamiany w przypadku kolizji.
    this.bouncer = new Vizi.BounceBehavior(
        { duration : FuturgoController.BOUNCE_DURATION }
    );
    this._object.addComponent(this.bouncer);
}
```

Natomiast metoda `update()` jest wywoływaną dla każdego składnika każdego obiektu w grafie sceny Vizi w każdej iteracji pętli wykonawczej programu. W kontrolerze samochodu metoda ta pełni kilka zadań. Po pierwsze, zapisuje aktualną pozycję samochodu, którą przywróci po kolizji lub przejechaniu przez nierówności terenu. Po drugie, aktualizuje prędkość pojazdu na podstawie wewnętrznego algorytmu fizycznego. Kolejną czynnością jest obliczanie nowej pozycji samochodu przy użyciu własności oznaczającej prędkość. A na koniec metoda `update()` sprawdza kolizje i analizowanie terenu.

```
FuturgoController.prototype.update = function()
{
    if (!this.enabled)
        return;

    var now = Date.now();
    var deltat = now - this.lastUpdateTime;

    this.savePosition();
    this.updateSpeed(now, deltat);
    this.updatePosition(now, deltat);
    this.testCollision();
    this.testTerrain();
    this.lastUpdateTime = now;
}
```

Do aktualizacji prędkości używany jest prosty algorytm pseudofizyczny, który imituje przyspieszenie i пед — listing 11.17. Im dłużej przytrzymany zostanie klawisz strzałki w góre, tym bardziej wzrasta przyspieszenie. Im dłużej przytrzymany zostanie klawisz strzałki w dół, tym bardziej samochód zwalnia. Jeśli po rozpoczęciu samochodu wszystkie klawisze zostaną puszczone, samochód będzie jechał siłą rozpedu. Jeśli po wykonaniu tych wszystkich obliczeń zmieni się szybkość lub przyspieszenie samochodu, rozesiane zostaną zdarzenia informujące o tej zmianie zainteresowane procedury. Kontroler kokpitu (opisany dalej w tym rozdziale) wykorzysta te informacje do zmiany wartości na zegarach w kokpicie.

Listing 11.17. Aktualizowanie szybkości samochodu

```
FuturgoController.prototype.updateSpeed = function(now, deltat) {
    var speed = this.speed, rpm = this.rpm;

    // Przyspiesza, jeśli odpowiedni klawisz jest wciśnięty.
    if (this.accelerate) {
```

```

        var deltaA = now - this.accelerateStartTime;
        this.acceleration = deltaA / 1000 * FuturgoController.ACCELERATION;
    }
} else {
    // Ped.
    var deltaA = now - this.accelerateEndTime;
    this.acceleration -= deltaA / 1000 * FuturgoController.INERTIA;
    this.acceleration = Math.max( 0, Math.min( FuturgoController.MAX_ACCELERATION,
        this.acceleration ) );
}

speed += this.acceleration;

// Zwalnia, jeśli zostanie naciśnięty klawisz hamowania.
if (this.brake) {
    var deltaB = now - this.brakeStartTime;
    var braking = deltaB / 1000 * FuturgoController.BRAKING;

    speed -= braking;
}
else {
    // Bezwładność.
    var inertia = deltat / 1000 * FuturgoController.INERTIA;
    speed -= inertia;
}

speed = Math.max( 0, Math.min( FuturgoController.MAX_SPEED, speed ) );
rpm = Math.max( 0, Math.min( FuturgoController.MAX_ACCELERATION,
    this.acceleration ) );

if (this.speed != speed) {
    this.speed = speed;
    this.dispatchEvent("speed", speed);
}

if (this.rpm != rpm) {
    this.rpm = rpm;
    this.dispatchEvent("rpm", rpm);
}
}
}

```

Aby zaktualizować pozycję samochodu, przesuwamy go wzdłuż linii widzenia (ujemna część osi z) na odległość obliczoną przy użyciu jego aktualnej prędkości. Ponadto obracamy samochód, obracając obiekt wokół osi y.

```

FuturgoController.prototype.updatePosition = function(now, deltat) {

    var actualMoveSpeed = deltat / 1000 * this.speed;
    var actualTurnSpeed = deltat / 1000 * this.turnSpeed;

    // Przesunięcie na osi z...
    this._object.transform.object.translateZ( -actualMoveSpeed );

    // ...ale z pozostawieniem pojazdu na podłożu.
    this._object.transform.position.y = this.groundY;

    // Skręt.
    if ( this.turnLeft ) {
        this._object.transform.object.rotateY( actualTurnSpeed );
    }
}

```

```

        if ( this.turnRight ) {
            this._object.transform.object.rotateY( -actualTurnSpeed );
        }
    }
}

```

Wykrywanie kolizji samochodu ze elementami sceny

Kod obsługujący wykrywanie kolizji samochodu z budynkami jest podobny do kodu kontrolera `Vizi.FirstPersonController`. Wywołuje metodę `objectFromRay()` systemu graficznego w celu obliczenia przecięcia promienia biegącego z aktualnej pozycji kamery do żądanej pozycji ze znajdującą się na scenie geometrią. Warto zwrócić uwagę na pierwszy przekazany do tej metody argument `this.scene`, który zawiera całą geometrię sceny, ale bez samochodu. Gdybyśmy do testu kolizji dodali także samochód, zawsze otrzymywaliśmy wynik pozytywny.

```

FuturgoController.prototype.testCollision = function() {

    this.movementVector.copy(this._object.transform.position)
        .sub(this.savedPos);
    this.eyePosition.copy(this.savedPos);
    this.eyePosition.y = this.groundY + this.avatarHeight;

    var collide = null;
    if (this.movementVector.length()) {

        collide = Vizi.Graphics.instance.objectFromRay(this.scene,
            this.eyePosition,
            this.movementVector,
            FuturgoController.COLLISION_MIN,
            FuturgoController.COLLISION_MAX);

        if (collide && collide.object) {
            var dist = this.eyePosition.distanceTo(collide.hitPointWorld);
        }
    }

    if (collide && collide.object) {
        this.handleCollision(collide);
    }
}
}

```

Implementowanie reakcji na kolizję

Kiedy spacerowaliśmy po mieście, przed przechodzeniem przez ściany budynków chronił nas kontroler trybu pierwszoosobowego. Gdy tylko została wykryta kolizja, następowało zatrzymanie ruchu. Jednak w przypadku samochodu lepiej zastosować mniej radykalne rozwiązanie. W realnym świecie, gdy samochód uderzy w ścianę, odbija się od niej i najczęściej się rozbija. W naszej symulacji sprawimy, że samochód `Futurgo` będzie się delikatnie odbijał od obiektów na scenie. Technika obsługi kolizji obiektów nazywa się **reakcją na kolizję** (ang. *collision response*).

Sposób implementacji reakcji na kolizję w kontrolerze samochodu pokazano na listingu 11.18. Najpierw rozsyłamy zdarzenie `collide` do wszystkich procedur nasłuchujących. Nasza aplikacja nasłuchuje ich w celu odtworzenia dźwięku zderzenia w odpowiednim momencie. Następnie za pomocą metody `restorePosition()` przywracamy pierwotną wartość pozycji samochodu, aby nie przebił się przez obiekty geometryczne. Przypomnę też, że metoda `realize()` dodała do samochodu zachowanie `Vizi.BounceBehavior`. Uruchamiamy je teraz, aby spowodować lekkie odbicie pojazdu od przeszkody. Samochód odbija się po linii prostej *do tyłu*. Należy

zwrócić uwagę na to, jak ustawiliśmy własność `bounceVector` na zanegowaną wartość wektora ruchu oraz że zastosowaliśmy skalowanie przez 0.333, aby imitować utratę siły spowodowaną zderzeniem. Na koniec wyłączamy silnik.

Listing 11.18. Obsługa kolizji poprzez reakcję na zderzenie

```
FuturgoController.prototype.handleCollision = function(collide) {  
  
    // Powiadomienie procedur nasłuchujących.  
    this.dispatchEvent("collide", collide);  
  
    // Powrót do poprzedniej pozycji.  
    this.restorePosition();  
  
    // Włączenie reakcji na zderzenie.  
    this.bouncer.bounceVector  
        .copy(this.movementVector)  
        .negate()  
        .multiplyScalar(.333);  
    this.bouncer.start();  
  
    // Wyłączenie silnika.  
    this.speed = 0;  
    this.rpm = 0;  
}
```

Implementowanie analizowania ukształtowania terenu

Środowisko miasta jest prawie całkowicie płaskie, ale zawiera kilka wzgórz. Nad poziom ulicy wznosi się np. krawężnik i musimy zdecydować, co ma się stać, gdy najedzie na niego nasz pojazd. Może się zatrzymać albo wjechać na przeszkodę, ale na pewno nie chcielibyśmy, aby przejeżdżał „przez” nią, jakby jej nie było. Zatrzymanie samochodu w momencie zderzenia z krawężnikiem byłoby łatwe do zrealizowania, ale za to mało zabawne. Fajniej będzie, jeśli samochód będzie wjeżdżał na krawężniki. W tym celu musimy jednak zaimplementować algorytm analizowania ukształtowania terenu.

Algorytmy **analizy ukształtowania terenu** (ang. *terrain following*) mają za zadanie utrzymywać kamerę lub awatara w stałej odległości nad podłożem. Gdy kamera rusza się w obrębie sceny, wysyłany jest promień w dół. Jeśli promień ten zderzy się z obiektem geometrycznym, przebytą przez niego odległość porównuje się z żądaną wysokością kamery. Gdy odległość ta jest mniejsza niż wymagana, kamerę się podnosi, co wygląda tak, jakby weszło się na schodek. Jeśli odległość jest większa niż wymagana, kamerę się opuszcza.

Kontroler samochodu `Futurgo` sprawdza ukształtowanie terenu w każdym wywołaniu swojej metody `update()`. Ponownie użyjemy metod systemu `Vizi` do testowania kolizji z geometrią sceny, ale tym razem użyjemy promienia skierowanego w dół (`downVector= [0, -1, 0]`). Możesz sprawdzić, jak to działa. Podjedź do dowolnego budynku, a zobaczysz, że pojazd wjedzie na znajdujący się przed nim krawężnik. Cofnij, a samochód zjedzie z powrotem na ulicę. Spójrz na listing 11.19 oraz ilustrację kolizji i analizowania ukształtowania terenu na rysunku 11.14.

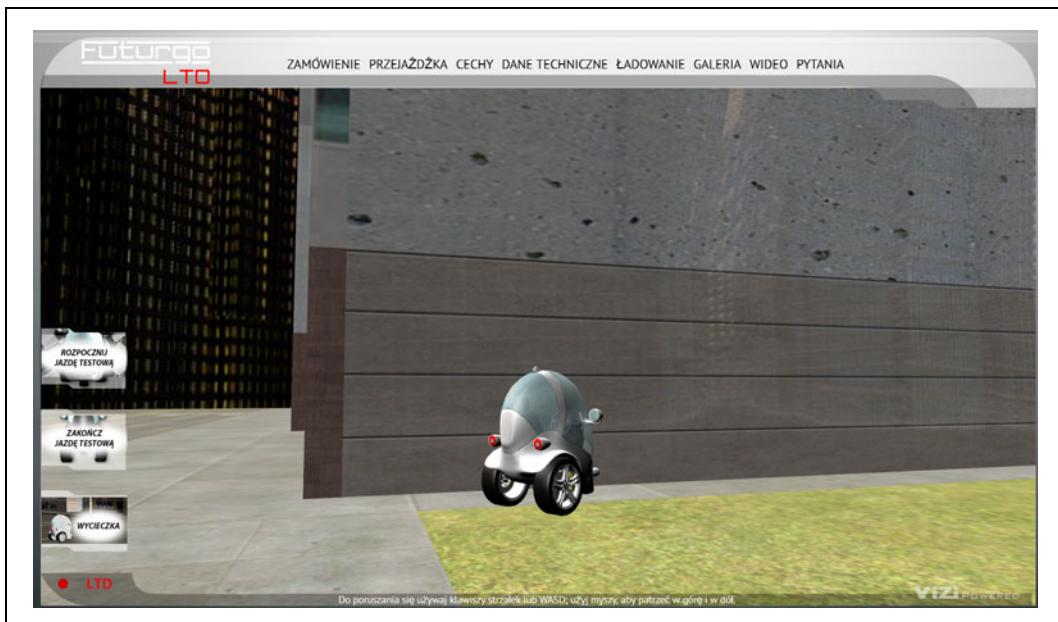
Listing 11.19. Analizowanie ukształtowania terenu w aplikacji `Futurgo`

```
FuturgoController.prototype.testTerrain = function() {  
  
    var EPSILON = 0.00001;
```

```
var terrainHit = Vizi.Graphics.instance.objectFromRay(this.scene,
    this.eyePosition,
    this.downVector);

if (terrainHit && terrainHit.object) {
    var dist = this.eyePosition.distanceTo(terrainHit.hitPointWorld);
    var diff = this.avatarHeight - dist;
    if (Math.abs(diff) > EPSILON) {
        console.log("distance", dist);

        this.eyePosition.y += diff;
        this._object.transform.position.y += diff;
        this.groundY = this._object.transform.position.y;
    }
}
```



Rysunek 11.14. Wykrywanie kolizji i analizowanie ukształtowania terenu: samochód zatrzymuje się na ścianach i wjeżdża na krawężniki. Efekt uzyskano przy użyciu techniki ray castingu

Dodawanie dźwięków do środowiska

Nasza aplikacja robi się coraz ciekawsza — możemy już kręcić się po ulicach futurystycznym samochodem — ale wciąż brakuje jej jednego szczegółu, czyli dźwięków. W aplikacji internetowej dźwięk może się wydawać luksusem, ale bez niego trudno sobie wyobrazić realistyczne środowisko trójwymiarowe. Na szczęście, proste efekty dźwiękowe można łatwo dodać za pomocą standardowych elementów HTML5.

Potrzebujemy tylko dwóch dźwięków: zapętlonego miejskiego szumu w tle oraz krótkiego odgłosu zderzenia. Zaczniemy od dodania do strony HTML elementów `<audio>` i `<source>` (plik `r11/futurgoCity.html`):

```
<audio volume="0.0" id="city_sound">
<!-- http://www.freesound.org/people/synthetic-oz/sounds/162704/ -->
<source src="../sounds/162704_syntheticoz_city-trimmed-looped.wav"
        type="audio/wav" />
Twoja przeglądarka nie obsługuje plików w formacie WAV w elemencie audio.
</audio>
<audio volume="0.0" id="bump_sound">
<!-- http://www.freesound.org/people/Calethos/sounds/31126/ -->
<source src="../sounds/31126_calethos_bump.wav" type="audio/wav" />
Twoja przeglądarka nie obsługuje plików w formacie WAV w elemencie audio.
</audio>
```

Teraz wystarczy tylko dopisać kod zmieniający głośność odgłosów i włączający ich odtwarzanie. Gdy wchodzimy do samochodu, odgłosy miasta stają się przytłumione. Gdy z niego wychodzimy, odgłosy te powinny z powrotem stawać się głośniejsze. Kiedy zderzymy się z czymś, powiniem być słyszalny odgłos zderzenia.

Implementacja dźwięków znajduje się w pliku źródłowym `r11/futurgoSound.js`. Umieszczony w nim kod jest prosty i zawiera wywołania standardowych metod DOM HTML5 dotyczących obsługi dźwięków. Na listingu 11.20 pokazano go w całości. Metody `interior()` i `exterior()` zwiększą i zmniejszą głośność dźwięku tła, a metoda `bump()` jednorazowo odtwarza odgłos zderzenia.

Listing 11.20. Zarządzanie dźwiękami na scenie Futurgo

```
FuturgoSound = function(param) {
    this.citySound = document.getElementById("city_sound");
    this.citySound.volume = FuturgoSound.CITY_VOLUME;
    this.citySound.loop = true;

    this.bumpSound = document.getElementById("bump_sound");
    this.bumpSound.volume = FuturgoSound.BUMP_VOLUME;
}

FuturgoSound.prototype.start = function() {
    this.citySound.play();
}

FuturgoSound.prototype.bump = function() {
    this.bumpSound.play();
}

FuturgoSound.prototype.interior = function() {
    $(this.citySound).animate(
        {volume: FuturgoSound.CITY_VOLUME_INTERIOR},
        FuturgoSound.FADE_TIME);
}
```

```

FuturgoSound.prototype.exterior = function() {
    $(this.citySound).animate(
        {volume: FuturgoSound.CITY_VOLUME},
        FuturgoSound.FADE_TIME);
}

FuturgoSound.prototype.bump = function() {
    this.bumpSound.play();
}

FuturgoSound.CITY_VOLUME = 0.3;
FuturgoSound.CITY_VOLUME_INTERIOR = 0.15;
FuturgoSound.BUMP_VOLUME = 0.3;
FuturgoSound.FADE_TIME = 1000;

```

Pozostało jeszcze tylko podłączyć te metody do aplikacji. Przypomnij sobie sekwencję czynności w metodzie `startTestDrive()` (plik `r11/futurgoCity.js`).

```
// Tłumi odgłosy miasta.
that.sound.interior();
```

Wyjście z samochodu powoduje wywołanie metody `exterior()` przywracającej pierwotną głośność dźwięków.

Klasa `FuturgoCity` obsługuje także odgłos kolizji. W tym celu do kontrolera samochodu została dodana procedura nasłuchu zdarzeń.

```
this.carController.addEventListener("collide", function(collide) {
    that.sound.bump();
});
```

Renderowanie dynamicznych tekstur

W naszej realistycznej aplikacji pozostała do zrobienia już tylko jedna rzecz. Samochód jest gotowy do jazdy i po zaimplementowaniu dźwięków myślałem, że zakończyłem pisanie kodu. Gdy jednak wsiadłem do pojazdu, uderzył mnie brak życia w jego wnętrzu. Szybko zdałem sobie sprawę, że odpowiadają za to nieruchome wskaźniki na kokpicie. Może mam wygórowane oczekiwania, ale trudno. Gdy samochód jedzie, wskaźówki na jego kokpicie powinny się poruszać. Musimy więc animować kokpit, a przynajmniej jego tekstury.

W tym podrozdziale utworzymy **teksturę proceduralną** (ang. *procedural texture*), czyli rysowaną dynamicznie za pomocą kodu źródłowego (normalne tekstury są statycznymi obrazami pobieranymi z plików). W tym celu skorzystamy z dobrze znanych technik renderowania na kanwie dwuwymiarowej. Przy użyciu API kanwy dwuwymiarowej kokpit będzie generował proceduralną teksturę reprezentującą aktualną prędkość i prędkość obrotową samochodu na zegarach.

Początkowo tekstura kokpitu przedstawiała zegary z nieruchomoimi wskaźówkami. Poprosiłem TC o oddzielenie wskaźówki od reszty kokpitu i zapisanie jej w osobnym pliku. Po krótkim czasie otrzymałem odpowiednie pliki. TC nie musiał nawet nic zmieniać w grafice trójwymiarowej, wystarczyło, że zmienił tekstury. Przesłane przez niego dwa pliki są pokazane na rysunkach 11.15 i 11.16.



Rysunek 11.15. Tekstura zegarów



Rysunek 11.16. Tekstura obrotowej wskazówki

Do animowania kokpitu utworzymy nowy składnik Vizi o nazwie `FuturgoDashboard`. Będzie to skrypt tworzący element HTML kanwy ładujący dwie mapy bitowe w metodzie `realize()` i aktualizujący wygląd zegarów w metodzie `update()` na podstawie aktualnej prędkości i prędkości obrotowej. Wartości te będziemy śledzić za pomocą procedur nasłuchu zdarzeń dodanych do kontrolera `FuturgoController`.

Na listingu 11.21 przedstawiony został odpowiedni kod źródłowy. Metoda `realize()` tworzy nowy element kanwy i obiekt tekstury Three.js do jego przechowywania. Następnie ustawiamy tę nową teksturę jako własność `map` materiału kokpitu. Później zawartość tej kanwy będziemy mogli zmieniać przy użyciu standardowych wywołań API Canvas 2D. Zmiany te będą widoczne na teksturze wyświetlanej na obiekcie.

Listing 11.21. Tworzenie tekstuury z kanwy dla kokpitu

```
FuturgoDashboardScript.prototype.realize = function()
{
    // Definicje zegarów.
    var gauge = this._object.findNode("head_light_L1");
    var visual = gauge.visuals[0];
```

```

// Utworzenie nowego elementu kanwy do rysowania.
var canvas = document.createElement("canvas");
canvas.width = 512;
canvas.height = 512;

// Utworzenie nowej tekstury Three.js z kanwy.
var texture = new THREE.Texture(canvas);
texture.wrapS = texture.wrapT = THREE.RepeatWrapping;
visual.material.map = texture;

this.texture = texture;
this.canvas = canvas;
this.context = canvas.getContext("2d");

```

Poniżej przedstawiona jest dalsza część metody `realize()`, w której ładowane są tekstury. Użyta w niej została własność DOM. Najpierw zdefiniowana została procedura obsługi zdarzeń `onload`, dzięki której przekazywana jest informacja, kiedy obraz zostanie załadowany i będzie gotowy do użycia. Następnie ustawniona została własność `src` w celu załadowania tego obrazu.

```

// Ładuję tekstury kokpitu i zegarów.
this.dashboardImage = null;
this.dialImage = null;

var that = this;
var image1 = new Image();
image1.onload = function () {
    that.dashboardImage = image1;
    that.needsUpdate = true;
}
image1.src = FuturoDashboardScript.dashboardURL;

var image2 = new Image();
image2.onload = function () {
    that.dialImage = image2;
    that.needsUpdate = true;
}
image2.src = FuturoDashboardScript.dialURL;

// Wymusza początkową aktualizację.
this.needsUpdate = true;
}

```

Czas przejść do rysowania, co pokazano na listingu 11.22. W każdym wywołaniu metody skryptu `update()` sprawdzamy, czy trzeba ponownie narysować teksturę. Decyzję podejmujemy na podstawie zmiany prędkości lub prędkości obrotowej kontrolera pojazdu. Jeśli zmiana taka nastąpiła, wywołujemy metodę `draw()`, aby rozpoczęć rysowanie na teksturze za pomocą API kanwy. Metoda ta najpierw kasuje zawartość kanwy i zastępuje ją bieżącym kolorem tekstu. Następnie, jeśli mapa bitowa kokpitu została załadowana, rysuje ją na kanwie z wykorzystaniem metody `drawImage()` kontekstu i pokrywa nią całą kanwę.

Listing 11.22. Rysowanie obrazu tła kokpitu

```

FuturoDashboardScript.prototype.draw = function()
{
    var context = this.context;
    var canvas = this.canvas;

    context.clearRect(0, 0, canvas.width, canvas.height);
    context.fillStyle = this.backgroundColor;
    context.fillRect(0, 0, canvas.width, canvas.height);

```

```
context.fillStyle = this.textColor;

if (this.dashboardImage) {
    context.drawImage(this.dashboardImage, 0, 0);
}
```



Jeśli potrzebujesz odświeżenia wiadomości na temat API kanwy, wróć do rozdziału 7., w którym znajduje się opis podstawowych technik rysowania na kanwie.

Teraz na wierzchu narysujemy wskazówkę. Cały czas rejestrujemy szybkość samochodu i jego prędkość obrotową (zaraz rozwinię ten temat), więc możemy użyć tych wartości do obliczenia kąta obrotu wskazówki. Przypominam, że API Canvas 2D zawiera metody `save()` i `restore()` służące odpowiednio do zapisywania bieżącego stanu kontekstu przed wykonaniem zestawu wywołań metod dotyczących rysowania oraz do przywracania tego stanu po zakończeniu rysowania. Otoczymy rysowanie każdej wskazówki tymi wywołaniami. Po zapisaniu stanu wykonujemy na kontekście dwuwymiarowe przekształcenia polegające na przesunięciu obrazu wskazówki, który mamy zamiar narysować, na odpowiednie miejsce na zegarze i obróceniu wskazówki o kąt odpowiadający aktualnej prędkości i prędkości obrotowej pojazdu. Następnie rysujemy ten obraz i przywracamy kontekst. Czynności te powtarzamy dla każdego zegara. (Wartości przesunięcia obliczyłem na podstawie rozmiaru obrazu wskazówki, a lokalizację określiłem metodą prób i błędów w programie graficznym).

```
var speeddeg = this._speed * 10 - 120;
var speedtheta = THREE.Math.degToRad(speeddeg);
var rpmdeg = this._rpm * 20 - 90;
var rpmtheta = THREE.Math.degToRad(rpmdeg);

if (this.dialImage) {
    context.save();

    context.translate(FuturgoDashboardScript.speedDialLeftOffset,
        FuturgoDashboardScript.speedDialTopOffset);
    context.rotate(speedtheta);
    context.translate(-FuturgoDashboardScript.dialCenterLeftOffset,
        -FuturgoDashboardScript.dialCenterTopOffset);
    context.drawImage(this.dialImage, 0, 0);
    context.restore();

    context.save();

    context.translate(FuturgoDashboardScript.rpmDialLeftOffset,
        FuturgoDashboardScript.rpmDialTopOffset);
    context.rotate(rpmtheta);
    context.translate(-FuturgoDashboardScript.dialCenterLeftOffset,
        -FuturgoDashboardScript.dialCenterTopOffset);
    context.drawImage(this.dialImage, 0, 0); // 198, 25, 115);
    context.restore();
}
}
```

Pozostało już tylko połączyć kontrolera samochodu z kokpitem, aby można było nasłuchiwać zmian prędkości i prędkości obrotowej pojazdu.

Aplikacja ustawia własność `carController` kokpitu po jego utworzeniu:

```
this.dashboardScript.carController = this.carController;
```

Kontroler `carController`, pokazany na listingu 11.23, jest właściwością JavaScript utworzoną za pomocą metody `Object.defineProperties()`. Ustawienie tej właściwości powoduje wywołanie metody dostępowej obiektu `setCarController()`. Metoda ta zapisuje kontrolera w prywatnej właściwości `this._carController` i dodaje procedury nasłuchu zdarzeń informujących o prędkości (`speed`) i prędkości obrotowej (`rpm`) samochodu. Procedury te zapisują nowe wartości i wskazują, że trzeba ponownie narysować kokpit poprzez ustawienie jego właściwości `needsUpdate`. Teraz przyspieszenia i zwolnienia samochodu będą widoczne na zegarach.

Listing 11.23. Kontroler kokpitu tworzący procedury nasłuchu zdarzeń dotyczących zmian prędkości jazdy samochodu i jego prędkości obrotowej

```
FuturgoDashboardScript.prototype.setCarController =  
function(controller) {  
  
    this._carController = controller;  
  
    var that = this;  
    controller.addEventListener("speed", function(speed) {  
        that.setSpeed(speed); });  
    controller.addEventListener("rpm", function(rpm) {  
        that.setRPM(rpm); });  
}
```



Możliwości dostępne podczas używania kanwy jako tekstury WebGL są nie do przeoczenia. Programista może używać dobrze znanego mu API do dynamicznego rysowania tekstur w języku JavaScript, co pozwala na tworzenie niesamowitych efektów. Projektanci biblioteki WebGL w tym zakresie spisali się znakomicie. Ponadto biblioteka ta obsługuje tekstury z elementu HTML `<video>`, co umożliwia tworzenie jeszcze ciekawszych efektów.

Podsumowanie

To był bardzo długi, ale i treściwy rozdział.

Dowiedziałeś się, jak na stronie internetowej utworzyć realistyczne trójwymiarowe środowisko z panoramicznym tłem, odbiciami mapy środowiska, nawigacją, dźwiękami i ruchomym obiektem reagującym na działania użytkownika. Udoskonaliłeś naszą przeglądarkę modeli, dodając funkcje umożliwiające przeglądanie struktury grafu sceny i właściwości obiektów. Nauczyłeś się tworzyć uproszczone wersje kilku klasycznych algorytmów i efektów stosowanych w trójwymiarowych grach, takich jak nawigacja pierwszoosobowa, wykrywanie kolizji, analizowanie ukształtowania terenu, renderowanie pudła nieba oraz tworzenie tekstur proceduralnych.

Tworzenie trójwymiarowych środowisk w przeglądarce internetowej to trudna, ale wykonalna praca. Teraz już wiesz, na czym polega.

Tworzenie aplikacji dla urządzeń przenośnych

Ostatnie dziesięciolecie to nie tylko okres burzliwego rozwoju technologii HTML5, ale również czas jeszcze bardziej radykalnych zmian w dziedzinie telefonów i tabletów. Nowatorskie projekty iPhone'a i iPada, które powstały w firmie Apple, zatarły granice między urządzeniami przenośnymi i zwykłymi komputerami. Obecnie urządzeń przenośnych sprzedaje się więcej niż tradycyjnych komputerów, ponieważ konsumenti preferują prostsze i mniejsze sprzęty, które mogą zabrać ze sobą, aby pobrać, obejrzeć film, posłuchać muzyki, przejrzeć strony internetowe, a nawet przeprowadzić rozmowę telefoniczną. Te nowoczesne podręczne komputery przyczyniły się także do rozwoju wielu nowych funkcji, takich jak usługi lokalizacyjne, interfejsy dotykowe czy wykrywanie orientacji urządzenia.

Aby można było korzystać z tych nowych udogodnień tabletów i smartfonów, programista najczęściej musi nauczyć się nowego języka programowania i obsługi specjalnego systemu operacyjnego. Gdy np. chce tworzyć aplikacje dla urządzeń firmy Apple, powinien poznać interfejsy API systemu operacyjnego iOS oraz opanować programowanie w języku Objective-C (albo utworzyć mostek do niego z innego macierzystego języka, np. C++). Programista piszący aplikacje dla systemu Android także musi poznać zestaw specjalnych interfejsów API oraz nauczyć się tworzenia aplikacji w Javie. To samo dotyczy wielu innych platform. Jednak od pewnego czasu na platformach przenośnych obsługiwana jest w ograniczonym zakresie technologia HTML5. Obsługa ta opiera się na stosowaniu kontrolek WebKit, których można używać w aplikacjach. Dzięki temu warstwę prezentacyjną i część logiki programu można zdefiniować przy użyciu kodu HTML, CSS i JavaScript, chociaż nadal konieczne jest też pisanie kodu macierzystego, aby dostać się do funkcji platformy, np. obsługujących grafikę trójwymiarową opartą na OpenGL, których aktualnie nie ma w przeglądarkach internetowych.

Ostatnio jednak przeglądarki nadrabają opóźnienie. W specyfikacjach HTML5 znalazła się większość funkcji, które początkowo były utworzone wyłącznie dla platform przenośnych. Wygląda na to, że dwa kiedyś całkiem osobne światy programowania dla konkretnych urządzeń i programowania sieciowego powoli się do siebie zbliżają. Dla wielu programistów jest to doskonała wiadomość. Będą mogli używać dobrze znanych języków HTML5 i JavaScript i jednocześnie tworzyć aplikacje działające w różnych środowiskach. Grafika trójwymiarowa to jeden z najnowszych dodatków do tego zestawu narzędzi. Technologia CSS3 na platformach przenośnych jest powszechnie obsługiwana, a technologia WebGL cieszy się niewiele gorszymi osiągnięciami w tym zakresie. W tym rozdziale omówię problemy dotyczące tworzenia trójwymiarowych aplikacji HTML5 dla urządzeń przenośnych.

Przenośne platformy trójwymiarowe

Macierzyste interfejsy API urządzeń przenośnych są na razie bardziej funkcjonalne od HTML5, ale różnice te powoli się zacierają. Grafika trójwymiarowa jest obsługiwana przez większość platform mobilnych, chociaż w obsłudze tej występują różne ograniczenia. Większość przeglądarek obsługuje WebGL, ale są wyjątki, np. mobilna wersja przeglądarki Safari. W czasie pisania tej książki sytuacja technologii HTML5 i grafiki trójwymiarowej na urządzeniach przenośnych przedstawała się następująco.

- Biblioteka WebGL jest obsługiwana przez wiele przeglądarek w wersji mobilnej, ale nie przez wszystkie. Szczegółowe informacje zostały podane w tabeli 12.1.

Tabela 12.1. Obsługa biblioteki WebGL w przenośnych urządzeniach i systemach operacyjnych

Platforma lub urządzenie	Obsługiwane przeglądarki
Amazon Fire OS (oparta na Androidzie)	Amazon Silk (tylko dla urządzeń Kindle Fire HDX)
Android	Mobilna wersja przeglądarki Chrome, mobilna wersja przeglądarki Firefox
Apple iOS	Brak obsługi w mobilnej wersji przeglądarki Safari i Chrome; obsługa w systemie szkieletowym iAds do tworzenia reklam HTML5 do użytku wewnętrz aplikacji
BlackBerry 10	Przeglądarka BlackBerry
Firefox OS	Mobilna wersja przeglądarki Firefox
Intel Tizen	Przeglądarka Tizen
Windows RT	Internet Explorer (wymaga systemu Windows RT 8.1 lub nowszego)

- Przekształcenia trójwymiarowe, przejścia i animacje CSS są obsługiwane przez wszystkie przeglądarki mobilne. Przykłady z rozdziału 6. powinny działać w każdym zaktualizowanym środowisku przenośnym. Jeśli masz niewygórowane potrzeby w zakresie grafiki trójwymiarowej, chcesz np. tylko zastosować parę trójwymiarowych efektów do dwuwymiarowych elementów strony, istnieje duża szansa, że lepiej zrobisz, używając CSS3 zamiast WebGL, ponieważ druga z tych technologii nie jest obsługiwana przez wszystkie platformy mobilne.
- Interfejs API Canvas 2D jest obsługiwany przez wszystkie przeglądarki mobilne. Można więc go wykorzystać jako wyjście awaryjne w przypadkach braku obsługi WebGL, chociaż wiąże się to z pogorszeniem wydajności, ponieważ element kanwy dwuwymiarowej nie ma wsparcia sprzętowego.

Największą luką w powyższej tabeli jest brak obsługi biblioteki WebGL w mobilnych wersjach przeglądarek Safari i Chrome w systemie iOS. Wprawdzie system Android zagarnia coraz większą część rynku, ale system iOS także jest bardzo popularny i przykuwa uwagę wielu programistów. Ta niekorzystna sytuacja może się wkrótce zmienić, ale aktualnie aplikacje WebGL nie działają w przeglądarkach internetowych w systemie iOS.

Na platformach, których przeglądarki internetowe nie obsługują technologii WebGL, można używać specjalnych adapterów, tzw. rozwiązań hybrydowych dostarczających interfejs API WebGL do aplikacji. Programista pisze w języku JavaScript kod współpracujący z kodem macierzystym odpowiedzialnym za implementację tego API. W wyniku nie powstanie aplikacja przeglądarkowa, ale program działający z prędkością kodu macierzystego, który można na-

pisać tak samo szybko jak każdą inną aplikację w JavaScriptie. Jedną z takich technologii, o nazwie CocoonJS, opiszę dalej w tym rozdziale.

Podczas wdrażania aplikacji dla platform mobilnych obsługujących technologię WebGL do wyboru mamy dwie możliwości: aplikacje przeglądarkowe oraz spakowane aplikacje zwykle zwane **aplikacjami sieciowymi** (ang. *web apps*). Aplikacje przeglądarkowe oparte na WebGL dla urządzeń przenośnych tworzy się w taki sam sposób jak zwykłe i dostarcza się jako zestaw plików z serwerów. Przy tworzeniu aplikacji sieciowych za pomocą narzędzi platformy pakuje się pliki — zwykle takie same jak te, które podawałyby się z serwera, czasami wz bogacone o ikonę i jakieś metadane — które następnie są rozprowadzane przez sklep z aplikacjami danej platformy lub za pośrednictwem innej tego typu usługi.

Niezależnie od metody wdrożenia, przy tworzeniu aplikacji trójwymiarowej dla urządzeń mobilnych należy uwzględnić pewne specyficzne kwestie. Powinno się dodać obsługę funkcji typowych dla danego urządzenia, takich jak interfejs dotykowy, lokalizacja i wykrywanie orientacji. Ponadto należy znacznie więcej uwagi poświęcić wydajności, ponieważ urządzenia te, w porównaniu z komputerami stacjonarnymi, z reguły mają mniej pamięci i mniej wydajne procesory CPU i GPU. Rozwijam te tematy dalej w tym rozdziale.

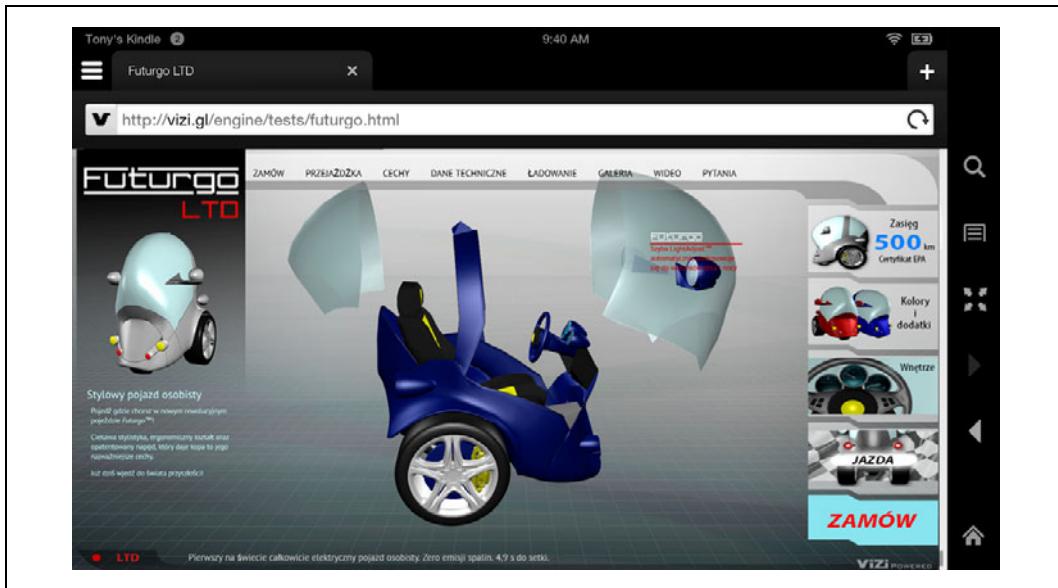


Branża platform mobilnych obsługujących HTML5 jest bardzo dynamiczna. Można odnieść wrażenie, że codziennie pojawia się nowa taka platforma. Przydatny przegląd aktualnej sytuacji i garść podstawowych informacji można znaleźć na stronie http://en.wikipedia.org/wiki/HTML5_in_mobile_devices.

Tworzenie aplikacji dla mobilnych wersji przeglądarek internetowych

Jeśli umiesz budować aplikacje WebGL dla przeglądarek internetowych na komputery stacjonarne, to aby przejść na platformy mobilne, wystarczy wpisać odpowiedni adres URL w pasku adresu przeglądarki. Jeśli tylko dana platforma obsługuje WebGL, Twoja aplikacja powinna po prostu działać. Z wydajnością może być różnie. Urządzenia i systemy operacyjne obsługują różne konfiguracje sprzętowe: niektóre są dość słabe, np. telefony GeeksPhone z systemem operacyjnym Firefox OS, a inne, np. tablety Samsung Galaxy i Google Nexus, cechują się bardzo dobrymi osiągami. Jednak w każdym przypadku coś powinno zostać wyrenderowane na ekranie, aby od tego można było zacząć.

Jednym z najbardziej imponujących urządzeń, jakie testowałem, był udostępniony w sprzedaży w październiku 2013 r. tablet Kindle Fire HDX firmy Amazon. Jest to najnowszy produkt z linii Kindle Fire, wyposażony w czterordzeniowy procesor Snapdragon i jednostkę GPU Adreno 330 firmy Qualcomm, doskonale obsługujący technologię HTML5. Siedmiocalowa wersja bardzo dobrze radziła sobie z przykładami opisanymi w tej książce. Na rysunku 12.1 przedstawiony został zrzut ekranu z aplikacji Futurogo (rozdział 10.) uruchomionej właśnie na tablecie Kindle Fire HDX w przeglądarce Silk. Obraz ten niczym nie różni się od przedstawionego w rozdziale 10. zrzutu z przeglądarki komputera stacjonarnego. Aby obrócić pojazd, należy przeciągnąć palcem po obszarze kanwy. Aby zmniejszyć lub powiększyć samochód, należy zsunąć lub rozsunąć palce na kanwie. Aby uruchomić animacje, należy dotknąć przycisków *Wnętrze* i *Jazda*.



Rysunek 12.1. Aplikacja Futurgo uruchomiona w tablecie Kindle Fire HDX

A dotknięcie wybranej części karoserii powoduje wyświetlanie nakładki z informacjami o niej. Wydajność jest świetna i znacznie przekracza oczekiwania, jakie można mieć w stosunku do tak lekkiego, zaledwie siedmiocalowego urządzenia.

Chcę podkreślić, że *nic nie robiłem*, aby uruchomić ten program w omawianym tablecie. Po prostu wpisałem adres URL w przeglądarce i po kilku sekundach ujrzałem kompletną stronę z animacjami. Tego typu urządzenia przenośne nie mają myszy, więc musiałem zaimplementować obsługę interfejsu dotykowego.

Dodawanie obsługi interfejsu dotykowego

Mobilne wersje przeglądarek pracujących z HTML5 automatycznie obsługują gesty dotykowe na elementach strony poprzez generowanie odpowiednich zdarzeń `mousedown`. Dzięki temu znajdujące się po prawej stronie okna aplikacji przyciski do uruchamiania animacji działały od razu. Jednak przeglądarki nie generują automatycznie zdarzeń myszy dla elementów kanwy. Tym trzeba zająć się osobiście i zaimplementować obsługę **zdarzeń dotykowych**.

Zdarzenia dotykowe dodano do mobilnych wersji przeglądarek internetowych, gdy interfejsy dotykowe stały się popularne. Są one podobne do zdarzeń myszy pod tym względem, że również dostarczają współrzędne x i y klienta, strony oraz ekranu. Ponadto zawierają jeszcze inne informacje, tzn. większość urządzeń przyjmuje dane dotykowe z więcej niż jednego źródła na raz (np. po jednym dla każdego palca dotykającego ekranu), więc w zdarzeniach znajdują się informacje dotyczące każdego z tych źródeł.

Ponadto przeglądarki definiują kilka nowych typów zdarzeń, których wykaz znajduje się w tabeli 12.2.

Tabela 12.2. Zdarzenia dotykowe przeglądarki internetowej

Zdarzenie	Opis
touchstart	Zdarzenie wyzwalane po wykryciu dotyku (np. po pierwszym zetknięciu palca z ekranem)
touchmove	Zdarzenie wyzwalane po zmianie pozycji dotyku (np. po przeciągnięciu palcem po ekranie)
touchend	Zdarzenie wyzwalane po zakończeniu dotyku (np. po zdjęciu palca z ekranu)
touchcancel	Zdarzenie wyzwalane po tym, jak źródło dotyku wyjdzie poza obszar czuły na dotyk lub dotyk zostanie przerwany w jakiś inny specyficzny sposób (np. wykryto zbyt dużo punktów dotyku)



Kompletna specyfikacja zdarzeń dotyku przeglądarek internetowych znajduje się na stronach W3C pod adresem <http://bit.ly/w3-touch-events>.

Pamiętaj, że obsługa zdarzeń dotyku w niektórych przeglądarkach wciąż jeszcze jest niedokończona, a więc można natknąć się na różne problemy. Przykładowo przeglądarka Internet Explorer dla komputerów stacjonarnych obsługuje zdarzenia dotyku dla komputerów osobistych obsługujących interfejs dotykowy. Jednak wśród typów zdarzeń DOM występują pewne różnice, przez co konieczne jest używanie własności CSS specyficznych dla przeglądarek (z przedrostkiem `-ms`). Szczegółowe informacje można znaleźć w dokumentacji przeglądarki.

Aby dodać do aplikacji obsługę dotyku, musimy zaimplementować procedury obsługi wcześniej wspomnianych zdarzeń. Dodamy je do kontrolera modelu używanego przez przeglądarkę Vizi, aby użytkownik mógł obracać oraz powiększać i zmniejszać model. Ponadto zaimplementujemy obsługę dotyku w samej aplikacji Futurgo, żeby reagowała na dotknięcia części samochodu przez użytkownika.

Implementowanie dotykowej rotacji modelu w przeglądarce modeli

Jedną z najfajniejszych funkcji standardowej wersji aplikacji Futurgo jest możliwość obracania modelu za pomocą myszy oraz jego powiększania i zmniejszania przy użyciu kółka myszy i gładzików. W urządzeniach dotykowych nie ma żadnego z tych urządzeń, więc musimy w zamian zaimplementować obsługę dotyku.

Przypomnij sobie z rozdziału 10., że w aplikacji Futurgo używany jest obiekt `Vizi.Viewer` i jego wbudowany **kontroler modelu** umożliwiający manewrowanie modelem za pomocą myszy. Teraz zmodyfikujemy tego kontrolera tak, by obsługiwał zdarzenia dotykowe. Kod źródłowy opisywanej klasy znajduje się w źródłach Vizi w pliku `src/controllers/orbitControls.js`.

Zaczniemy od dodania procedury nasłuchu zdarzeń `touchstart`, która będzie wywoływać metodę `onTouchStart()`.

```
this.documentElement.addEventListener( 'touchstart', onTouchStart, false );
```

Pozostałe potrzebne procedury nasłuchu zdarzeń zostaną dodane w metodzie `onTouchStart()`. (Zmienna `scope` jest zmienną zamknięcia JavaScript zawierającą wartość `this` dla obiektu sterowania orbitą).

```
scope.documentElement.addEventListener( 'touchmove', onTouchMove, false );
scope.documentElement.addEventListener( 'touchend', onTouchEnd, false );
```

Teraz możemy obsłużyć zdarzenia dotykowe. Na listingu 12.1 znajduje się kod metody `onTouchStart()`. Zasadniczo tworzymy falsozywe zdarzenie naciśnięcia przycisku myszy i wywołujemy procedurę obsługi zdarzeń `onMouseDown()` używaną przez kod obsługi zdarzeń myszy.

Szczegóły każdego źródła dotyku są zapisywane na liście touches zdarzenia będącej tablicą obiektów typu Touch. W tym przypadku przyjmujemy pojedyncze dotknięcie, a więc ignorujemy wszystkie obiekty na tej liście oprócz pierwszego (o numerze 0). Wartości z tego obiektu są kopiowane do naszego fałszywego zdarzenia myszy i przekazywane do metody onMouseDown(). Następnie zdarzenie jest obsługiwane, tak jak zwykle zdarzenie naciśnięcia przycisku myszy.

Listing 12.1. Obsługa zdarzenia dotyku poprzez utworzenie fałszywego zdarzenia naciśnięcia przycisku myszy

```
// Imitacja zdarzenia naciśnięcia lewego przycisku myszy.  
var mouseEvent = {  
    'type': 'mousedown',  
    'view': event.view,  
    'bubbles': event.bubbles,  
    'cancelable': event.cancelable,  
    'detail': event.detail,  
    'screenX': event.touches[0].screenX,  
    'screenY': event.touches[0].screenY,  
    'clientX': event.touches[0].clientX,  
    'clientY': event.touches[0].clientY,  
    'pageX': event.touches[0].pageX,  
    'pageY': event.touches[0].pageY,  
    'button': 0,  
    'preventDefault' : function() {}  
};  
  
onMouseDown(mouseEvent);
```

Jakby to powiedział nieśmiertelny Spock: „Proste, ale skuteczne”.

Podobne tanie sztuczki stosujemy dla zdarzeń touchmove i touchend, tylko używamy tablicy event.changedTouches. Zawiera ona nowe wartości dotyczące każdego źródła dotyku, które się poruszyło. Tym razem także przyjmujemy założenie, że czynność dotknięcia była jednorazowa. To nie przeszkadza, ponieważ wobec zdarzeń wielodotykowych mamy inne plany. Szczegóły znajdziesz w kodzie źródłowym onTouchMove() i onTouchEnd().

Implementowanie zoomu opartego na wielodotyku

Większość urządzeń obsługuje więcej niż jedno źródło dotyku — jest to tzw. **wielodotyk** (ang. *multitouch*). Typowym zastosowaniem tej techniki jest zsuwanie lub rozsuwanie dwóch palców na ekranie w celu zwiększenia lub zmniejszenia wyświetlonych na nim obiektów. Zaimplementujemy tę technikę w naszym kontrolerze modelu Vizi.

Programowanie wielodotyku jest nieco bardziej pracochłonne niż implementacja obsługi pojedynczych dotknięć, ponieważ wymaga śledzenia ruchu więcej niż jednego źródła dotyku. Każdy obiekt typu Touch na liście touches lub changedTouches zawiera własność identifier będącą identyfikatorem, który na pewno nie zmieni się przez cały czas trwania czynności (od touchstart przez touchmove po touchend lub touchcancel).

Spójrzmy na kod źródłowy. Na początku metody touchStart() sprawdzamy, czy jest więcej niż jedno źródło dotyku. Jeśli tak, przyjmujemy, że użytkownik wykonał gest zoomu, a nie gest obrotu modelu. Przy użyciu dwóch pierwszych elementów tablicy touches obliczamy odległość między źródłami dotyku i zapisujemy ją we właściwości touchDistance. Później na jej podstawie określmy, czy użytkownik zsunął, czy rozsunął palce.

```
if ( event.touches.length > 1 ) {  
    scope.touchDistance = calcDistance(event.touches[0],
```

```

        event.touches[1]);
scope.touchId0 = event.touches[0].identifier;
scope.touchId1 = event.touches[1].identifier;
}

```

Dodatkowo musimy zapisać identyfikatory (będące łańcuchami) obu obiektów we własnościach touchId0 i touchId1. Potrzebujemy ich, by sprawdzić, które źródła dotyku się poruszyły, gdy wystąpią kolejne zdarzenia touchmove. Nie ma gwarancji, że poszczególne obiekty dotyku w nowych listach changedTouches zostaną zapisane w takiej samej kolejności jak po zdarzeniu touchstart. Jedyny pewny identyfikator każdego obiektu typu Touch to wartość jego własności identifier. Dlatego właśnie zapisujemy ją sobie na później.

Teraz zajmiemy się obsługą zdarzenia touchmove, co pokazano na listingu 12.2. Działanie metody onTouchMove() zaczyna się od sprawdzenia, czy wykryte zdarzenie to wielodotyk. Jeśli tak, szukamy w tablicy changedTouches dwóch zapisanych wcześniej identyfikatorów touchId0 i touchId1. Interesują nas obiekty typu Touch opatrzone tymi identyfikatorami. Mając je, możemy za pomocą funkcji calcDistance() obliczyć nową odległość. Otrzymany wynik porównujemy z poprzednim. Dodatnia różnica oznacza, że użytkownik rozsunął palce, więc należy powiększyć model (przybliżyć). Jeśli różnica jest ujemna, znaczy to, że użytkownik zsunął palce i model trzeba zmniejszyć (oddalić).

Listing 12.2. Obsługa zoomu

```

if ( event.changedTouches.length > 1 ) {
    var touch0 = null;
    var touch1 = null;
    for (var i = 0; i < event.changedTouches.length; i++) {
        if (event.changedTouches[i].identifier == scope.touchId0)
            touch0 = event.changedTouches[i];
        else if (event.changedTouches[i].identifier == scope.touchId1)
            touch1 = event.changedTouches[i];
    }
    if (touch0 && touch1) {
        var touchDistance = calcDistance(touch0, touch1);
        var deltaDistance = touchDistance - scope.touchDistance;
        if (deltaDistance > 0) {
            scope.zoomIn();
        }
        else if (deltaDistance < 0) {
            scope.zoomOut();
        }
        scope.touchDistance = touchDistance;
    }
}

```

Zobaczmy teraz, jak oblicza się odległość między źródłami dotyku. Na listingu 12.3 pokazano kod źródłowy funkcji calcDistance(), w którym odległość między punktami obliczana jest przy użyciu twierdzenia Pitagorasa.

Listing 12.3. Obliczanie odległości między rozstawionymi palcami

```

function calcDistance( touch0, touch1 ) {
    var dx = touch1.clientX - touch0.clientX;
    var dy = touch1.clientY - touch0.clientY;
    return Math.sqrt(dx * dx + dy * dy);
}

```

Wyłączanie możliwości skalowania na stronie internetowej

Pozostał do dopracowania jeszcze jeden ważny szczegół. Przeglądarki internetowe w urządzeniach mobilnych standardowo umożliwiają powiększanie i zmniejszanie elementów stron internetowych za pomocą dotyku. Jednak funkcja ta będzie kolidować z naszą implementacją. Na szczęście, można ją wyłączyć na poziomie kodu HTML. Wystarczy dodać poniższy element HTML5 <meta>, aby uniemożliwić skalowanie strony przez użytkownika (plik *r12/futurgo.html*):

```
<meta name="viewport"  
      content="width=device-width, initial-scale=1.0, user-scalable=no">
```

Dodawanie zdarzeń dotyku Vizi.Picker do modelu Futurgo

Standardowa wersja aplikacji Futurgo zawiera bardzo przydatne nakładki z informacjami dotyczącymi różnych części modelu. Wystarczy ustawić kursor na wybranej części (przedniej szybie, karoserii albo kole), aby wyświetliły się element <div> zawierający szczegółowe dane na jej temat, ale w urządzeniach przenośnych nie ma myszy, więc funkcja ta nie działa. Możemy jednak sprawić, by nakładki wyświetlały się po dotknięciu wybranych części. Klasa *Vizi.Picker* ma wbudowane mechanizmy obsługi zdarzeń dotykowych. Kod dodany w celu zaimplementowania obsługi dotykowego wyświetlania nakładek znajduje się w pliku *r12/futurgo.js* iaczyna w wierszu 44. Jest też oznaczony pogrubieniem na listingu 12.4.

Listing 12.4. Dodawanie zdarzeń dotyku Vizi.Picker do modelu Futurgo

```
// Zmienia przeźroczystość okien po uruchomieniu aplikacji.  
var that = this;  
scene.map(/windows_front|windows_rear/, function(o) {  
    var fader = new Vizi.FadeBehavior({duration:2, opacity:.8});  
    o.addComponent(fader);  
    setTimeout(function() {  
        fader.start();  
    }, 2000);  
  
    var picker = new Vizi.Picker;  
    picker.addEventListener("mouseover", function(event) {  
        that.onMouseOver("glass", event); });  
    picker.addEventListener("mouseout", function(event) {  
        that.onMouseOut("glass", event); });  
    picker.addEventListener("touchstart", function(event) {  
        that.onTouchStart("glass", event); });  
    picker.addEventListener("touchend", function(event) {  
        that.onTouchEnd("glass", event); });  
    o.addComponent(picker);  
});
```

Procedury obsługi zdarzeń dotyku są proste: ponownie wykorzystamy w nich tanią sztuczkę przekazania sterowania do istniejącej procedury obsługi zdarzeń myszy.

```
Futurgo.prototype.onTouchEnd = function(what, event) {  
    console.log("touch end", what, event);  
    this.onMouseOver(what, event);  
}
```



Cale szczęście, że w metodzie `onMouseOver()` nie ma niczego, co wymagałoby prawdziwego zdarzenia DOM `MouseEvent`, bo nasz kod by nie zadziałał. W tym przypadku się udało, ale *unikaj* stosowania takich sztuczek w kodzie produkcyjnym, ponieważ wynikające z nich błędy mogą ujawnić się w najmniej oczekiwany momencie.

Debugowanie mobilnej funkcjonalności w stacjonarnej wersji przeglądarki Chrome

Gdy nauczyliśmy się obsługiwać zdarzenia dotyku, dodanie tej obsługi do rdzenia systemu Vizi i aplikacji Futurgo było już łatwe. Nawet obsługa wielodotykowego zoomu, mimo że wymagała trochę pracy, nie była wielkim wyzwaniem. Mimo że praca była stosunkowo łatwa, nie możemy zapominać, że jesteśmy tylko ludźmi i popełniamy błędy. Dlatego wszystkie nowo dodane funkcje trzeba dokładnie przetestować i ewentualnie poprawić.

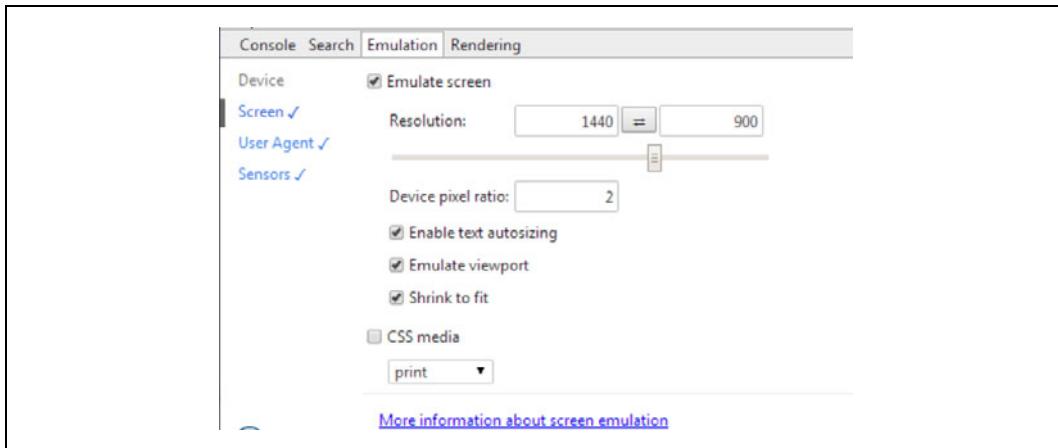
Każda z mobilnych platform wymienionych w tabeli 12.1 obsługuje inny rodzaj debuggera, który można podłączyć do aplikacji działającej na urządzeniu. Niektóre z tych systemów działają bardzo dobrze, ale praca z niektórymi była dla mnie traumatycznym przeżyciem. Wcześniej czy później Ty również będziesz musiał przez to przejść. Nie zamieściłem tu instrukcji obsługi któregokolwiek z tych narzędzi, więc szczegółowych informacji musisz poszukać w odpowiednich źródłach.

Byłoby dobrze, gdyby przed przesłaniem aplikacji na urządzenie można ją było wstępnie przetestować na komputerze stacjonarnym. W sukurs przyjdzie przeglądarka Chrome, która zawiera narzędzia do emulowania wybranych funkcji wersji mobilnej, np. zdarzeń dotykowych. Gdy włączona jest emulacja dotyku, zdarzenia dotykowe można wyzwalać za pomocą myszy. Oto krótkie wprowadzenie.

1. Uruchom aplikację w przeglądarce Chrome.
2. Uruchom debugger Chrome.
3. Kliknij znajdującą się po prawej stronie ikonę strzałki w prawo z trzema kreskami o etykiecie *Show drawer* (wyświetl szufladkę). W efekcie powinno zostać wyświetcone dodatkowe okno z czterema kartami. Kliknij kartę *Emulation* (emulacja), a następnie po lewej stronie kliknij odnośnik *Device* (urządzenie). W oknie, które się pojawi, wybierz urządzenie (możesz pozostawić wybór domyślnej *Google Nexus*) i kliknij przycisk *Emulate* (emuluj).
4. Następnie kliknij odnośnik *Screen* (ekran) i w polach *Resolution* (rozdzielcość) wpisz większą niż domyślna rozdzielcość ekranu, np. 1440×900 pikseli — rysunek 12.2.
5. Od tej pory zdarzenia myszy będą imitowały zdarzenia dotyku. Podczas pracy nie zamykaj debugera.



Emulacja zdarzeń dotyku w przeglądarce Chrome działa tylko wtedy, gdy uruchomiony jest debugger. Po jego zamknięciu zdarzenia dotyku nie są generowane.



Rysunek 12.2. Włączanie emulacji zdarzeń dotyku w stacjonarnej wersji przeglądarki Chrome

Teraz w przeglądarce Chrome włączone jest emulowanie zdarzeń dotyku, dzięki czemu wszystkie zdarzenia myszy będą zamieniane na dotyk i jako takie wysypane do aplikacji. Spójrz na rysunek 12.3. Widać na nim efekt kliknięcia myszą jednej z części modelu. W konsoli pojawiły się informacje o wyzwolonych zdarzeniach touchstart i touchend (zakreślone). Ta prosta funkcja pozwala wstępnie przetestować działanie kodu, zanim wypróbuje się aplikację na urządzeniu przenośnym. Niestety, mamy możliwość testowania tylko pojedynczych zdarzeń dotyku.



Rysunek 12.3. Debugowanie zdarzeń dotyku w aplikacji Futurgo w stacjonarnej wersji przeglądarki Chrome

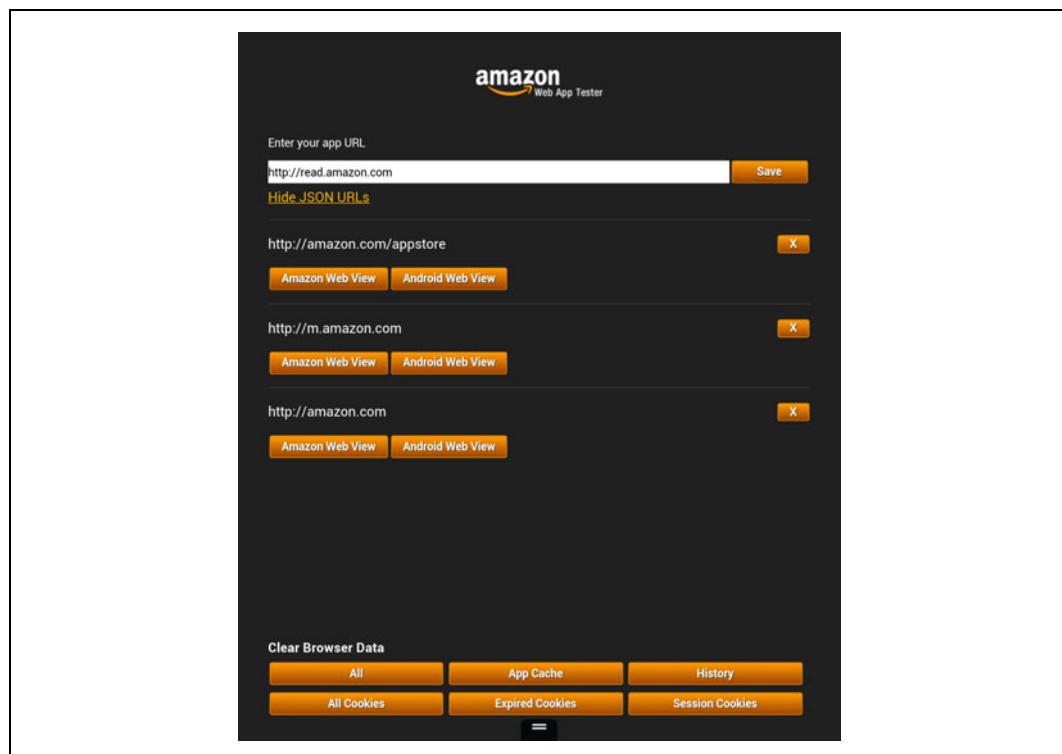
Tworzenie aplikacji sieciowych

Czasami aplikację trzeba spakować w jedną paczkę do wdrożenia na urządzeniu. Jest to konieczne np. wtedy, gdy chcemy zaoferować możliwość zakupów w programie albo jakieś inne usługi, których nie da się świadczyć w przeglądarce internetowej. Można też zrobić to, by umieścić na ekranie ikonę do bezpośredniego uruchamiania aplikacji. Większość nowych platform mobilnych obsługuje aplikacje utworzone przy użyciu technologii HTML5 i JavaScript oraz spakowanych jako **aplikacje sieciowe**.

Tworzenie aplikacji sieciowych i narzędzia do ich testowania

Narzędzia programistyczne do tworzenia aplikacji sieciowych w HTML5 są inne na każdej platformie. W każdym środowisku uruchamianie, testowanie, debugowanie i pakowanie aplikacji odbywa się w nieco odmienny sposób.

Firma Amazon udostępnia narzędzie o nazwie Web App Tester dla systemu Fire OS instalowanego w urządzeniach Kindle. Fire OS to oparty na Androidzie system operacyjny utworzony przez firmę Amazon do obsługi urządzeń Kindle Fire. Narzędzie Web App Tester jest aplikacją dla tego systemu, którą można pobrać ze sklepu Amazon. Szczegółowe informacje znajdują się na stronie <https://developer.amazon.com/sdk/webapps/tester.html>. Na rysunku 12.4 można zobaczyć zrzut ekranu z tego narzędzia.



Rysunek 12.4. Aplikacja Web App Tester firmy Amazon

Obsługa programu jest banalnie łatwa: wystarczy wpisać adres URL strony internetowej, aby otworzyć ją w trybie pełnoekranowym. Program zapamiętuje wpisany adres, więc następnym razem nie trzeba go już ręcznie wpisywać.



Narzędzia programistyczne do tworzenia aplikacji sieciowych różnych platform znacznie się różnią. Dotyczy to nawet różnych wersji systemu Android. Przykładowo system Fire OS dla urządzeń Kindle bazuje na systemie Android, ale firma Amazon dodała do niego wiele własnych narzędzi wspomagających programowanie, testowanie i pakowanie aplikacji. Informacje na temat innych przeróbek systemu Android można znaleźć w ich dokumentacji. Warto też zjrzeć na stronę dla programistów Androida pod adresem <http://bit.ly/dev-android-webapps>.

Pakowanie aplikacji sieciowych do dystrybucji

Przetestowaną i wolną od błędów aplikację można przekazać do wdrożenia. Jest to kolejna dziedzina, w której między platformami występują znaczne różnice. Firma Amazon np. udostępnia portal Mobile App Distribution Portal, w którym zarejestrowani programiści mogą tworzyć aplikacje dla urządzeń Kindle Fire i Android. Aby opublikować aplikację w tym portalu, należy przejść kilka etapów. Najpierw tworzy się **plik manifestu** aplikacji, czyli plik zawierający informacje o jej zawartości i właściwościach. Poniżej znajduje się przykładowa treść takiego pliku dla bardzo prostej aplikacji. Jedyne wymagane pole to `verification_key`, które zawiera wartość wygenerowaną przez Amazon w procesie publikacji. Pozostałe metadane dotyczące aplikacji, np. ikony i opis, dostarcza się przez internet w czasie wysyłania aplikacji i nie wpisuje się ich w manifeście. Dokładne informacje na temat budowy plików manifestu firmy Amazon można znaleźć na stronie <https://developer.amazon.com/sdk/webapps/manifest.html>.

```
{  
    "verification_key":  
        "wpisz klucz weryfikacyjny z karty App File(s)",  
    "launch_path": "index.html",  
    "permissions": [  
        "iap",  
        "geolocation",  
        "auth"  
    ],  
    "type": "web",  
    "version": "0.1a",  
    "last_update": "2013-04-08 13:30:00-0800",  
    "created_by": "webappdev"  
}
```

W systemie operacyjnym Firefox OS proces dystrybucji przebiega inaczej. Opiera się na portalu Firefox Marketplace oraz innej składni manifestu. Oto przykładowa treść takiego pliku.

```
{  
    "name": "Nazwa_aplikacji",  
    "description": "Opis aplikacji",  
    "version": 1,  
    "installs_allowed_from": ["*"],  
    "default_locale": "pl",  
    "launch_path": "/index.html",  
    "fullscreen": "true",  
    "orientation": ["landscape"],  
    "icons": {
```

```
        "128": "/images/icon-128.png"
    },
    "developer": {
        "name": "Twoja nazwa",
        "url": "http://adres.twojej.firmy.pl"
    }
}
```

W plikach manifestu Firefoksa podaje się pliki należące do pakietu, ikonę aplikacji, nazwę i opis aplikacji oraz wybrane informacje dotyczące programisty. Szczegółowe informacje na temat budowania aplikacji dla systemu Firefox OS można znaleźć na stronie https://developer.mozilla.org/en-US/Apps/Developing/Packaged_apps.

Tworzenie aplikacji hybrydowych

Interfejsy API HTML5 i platform mobilnych powoli zbliżają się do siebie. Możliwe że w niedalekiej przyszłości będzie można napisać program przy użyciu HTML5 i wdrożyć go na każdej mobilnej platformie przy użyciu dostarczonych przez producenta narzędzi. Jednak tak może być dopiero w przyszłości, a na razie platformy różnią się między sobą także w kwestiach dotyczących obsługi technologii trójwymiarowych. Jak już pisalem, biblioteka WebGL jest obsługiwana przez większość platform mobilnych, ale nie przez wszystkie.

Z wymienionych, a także innych powodów można spróbować użycia jednej z technologii włączających obsługę WebGL w różnych urządzeniach, dostarczając macierzystą bibliotekę (**adapter**) udostępniającą interfejs API WebGL do użytku z poziomu JavaScriptu. Z pomocą adaptera można połączyć kod w językach JavaScript i HTML z kodem macierzystym w jedną spakowaną aplikację — hybrydę, jeśli ktoś woli.

Powodów, dla których programista może sięgnąć po rozwiązanie hybrydowe, jest kilka.

Brak obsługi przez przeglądarkę

Mimo że system operacyjny iOS jest ostatnim systemem bez obsługi WebGL, jest bardzo ważny. Na platformach, takich jak iOS i np. starsze wersje Androida, które mogą, ale nie muszą obsługiwać technologii WebGL, rozwiązanie hybrydowe pozwala na tworzenie i wdrażanie trójwymiarowych aplikacji WebGL przy użyciu JavaScriptu.

Wydajność

Biblioteki adaptacyjne są zazwyczaj nieco efektywniejsze od równoważnych przeglądarkowych aplikacji WebGL. Dzieje się tak z dwóch powodów. Po pierwsze, biblioteki te mogą oferować lepiej zoptymalizowane i dostosowane maszyny wirtualne JavaScriptu. Po drugie, mogą pomijać dodatkowe warstwy zabezpieczeń WebGL, które są wymagane w przeglądarkach — niezbędne w aplikacjach sieciowych, ale niepotrzebne w aplikacjach macierzystych.

Wdrażanie w postaci aplikacji

Jeśli Twoim celem jest udostępnianie produktu jako aplikacji mobilnej, a nie strony internetowej, zastosowanie rozwiązania hybrydowego może być bardzo dobrym pomysłem. Czasami wartością dodaną takiego podejścia jest dostęp z poziomu JavaScriptu do funkcji (np. zakupów w aplikacji, macierzystych reklamowych pakietów SDK itp.), które w czysto przeglądarkowej wersji byłyby niedostępne.

Istnieje kilka bibliotek adaptacyjnych umożliwiających zastosowanie rozwiązania hybrydowego. Podczas gdy wiele z nich udostępnia obsługę sprzętową dla kanwy i inne specjalne funkcje — z których najbardziej znana jest PhoneGap firmy Adobe (<http://phonegap.com/>) — tylko niektóre oferują wsparcie dla WebGL. Dwie najbardziej godne uwagi to CocoonJS i Ejecta. Narzędzia te służą do tego samego, ale każde z nich działa w inny sposób. Oto krótkie porównanie.

CocoonJS (<http://www.ludei.com/tech/cocoonjs>)

CocoonJS działa w systemach Android i iOS. Ukrywa szczegóły podstawowego systemu operacyjnego w łatwym w użyciu kontenerze aplikacji na kod HTML5 i JavaScript. Do stacza implementacje kanwy, WebGL, Web Audio, WebSockets i wiele innych. Ponadto biblioteka CocoonJS zawiera system do budowy projektów działających w chmurze, więc wystarczy tylko podpisać swój projekt i go zbudować. Programista nie musi uczyć się obsługi specjalnych narzędzi platform, takich jak Xcode dla systemu iOS. CocoonJS to projekt o zamkniętym kodzie źródłowym, nad którym pieczę sprawuje firma Ludei z San Francisco.

Ejecta (<http://impactjs.com/ejecta>)

Ejecta to biblioteka typu open source o funkcjonalności bardzo podobnej do CocoonJS, ale działająca tylko w systemie iOS. Zrodziła się z projektu ImpactJS, czyli silnika do tworzenia gier w HTML5. Obsługa Ejecty jest nieco trudniejsza niż CocoonJS i wymaga znajomości środowiska Xcode oraz macierzystych interfejsów API platformy.

Mimo że sam projekt Ejecta jest otwarty, jego zależność od systemu iOS i środowiska Xcode sprawia, iż nie opiszę go w tej książce. Dlatego przykładową aplikację hybrydową utworzymy na bazie biblioteki CocoonJS.

CocoonJS jako technologia tworzenia gier i aplikacji HTML dla urządzeń mobilnych

CocoonJS to technologia adaptacyjna do tworzenia hybrydowych aplikacji HTML5 dla urządzeń mobilnych. Pełni rolę macierzystego kontenera dla kodu HTML5, tzn. aplikacja lub gra jest wykonywana jako aplikacja macierzysta, ale wewnętrznie wykonuje kod JavaScript i HTML. CocoonJS działa w systemach iOS i Android, w obu oferuje identyczne środowisko wykonawcze.

Przy użyciu biblioteki CocoonJS programista może dostarczyć plik HTML i związanego z nim kod JavaScript w celu wyrenderowania go w trybie pełnoekranowym na dwu- lub trójwymiarowej kanwie przy użyciu standardowych dwuwymiarowych interfejsów API i WebGL. Dodatkowo obsługiwane są technologie Web Audio, ładowanie obrazów, XMLHttpRequest oraz WebSockets. CocoonJS implementuje macierzyste, wspomagane sprzętowo wersje wymienionych interfejsów API oraz zawiera specjalną **maszynę wirtualną** JavaScriptu zoptymalizowaną przez firmę Ludei w celu osiągnięcia jak najlepszej wydajności. Na rysunku 12.5 zaprezentowano zrzut modelu Futurgo wyświetlonego jako pełnoekranowa macierzysta aplikacja dla systemu iOS na urządzeniu Apple iPad 4.

Dla ułatwienia programowania i testowania biblioteka CocoonJS obsługuje aplikację uruchamiającą o nazwie CocoonJS Launcher, w której można podejrzeć wynik pracy, wpisując adres URL albo upuszczając archiwum ZIP zawierające całą treść aplikacji. Program CocoonJS Launcher, widoczny na rysunku 12.6, można pobrać ze sklepów z aplikacjami Apple i Android. Aby przetestować aplikację, kliknij przycisk **YOUR APP** (Twoja aplikacja) i wpisz adresy URL swoich



Rysunek 12.5. Model Futurgo uruchomiony jako macierzysta aplikacja na iPada utworzona przy użyciu biblioteki CocoonJS

plików testowych w oknie tekstowym albo otwórz upuszczone do Launchera pliki ZIP za pomocą programu iTunes lub narzędzi SDK Androida. Szczegółowe informacje można znaleźć w dokumentacji biblioteki CocoonJS.



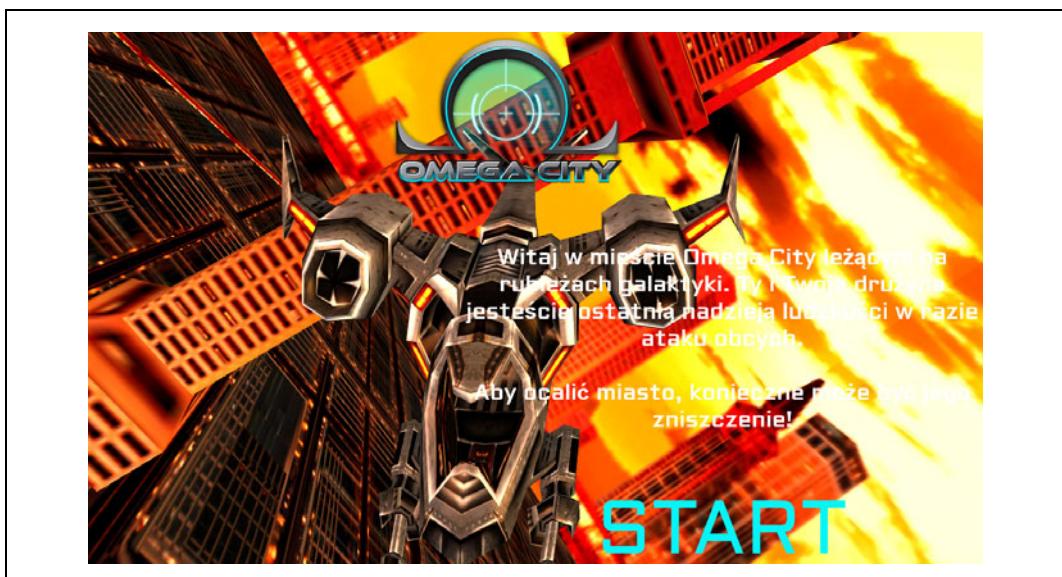
Rysunek 12.6. Ekran główny aplikacji CocoonJS Launcher

Po podejrzeniu i przetestowaniu aplikacji w Launcherze można ją przekształcić w aplikację macierzystą za pomocą usługi chmurowej firmy Ludei. W tym celu należy wysłać pliki aplikacji do tej usługi, aby za kilka minut pobrać paczkę gotową do dystrybucji w sklepach iOS, Amazon, GooglePlay i innych.

Składanie aplikacji przy użyciu biblioteki CocoonJS

Mimo iż twórcy biblioteki CocoonJS utrzymują, że przy jej użyciu można utworzyć dowolny rodzaj aplikacji macierzystej i HTML5, ich głównym celem do tej pory było umożliwienie tworzenia wysoko wydajnych gier. Zatem teraz utworzymy prostą grę, aby zobaczyć, jak się to robi. Wprawdzie będzie to raczej demo gry, nie prawdziwa gra, ale przecież chodzi tylko o przedstawienie procesu. Zanim zagłębimy się w szczegóły używania biblioteki CocoonJS, przyjrzymy się wersji gry działającej na komputerze stacjonarnym. Następnie przystosujemy ją do użytku z CocoonJS.

Otwórz w przeglądarce internetowej plik *r12/omegacity/omegacity.html*. Zobaczysz ekran startowy pokazany na rysunku 12.7. Model powinien wyglądać znajomo, ponieważ jest to scena wirtualnego miasta załadowana z pliku glTF przy użyciu przykładowego programu z rozdziału 8. (*r08/pipelinetreejsgltfscene.html*).

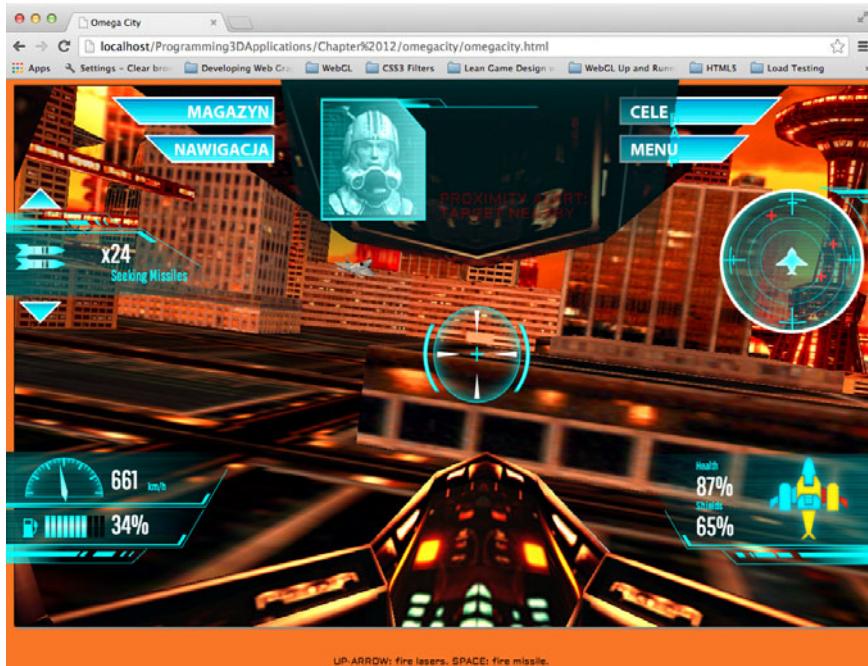


Rysunek 12.7. Ekran startowy gry Omega City. Grafika dwuwymiarowa i projekt autorstwa GameSalad (<http://www.gamesalad.com/>), scena wirtualnego miasta opublikowana dzięki uprzejmości 3DRT (<http://bit.ly/1hTUDXJ>), dźwięki z portalu FreeSound (<http://www.freesound.org/>); wszelkie prawa zastrzeżone



Ten wspaniały model został zamówiony w firmie 3DRT (<http://www.3drt.com/>). Pokazane tu demo powstało przy współpracy z firmą GameSalad z Austin w USA (<http://www.gamesalad.com/>), będącą autorem łatwego w obsłudze narzędzia do tworzenia dwuwymiarowych gier HTML i mobilnych. Przypominam, że model ten, podobnie jak wszystkie inne przedstawione w tej książce, podlega prawom autorskim i nie może być wykorzystywany w innych aplikacjach ani w innych celach niż nauka programowania z tej książki, chyba że zakupi się osobną licencję.

Witaj w mieście Omega City leżącym na rubieżach galaktyki. Ty i Twoja drużyna jesteście ostatnią nadzieję ludzkości w razie ataku obcych. Aby ocalić miasto, konieczne może być jego zniszczenie! Kliknij przycisk *START*, aby rozpocząć grę. Zostanie wyświetlony ekran główny widoczny na rysunku 12.8. Statek jest sterowany przez autopilota, więc gracz może tylko strzelać z broni. Naciśnij strzałkę w góre, aby wystrzelić z lasera. Zobaczyś dwie niebieskie wiązki zbiegające się na środku pola widzenia. Naciśnij spację, aby wystrzelić pocisk. Najpierw usłyszysz odgłos lądowania, a następnie spod statku wyleci pocisk, który po zderzeniu z celem wywoła potężną eksplozję widoczną jako zielona luna. To proste rzeczy, na przykładzie których chcę tylko pokazać, jak tworzy się aplikacje hybrydowe dla systemu iOS przy użyciu biblioteki CocoonJS.



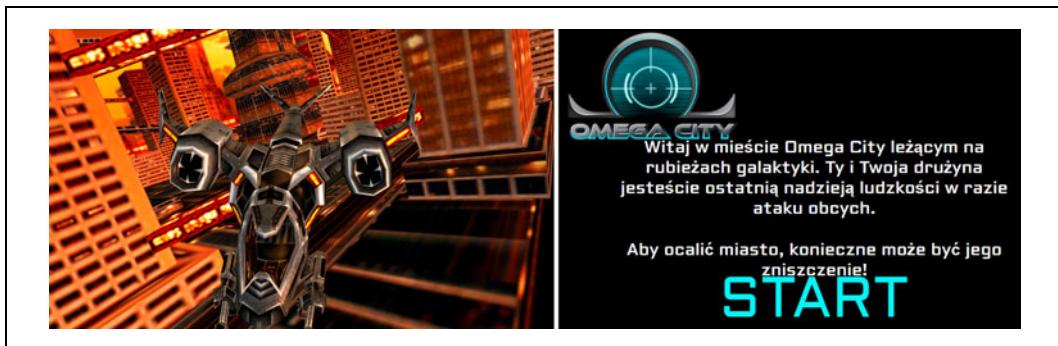
Rysunek 12.8. Gra Omega City uruchomiona na komputerze stacjonarnym

Tworzenie widoku głównego i nakładek

Może zauważysz drobną różnicę między modelem Futurgo działającym na tablecie Kindle Fire HDX w oparciu o czysty HTML5 (rysunek 12.1) a tym samym modelem działającym w systemie iOS jako aplikacja macierzysta utworzona przy użyciu biblioteki CocoonJS (rysunek 12.5). Wersja na tablet Kindle Fire wygląda dokładnie tak samo jak na komputery stacjonarne, tzn. widać w niej fioletowy gradient za modelem, natomiast w wersji dla systemu iOS tło jest czarne. Jest to spowodowane tym, że CocoonJS nie jest pełną przeglądarką i silnikiem składającym się z HTML5, tylko macierzystą implementacją elementu kanwy mającą umożliwić programistom JavaScript tworzenie grafiki dwu- i trójwymiarowej. CocoonJS wczytuje i analizuje znaczniki HTML, ale większość z nich ignoruje wraz ze stylami.

Biblioteka CocoonJS poprawnie interpretuje element HTML kanwy oraz dołączone pliki JavaScript i to wszystko. Nie należy się spodziewać, że arkusze stylów też będą działały. Tło w modelu Futurogo jest dodane za pomocą kodu CSS do elementu <div> kontenera, a CocoonJS ignoruje ten arkusz stylów. Gdybyśmy koniecznie chcieli mieć taki obraz w tle także w aplikacji opartej na CocoonJS, musielibyśmy go narysować na kanwie. Moglibyśmy np. dodać obiekt Three.js daleko w tle albo zdefiniować pudło nieba przy użyciu systemu Vizi (rozdział 11.), ale w tak prostej demonstracyjnej aplikacji gra jest niewarta świeczki.

Mimo że CocoonJS nie obsługuje arkuszy stylów dla elementów w tle, w firmie Ludei zdają sobie sprawę, że wielu programistów do tworzenia interfejsu użytkownika gier używa elementów HTML. Dlatego umożliwiono dodanie drugiego pliku HTML, renderowanego w oknie WebView na głównej kanwie. Kluczowe w tej technice jest podzielenie kanwy i nałożenie na nią elementów HTML z dwóch osobnych plików lub widoków. Na rysunku 12.9 pokazano zawartość każdego z nich, jeden obok drugiego.



Rysunek 12.9. Od lewej: kanwa i nakładka dla wersji gry Omega City opartej na bibliotece CocoonJS

Aby przystosować grę Omega City do użytku z biblioteką CocoonJS, należy podzielić oryginalny plik *omegacity.html* na dwa pliki: *index.html* i *wv.html*. Plik *index.html* zawiera kod dotyczący głównej kanwy, a *wv.html* — kod nakładki. Następnie dodajemy kod pomocniczy CocoonJS z plików JavaScript dostarczonych przez firmę Ludei. Pliki te obsługują dodawanie nakładki przy użyciu kontrolki WebView oraz zawierają narzędzia umożliwiające komunikację między tymi dwoma widokami — trochę dalej rozwinię ten temat.

Biblioteka CocoonJS współpracuje także z przeglądarkami komputerów stacjonarnych, dzięki czemu można podejrzeć efekt swoich prac na komputerze, zanim przejdzie się do testowania w specjalnej aplikacji.

Kod widoku głównej kanwy znajduje się w pliku *r12/omegacity-iOS/index.html* oraz na listingu 12.5. Najpierw dołączamy kilka plików biblioteki CocoonJS. Następnie po załadowaniu strony tworzymy obiekt gry, który będzie renderowany w głównej kanwie WebGL. Kod źródłowy obiektu gry znajduje się w pliku *r12/omegacity-iOS/omegacity.js*. Później tworzymy obiekt do obsługi widoku nakładki, czyli **ekranu HUD** (ang. *heads-up display*) oraz obiekt do obsługi dźwięków. (Należy zwrócić uwagę na przedrostek proxy w nazwie klasy *ProxyHUD*. Zaraz to wyjaśnię).

Listing 12.5. Kod głównego widoku aplikacji opartej na bibliotece CocoonJS

```
<script src=".libs/cocoon_cocoonsextensions/CocoonJS.js">
</script>
<script src=".libs/cocoon_cocoonsextensions/CocoonJS_App.js">
```

```

</script>
<script
src="./libs/cocoon_cocoonsExtensions/CocoonJS_App_ForCocoonJS.js">
</script>
<script src="./libs/vizi/vizi.js"></script>
<script src="omegacity.js"></script>
<script src="omegacityProxyHUD.js"></script>
<script src="omegacitySound.js"></script>
<script>

    var game = null;
    var hud = null;
    var sound = null;
    var gameLoadComplete = false;
    var wvLoadComplete = false;

var handleLoad = function() {

    var container = document.getElementById("container");

    game = new OmegaCity({ container : container,
        loadCallback : onLoadComplete,
        loadProgressCallback : onLoadProgress,
    });

    hud = new ProxyHUD({game : game});

    sound = new OmegaCitySound({game : game});
}

```

Po utworzeniu obiektów gry ładujemy widok nakładki z pliku *wv.html* za pomocą wywołania metody aplikacji CocoonJS `loadInTheWebView()`.

```

setTimeout(function() {

    CocoonJS.App.onLoadInTheWebViewSucceed.addEventListener(
        function(url) {
            CocoonJS.App.showTheWebView();
            Vizi.System.log("Wczytano widok.");
            wvLoadComplete = true;

            if (gameLoadComplete) {
                gameReady();
            }
        }
    );
    CocoonJS.App.onLoadInTheWebViewFailed.addEventListener(
        function(url) {
            Vizi.System.log("Nie wczytano widoku.", url);
        }
    );
    CocoonJS.App.loadInTheWebView("wv.html");
}, 10);

sound.enterState("load");
game.load();
}

```

Nakładka zawiera cały kod HTML i JavaScript widoku (listing 12.6). Otwórz plik *r12/omegacity-iOS/wv.html*, aby go obejrzeć. Po kodzie obiektów ekranu HUD dodajemy pliki biblioteki CocoonJS pomocne w obsłudze tego widoku, a następnie tworzymy parę własnych obiektów,

z których będziemy korzystać tylko w interfejsie użytkownika. (Znów pojawiają się obiekty proxy, do których wkrótce dojdziemy).

Listing 12.6. Kod nakładki aplikacji opartej na bibliotece CocoonJS

```
<!-- Informacja o wczytywaniu -->
<div id="loadStatus" style="display:none">
    Loading...
</div>
<!-- Ekran z przyciskiem START -->
<div id="startScreen" style="display:none">
    <!-- Logo -->
    <div id="logowtext"></div>
    <div id="startScreenText">
        Witaj w mieście Omega City leżącym na rubieżach galaktyki. Ty i Twoja drużyna jesteście ostatnią  

        nadzieję ludzkości w razie ataku obcych.<br><br>
        Aby ocalić miasto, konieczne może być jego zniszczenie!
    </div>
    ... <!-- Dalszy kod HTML -->
    <script src=".libs/cocoon_cocoonsExtensions/CocoonJS.js">
    </script>
    <script src=".libs/cocoon_cocoonsExtensions/CocoonJS_App.js">
    </script>
    <script src=".libs/cocoon_cocoonsExtensions/CocoonJS_App_ForWebView.js">
    </script>
    <script src="omegacityGameProxy.js"></script>
    <script src="omegacityProxySound.js"></script>
    <script src="omegacityHUD.js"></script>
    <script>

        var hud = null;
        var game = null;
        var sound = null;

    var onload = function() {

        hud = new OmegaCityHUD();
        sound = new ProxySound();
        game = new OmegaCityGameProxy();
    }
}
```

Aby połączyć oba widoki, trzeba zrobić jeszcze jedno: należy sprawić, żeby nakładka była przezroczysta. W tym celu za pomocą CSS ustawiamy przezroczysty kolor tła wszystkich elementów (plik *r12/omegacity-iOS/css/omegacity.css*).

```
body {
    background-color:rgba(0, 0, 0, 0);
    color:#11F4F7;
    padding:0;
    margin-left:0;
    margin-right:0;
    overflow:hidden;
}
```

Obsługiwanie komunikacji między kanwą a nakładką

Widok nakładki w bibliotece CocoonJS jest zaimplementowany jako kontrolka WebView umieszczona na głównym widoku kanwy. W efekcie maszyna wirtualna JavaScriptu obsługująca widok kanwy jest całkowicie niezależna od maszyny wirtualnej JavaScriptu obsługującej widok

nakładki. Innymi słowy, te dwa silniki działają w różnych kontekstach i najprawdopodobniej używają dwóch całkiem różnych maszyn wirtualnych JavaScript! Widok główny korzysta z maszyny wirtualnej biblioteki CocoonJS, a kontrolka WebView — z macierzystej maszyny wirtualnej platformy. Jeśli więc w widoku głównym napiszemy kod odwołujący się do funkcji znajdujących się w nakładce, wystąpi błąd, ponieważ implementacje tych funkcji nie zostaną znalezione. Podobnie jest też w drugą stronę. Jednak twórcy biblioteki CocoonJS zadali o możliwość komunikacji między widokami, implementując mechanizm przesyłania komunikatów. Na szczęście, nie trzeba poznawać szczegółów działania, aby z niego korzystać.

W bibliotece CocoonJS znajduje się metoda aplikacji o nazwie `forwardAsync()` służąca do przesyłania łańcuchów między kontekstami. Przesyłane łańcuchy są wykonywane jako kod JavaScript za pomocą funkcji `eval()`. Aby więc wywołać funkcję z drugiego kontekstu, wystarczy utworzyć łańcuch reprezentujący takie wywołanie.

Dla zwiększenia czytelności kodu wszystkie wywołania metody `forwardAsync()` opakujemy w proste wywołanie metody na obiekcie proxy (pośrednim): wywołanie metody obiektu pośredniego powoduje wywołanie metody `forwardAsync()`, która z kolei wysyła wiadomość do drugiego („zdalnego”) kontekstu. Po wykonaniu kodu reprezentowanego przez tę wiadomość następuje wywołanie w zdalnym kontekście funkcji, która może w końcu wywołać metodę zdalnego obiektu.

Aby zobaczyć, jak to działa na konkretnym przykładzie, spójrz na kod uruchamiający grę po naciśnięciu przez użytkownika przycisku *START* (plik *r12/omegacity-iOS/omegacityProxyHUD.js*). Jest to metoda z klasy `OmegaCityGameProxy` przesyłająca wiadomość z nakładki do widoku głównego.

```
OmegaCityGameProxy.prototype.play = function() {
    CocoonJS.App.forwardAsync("playGame();");
}
```

W głównym widoku znajduje się kod obsługujący wiadomość `playGame()`. Nakazuje on mechanizmowi obsługi dźwięków wyłączenie odgłosów gry oraz głównemu obiektyowi gry rozpoczęcie działania.

```
function playGame() {
    sound.enterState("play");
    game.play();
}
```

W grze również występują zdarzenia, które mogą zmieniać coś na ekranie HUD, np. zmniejszać wartość licznika pocisków po wystrzeleniu jednego z nich. A gdy w pobliżu pojawi się statek obcych, na górze po lewej stronie wyświetlane jest specjalne ostrzeżenie w postaci czerwonego migającego tekstu. Te metody ekranu HUD implementujemy przy użyciu obiektu pośredniego wysyłającego wiadomości z widoku głównego do nakładki.

```
ProxyHUD.prototype.enterState = function(state, data) {
    CocoonJS.App.forwardAsync("hudEnterState('" + state + "','" +
        data + "');");
}
```

Kod nakładki obsługuje wiadomość `hudEnterState()` poprzez wywołanie metody obiektu `HUD enterState()`.

```
function hudEnterState(state, data) {
    console.log("Stan HUD: " + state + " " + data);
    hud.enterState(state, data);
}
```



Przedstawione techniki programowania mogą wydawać się dziwne, ale są powszechnie stosowane w systemach opartych na **komunikacji międzyprocesowej** (ang. *interprocess communication — IPC*), w których stosuje się **zdalne wywoływanie procedur** (ang. *remote procedure call — RPC*) w celu nawiązania komunikacji między osobnymi procesami komputera.

Dwuwidokowa architektura biblioteki CocoonJS wymusza użycie techniki RPC tylko wtedy, gdy programista buduje nakładkę HTML5 w aplikacji hybrydowej. Pisanie wywołań pośrednich w dwóch kierunkach jest żmudne i warto by je zautomatyzować. Programiści z Ludei w rozmowie ze mną sugerowali, że nad tym pracują.

Tworzenie hybrydowych aplikacji WebGL — konkluzja

W tym podrozdziale poznałeś techniki tworzenia trójwymiarowych aplikacji mobilnych przy użyciu technologii HTML5 za pomocą metody hybrydowej. Prześledziłeś przykład tworzenia macierzystej aplikacji z użyciem kontrolki WebView i specjalnej biblioteki emulującej API WebGL. Techniki te stosuje się przy tworzeniu aplikacji dla takich środowisk jak system operacyjny iOS, w którym przeglądarki Safari i Chrome nie obsługują technologii WebGL.

Poznałeś bibliotekę CocoonJS firmy Ludei jako jedno z dostępnych rozwiązań do tworzenia aplikacji hybrydowych. Z pomocą tej biblioteki można łatwo zbudować aplikację bez uczenia się obsługi macierzystych interfejsów API typu Cocoa dla systemu iOS. Mimo to, musieliszmy wykonać parę dodatkowych czynności związanych z obsługą nakładki HTML5. Biblioteka CocoonJS nie jest kompletną przeglądarką HTML5, a jedynie mechanizmem do renderowania na kanwie i dlatego musieliszmy wydzielić wszystkie elementy HTML5 dotyczące interfejsu użytkownika do kontrolki WebView, a następnie zaimplementować komunikację między tym widokiem a główną kanwą za pomocą specjalnych interfejsów API JavaScript. Wprawdzie rozwiązanie to nie jest idealne, ale w większości przypadków wystarcza. Niestety, biblioteka CocoonJS nie ma otwartego kodu źródłowego, a będąca jej właścicielem firma zastanawia się nad możliwościami jej licencjonowania. Alternatywą jest otwarta biblioteka Ejecta, ale żeby jej używać, trzeba mieć dużą wiedzę na temat programowania w systemie iOS. Ponadto produkt ten jest jeszcze niedokończony.

Konkluzja jest taka, że nie ma idealnego rozwiązania. Są natomiast sensowne możliwości do wyboru, więc każdy powinien znaleźć coś odpowiadającego jego potrzebom i możliwościom finansowym.

Wydajność mobilnych aplikacji trójwymiarowych

Platformy mobilne mają zasoby bardziej ograniczone niż komputery stacjonarne. Zazwyczaj dysponują mniejszą ilością pamięci fizycznej oraz mniej wydajnymi jednostkami CPU i GPU. Ponadto w niektórych przypadkach poważnym ograniczeniem może być połączenie internetowe. Dlatego niezależnie od tego, czy tworzysz aplikację przeglądarkową, spakowaną aplikację sieciową HTML5, czy aplikację hybrydową przy użyciu bibliotek CocoonJS lub Ejecta, musisz cały czas dbać o optymalną wydajność swojego produktu.

Szczegółowy opis zagadnień wydajności wybiega poza zakres tematyczny tej książki, ale później zamieściłem parę wskazówek dotyczących najważniejszych problemów i technik (kolejność opisywanych tematów jest przypadkowa).

Zarządzanie pamięcią w JavaScript

W języku JavaScript usuwanie nieużywanych obiektów odbywa się automatycznie. Oznacza to, że programista nie musi własnoręcznie alokować obiektów, ponieważ robi to za niego maszyna wirtualna, która dodatkowo usuwa je i przywraca do użytku zajmowaną przez nie pamięć, gdy przestaną być używane. Proces ten nazywa się **usuwaniem nieużytków** (ang. *garbage collection*). Standardowo mechanizm usuwania nieużytków włącza się, gdy maszyna wirtualna uzna to za stosowne. Gdy maszyna wirtualna uruchomi ten mechanizm, aplikacje mogą chwilowo tracić płynność działania. Istnieje kilka technik, które pozwalają zminimalizować czas trwania procesu usuwania nieużytków. Oto one.

- Zaallokowanie całej potrzebnej pamięci podczas uruchamiania aplikacji.
- Utworzenie pul obiektów wielokrotnego użytku, które mogą być przywracane według uznania programisty.
- Zwracanie skomplikowanych wartości przez funkcje w miejscu poprzez przekazywanie obiektów zamiast zwracania nowych obiektów JavaScript.
- Unikanie używania zamknięć (tzn. obiektów powiązanych z innymi obiektami znajdującymi się poza zakresem używającej je funkcji).
- Unikanie używania operatora new zawsze wtedy, gdy jest to możliwe.

Platformy mobilne z racji ograniczonych zasobów pamięciowych są szczególnie wrażliwe na działanie mechanizmu usuwania nieużytków.

Mniejsza moc jednostek CPU i GPU

Producenci, starając się, aby ich produkty były jak najlżejsze i jak najtańsze, używają słabszych i mniej kosztownych części, np. procesorów głównych (CPU) i graficznych (GPU). Wprawdzie platformy mobilne i tak zaskakują wydajnością, ale do komputerów stacjonarnych wciąż im daleko. Aby nie obciążać nadmiernie słabszych jednostek CPU i GPU oraz zapewnić lepsze walory użytkowe swoich aplikacji, a także potencjalnie oszczędzić trochę energii akumulatora, można zastosować następujące techniki.

- **Zmniejsz rozdzielcość treści trójwymiarowej.** Treść trójwymiarowa może znacznie obciążyć zarówno jednostkę CPU, jak i GPU. Zwłaszcza w telefonach komórkowych używanie grafik o wysokiej rozdzielcości może nie mieć sensu, ponieważ ekranы montowane w tych urządzeniach i tak mają niższą rozdzielcość. Po co w takim razie marnować zasoby? Ponadto dzięki przesyłaniu mniejszej ilości danych zmniejszymy obciążenie sieci. Z drugiej strony, nowe tablety mają ekranы o bardzo wysokiej rozdzielcości, więc należy znaleźć złoty środek.
- **Dokładnie przyjrzyj się swoim algorytmom.** W bardzo szybkim komputerze niewydajny kod może pozostać niewykryty, ale urządzenia mobilne szybko go ujawnią. Dotknij np. metalowej części karoserii pojazdu Futurgo na tablecie Kindle Fire HDX. Czasami zauważysz krótkie opóźnienie spowodowane poszukiwaniem przez program dotkniętego obiektu. Jest to efekt uboczny implementacji wybierania obiektów w bibliotece Three.js. Algorytmy te nie są zoptymalizowane i w niewielkim urządzeniu nie grzeszą wydajnością. Kiedyś pewnie doczekają się poprawki albo nowej lub lepszej implementacji w systemie szkieletowym typu Vizi, ale na razie trzeba uważać na takie pułapki i — jeśli trzeba — szukać alternatywnych rozwiązań.

- **Uprość shadery.** Shadery GLSL bywają skomplikowane do tego stopnia, że po komplikacji mogą przekroczyć możliwości sprzętowe niektórych słabszych urządzeń. Pisząc programy dla tych platform, staraj się maksymalnie upraszczać programy cieniące.

Ograniczenia zasobów sieciowych

Z myślą o urządzeniach, w których używa się komórkowych połączeń internetowych i ograniczonych limitów transferu, należy ograniczać ilość przesyłanych danych. Treść trójwymiarowa ma duże rozmiary i jej przesyłanie może znacznie obciążać mało wydajne łączę. Dlatego projektując aplikację, warto rozważyć możliwość zastosowania następujących technik.

- **Pakowanie zasobów:** najlepiej dostarczać aplikacje w postaci pojedynczych pakietów. Wówczas treść jest wysyłana jednorazowo podczas instalacji programu.
- **Wykorzystanie pamięci podręcznej przeglądarki.** Staraj się tak projektować swoje zasoby, aby dało się je zapisać w pamięci podręcznej przeglądarki i uniknąć niepotrzebnego wielokrotnego pobierania.
- **Wysyłanie zasobów partiami.** Ta już klasyczna metoda optymalizacji aplikacji sieciowych umożliwia zmniejszenie liczby żądań sieciowych i połączeń z serwerem. Jeśli np. trzeba przesyłać kilka map bitowych do implementacji paska postępu, można je zapisać w jednym dużym obrazie, tzw. spricie CSS (wszystkie obrazy są zapisane w jednym pliku i wybiera się je za pomocą kodu CSS).
- **Używanie formatów binarnych i kompresowanie danych.** Jednym z najważniejszych powodów, dla których rozpoczęto prace nad formatem glTF, opisany w rozdziale 8., była konieczność zredukowania rozmiaru plików poprzez zastosowanie reprezentacji binarnej, w celu skrócenia czasu ich pobierania. Technikę tę można połączyć z kompresją po stronie serwera oraz specjalnymi algorytmami, np. do kompresji geometrii trójwymiarowej, aby uzyskać jeszcze lepsze efekty.

Podsumowanie

W tym rozdziale poznaleś podstawy nowych technik programowania mobilnych aplikacji trójwymiarowych przy użyciu technologii HTML5 i WebGL. Platformy mobilne powoli doganiają komputery stacjonarne pod względem wydajności, a jednocześnie przyczyniają się do uzupełniania HTML5 w wiele różnych, ciekawych funkcji. Większość tych środowisk obsługuje już technologie trójwymiarowe: z CSS3 można korzystać wszędzie, a WebGL działa we wszystkich mobilnych wersjach przeglądarek oprócz Safari i Chrome dla systemu iOS.

Techniki tworzenia aplikacji WebGL dla urządzeń przenośnych są bardzo proste. Najczęściej wystarczy zbudować zwykłą aplikację, która powinna działać wszędzie. Trzeba tylko zamienić mechanizmy obsługi myszy na mechanizmy obsługi zdarzeń dotykowych. W rozdziale pokazałem, jak dodać obsługę zdarzeń dotykowych do przeglądarki Vizi. Zaimplementowałem obsługę przeciągnięć palcem w celu obracania modelu oraz obsługę gestu zsuwania i rozsuwania palców w celu powiększania i zmniejszania obrazu. Ponadto włączyliśmy możliwość wyświetlania dodatkowych informacji o częściach modelu za pomocą pojedynczego krótkiego dotknięcia. Aby ułatwić sobie pracę i testowanie aplikacji dotykowych, można włączyć emulację zdarzeń dotyku w przeglądarce Chrome. Umiemy też już tworzyć spakowane aplikacje WebGL, tzw. aplikacje sieciowe, przy użyciu specjalnych technologii pakowania i dystrybucji udostępnianych przez właścicieli platform, takich jak Mobile App Distribution Portal firmy Amazon.

Dla przeglądarek nieobsługujących technologii WebGL można użyć adapterów typu CocoonJS lub Ejecta służących do tworzenia aplikacji „hybrydowych” łączących kod HTML5 z kodem macierzystym. Dzięki temu można korzystać z języka JavaScript, a mimo to szybko wdrażać macierzyste aplikacje i przy okazji zyskać dostęp do funkcji platformy, które nie są normalnie osiągalne w przeglądarce, np. zakupów w aplikacji.

W ostatniej części rozdziału napisałem, jak unikać problemów z wydajnością aplikacji przeznaczonych dla platform mobilnych. Mimo że wydajność tych platform w ostatnich latach znacznie się poprawiła, nadal nie dorównują one komputerom stacjonarnym. Dlatego należy mieć na uwadze kwestie dotyczące wydajności, a szczególnie zarządzania pamięcią, obciążenia jednostek CPU i GPU oraz obciążenia sieci.

Źródła informacji

W tym dodatku znajduje się podzielona na kategorie lista źródeł informacji dla programistów. Wiele z wymienionych stron sam często odwiedzam, aby znaleźć najświeższe wiadomości z branży, dowiedzieć się o nowych bibliotekach i narzędziach, obejrzeć najnowocześniejsze wersje demo oraz poznać techniki opracowywane przez wiodących programistów.

WebGL

Specyfikacja WebGL

Opiekę nad standardem WebGL sprawuje grupa Khronos, czyli organizacja rozwijająca także takie technologie jak OpenGL, COLLADA i wiele innych. Najnowszą wersję oficjalnej specyfikacji można znaleźć w witrynie grupy Khronos pod adresem <http://www.khronos.org/registry/webgl/specs/latest/1.0/>.

Listy mailingowe i fora o tematyce WebGL

Organizacja Khronos prowadzi publiczną listę mailingową przeznaczoną do dyskusji na temat najnowszych wersji specyfikacji WebGL. Listę public_webgl@khronos.org można subskrybować, postępując zgodnie z instrukcjami zamieszczonymi na stronie <http://www.khronos.org/webgl/public-mailing-list/>.

Ponadto istnieje też grupa dyskusyjna Google o tematyce dotyczącej bardziej ogólnie programowania przy użyciu technologii WebGL. Można się do niej zapisać na stronie <http://goo.gl/CJlvc4>.

Blogi o tematyce WebGL i serwisy z wersjami demo

Istnieje wiele fantastycznych blogów poświęconych programowaniu przy użyciu technologii WebGL. Oto niektóre z tych, które regularnie odwiedzam.

Learning WebGL (<http://learningwebgl.com/blog/>)

To jedna z najstarszych stron internetowych poświęconych technologii WebGL. Jej twórcą jest Giles Thomas, ale aktualnie jest w moich rękach. Powinna być pierwszą lekturą dla tych, którzy potrzebują wprowadzenia do podstaw niskopoziomowego programowania przy

użyciu WebGL i obsługi API tej biblioteki. Ponadto w serwisie można znaleźć tygodniowe zestawienie najnowszych aplikacji demonstracyjnych i projektów programistycznych ze świata WebGL.

Learning Three.js (<http://learningthreejs.com/>)

Blog Jerome'a Etienne'a poświęcony praktycznym technikom programowania przy użyciu biblioteki Three.js.

TojiCode (<http://blog.tojicode.com/>)

Blog Brandonego Jonesa, pracownika firmy Google. Na stronie można znaleźć mnóstwo artykułów zgłębiających temat interfejsu API WebGL i również wiele wysokiej jakości publikacji na inne tematy związane z programowaniem.

Three.js w Reddit (<http://www.reddit.com/r/threejs>)

Strona Reddit poświęcona bibliotece Three.js jest prowadzona przez Thea Armoura, który często ją aktualizuje. Można na niej znaleźć wiele wersji demo, poznać różne techniki oraz przeczytać niezliczoną ilość artykułów i wiadomości.

WebGL.com (<http://webgl.com>)

Prowadzona przez Dariena Acostę z Nowego Jorku witryna dla odkrywców nowych gier, wersji demo i aplikacji opartych na WebGL.

Wersje demo aplikacji WebGL w portalu Mozilla

(<https://developer.mozilla.org/pl/demos/tag/tech:webgl>)

Aplikacje demonstracyjne utworzone przez Mozilla Labs i partnerów.

Eksperymenty WebGL w przeglądarce Chrome (<http://www.chromeexperiments.com/webgl>)

Wersje demo utworzone przez firmę Google i partnerów.

Zajęcia poświęcone WebGL w HTML5 (<http://www.cs.put.poznan.pl/aurbanski/bai.php>)

Strona z wykładami na temat WebGL w HTML5 wykładowcy jednej z polskich politechnik.

Wprowadzenie do WebGL (<http://student.agh.edu.pl/~kiepas/files/tutorialWebGL.pdf>)

Praca studentów Akademii Górnictwo-Hutniczej na temat WebGL i HTML5.

WebGL w serwisach społecznościowych

Jestem gospodarzem grupy spotkaniowej WebGL w okolicy zatoki San Francisco (<http://meetup.com/WebGL-Developers-Meetup/>). Istnieją też podobne grupy w Los Angeles, Nowym Jorku, Bostonie, Londynie i wielu innych miejscowościach. Spotkania takie są dobrym miejscem poznania osób o podobnych zainteresowaniach. Jeśli nie mieszkasz w pobliżu San Francisco, poszukaj grupy działającej blisko Twojego miejsca zamieszkania w portalu Meetups.com!

Istnieją też grupy w serwisach LinkedIn (<http://www.linkedin.com/groups?gid=2426944>) i Facebook (<https://www.facebook.com/groups/webgl/>).

CSS3

Specyfikacje CSS3

Specyfikacjami CSS3 (przekształcenia, przejścia, animacje, filtry itd.) zarządza konsorcjum W3C (ang. *World Wide Web Consortium*):

<http://www.w3.org/TR/css3-transforms/>
<http://www.w3.org/TR/css3-transitions/>
<http://www.w3.org/TR/css3-animations/>
<http://www.w3.org/TR/filter-effects/>

Opisane w rozdziale 6. filtry własne użytkownika CSS są promowane głównie przez firmę Adobe. Nie są one powszechnie obsługiwane przez przeglądarki internetowe i aktualnie obsługuje je tylko przeglądarka Chrome. Dlatego należy używać ich z rozwagą. Najświeższe informacje na ich temat można znaleźć na stronie <http://adobe.github.io/web-platform/samples/css-customfilters/>.

Blogi o tematyce CSS3 i serwisy z wersjami demo

Najlepszą stronę z objaśnieniem przekształceń trójwymiarowych w CSS utworzył David DeSandro z Twittera. Jej adres to <http://24ways.org/2010/intro-to-css-3d-transforms/>.

Kilka świetnych przykładów zastosowania trójwymiarowych efektów CSS można znaleźć na blogu Codrops (<http://tympanus.net/codrops/>). W serwisie znajdziesz m.in. trójwymiarową półkę na książki (<http://tympanus.net/codrops/2013/01/08/3d-book-showcase>) opisaną w rozdziale 6.

Parę ciekawych przykładów użycia technologii trójwymiarowej CSS znajduje się też w serwisie Dirk'a Webera pod adresem <http://www.elegtriq.com>.

Keith Clark przekroczył granice niemożliwego i utworzył fantastyczną strzelankę przy użyciu tylko trójwymiarowych technik CSS (<http://blog.keithclark.co.uk/creating-3d-worlds-with-html-and-css>).

Wiele ciekawych artykułów na temat przejść i animacji CSS pisze Kirupa Chinnathambi z firmy Microsoft. Szczególnie godne uwagi są publikacje dostępne pod adresami <http://bit.ly/kirupa-transitions> i <http://bit.ly/kirupa-animations>.

Kilka godnych uwagi artykułów opublikowała też firma Bradshaw Enterprises (<http://css3.brashawenterprises.com>). Wśród jej publikacji można znaleźć porady dotyczące używania przejść, przekształceń, animacji i filtrów CSS3.

Kanwa

Specyfikacja kontekstu kanwy dwuwymiarowej

Specyfikacja interfejsu API kanwy dwuwymiarowej znajduje się pod opieką organizacji W3C. Jej najnowszą wersję można znaleźć na stronie <http://www.w3.org/TR/2dcontext2/>.

Kursy obsługi kanwy dwuwymiarowej

Jak napisałem w rozdziale 7., programiści mogą tworzyć trójwymiarowe aplikacje renderowane za pomocą interfejsu API kanwy dwuwymiarowej przy użyciu biblioteki Three.js lub K3D/Phoria (której opis znajduje się nieco dalej). Biblioteki te ukrywają szczegóły dotyczące renderowania na kanwie dwuwymiarowej i udostępniają wysokopoziomowe konstrukcje trójwymiarowe do programowania. Jeśli jednak chcesz dowiedzieć się, jak dokładnie działa API kanwy dwuwymiarowej, w internecie znajdziesz mnóstwo materiałów na ten temat. Oto kilka adresów, które przydały mi się podczas szukania informacji do tej książki:

<http://bit.ly/canvas-tutorial>

<http://bit.ly/draw-graphics-w-canvas>

http://www.w3schools.com/html/html5_canvas.asp

<http://diveintohtml5.info/canvas.html>

Systemy szkieletowe, biblioteki i narzędzia

Biblioteki do programowania trójwymiarowego

Ostatnio pojawiło się kilka otwartych trójwymiarowych bibliotek JavaScript. Oto lista kilku najlepszych z nich w przypadkowej kolejności.

Three.js (<http://threejs.org/>)

Jest to zdecydowanie najpopularniejsza biblioteka z obsługą grafu sceny do tworzenia aplikacji WebGL. Użyto jej do budowy wielu powszechnie znanych przykładowych aplikacji demonstrujących możliwości technologii WebGL. Zawiera łatwy w użyciu zestaw gotowych obiektów, z których często korzysta się w typowych projektach. Jest szybka, a w jej budowie zastosowano wiele najlepszych technik programistycznych. Ponadto dzięki licznym wbudowanym typom obiektów i narzędziom pomocniczym biblioteka Three.js stwarza duże możliwości dla programisty. Wtyczkowy system renderowania umożliwia renderowanie treści trójwymiarowej (z pewnymi ograniczeniami) na kanwie dwuwymiarowej, w SVG oraz za pomocą CSS3 przy użyciu przekształceń trójwymiarowych. Three.js to dobrze prowadzona biblioteka, którą rozwija kilku programistów.

SceneJS (<http://www.scenejs.org/>)

Silnik do programowania trójwymiarowego w JavaScriptie o otwartym kodzie źródłowym, udostępniający interfejs grafu sceny bazujący na formacie JSON. Jego specjalnością jest efektywne renderowanie dużych liczb osobnych obiektów, co jest wymagane w szczegółowo modelach używanych w inżynierii i medycynie. Ponadto SceneJS obsługuje fizykę i zawiera niektóre konstrukcje działające na poziomie wyższym niż te w Three.js, np. model zdarzeń i API grafu sceny w stylu jQuery.

GLGE (<http://www.glege.org/>)

GLGE to biblioteka JavaScript, która ma ułatwić używanie oraz skrócić czas przygotowywania biblioteki WebGL, aby programiście pozostało więcej czasu na tworzenie treści. GLGE jest dobra do podstawowych zastosowań, ale ma funkcjonalność uboższą niż Three.js i SceneJS.

K3D i Phoria (<http://www.kevs3d.co.uk/dev/phoria/>)

K3D i jej następczyni Phoria to biblioteki do renderowania grafiki trójwymiarowej przy użyciu samego interfejsu API kanwy dwuwymiarowej. Phoria jest produktem Kevin Roasta z Wielkiej Brytanii (<http://www.kevs3d.co.uk/dev/>; na Twitterze szukaj @kevinroast). Roast to programista interfejsów użytkownika zafascynowany grafiką. Phoria nie jest wprawdzie tak rozwiniętym produktem jak Three.js, ale i tak robi duże wrażenie. Jest bardzo szybka i doskonale spisuje się w zakresie cieniowania oraz obsługi tekstur. Ponieważ jednak bazuje na programowym mechanizmie renderującym, jej możliwości w zakresie generowania grafiki trójwymiarowej są ograniczone. Niektórych efektów praktycznie nie da się dobrze zaimplementować.

Silniki gier trójwymiarowych

Do obiegu trafia coraz więcej różnych silników gier opartych na technologii WebGL. Biblioteki te są dobrym wyborem dla twórców gier i skomplikowanych aplikacji trójwymiarowych, ale są raczej zbyt skomplikowane do tworzenia prostych projektów. (Więcej na ten temat piszę w następnym punkcie poświęconym systemom szkieletowym). Jeśli nie napisałem inaczej, oznacza to, że dany silnik ma otwarty kod źródłowy.

playcanvas (<http://playcanvas.com/>)

Firma PlayCanvas z Londynu opracowała bogato wyposażony silnik i działające w chmurze narzędzie do tworzenia treści. Narzędzie to zawiera scenę, na której może współpracować wiele osób w czasie rzeczywistym. Ponadto produkt jest zintegrowany z serwisami GitHub i Bitbucket oraz umożliwia publikowanie w mediach społecznościowych za pomocą kliknięcia jednego przycisku. Podczas pisania tej książki udostępniany jest kod źródłowy klienta, ale nie wiadomo nic na temat licencjonowania.

Turbulenz (<http://biz.turbulenz.com/developers/>)

Turbulenz to niezwykle rozbudowany, otwarty i dostępny bez opłat licencyjnych silnik gier, który można pobrać z internetu w postaci pakietu SDK. Będącą jego właścicielem firma pobiera opłaty licencyjne za publikowanie treści w jej sieci (<https://turbulenz.com/#>). Turbulenz to najbardziej rozwinięty z wszystkich opisywanych interfejsów API, zawiera bardzo dużą liczbę klas, ale jest też trudny do opanowania. To produkt zdecydowanie przeznaczony dla doświadczonych programistów. Kliencka część biblioteki ma otwarty kod źródłowy, a pozostałe części systemu (serwer, wirtualna gospodarka itd.) zarezerwowano w celu czerpania zysków.

Goo Engine (<http://www.gootechnologies.com/>)

Niedawno firma Goo Technologies udostępniła wybranemu gronu wersję beta swojego silnika i narzędzie do tworzenia treści. Dodatkowo firma udostępnia łatwy w obsłudze interfejs do tworzenia treści przeznaczony dla twórców zwykłych stron. Aktualnie produkt Goo Technologies nie ma otwartego kodu źródłowego.

Verold (<http://www.verold.com/>)

Lekka platforma do publikowania interaktywnej treści trójwymiarowej utworzona przez firmę Verold Inc. z Toronto. Na stronach firmy czytamy, że jest to „bezwtyczkowy, rozszerzalny system z obsługą prostego kodu JavaScript, przeznaczony dla hobbystów, studentów, nauczycieli, specjalistów od komunikacji wizualnej i specjalistów od marketingu internetowego do osadzania animowanej treści trójwymiarowej na stronach internetowych”.

Podobnie jak Goo, produkt Verold jest skierowany do szerokiego grona odbiorców niebędących specjalistami grafiki i ma uproszczony interfejs do skomplikowanego silnika gier. Aktualnie Verold nie ma otwartego kodu źródłowego.

Babylon.js (<http://www.babylonjs.com/>)

Prywatny projekt pracownika firmy Microsoft o nazwisku David Catuhe. Jest to łatwy w obsłudze silnik, który pod względem funkcjonalności i walorów użytkowych można sklasyfikować gdzieś między biblioteką Three.js a najbardziej zaawansowanymi silnikami gier. Na stronie demonstracyjnej znajduje się wiele przykładowych aplikacji, od kosmicznych strzelanek po pokazy architektury.

KickJS (<http://www.kickjs.org/>)

Silnik gier i biblioteka renderingu o otwartym kodzie źródłowym autorstwa Mortena Nobla-Jørgensena. Jest to projekt, który wyrósł z pracy akademickiej autora. KickJS jest słabiej rozwinięty i ma społeczność działającą mniej preźnie niż wcześniej opisane produkty. Uwzględniałem go w tym spisie, ponieważ w jego budowie zastosowano najwięcej nowoczesnych, najlepszych technik programistycznych.

Systemy do tworzenia prezentacji

Popyt na narzędzia do szybkiego tworzenia treści trójwymiarowej sprawił, że powstało kilka eksperymentalnych systemów szkieletowych do budowania prezentacji. W tych produktach, w odróżnieniu od typowych silników, najważniejsza jest szybkość tworzenia treści graficznej i łatwość osadzania jej na stronach internetowych, aby bez problemu można było opracowywać wizualizacje danych i produktów, proste animacje itd.

Voodoo.js (<http://www.voodoojs.com/>)

Zadaniem biblioteki Voodoo.js jest ułatwienie tworzenia treści trójwymiarowej i osadzania jej na stronach internetowych. Biblioteka ma niezwykle prosty interfejs API do dodawania trójwymiarowych modeli do stron — wystarczy wpisać adres URL modelu, identyfikator elementu `<div>` i kilka parametrów konfiguracyjnych. Biblioteka Voodoo.js nadaje się praktycznie tylko do wstawiania modeli na strony internetowe, ale w tej dziedzinie jest naprawdę dobra.

tQuery (<http://jeromeetienne.github.io/tquery/>)

Biblioteka tQuery to produkt Jerome'a Etienne'a, który prowadzi popularny blog LearningThree.js (<http://learningthreejs.com/>). Wzorowana na bibliotece jQuery biblioteka tQuery ma z założenia łączyć „możliwości biblioteki Three.js z walorami użytkowymi interfejsu API biblioteki jQuery”, czyli ma być łatwym w obsłudze interfejsem API do grafu sceny Three.js. Użyto w niej stylu programowania opartego na łączaniu wywołań funkcji w łańcuchy. Obsługuje wysokopoziomowe interaktywne zachowania poprzez wywołania zwrotne. Korzystając z niej, można zaoszczędzić pisania wielu wierszy kodu Three.js. Raczej nie powinno się jej nazywać systemem szkieletowym, ponieważ bardziej przypomina nieinwazyjną bibliotekę pomocniczą w stylu jQuery. Biblioteka tQuery dla programistów szukających sposobów na zmniejszenie ilości kodu może być prawdziwym skarbem.

PhiloGL (<http://www.senchalabs.org/philogl/>)

PhiloGL to eksperymentalny pakiet utworzony przez specjalistę wizualizacji danych Nicolasa Garcię Belmonte podczas pracy w laboratoriach firmy Sencha Inc. Celem PhiloGL jest „sprawienie, aby programowanie WebGL było tak samo przyjemne i łatwe, jak przy

użyciu innych popularnych systemów szkieletowych". Belmonte objął swoje podejście do projektowania w pierwszym wpisie na blogu (<http://bit.ly/sencha-philoGL>). Jego produkt nawet w fazie eksperymentalnej jest godzien uwagi. Firma Sencha Inc. zajmuje się tworzeniem najwyższej klasy szkieletów interfejsu użytkownika, więc jej pracownicy wiedzą co nieco o tworzeniu efektywnych interfejsów użytkownika przy użyciu HTML5. Strona internetowa PhiloGL zawiera kilka przykładów działających aplikacji, np. przeróbki wszystkich kursów ze strony Learning WebGL (<http://www.learningwebgl.com/>).

Vizi (<https://github.com/tparisi/Vizi>)

Vizi to szkieletowy system prezentacyjny mojego autorstwa. Jest owocem kilku lat pracy nad innymi szkieletami i silnikami trójwymiarowymi (np. VRML i X3D). Przy jego budowie użyte zostały najlepsze techniki tworzenia silników gier, a szczegółowo mówiąc, zastosowano architekturę opartą na kompozycji i agregacji zamiast tradycyjnych klas. Celem tego systemu jest umożliwienie szybkiego tworzenia ciekawych aplikacji trójwymiarowych. Podobnie jak Voodoo.js, Vizi umożliwia osadzenie modelu na stronie za pomocą zaledwie kilku wierszy kodu, ale dodatkowo zawiera kompletny wysokopoziomowy interfejs API do obsługi interakcji, animacji i zachowań dowolnych elementów na scenie.

Narzędzia do tworzenia treści trójwymiarowej

Klasyczne programy do modelowania i animowania

Firma Autodesk (<http://www.autodesk.com/>) produkuje pełen asortyment programów do tworzenia modeli i animacji trójwymiarowych. Ich ceny są bardzo wysokie, chociaż od niedawna firma zaczęła też oferować godne uwagi wersje edukacyjne i próbne.

Oprócz profesjonalnych pakietów firmy Autodesk dostępnych jest jeszcze kilka innych przyzwoitych programów w bardziej przystępnych cenach.

Blender (<http://www.blender.org/>)

Darmowy program o otwartym kodzie źródłowym, działający we wszystkich najważniejszych systemach operacyjnych i dostępny na licencji GNU General Public License (GPL). Autorem Blendera jest holenderski programista Ton Roosendaal, a opiekę nad nim sprawuje holenderska fundacja non profit o nazwie Blender Foundation. Program ten jest bardzo popularny. Fundacja szacuje, że ma około dwóch milionów użytkowników. Korzystają z niego artyści i inżynierowie zarówno amatorzy oraz studenci, jak i profesjonalisci.

SketchUp (<http://www.sketchup.com/>)

SketchUp to łatwy w obsłudze program do tworzenia trójwymiarowych modeli. Znajduje zastosowanie w projektowaniu architektury, inżynierii oraz w mniejszym stopniu w tworzeniu gier. Na stronie produktu można znaleźć wersje darmowe i niedrogie wersje profesjonalne.

Poser (<http://poser.smithmicro.com/>)

Poser to narzędzie ze średniej półki do animowania postaci. Podobnie jak SketchUp, jest oferowany w atrakcyjnych cenach i przeznaczony dla mniej wymagających odbiorców. Ma intuicyjny interfejs do projektowania i animowania postaci. Dodatkowo program zawiera bogatą bibliotekę gotowych ludzkich i zwierzęcych postaci oraz zestaw scen, rekwizytów, pojazdów, kamer i konfiguracji oświetlenia. Używa się go do tworzenia realistycznych nieruchomych obrazów i animacji.

Środowiska przeglądarkowe

Wraz z technologiami chmurowymi i technologią WebGL zaczęły pojawiać się nowe rodzaje narzędzi do tworzenia treści: przeglądarkowe trójwymiarowe środowiska zintegrowane. Wymienione poniżej produkty są wciąż w początkowej fazie rozwoju, ale zapowiadają się bardzo obiecująco.

Goo Create (<http://www.gootechnologies.com/>)

Opisanemu wcześniej silnikowi Goo towarzyszy łatwy w obsłudze interfejs do tworzenia treści przeznaczony do użytku ogólnego. Goo Create zawiera kilka gotowych modeli i animacji, na bazie których łatwiej rozpocząć własną pracę.

Verold Studio (<http://www.verold.com/>)

Verold Studio to przeglądarkowe narzędzie do tworzenia treści trójwymiarowej i środowisko programistyczne towarzyszące opisanemu wcześniej silnikowi gier Verold.

Sketchfab (<http://sketchfab.com/>)

Sketchfab to usługa sieciowa do publikowania i udostępniania na bieżąco w internecie interaktywnych trójwymiarowych modeli bez używania jakiekolwiek wtyczki. Wykonując kilka kliknięć, autor może wysłać trójwymiarowy model w jednym z kilku formatów na stronę internetową, aby następnie pobrać kod HTML potrzebny do osadzenia tego modelu na swoich stronach.

SculptGL (<https://github.com/stephomi/sculptgl>)

Darmowe i otwarte sieciowe narzędzie do modelowania wyposażone w łatwy w obsłudze interfejs do tworzenia prostych modeli rzeźb. Istnieje możliwość eksportowania prac do różnych formatów oraz bezpośredniego ich opublikowania w Verold i Sketchfab.

Animacyjne systemy szkieletowe

Dzisiejsze aplikacje do animowania treści powinny używać funkcji `requestAnimationFrame()`. Aby zapewnić jej spójną obsługę we wszystkich przeglądarkach, Paul Irish utworzył świetny wypelniacz (<http://paulirish.com/2011/requestanimationframe-for-smart-animating>).

Do budowania prostych animacji z klatkami pośrednimi wystarczy popularna otwarta biblioteka Tween.js (<https://github.com/sole/tween.js>) autorstwa Soledad Penadés.

Do tworzenia animacji opartej na klatkach kluczowych można użyć klas wbudowanych biblioteki Three.js i paru dodatkowych, dostępnych w przykładach dołączonych do tego produktu. Dziedzina ta z pewnością będzie dynamicznie się zmieniać, gdy więcej narzędzi pojawi się w internecie i dojrzeją bardziej przyjazne sieci formaty danych, takie jak glTF.

Diagnozowanie i profilowanie aplikacji WebGL

W nowych wersjach przeglądarek dostępne są rozmaite narzędzia do diagnozowania i profilowania aplikacji WebGL. Architekt graficzny z firmy AGI Patrick Cozzi (twórca wirtualnego globu i silnika map WebGL o nazwie Cesium) opublikował na stronie <http://www.realtimerendering.com/blog/webgl-debugging-and-profiling-tools/> świetny przegląd przeglądarkowych narzędzi pomocnych w pracy z WebGL.

Programowanie aplikacji trójwymiarowych dla urządzeń przenośnych

Kluczowe przy tworzeniu dobrych trójwymiarowych aplikacji mobilnych jest dodanie obsługi zdarzeń dotykowych. Specyfikację przeglądarkowych zdarzeń dotyku można znaleźć wśród rekomendacji konsorcjum W3C na stronie <http://www.w3.org/TR/2013/REC-touch-events-20131010/>.

W portalu dla programistów aplikacji dla systemu Android znajdują się wyczerpujące informacje na temat tworzenia aplikacji sieciowych przy użyciu technologii HTML5 — <http://developer.android.com/guide/webapps/index.html>.

Firma Amazon udostępnia rozbudowany system do publikowania aplikacji sieciowych, w ramach którego dostarcza aplikację Web App Tester dla bazującego na Androidzie systemu operacyjnego Kindle Fire OS (<https://developer.amazon.com/sdk/webapps/tester.html>) oraz utrzymuje portal dystrybucyjny do pakowania i rozpowszechniania aplikacji (<https://developer.amazon.com/sdk/webapps/manifest.html>).

W środowiskach standardowo nieobsługujących technologii WebGL, np. w systemie operacyjnym iOS, dostępne są rozwiązania hybrydowe do tworzenia aplikacji łączących HTML5 i JavaScript z kodem macierzystym. PhoneGap (<http://phonegap.com/>) firmy Adobe jest wiodącym produktem wśród bibliotek hybrydowych, ale aktualnie nie obsługuje WebGL. Aby korzystać z tej technologii w systemie iOS, należy użyć jednego z hybrydowych systemów szkieletowych. Oto one.

CocoonJS (<http://www.ludei.com/tech/cocoonjs>)

CocoonJS działa w systemach Android i iOS. Ukrywa szczegóły podstawowego systemu operacyjnego w łatwym w użyciu kontenerze aplikacji na kod HTML5 i JavaScript. Dostarcza implementacje kanwy, WebGL, Web Audio, WebSockets i wiele innych. Ponadto biblioteka CocoonJS zawiera system do budowy projektów działających w chmurze, więc wystarczy tylko podpisać swój projekt i go zbudować. Programista nie musi uczyć się obsługi specjalnych narzędzi platform, takich jak Xcode dla systemu iOS. CocoonJS to projekt o zaknietym kodzie źródłowym, nad którym pieczę trzyma firma Ludei z San Francisco.

Ejecta (<http://impactjs.com/ejecta>)

Ejecta to biblioteka typu open source o funkcjonalności bardzo podobnej do CocoonJS, ale działającą tylko w systemie iOS. Zrodziła się z projektu ImpactJS, czyli silnika do tworzenia gier w HTML5. Obsługa Ejecty jest nieco bardziej skomplikowana niż CocoonJS i wymaga znajomości środowiska Xcode oraz macierzystych interfejsów API platformy.

Specyfikacje trójwymiarowych formatów plików

Trójwymiarowe formaty plików można podzielić na trzy kategorie: formaty służące do reprezentowania pojedynczych obiektów modeli, formaty do przechowywania pojedynczych animacji oraz formaty do przechowywania całych scen zawierających wiele modeli, hierarchii przekształceń, kamer, światel i animacji. Istnieje wiele trójwymiarowych formatów plików, zbyt wiele, by opisać tu wszystkie.

Poniżej znajduje się lista formatów, które najlepiej nadają się do użytku w aplikacjach sieciowych.

Formaty do przechowywania modeli

- Wavefront OBJ (http://en.wikipedia.org/wiki/Wavefront_.obj_file),
- STL — trójwymiarowy tekstowy format plików na potrzeby drukowania (http://en.wikipedia.org/wiki/STL_%28file_format%29).

Formaty do przechowywania animacji

- Formaty do przechowywania animacji postaci MD2 (<http://tfc.duke.free.fr/coding/md2-specs-en.html>) i MD5 (<http://tfc.duke.free.fr/coding/md5-specs-en.html>) firmy id Software,
- Format animacyjny BVH firmy BioVision do przechwytywania ruchu (<http://research.cs.wisc.edu/graphics/Courses/cs-838-1999/Jeff/BVH.html>).

Formaty do przechowywania kompletnych scen

- VRML (<http://bit.ly/web3d-vrml>) i X3D (<http://bit.ly/web3d-x3d>) — pierwsze formaty trójwymiarowe do użytku w sieci,
- COLLADA — format do wymiany zasobów cyfrowych (http://www.khronos.org/files/collada_spec_1_4.pdf),
- glTF (ang. *Graphics Library Transmission Format*) — <http://gltf.gl/>.

Inne powiązane technologie

Trójwymiarowe aplikacje nie powstają w próżni. Istnieją jeszcze inne ciekawe technologie sieciowe, którymi czasami warto wzbogacić swoje projekty. Oto kilka z nich.

Interfejs API Pointer Lock

W pełnoekranowych trójwymiarowych aplikacjach, takich jak gry, czasami potrzebna jest większa kontrola nad zdarzeniami myszy niż w zwykłych aplikacjach. Dlatego twórcy przeglądarki wprowadzili interfejs API o nazwie Pointer Lock umożliwiający ukrywanie kursora i używanie niskopoziomowych zdarzeń ruchu myszy w sposób wymagany podczas programowania gier.

Dobre wprowadzenie do interfejsu API Pointer Lock napisał John McCutchan z Google. Artykuł znajduje się pod adresem <http://html5rocks.com/en/tutorials/pointerlock/intro/>.

Aktualną specyfikację W3C API Pointer Lock można znaleźć pod adresem <http://www.w3.org/TR/pointerlock/>.

Interfejs API Page Visibility

Trójwymiarowe aplikacje wyświetlające 60 klatek na sekundę mogą zająć znaczną część cyklu procesora. A przecież jeśli karta z taką aplikacją nie jest wyświetlana, to nie ma sensu renderować sceny. Poza tym czasami aplikacja może wymagać kontynuacji pewnych obliczeń, ale ze znacznie mniejszą intensywnością. Dlatego w najnowszych przeglądarkach wprowadzono

nowy interfejs API o nazwie Page Visibility umożliwiający programistom sprawdzenie, kiedy strona lub karta jest niewidoczna, i podjęcie w związku z tym odpowiednich czynności mających na celu oszczędzenie zasobów sprzętowych.

Dobre wprowadzenie do technologii Page Visibility znajduje się w serwisie dla programistów firmy Google na stronie <https://developers.google.com/chrome/whitepapers/pagevisibility>.

Aktualna specyfikacja interfejsu znajduje się w serwisie W3C pod adresem <http://www.w3.org/TR/page-visibility/>.

Technologie WebSockets i WebRTC

Programiści trójwymiarowych gier dla wielu graczy, wirtualnych światów i aplikacji do współpracy na bieżąco muszą implementować mechanizmy komunikacji między klientem i serwerem. Służą do tego dwie technologie, WebSockets i WebRTC.

WebSockets (albo inaczej specyfikacja WebSocket) to standardowa przeglądarkowa implementacja protokołu TCP/IP. Technologii tej można używać do obsługi dwukierunkowej komunikacji między serwerem i klientem. Twórcy protokołu TCP/IP nie planowali możliwości obsługi komunikacji na bieżąco i dlatego lepszym rozwiązaniem może być technologia WebRTC (opisana jako następna). Wszystko jednak zależy od potrzeb w konkretnym przypadku. Na stronie <http://code.tutsplus.com/tutorials/start-using-html5-websockets-today--net-13270> znajduje się samouczek technologii WebSockets. Dodatkowych informacji można też poszukać na stronie głównej projektu <http://www.websocket.org/>.

WebRTC to standard błyskawicznego przesyłania wiadomości między klientem a serwerem sieciowym. Technologia ta może być lepsza do obsługi wielu użytkowników niż WebSocket, ponieważ powstała właśnie z myślą o wymianie wiadomości na bieżąco. Na stronie <http://www.html5rocks.com/en/tutorials/webrtc/basics/> znajduje się krótkie wprowadzenie. Główną stronę projektu utrzymuje firma Google pod adresem <http://www.webrtc.org/>, a aktualna rekomendacja W3C jest dostępna na stronie <http://www.w3.org/TR/webrtc/>.

Web Workers

Technologia Web Workers (<http://www.w3.org/TR/workers>) pomaga w programowaniu wielowątkowym w języku JavaScript. W aplikacjach trójwymiarowych można ją wykorzystać do wykonywania w tle wybranych zadań, np. ładowania modeli albo przeprowadzania symulacji fizycznych. W ten sposób można zapewnić płynne działanie interfejsu użytkownika nawet wtedy, gdy aplikacja wykonuje jakieś skomplikowane obliczenia.

Z używaniem technologii Web Workers wiążą się pewne drobne trudności, takie jak np. kłopoty z przekazywaniem obiektów pamięci między wątkami. Szczegółowy opis tych zagadnień znajduje się w świetnym artykule opublikowanym w serwisie HTML5 Rocks pod adresem <http://updates.html5rocks.com/2011/12/Transferable-Objects-Lightning-Fast>.

IndexedDB i API systemu plików

Pliki trójwymiarowe bywają bardzo duże i dlatego warto wypróbować nowe technologie HTML5, które mogą pomóc w zaoszczędzeniu na transferze poprzez zapisanie części danych na dysku użytkownika. Na pamięci podręcznej przeglądarki nie można polegać, ponieważ

jest za mała i nie można jej kontrolować z poziomu aplikacji. Zawartość pamięci podręcznej może zostać w każdej chwili skasowana przez użytkownika lub „wypchnięta” przez nowsze dane.

Ray Camden, propagator technik programowania z firmy Adobe i jeden z recenzentów tej książki, wpadł na pomysł, aby użyć IndexedDB, czyli przeglądarkowego interfejsu API do bazy danych, w celu przechowywania danych. Napisał na ten temat artykuł w odniesieniu do tworzenia bogatych aplikacji SVG (<http://bit.ly/camden-richSVG>). Specyfikacja IndexedDB znajduje się pod adresem <http://www.w3.org/TR/IndexedDB/>.

Jednak IndexedDB to nie system plików, tylko interfejs API do bazy danych. Jeśli potrzebna jest możliwość zapisywania treści na dysku użytkownika i pobierania jej przy użyciu interfejsu API w stylu systemu plików, można użyć eksperymentalnego API o nazwie FileSystem (<http://www.w3.org/TR/file-system-api/>). Przy jego użyciu można tworzyć aplikacje odczytujące i zapisujące pliki w hierarchii folderów na dysku użytkownika. W serwisie HTML5 Rocks znajduje się świetny artykuł na ten temat (<http://www.html5rocks.com/en/tutorials/file/filesystem/>). Na razie interfejs API FileSystem jest obsługiwany tylko przez przeglądarki Chrome i Opera. Ponadto nie należy go mylić z API File (<http://www.w3.org/TR/FileAPI/>), który daje możliwość tylko odczytu plików z lokalnego systemu plików.

Skorowidz

3 Dreams of Black, 53
3D Systems, 192
3D Warehouse, 189
3DRT, 190, 316
3ds Max, 73, 111, 182

A

albedo, 80
Amazon Silk, 21
analiza ukształtowania terenu, 293
Android, 301, 302, 311
animacja, 19, 27, 45, 78, 99, 179,
192, 201, 252, 286
CSS, 133, 147
czasowa, 104
funkcja prędkości, *Patrz*:
funkcja prędkości animacji
klatkowa, 103
mimiki, 100
obiektów połączonych, 113
oparta na
celach morfingu, 100, 118
klatkach kluczowych, 100,
110, 111, 113, 179, 190
krzywych sklejanych, 117
shaderze, 100, 125
po linii ścieżek, 100, 116, 118
postaci, 100, 182, 190, 214
przyspieszanie, 107
szkieletowa, *Patrz też*: skinning
zwalnianie, 107
animator, 179
antialiasing, 165
wielopróbkowy, 61
API
Canvas 2D, 23, 157, 160, 164,
299, 302
graficzne, 31
JavaScript, 20
OpenGL, 19

Three.js, 57
WebGL aplikacja mobilna, 302
aplikacja
hybrydowa, 313, 322
manifest, 312
mobilna, 302, 303, 322
prezentacyjna, 220
sieciowa, 177, 303
dystrybucja, 312
mobilna, 303
tworzenie, 311
wizualna, 223
trójwymiarowa, 213
testowanie, 238
tworzenie, 233, 234, 235
Arnaud Remi, 198
articulated animation, *Patrz*:
animacja obiektów połączonych
artysta techniczny, 180
asm.js, 20
ATI, 22
awatar, 214, 216
odległość od podłożu, 293

B

Babylon.js, 219
backface, *Patrz*: ściana tylna
Belmonte Nicolas Garcia, 222
Béziera krzywa, 117
biblioteka, 214
adaptacyjna dla aplikacji
hybrydowych, 314
Cango3D, 167
CocoonJS, *Patrz*: CocoonJS
CubeGeometry, 61
DirectX, 33
grafiki trójwymiarowej, 67
K3D, 167, 168
macierzysta, 313
Nihilogic, 167
OpenGL, *Patrz*: OpenGL

Three.js, *Patrz*: Three.js
trójwymiarowa, 214
Tween.js, *Patrz*: Tween.js
WebGL, *Patrz*: WebGL
Biovision Hierarchical Data,
Patrz: plik BVH
BlackBerry 10, 21, 302
blend weight, *Patrz*: waga mieszania
Blender, 73, 111, 182, 183
blokowanie
kursora, 283
myszy, 283
bone, *Patrz*: kość
bounding volume, *Patrz*: bryła
brzegowa
bryła, 41
brzegowa, 73
B-spline, *Patrz*: krzywa B-sklejana
bufor, 36, 200
głębi, 40, 96, 165
indeksów, 43
kolorów, 40, 165
widok, 200
współrzędnych teksturowych, 49
z, 165
bufor głębi, 166
BVH Motion Creator, 193

C

Cabello Ricardo, 56, 155
Canvas 2D, 20, 157, 160, 164, 299, 302
Canvas 3D, 32
Catmull Ed, 117
Catmulla-Roma krzywa, 117
cel morfingu, 100, 118, 119
Chrome, 21, 302
cieniowanie, 87
bez oświetlenia, 91
Blinna, 212
Fresnela, 91

- cieniowanie
Lamberta, 80, 91
Phonga, 65, 80, 91, 228
trójkątów, 165
- cień, 87
intensywność, 90
renderowanie, 87
- CocoonJS, 314, 316, 318, 320
- CocoonJS Launcher, 314
- Codrops, 151
- collision detection, *Patrz*: kolizja wykrywanie
- collision response, *Patrz*: kolizja reakcja
- Core Animation, 131
- Cozzi Patrick, 198
- CSS, 23
animacja, *Patrz*: animacja CSS
filtry własne, 28, 153, 154
przejście, 133, 143
przekształcenie, *Patrz*: przekształcenie CSS
shader, 153
- CSS Custom Filters, 28, 153
- CSS Shaders, 153
- CSS transitions, *Patrz*: CSS przejście
- CSS3, 19, 131, 132, 215
- ## D
- DAG, 75
- Danger Mouse, 53
- DCC, 181
- deferred rendering, *Patrz*: rendering opóźniony
- Denoyel Alban, 187
- Despoulain Thibaut, 55
- diffuse color, *Patrz*: kolor rozproszony
- digital content creation tools, *Patrz*: DCC
- directed acyclic graph, *Patrz*: DAG
- directional light, *Patrz*: oświetlenie kierunkowe
- drzewo, 75
- dynatree, 264
- dyrektor techniczny, 180
- dźwięk, 294, 318
- ## E
- easing, *Patrz*: funkcja prędkości animacji
- efekt Fresnela, 92
- efekty dźwiękowe, 258, 294
- Ejecta, 314
- ekran HUD, 318
- ## F
- ekstruzja, 68
- element
canvas, 35
DOM Image, 46
- Emscripten, 20
- Etienne Jerome, 220
- Eulera kąt, *Patrz*: kąt Eulera
- ## G
- figura dwuwymiarowa, 69
- filtr
biliniarny, 165
własny, 29
- Firefox, 21, 302
- first-person navigation, *Patrz*: nawigacja pierwszoosobowa
- first-person perspective, *Patrz*: perspektywa pierwszej osoby
- first-person shooter, *Patrz*: FPS
- format
animacyjny, 190, 192
binarny, 207
do przechowywania całych scen, 190, 193
modelowy, 190, 192
pliku, *Patrz*: plik
- FPS, 280, 283
- frame, *Patrz*; klatka
- Fresnela
cieniowanie, *Patrz*: cieniowanie Fresnela
efekt, *Patrz*: efekt Fresnela
shader, *Patrz*: shader Fresnela
- frustum, 27
- funkcja
animate, 78, 104
computeFaceNormals, 73
document.createElement, 158
document.getElementById, 35
dwuwymiarowego API kanwy, 160
ImageUtils.loadTextureCube, 83
prędkości animacji, 109, 110
reflect, 94
refract, 94
requestAnimationFrame, 20, 62, 100, 101, 102, 103
setInterval, 20, 101
setTimeout, 20, 101, 102
sklejana, 111
- ## H
- geometria
statyczna, 73
THREE.BufferGeometry, 73
- Ginier Stephane, 187
- GL Shading Language, *Patrz*: GLSL
- GLSL, 91
- GLSL ES, 28
- Goo Engine, 218
- GPU, 27
- gra FPS, *Patrz*: FPS
- graf
acykliczny skierowany, *Patrz*: DAG
sceny, *Patrz*: scena graf
- grafika trójwymiarowa, 22
tworzenie, 177
- Graphics Library Transmission Format, 198
- Gunning Brent, 220
- ## I
- heads-up display, *Patrz*: ekran HUD
- HexGL, 55
- hierarchia przekształceń, 75, 138, 285
- ## J
- interaktywność, 228
- Internet Explorer 11, 21, 302
- interpolacja, 106
bazująca na funkcji sklejanej, 111
krzywej sklejanej, 116
liniowa, 106, 109, 111
nielinowa, 107
- interprocess communication, *Patrz*: komunikacja międzyprocesowa
- iOS, 301, 302
- IPC, *Patrz*: komunikacja międzyprocesowa
- Irish Paul, 102
- ## J
- język
CSS, 131
GLSL, 38, 154, 188
HTML, 131
Python, 183
VRML, 194
X3D, 194
- Jones Nora, 53
- jQuery, 131, 215
- JSFiddle, 188

K

kamera, 26, 27, 37, 61, 153, 200, 261
dodatkowa, 258, 284
domyslna, 214, 240
kontroler, 281
 modelowy, 281
 pierwszoosobowy, 281,
 283, 292
odlegośc od podłoża, 293
przełączanie, 216
kanal alfa, 152
kanwa, 19, 215, 320
 dwuwymiarowa, 157, 164
 kontekst, 158
kąt Eulera, 79
Keyframe.js, 111
Kronos Group, 32
klasa
 bazowa
 THREE.CircleGeometry, 70,
 71
 THREE.Geometry, 69
 ExtrudeGeometry, 68
 MeshBasicMaterial, 79, 85
 MeshLambertMaterial, 79, 85
 MeshPhongMaterial, 79, 85
 oświetleniowa, 84
 Path, 68, 69
 Shape, 68, 69
 THREE.BufferGeometry, 73
 THREE.EffectComposer, 97
 THREE.Object3D, 75, 76, 78
 THREE.OrbitControls, 240
 THREE.ShaderMaterial, 91, 125
 THREE.WebGLRenderTarget, 97
translate, 135
Vizi.Application, 240
Vizi.FadeBehavior, 249, 250
Vizi.FirstPersonControllerScript,
 283
Vizi.Loader, 241, 242
Vizi.Object, 288
Vizi.Picker, 251, 308
Vizi.RotateBehavior, 251
Vizi.Script, 288
Vizi.Viewer, 240, 281
klatka, 103
 kluczowa, 100, 110, 113, 179, 190
 pośrednia, *Patrz:* tweening
 szybkość zmiany, 103
kolizja, 153, 216
 reakcja, 292
 wykrywanie, 283, 284, 292
kolor
 odbicia, 81
 rozproszenia, 81
 rozproszony, 85

L

Lightwave, 259
Luppi Daniel, 53

L

macierz
 model-widok, 37
 projekcji, 26
 przekształceń, 26
mapa
 nierówności, 81
 normalnych, 81
 środowiskowa, 83, 212
 teksturowa, *Patrz:* tekstura
mapowanie
 cieni, 87, 90
 tekstura, *Patrz:* teksturowanie
 UV, *Patrz:* teksturowanie
maszyna wirtualna JavaScript, 20,
 314

materiał, 25, 61, 79, 200
 kolor rozproszenia, *Patrz:* kolor
 rozproszenia
 Phong, 65
 własność, 85
 ambient, 85
 emissive, 85
Maya, 73, 111, 182
 eksport do COLLADA, 237
McKegney Ross, 186
mesh, *Patrz:* siatka
metoda
 forwardAsync, 321
 generowania ścieżek, 69
 getContext, 35
 lineTo, 69

M

moveTo, 69
restore, 299
save, 299
setTimeout, 286
międzyklatka, 100, 106
 wstawianie, 106
Milk Chris, 53
Minesweeper, 107
mipmapowanie, 48, 165
Miyazaki Aki, 193
model, 24
modelarz, 178
modelowanie trójwymiarowe, 178
ModelView matrix, *Patrz:* macierz
 model-widok
morphing, 100, 118, 119, 124
morph target, *Patrz:* cel morfingu
morph target animation,
 Patrz: animacja oparta
 na celach morfingu
motion capture, *Patrz:*
 przechwytywanie ruchu
MotionBuilder, 182
mouse lock, *Patrz:* blokowanie
 myszy
mouse look, *Patrz:* patrzenie myszą
Mr.doob, *Patrz:* Cabello Ricardo
MSAA, *Patrz:*
 antialiasing:wielopróbkowy
multipass rendering, *Patrz:*
 rendering wieloprzebiegowy
multisample antialiasing, *Patrz:*
 antialiasing wielopróbkowy
multiteksturowanie, 81, 126
multitouch, *Patrz:* wielodotyk

N

nawigacja pierwszoosobowa, 279,
 281, 283
Nobel-Jrgensen Morten, 219
normalna, 73
 krzywej, 117
 mapa, *Patrz:* mapa normalnych
 przemieszczanie, 81
NVIDIA, 22

O

O'Callahan Robert, 101
obiekt
 animowanie, 99
 po linii ścieżek, 100
 modyfikowanie właściwości, 99
 podstawowy, 36, 40
 THREE, 61

objętość widokowa, 27
obszar widoku, 26, 36
OpenCOLLADA, 237, 238
OpenGL, 33
OpenGL ES Shader Language,
 Patrz: GLSL ES
ostrosłup ścięty, 27
oświetlenie, 25, 64, 80, 153, 200
 brak, 80
 gotowe, 80
 kierunkowe, 65, 84
 otaczające, 84, 85
 punktowe, 84, 85
 reflektorowe, 84, 85, 90
 własność
 color, 85
 intensity, 85
OutsideOfSociety, 120

P

Passet Pierre-Antoine, 187
patrzenie myszą, 283
Penadés Soledad, 107
Penner Robert, 110
perspektywa, 26
 pierwszej osoby, 280
Pesce Mark, 194
pętla wykonawcza, 62, 99, 216
PhiloGL, 222, 223
PhoneGap, 314
Phong Bui Tuong, 65
Pinson Cédric, 187
PixelCG Tips and Tricks, 68
PlayCanvas, 218
plik
 .bin, 207
 .dae, 114
 .obj, 74, 190, 198
 BVH, 192, 193
 COLLADA, 114, 184, 195, 208,
 237, 261
 konwertowanie na glTF, 237
 wczytywanie, 115
FBX, 182, 201
glTF, 198, 200, 211, 237, 261
JPEG, 46, 47
JSON, 57, 202, 208
manifestu, 312
MD2, 120, 192
MD5, 192
MTL, 74
PNG, 46, 152
STL, 192
VRML, 194
Wavefront OBJ, 74, 190
X3D, 194

Plus 360 Degrees, 54
płaszczyzna odcięcia, 27
podkładka, 102
pointer lock, *Patrz*: blokowanie
 kursora
polyfill, *Patrz*: podkładka
Poser, 184
post-processing, *Patrz*:
 przetwarzanie końcowe
prefabrykat, 274
primitive, *Patrz*: obiekt
 podstawowy
procedura wywoływanie zdalne,
 Patrz: RPC
procedural texture, *Patrz*: tekstura
 proceduralna
procesor graficzny, *Patrz*: GPU
program
 cieniąjący, *Patrz*: shader
 narzędzie do tworzenia cyfrowej
 treści, *Patrz*: DCC
projection matrix, *Patrz*: macierz
 projekcji
przechwytywanie ruchu, 193
przeglądarka mobilna, 302
przeglądarkowe środowisko
 zintegrowane, 185
przekształcenie, 25
 CSS, 133, 134, 135, 143
 trójwymiarowe, 21
 hierarchia, *Patrz*: hierarchia
 przekształceń
 obrót, 79, 136
 perspektywa, 136
 przesunięcie, 78
 skala, 78
przepływ sterowania, 214
przetwarzanie końcowe, 96, 97, 180
przezroczystość, 249
pudło nieba, 84, 258, 272

Q

Qualcomm, 22

R

Rails, 215
rama TNB, 117
refleksy, 80, 81
remote procedur call, *Patrz*: RPC
renderer, 61
rendering, 96, 213
 kanwowy, 169, 171, 172
 opóźniony, 97
 programowy, 165

przetwarzanie końcowe, *Patrz*:
 przetwarzanie końcowe
trójwymiarowy, 19, 151, 152
wieloprzebiegowy, 96, 97
Renderosity, 189
repozytorium 3D, 188
rig, *Patrz*: rusztowanie
RO.ME, 53, 54
Roast Kevin, 168
Robinet Fabrice, 198
Roosendaal Ton, 183
RPC, 322
Russell Kenneth, 32
rusztowanie, 179

S

Safari, 21, 302
scena, 61
 graf, 75, 215, 260, 261
 korzeń, 75
 struktura, 75
scene graph, *Patrz*: scena graf
SculptGL, 187
Second Life, 9
Sencha Inc., 222
serwer sieciowy, 47
shader, 27, 28, 38, 79, 100, 124, 180,
 188
 cube, 84
 fragmentów, 38, 93, 95
Fresnela, 92
GLSL, 91
glebi pola, 54
kreskówkowy, 54
pixeli, *Patrz*: shader
 fragmentów
pomocniczy, 84
programowalny, 91
tekstura, 50
tworzenie, 38, 39, 79
wierzchołków, 38, 93, 94
ShaderFusion, 180
Shadertoy, 188
shadow mapping, *Patrz*:
 mapowanie cieni
Shockwave 3D, 9
siatka, 24, 61, 200
 importowanie, 74
 powierzchnia, 24
 trójwymiarowa, 24, 25, 37,
 Patrz też: model
Silicon Graphics Open Inventor,
 194
silnik
 Gecko, 158
gier, 217, 223

Unity, 180, 274
Unreal, *Patrz*: Unreal
single mesh animation, 121
skeletal animation, *Patrz*: animacja szkieletowa, skinning
skeleton, *Patrz*: szkielet
Sketchfab, 187
SketchUp, 184
skinning, 100, 121, 124, 179
skybox, *Patrz*: pudełko nieba
spline curve, *Patrz*: krzywa sklejana, *Patrz*: krzywa składana
Swappz Interactive, 186
system
 operacyjny
 Android, *Patrz*: Android
 iOS, *Patrz*: iOS
 Rails, *Patrz*: Rails
 szkieletowy, 214, 220, 258
 dane wejściowe, 216
 interfejs użytkownika, 222
 model nawigacji, 216
 prezentacyjny, 220
 przeglądarka, 216
 rozszerzalność, 214
 środowisko, 215
 Zend, *Patrz*: Zend
szkielet, 100, 121
trójwymiarowy, 214

S

ściana tylna, 140
środowisko
 trójwymiarowe, 257
 testowanie, 258, 260, 261,
 269, 293
 tworzenie, 258, 259, 293
światło, *Patrz*: oświetlenie

T

tablica
 liczb JavaScript, 73
 typowana, 37, 73
TC Chang, 234, 259
TCP/IP, 20
technical artist, *Patrz*: artysta techniczny
technical director, *Patrz*: dyrektor techniczny
tekstura, 25, 46, 62, 72, 79
 dynamiczna, 258, 296
 filtrowanie, 48, 165
 generowana programowo, 166
 proceduralna, 296

sześcienna, 83
wielokrotna, *Patrz*: multiteksturowanie
współrzędne, 49, 72
teksturowanie, 178
terrain following, *Patrz*: analiza ukształtowania terenu
Three.js, 53, 54, 55, 56, 155, 167,
 171, 177, 193, 202, 207, 208, 213
 folder
 build, 58
 docs, 58
 editor, 58
 examples, 59
 src, 59
 utils, 59
 instalowanie, 58
 przekształcenia, *Patrz*: przekształcenie
 przekształcenie
 renderer, 61, 169
 renderowanie, 67
 scena trójwymiarowa, 57
 struktura projektu, 58
time-based animation, *Patrz*: animacja czasowa
Tizen, 21, 302
tło, 73
TNB frame, *Patrz*: rama TNB
jQuery, 220, 223
transform, *Patrz*: przekształcenie
transform hierarchy, *Patrz*: hierarchia przekształceń
Trimble, 189
trójkąt
 cieniowanie, 165
 pas, 36, 37
 przekształcanie, 165
 sortowanie, 165
 tablica, *Patrz*: trójkąt zestaw
 zestaw, 36
TurboSquid, 189, 259
Turbulenz, 218
tween, *Patrz*: międzyklatka
Tween.js, 107, 109, 111
tweening, 106, 111
typ geometryczny, 67
typed array, *Patrz*: tablica typowana

U

układ współrzędnych, 22, 23, 133
Unreal, 20
urządzenie przenośne, 301, 303
usuwanie nieużytków, 323

V

Verold Studio, 186
view volume, *Patrz*: objętość widokowa
viewport, *Patrz*: obszar widoku
Virtual Reality Markup Language,
 Patrz: język VRML
Vizi, 223, 239, 258
 animacja, 244, 252, 286
 architektura, 224
 interaktywność, 228, 252, 254
 obsługa, 226
 zmianianie kolorów, 254
Voodoo.js, 220
VRML, 9
Vukićević Vladimir, 32

W

W3C, 20
waga mieszania, 122
Web App Tester, 311
web apps, *Patrz*: aplikacja: sieciowa
Web Workers, 20
WebGL, 19, 20, 21, 23, 31, 34, 56,
 172, 177, 198, 201, 215
 kontekst, 35, 61, 158
 oświetlenie, 86
 podstawy, 32
 silnik gier, 218
 tekstura, 62
 zabezpieczenia, 47
WebSockets, 20
wektor
 normalny, *Patrz*: normalna
 position, 78
 rotation, 78
 scale, 78
 White Jack, 53
 widok bufora, 200
 widżet dynatree, 264
 wielodotyk, 306
 wielowątkowość, 20
 wierzchołek, 24
 Windows RT, 302
 World Wide Web Consortium,
 Patrz: W3C
współrzędne UV, 72, *Patrz* też:
 tekstura współrzędne
wywołanie zwrotne, 214

Z

zachowanie domyślne, 214
z-buffer, *Patrz*: bufor głębi
zdarzenie dotykowe, 304, 305
Zend, 215