O'REILLY®

Aplikacje 3D

Przewodnik po HTML5, WebGL i CSS3

TWÓJ PRZEWODNIK PO GRAFICE 3D W HTML!

HELION

Tony Parisi

Tytuł oryginału: Programming 3D Applications with HTML5 and WebGL

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-246-9668-0

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

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/apli3d.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/apli3d
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 4.

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 przekształceniami, materiały, tekstury i światła, generowane na bieżąco cienie, programowalne shadery oraz elastyczny system renderingu umożliwiający stosowanie technik wielopowtórzeniowych i opóźnień w celu uzyskania zaawansowanych efektów specjalnych.

Geometria i siatki

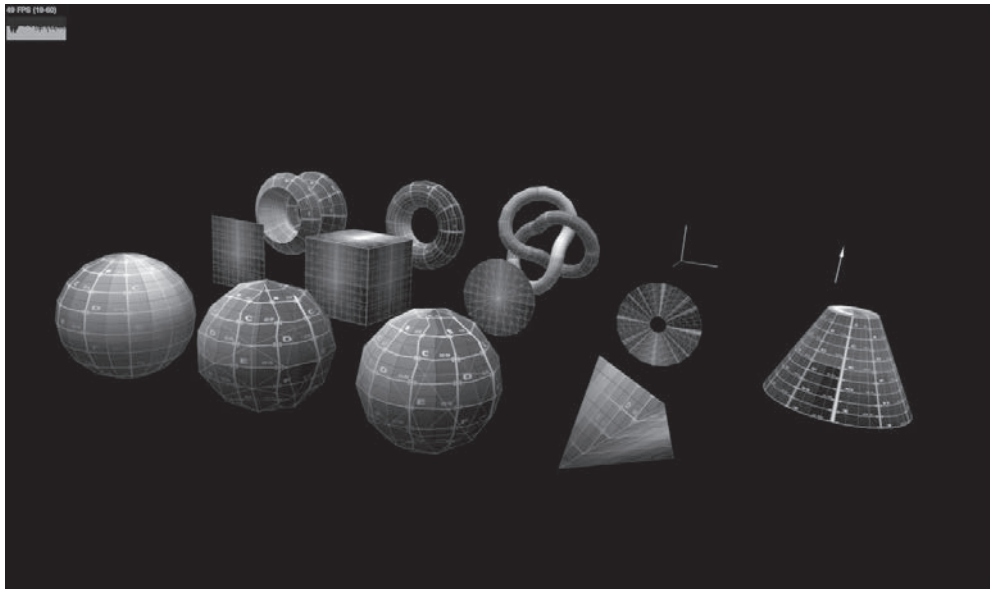
Jedną z największych zalet posługiwania się biblioteką Three.js w porównaniu z używaniem wprost API WebGL jest oszczędność pracy potrzebnej do tego, by utworzyć i narysować figury geometryczne. Przypomnij sobie całe strony kodu z rozdziału 2., napisane w celu utworzenia danych kształtów i tekstur dla 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.

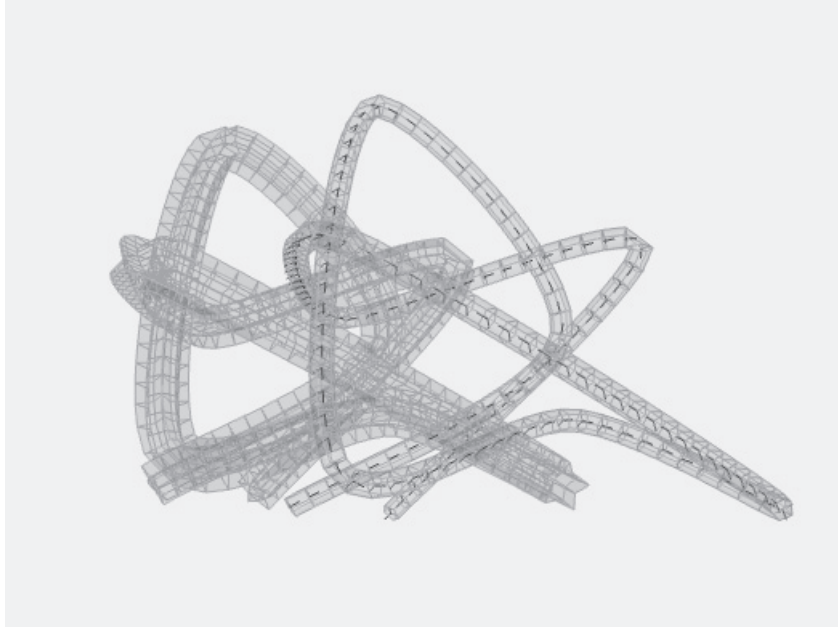
Aby zobaczyć na żywo, jak wyglądają gotowe obiekty geometryczne biblioteki Three.js, otwórz przykładowy plik projektu Three.js znajdujący się w folderze *examples/webgl_geometries.html* (rysunek 4.1). Każdy obiekt siatki zawiera inny typ geometrii, a tekstura ukazuje sposób generowania współrzędnych 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 doskonały sposób na generowanie naturalnie wyglądających kształtów. Szczegółowy opis krzywych składanych znajduje się w rozdziale 5.



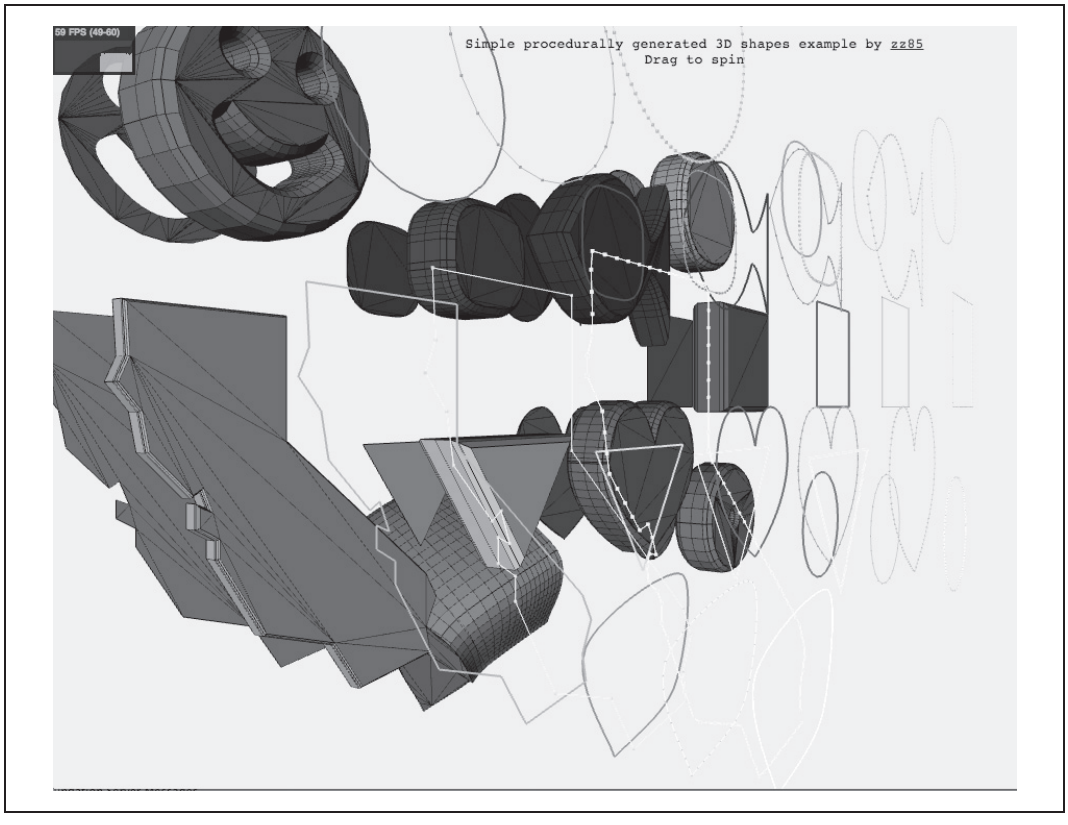
Rysunek 4.2. Ekstruzje utworzone przy użyciu krzywej składanej z biblioteki Three.js

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 najprostszych 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 );
```

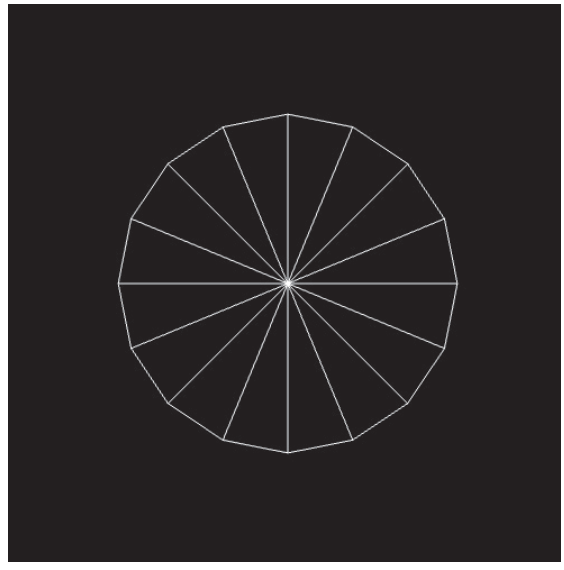
```
    for ( i = 0; i <= segments; i ++ ) {

        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 pamiętasz z rozdziału 2., że w WebGL używa się funkcji `gl.drawElements()` do renderowania trójkątów przy użyciu tablicy indeksowanej. To samo dzieje się tutaj, tylko wewnątrz mechanizmów biblioteki Three.js).



Rysunek 4.4. Trójkąty składające się na obiekt THREE.CircleGeometry

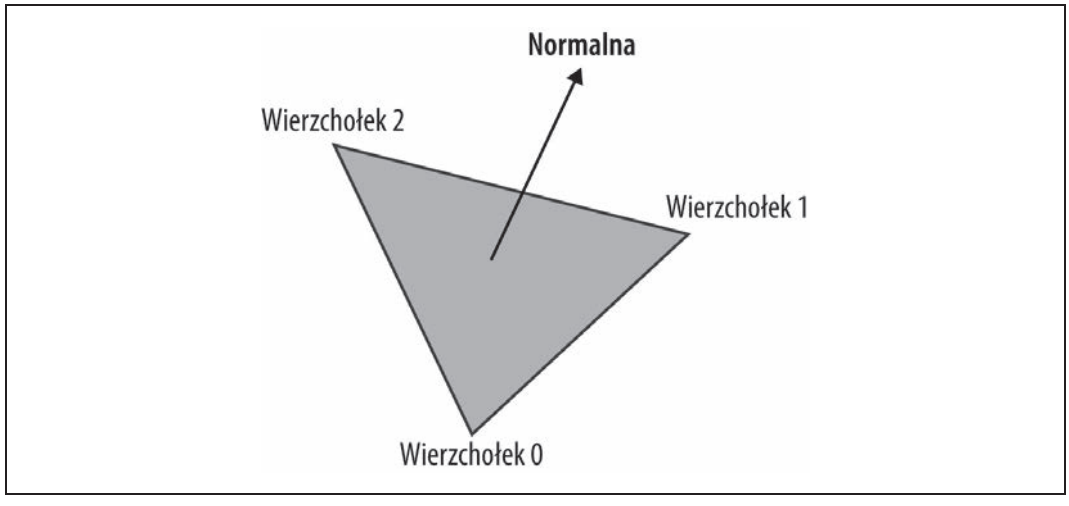
W każdej z pętli pominęliśmy jeden szczegół: generowanie współrzędnych teksturowych. Biblioteka WebGL „nie wie", jak mapować piksele tekstury na rysowane trójkąty, więc trzeba jej to podpowiedzieć. Pętle `for` generują współrzędne teksturowe, zwane również **współrzędnymi UV**, i zapisują je w tablicy `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 tekstury, 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.

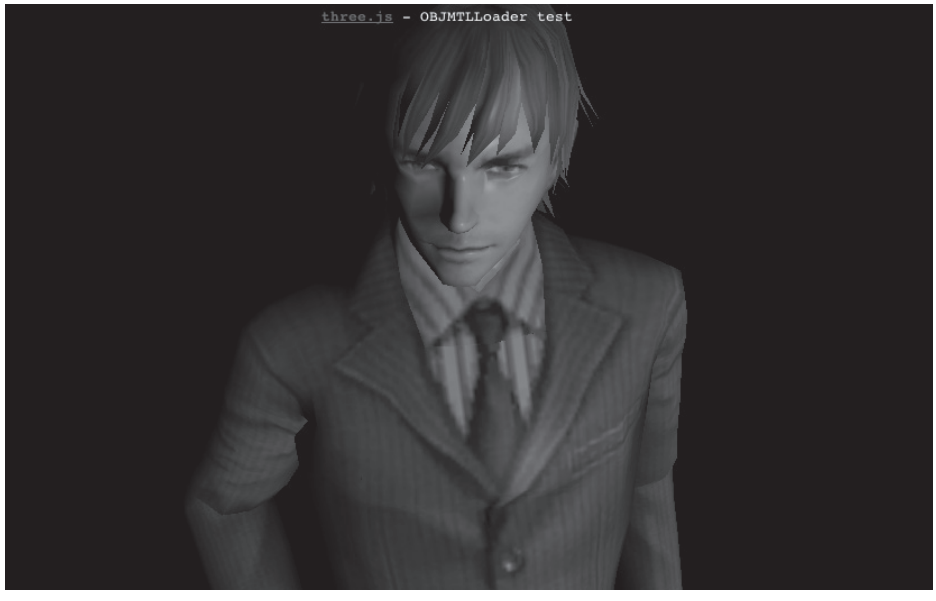
Geometria buforowana do optymalizacji renderowania siatki

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. teł i przedmiotów scen, gdy wiadomo, że wartości wierzchołków się nie zmienią, 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 wczytujące 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, świateł 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 dołączyć 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 nadrzędny-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

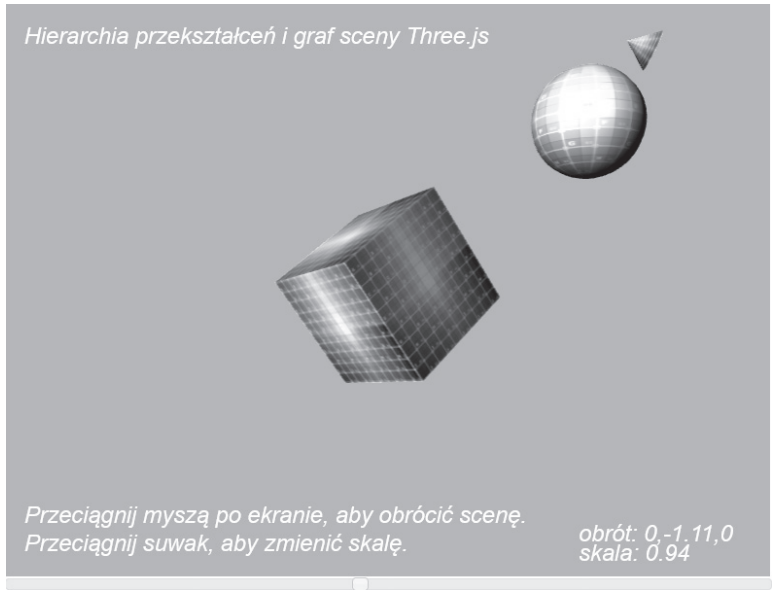
Grafy sceny są najbardziej przydatne do reprezentowania złożonych obiektów w hierarchiach. Wyobraź sobie np. robota, pojazd albo układ słoneczny. Każdy z tych obiektów składa się z pewnej liczby części — kończyn, nóg, satelitów — z których każda zachowuje się we właściwy sobie sposób. Graf sceny umożliwia, 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 nadrzędnym. Powiedzmy przykładowo, że tworzymy animację samochodu poruszającego się po określonej ścieżce. Bryła samochodu przesuwa się w wyznaczonym kierunku, ale jego koła kręcą 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 nadrzędny). 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 własnościach `position` (przesunięcie), `rotation` oraz `scale` zawiera informacje dotyczące przekształceń. Ustawiając ich wartości, można dany obiekt przesunąć, obrócić i przeskalować. Jeżeli obiekt ma potomków (dzieci i ich dzieci), dziedziczą one jego ustawienia przekształceń. Jeśli własności przekształceń obiektów podrzędnych 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 sferę. Ponadto przesunięto nieco stożek do góry i oddalono go od sfery za pomocą odpowiedniego ustawienia własności `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 teksturę.
    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);
}
```

Kolej na animacje. 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 siatką 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 własności `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, 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 przekształcenia 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 zauważyłeś 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 połyskliwość. 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.

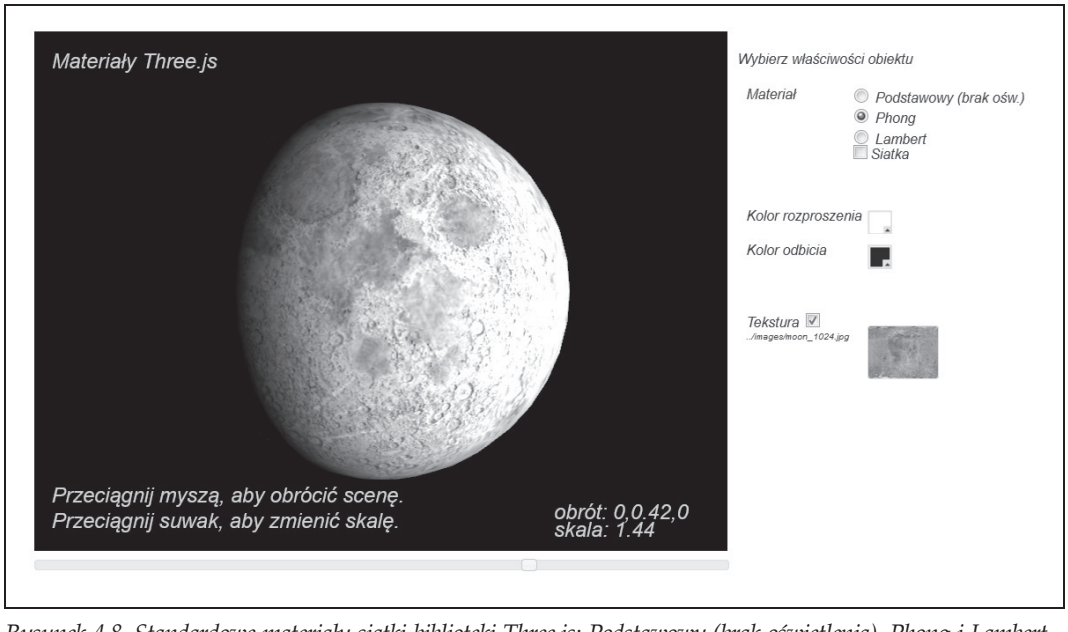
Biblioteka Three.js obsługuje najczęściej używane typy materiałów w klasach `MeshBasicMaterial`, `MeshPhongMaterial` oraz `MeshLambertMaterial`. (Przedrostek `Mesh` oznacza, że obiekty tych typów powinny być używane w połączeniu z obiektami siatki, a więc nie z liniami czy cząsteczkami; istnieją też 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 odwrócone od światła.

Cieniowanie Lamberta
: W cieniowaniu Lamberta 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 księżyce, które mają wysokie **albedo** (odbijają dużo światła od powierzchni).

Aby zobaczyć, jak wyglądają różne rodzaje materiałów, otwórz plik *r4/threejsmaterials.html*. Na rysunku 4.8 pokazano jasno oświetloną sferę z nałożoną teksturą imitującą powierzchnię księżyca. Księżyc 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

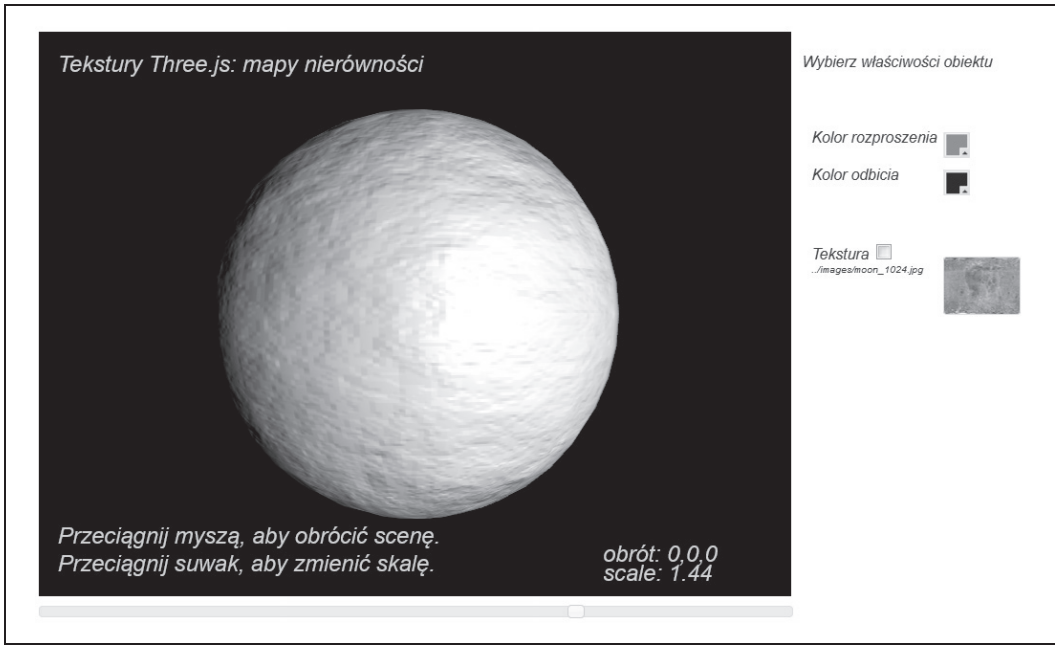
W poprzednim przykładzie pokazałem, jak zdefiniować wygląd powierzchni obiektu za pomocą tekstury. Większość typów materiałów w bibliotece Three.js umożliwia stosowanie wielu tekstur, co pozwala osiągnąć bardziej realistyczny efekt. W technice tej, zwanej **multiteksturowaniem**, chodzi o zwiększenie realizmu bez wykonywania nadmiernej ilości dodatkowych obliczeń. Alternatywą jest użycie większej liczby wielokątów lub wyrenderowanie 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 teksturę 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 własności `bumpMap` obiektu parametrów przekazywanego do konstruktora klasy `THREE.MeshPhongMaterial`.

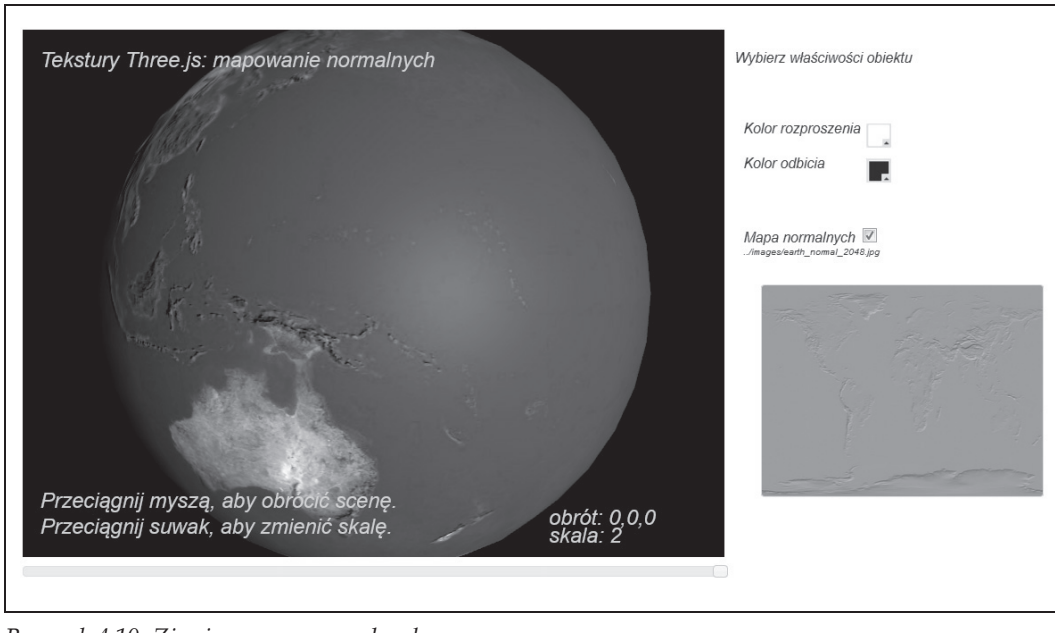
```
material= new THREE.MeshPhongMaterial({map: map,
    bumpMap: bumpMap});
```

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 rozdzielczość niż w danych wierzchołków siatki. Shader wprowadza te informacje normalnych 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/threejsnormalmap.html*. Użyta mapa normalnych



Rysunek 4.9. Mapowanie nierówności

jest widoczna na dole po prawej (rysunek 4.10). Zwróć uwagę na zarysy wzniesień 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.



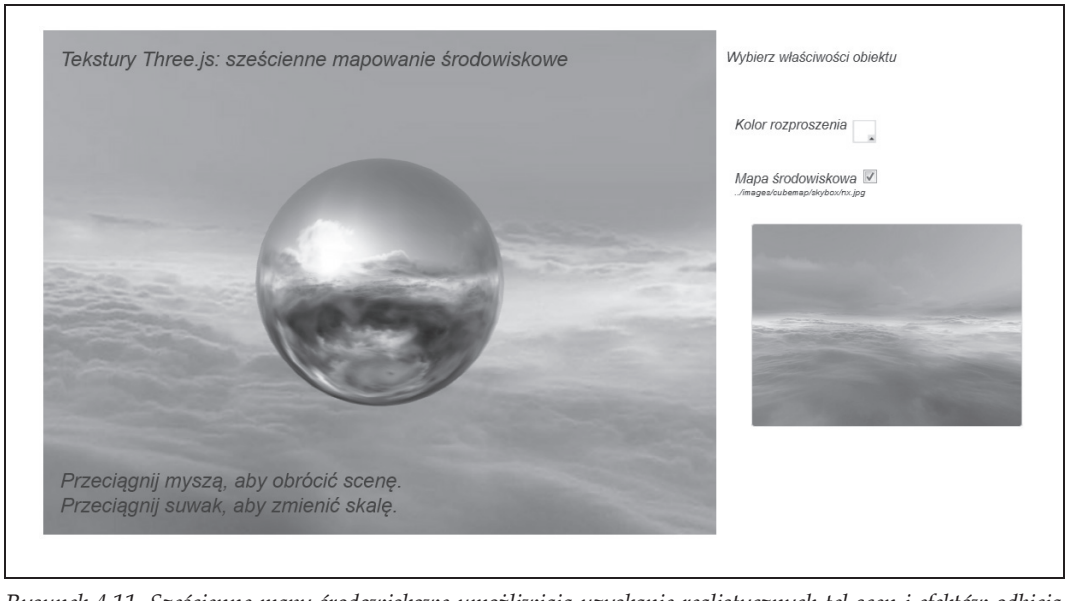
Rysunek 4.10. Ziemia z mapą normalnych

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

```
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/three-jsenvmap.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 wewnątrz kostki użytej jako tło sceny. Sztuczka polega na tym, że na materiale sfery użyto **tekstury sześciennej**, czyli utworzonej z sześciu odrębnych map bitowych połączonych w jeden obraz wewnątrz sześcianu. W tym przykładzie utworzono w ten sposób tło ilustrujące niebo. Poszczególne pliki składające się na ten produkt znajdują sie 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 teł 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: 0xffffff,
     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 teksturą 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 wewnątrz 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 reflektorowe** 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 określoną odległość.

Światło reflektorowe
: Światło reflektorowe ma pozycję i kierunek. Ponadto można określać 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 **kolorem 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.

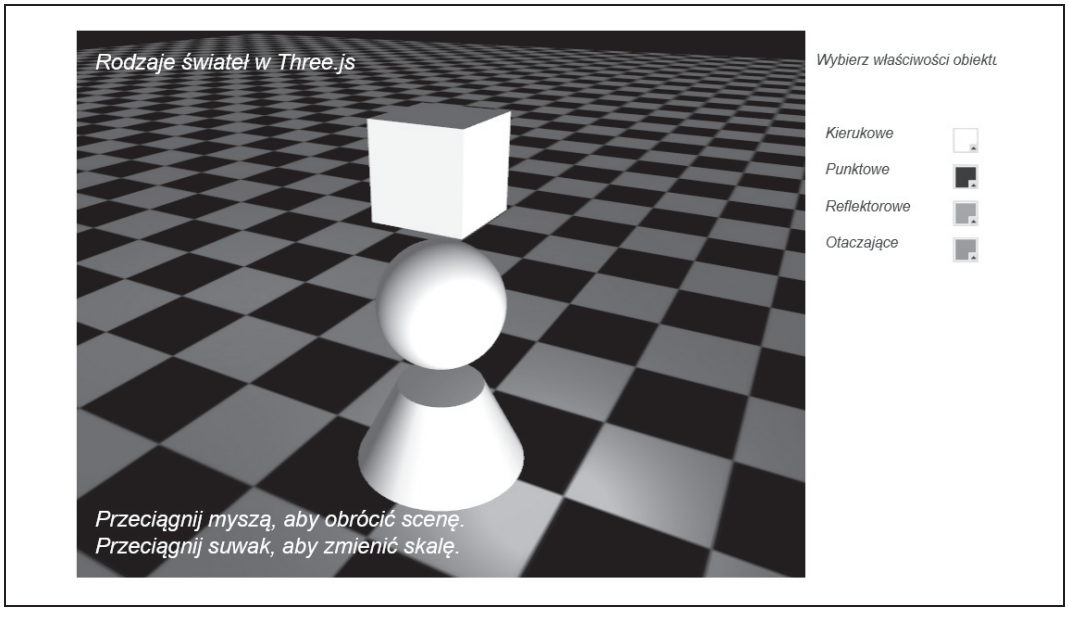
`emissive`
: Ta własność materiału określa kolor emitowany przez obiekt, niezależnie od występujących na scenie źródeł światła.

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 ignoruje światło.

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

Na poniższym listingu przedstawiony jest kod źródłowy dotyczący tworzenia świateł. Białe światło kierunkowe umiejscowione przed sceną oświetla jaskrawe białe obszary znajdujące się na przedzie obiektów geometrycznych. Niebieskie światło punktowe świeci z tyłu modelu. Zwróć 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

własności `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.

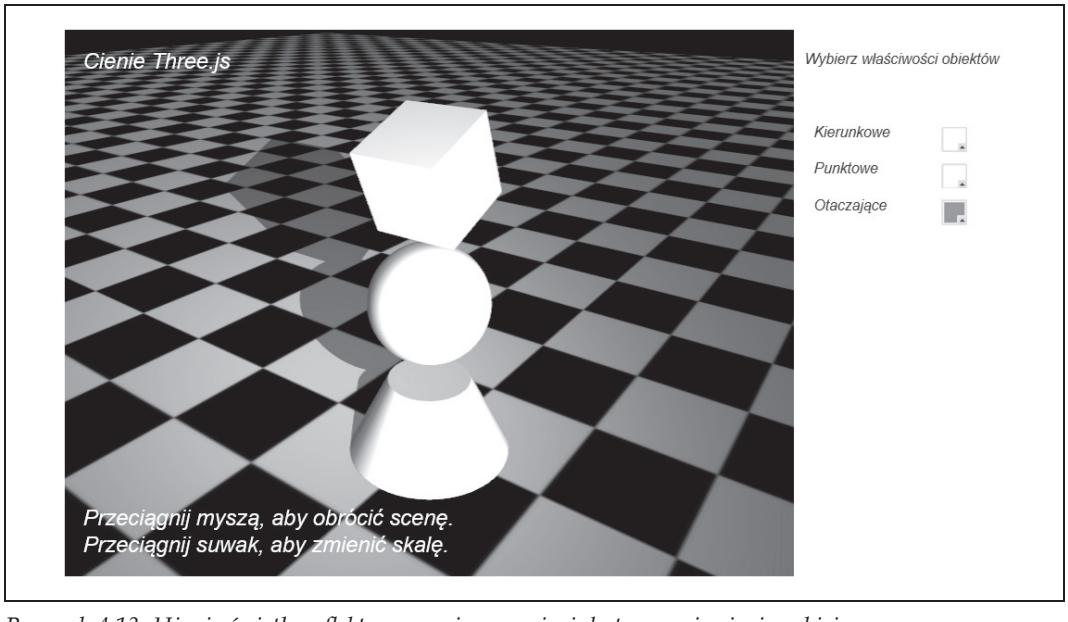
```
// Tworzy i dodaje wszystkie światła.
directionalLight.position.set(.5, 0, 3);
root.add(directionalLight);

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 bufory i shadery, a zadaniem programisty jest synteza efektów oświetleniowych za pomocą kodu shadera. Biblioteka Three.js zawiera oszałamiającą ilość narzędzi do pracy z materiałami i światłem. A jeśli weźmie się pod uwagę, że wszystko to napisano w JavaScripcie, 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 realizmu 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 świateł 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 świateł, 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 zacienione 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 świateł, które rzucają cienie. Można to zrobić dla typów świateł `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 = 0xffffff;
    var ambient = 0x888888;
    // Dodaje płaszczyznę 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 włączyliśmy 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 własność `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. Domyślnie wynosi ona `0.5` i jest to wartość odpowiednia w naszym przypadku. Następnie ustawiamy własności określające rozmiar mapy cieni. Jest to kolejna mapa bitowa tworzona przez Three.js, na której biblioteka renderuje zacienione obszary, które następnie miesza 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. Domyślnie siatki nie obsługują cieni, więc trzeba to zmienić za pomocą odpowiednich ustawień. W tym przykładzie chcemy, aby obiekty geometryczne rzucały 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 własności `shadowDarkness` światła. Musimy zatem samodzielnie aktualizować tę własność 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 teksturą, 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, specjaliści 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 Open GL i Open GL ES (na której bazuje API WebGL). Kod źródłowy w języku GLSL jest kompilowany i wykonywany na użytek WebGL przy użyciu metod obiektu kontekstu WebGL. Biblioteka Three.js ukrywa przed programistą kod GLSL, dzięki czemu można całkowicie pominąć 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 zawdzięczają nazwę efektowi Fresnela, zjawisku, które jako pierwszy opisał francuski fizyk Augustin-Jean Fresnel (1788 – 1827). Fresnel rozwinął 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*. Przeanalizujemy 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. Przypomnę, 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 Fresnela definiuje dane jednolite sterujące odbiciem i załamaniem światła (np. `mRefractionRatio` i `mFresnelScale`). Ponadto shader ten definiuje zmienną jednolitą dla tekstury sześciennej używanej 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 zwracane 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 konkatencję, 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, kompilowanie 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 osiągnęlibyśmy zadowalającej szybkości zmiany klatek. Dlatego właśnie tak często korzystamy z biblioteki WebGL.

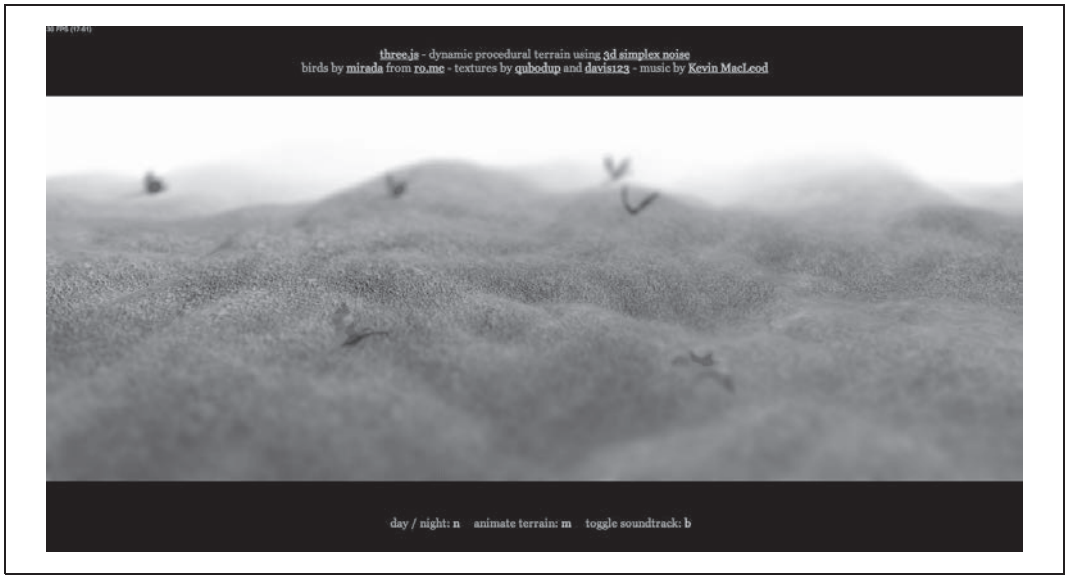
Sama biblioteka WebGL również oferuje kilka sposobów renderowania obrazów. Można np. zastosować renderowanie przy użyciu **bufora głębi** (ang. *z-buffer*), które polega na wykorzystaniu przez sprzęt dodatkowej pamięci, aby renderować na scenie wyłącznie piksele znajdujące się na przedzie. Wszystko zależy 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 nakładu pracy ze strony programisty. Jak można się zorientować, sądząc 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 wysokiej jakości 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 majestatycznie przelatują nad nieziemsko wyglądającą krainą, przecinając mgliste powietrze przy zachodzącym 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.

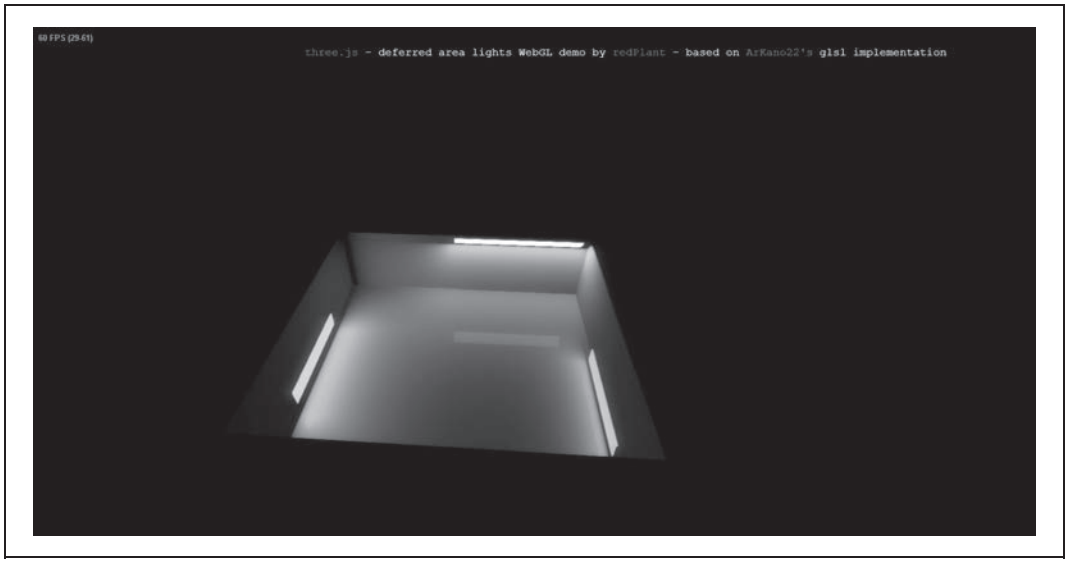
- Obsługa **wielu celów renderowania** poprzez obiekt `THREE.WebGLRenderTarget`. Dzięki wielu celom renderowania scenę można renderować wielokrotnie na pozaekranowych mapach bitowych, a następnie połączyć je wszystkie w ostateczny obraz. (Plik źródłowy: *src/renderers/WebGLRenderTarget.js*).
- Wieloprzebiegowa pętla renderowania zaimplementowana w klasie `THREE.EffectComposer`. Obiekt tej klasy zawiera przynajmniej jeden obiekt **przebiegu renderowania**, który wywołuje po kolei 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. Prześledziłeś praktyczne przykłady wykorzystania materiałów, tekstur oraz świateł. 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.

Skorowidz

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

Ś

T

U

V

W

Z

O'REILLY®

Aplikacje 3D

Przewodnik po HTML5, WebGL i CSS3

Do niedawna wyświetlanie zaawansowanej grafiki 3D w przeglądarce internetowej wymagało zainstalowania dodatkowych wtyczek oraz poznawania nowych narzędzi. Dzięki HTML5 i WebGL te czasy powoli odchodzą w niepamięć! Teraz możesz wykorzystać niesamowite możliwości tego duetu, by zaskoczyć użytkowników atrakcyjnymi efektami 3D!

Ta wyjątkowa książka została w całości poświęcona właśnie zagadnieniom związanym z grafiką 3D w przeglądarce internetowej. Sięgnij po nią i przekonaj się, jak wykorzystać API WebGL do renderowania trójwymiarowej grafiki w czasie rzeczywistym. W kolejnych rozdziałach poznasz bibliotekę języka JavaScript – Three.js, która w znaczący sposób ułatwia życie programistom. Informacje zawarte w dalszych rozdziałach pozwolą Ci skorzystać z zaawansowanych efektów w CSS3 i tworzyć animacje trójwymiarowe. Zaznajomisz się też ze szczegółami tworzenia aplikacji dla urządzeń mobilnych. Twoją uwagę z pewnością przykuje przegląd narzędzi do tworzenia trójwymiarowych modeli i animacji – zarówno tych klasycznych, jak i tych online. To doskonała lektura dla wszystkich deweloperów chcących wzbogacić swój warsztat o elementy grafiki 3D.

Dzięki tej książce:

- zapoznasz się z podstawami teorii grafiki 3D
- poznasz API WebGL
- wykorzystasz bibliotekę Three.js
- odkryjesz narzędzia przydatne w codziennej pracy
- jeszcze bardziej uatrakcyjnisz Twoją stronę

Poznaj potencjał HTML5 w zakresie grafiki 3D!

helion.pl
księgarnia internetowa

Nr katalogowy: 26715

Księgarnia internetowa:
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ
KOD KORZYŚCI

ISBN 978-83-246-9668-0

Cena 59,00 zł