Świat 3D w Javie

Podstawy programowania z wykorzystaniem API Java 3D

Java jest jednym z najczęściej wykorzystywanych języków programowania, który w połączeniu z wirtualną maszyną Javy daje aplikacjom w niej napisanym dużą przenośność. Java 3D, darmowe API do Javy, pozwala wykorzystać podstawowe zalety języka i platformy do renderowania obiektów 3D w bardzo elastyczny i logiczny sposób.

Dowiesz się:
- jak wyświetlać obiekty 3D przy pomocy Javy 3D,
- jak wykorzystywać podstawowe obiekty 3D dostępne w Javie 3D,
- jak tworzyć własne obiekty 3D,
- jak przypisywać obiektom kolory, tekstury, właściwości oświetlenia, etc.

Powinieneś wiedzieć:
- podstawy języka Java,
- podstawy geometrii analitycznej,
- wiedzieć czym cechuje się struktura drzewa.

Poziom trudności

Java 3D to darmowe, wszechstronne i potężne API pozwalające na wyświetlanie obiektów 3D w czasie rzeczywistym oraz interakcję z nimi. Wykorzystuje ona w pełni moc karty graficznej poprzez DirectX lub OpenGL, przez co renderowane sceny są naprawdę wysokiej jakości. Obsługiwane są funkcje takie jak antyaliasing, filtrowanie anizotropowe, cieniowanie, dynamiczne oświetlanie, słowem, Java 3D daje programiście naprawdę szerokie możliwości – począwszy od tworzenia prostych aplikacji, poprzez wizualizacje różnego rodzaju symulacji, aż po gry 3D. Co ważniejsze, Java 3D jest naprawdę dobrze przemyślanym API, choć może nie jest to widoczne przy pierwszym zetknięciu z nim, a programowanie z jego wykorzystaniem jest bardzo intuicyjne.

Instalacja

Należy ściągnąć odpowiedni instalator (lub ewentualnie zestaw binarek) ze strony *https://java3d.dev.java.net/binary-builds.html*, po czym go uruchomić. Java 3D zostanie zainstalowana i zintegruje się z JRE/SDK automatycznie.

Pierwszy program – J3DTextDemo – Hello 3D World!

Program ten, jak widać na obrazku, wyświetla trójwymiarowy napis Hello 3D World oświetlony z trzech stron – od przodu punktowym światłem zielonym, z lewej kierunkowym niebieskim, a z prawej kierunkowym żółtym. Zanim przejdziemy do samego obiektu 3D, jakim jest ten tekst, musimy przyjrzeć się klasom niezbędnym do stworzenia świata 3D, z którego będziemy korzystać. Fragment kodu za to odpowiedzialny widoczny jest na Listingu 1.

Kluczową rolę w tym kodzie pełni klasa SimpleUniverse. Jest ona swego rodzaju kontenerem, który po utworzeniu zawiera wszystkie niezbędne do renderowania świata 3D obiekty z przypisanymi podstawowymi ustawieniami. Nie będę szczegółowo opisywał każdego z nich, gdyż z punktu widzenia przedstawianych programów nie jest to konieczne. Warto jednak zauważyć, że każdy z obiektów składowych jest dostępny poprzez odpowiednie metody w SimpleUniverse, dzięki czemu możemy zmieniać tylko te jego elementy, które nas interesują.

Oto opis wykonywanych czynności przygotowawczych:

- Tworzymy obiekt `GraphicsConfiguration` – metodą statyczną klasy `SimpleUniverse` pobieramy konfigurację grafiki, która powinna być optymalna dla naszej konfiguracji sprzętowej.
- Tworzymy obiekt `Canvas3D`. Jest to komponent, który służy do renderowania świata 3D. Dodajemy go później do GUI w dowolnym, odpowiadającym nam miejscu. Dodatkowo ustawiamy preferowany rozmiar tego płótna.
- Tworzymy główną gałąź (BranchGroup) naszego świata 3D, po czym ją kompilujemy. Kompilacja sceny to swego rodzaju optymalizacja struktury obiektów przez nas dodanych. Czynność ta powinna być wykonana przed wyświetleniem danej grupy, gdyż może ona zwiększyć wydajność. Szczegóły dotyczące kompilacji można znaleźć pod adresem *http://java3d.j3d.org/tutorials/quick_fix/compile.html* oraz w dokumentacji API.
- Tworzymy nowy świat 3D przypisany do utworzonego wcześniej obiektu canvas3d.
- Dodajemy do świata utworzoną wcześniej gałąź z obiektami 3D.

Po wykonaniu tych czynności mamy w pełni funkcjonujący świat 3D, którego zawartość wyświetlana jest poprzez obiekt canvas3d. Należy jeszcze dodać go do naszego GUI: fragment zaprezentowany w Listingu 2.

Dodawanie obiektów do świata 3D

Najwyższa pora, by po stworzeniu obiektów pozwalających na renderowanie świata 3D, zająć się zawartością tego świata. W tym programie cała zawartość głównej gałęzi generowana jest w jednej metodzie – linijka po linijce. W przypadku bardziej skomplikowanych światów sugeruję, by stworzyć własne klasy reprezentujące obiekty 3D, dzięki czemu aplikacja stanie się dużo bardziej przejrzysta, a operowanie na nich dużo prostsze. Przykład takiej klasy znaleźć można w programie 3.

Świat 3D w Javie 3D tworzony jest w formie drzewa. Każdy obiekt 3D musi mieć przypisanego dokładnie jednego rodzica, przy czym korzeniem jest w naszym przypadku obiekt SimpleUniverse. Ilość potomków nie jest ograniczona. Strukturę tę w Javie 3D reprezentują klasy abstrakcyjne `Node`

– węzeł oraz dziedziczące po nim `Leaf` – liść (węzeł bez potomków, właściwy obiekt 3D) oraz `Group` – grupa (węzeł do którego można przypinać inne węzły, w tym liście).

Dwie najważniejsze i najczęściej stosowane klasy potomne klasy `Group to BranchGroup` oraz `TransformGroup`. Opiszę je teraz dokładniej.

`BranchGroup` jest takim węzłem, o którego poddrzewie powinno się myśleć jak o samodzielnej miniscenie. Obiektu tego używamy do oznaczenia konkretnego obiektu lub grupy obiektów, które tworzą pewną całość, np. jeśli mamy w świecie 3D lampę składającą się z abażuru, żarówki, stojaka oraz źródła światła, obiekty te powinny zostać zamknięte w oddzielnej `BranchGroupie`, która będzie reprezentować lampę jako całość. Oprócz wprowadzania logiki i ładu, takie konstruowanie sceny ma zalety praktyczne – dzięki temu możemy jedną operacją np. odłączyć od sceny całą grupę.

`TransformGroup` jest węzłem, który ma przypisaną transformację. Oznacza to, że wszyscy potomkowie tego węzła mogą zostać przesunięci i obróceni w dowolny (ale wszyscy w taki sam) sposób.

Oprócz węzłów-rodziców scena składa się także z liści. W przypadku J3DTextDemo wykorzystywane są następujące klasy dziedziczące po Leaf:

- `Shape3D`, który jest dość ogólną klasą reprezentującą dowolny kształt w świecie 3D. W naszym przypadku zawiera ona napis przygotowany przy pomocy klas `Text3D` i `Font3D`.
- `DirectionalLight`, który definiuje źródło światła padające w konkretnym kierunku z nieskończenie dalekiego źródła, w związku z czym promienie światła są do siebie równoległe.
- `PointLight`, który definiuje źródło światła w konkretnym miejscu w przestrzeni. Promienie światła rozchodzą się koncentrycznie we wszystkich kierunkach.
- Obie powyższe klasy rozszerzają klasę `Light`, która definiuje także kolor emitowanego światła.

Wiedząc to wszystko, przyjrzyjmy się strukturze sceny w J3DTextDemo (Rysunek 2)

Korzeniem jest obiekt klasy `Locale` (który jest częścią `SimpleUniverse`) i do niego przypinamy stworzoną przez nas grupę `wholeScene`. Bezpośrednio do niej przypinamy źródła światła, gdyż chcemy, by ich położenie było niezmienne. Gdybyśmy dodali utworzony tekst do głównej gałęzi bezpośrednio, okazałoby się, że jest zbyt duży, by zmieścić się na ekranie. Ponadto, tekst ten byłby ułożony dokładnie frontem do nas, w związku z czym w ogóle nie widzielibyśmy, że jest on obiektem 3D! Dlatego właśnie `shDispText` dołączony jest do `textScalingGroup`, która zmniejsza jego rozmiar, ta grupa zaś dołączona jest do `textRotatingGroup`, która go obraca.

Mam nadzieję, że struktura J3DTextDemo jest już dla Was zrozumiała. Program ten miał na celu zademonstrowanie jak proste i szybkie jest tworzenie scen 3D. Teraz skupimy się na tym, co dzieje się za kulisami, czyli jak to wszystko tak naprawdę działa.

Świat Javy 3D

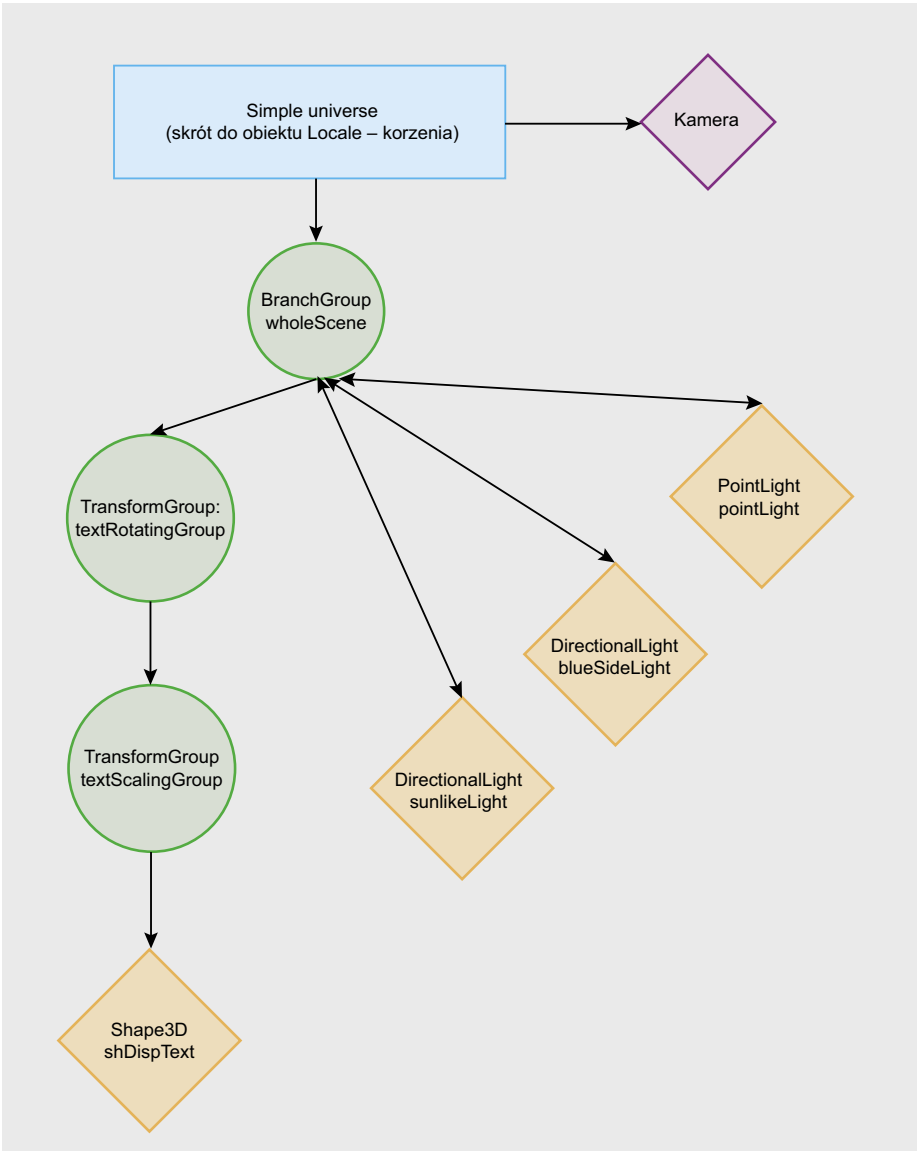
Pierwszą rzeczą, którą trzeba zapamiętać, jest układ współrzędnych obowiązujący w Javie 3D, widoczny obok ilustracji 3. Patrząc na scenę z domyślnego punktu, widzimy osie x i y jak w 2D, zaś nasze oczy skierowane są w stronę mniejszych wartości osi z.

Jest to układ współrzędnych nieco inny, niż ten, do którego jesteśmy przyzwyczajeni ze szkoły czy uczelni. By się z nim zaznajomić, proponuję modyfikować wierzchołki sześcianu w programie 2 i obserwować jak wpłynie to na kształt bryły.

Każdy obiekt 3D składa się z pewnej ilości polygonów – trójkątów umieszczonych w przestrzeni. Każdy taki trójkąt definiowany jest przez trzy punkty o współrzędnych kartezjańskich (x, y, z). By stworzyć kształt inny niż trójkąt, musimy stworzyć pewną ilość polygonów – np. na ścianę sześcianu (kwadrat) wystarczą dwa trójkąty, zaś na cały sześcian – 2*6 = 12 polygonów. Grupę polygonów składających się na obiekt będę nazywał jego geometrią - przykład jej definiowania znajduje się w programie 2.

Zakładając, że mamy gotowy zestaw polygonów, który chcemy umieścić w określonym miejscu w przestrzeni, mamy dwie możliwości: albo zmodyfikujemy geometrię obiektu i przesuniemy wszystkie jego wierzchołki *ręcznie*, tzn. zmodyfikujemy ich współrzędne, al-

Rysunek 1. *Hello 3D World – napis wyrenderowany przez Javę 3D*



Rysunek 2. *Drzewo sceny programu J3DTextDemo*

bo umieścimy ten obiekt wewnątrz obiektu `TransformGroup` i przypiszemy mu określoną transformację dokonującą tego samego. Z punktu widzenia wydajności obie metody są podobne, zaś z punktu widzenia prostoty rozwiązania, dużo lepiej umieścić dany kształt w `TransformGroupie`.

Do każdego obiektu `TransformGroup` przypisana jest pewna transformacja – obiekt Transform3D. Z matematycznego punktu widzenia jest macierzą 4 x 4, zaś współrzędne punktu lub wektora objętego daną transformacją to iloczyn macierzy z tym wektorem. Rozumienie mechanizmów matematycznych jest przydatne, ale nie jest konieczne, gdyż Java 3D daje nam możliwość tworzenia złożonych transformacji poprzez składanie tych podstawowych.

Listing 1. *Inicjalizacja świata 3D*

```
// pobieramy podstawową konfigurację graficzną z SimpleUniverse
GraphicsConfiguration config =
SimpleUniverse.getPreferredConfiguration();

// tworzymy nowy Canvas3D z podaną konfiguracją
Canvas3D canvas3d = new Canvas3D(config);

// ustawiamy preferowany rozmiar canvasa
canvas3d.setPreferredSize(new Dimension(800,200));

// tworzymy główną gałąź świata 3D
BranchGroup scene = createSceneGraph();

// po zakończeniu jej tworzenia dokonujemy kompilacji
scene.compile();

// tworzymy nowy świat 3D
SimpleUniverse universe = new SimpleUniverse(canvas3d);

// dodajemy do świata stworzoną wcześniej gałąź
universe.addBranchGraph(scene);
```

Listing 2. *Wyświetlanie stworzonego świata 3D na ekranie*

```
JFrame dialog = new JFrame();
dialog.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
dialog.setTitle("Hello 3D World!");
JPanel panel = new JPanel();
dialog.getContentPane().add(panel);

// dodajemy Canvas3D jak zwykły komponent
panel.add(new J3DTextDemo().getCanvas());
dialog.pack();
dialog.setVisible(true);
```

Listing 3. *Złożenie transformacji*

```
// ustalamy obrót w rotTransform - w osi x;
// zmienna curTransform zawiera transformację,
// którą mieliśmy wcześniej, chcemy do niej dodać nowe obroty
rotTransform.rotX(Math.PI/100*Math.sin(x));
// ustawiamy newTransform na złożenie transformacji aktualnej i obrotu
newTransform.mul(curTransform, rotTransform);

// ustalamy obrót w rotTransform - w osi y
rotTransform.rotY(Math.PI/100*Math.cos(x));
// ustawiamy new transform na złożenie siebie samego z rotTransform
newTransform.mul(rotTransform);

// ustalamy obrót w rotTransform - w osi z
rotTransform.rotZ(-Math.PI/100*Math.sin(x));
// ustawiamy new transform na złożenie siebie samego z rotTransform
newTransform.mul(rotTransform);
```

Gdy tworzymy nowy obiekt `Transform3D`, reprezentuje on przekształcenie identycznościowe, czyli niezmieniające w żaden sposób obiektów 3D. By zmienić działanie transformacji można ręcznie przypisać mu pewną macierz lub skorzystać z metod zaimplementowanych w Javie 3D. Są to między innymi:

- `rotX(double angle)` – przypisuje danej transformacji obrót w osi x o dany kąt,
- `rotY(double angle)`, `rotZ(double angle)` – analogicznie do powyższego, dla różnych osi,
- `setTranslation(Vector3f translation)` – ustawia przesunięcie w przestrzeni o dany wektor,
- `mul(Transform3D transform)` – łączy daną transformację z podaną w argumencie, dzięki czemu możemy otrzymać zarówno obrót, jak i przesunięcie,
- `mul(Transform3D transform1, Transform3D transform2)` – ustawia wartość transformacji na złączenie transformacji podanych jako parametry.

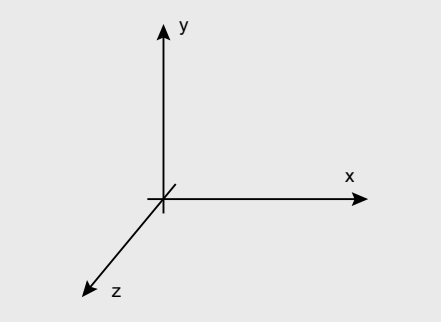
Przykład wykorzystania tych metod demonstruje fragment kodu z programu 2. –Listing 3.

Zmienna `newTransform`, po wykonaniu powyższych operacji, staje się złączeniem poprzedniej transformacji oraz obrotów wokół trzech osi o różne kąty. Zachęcam do poeksperymentowania z tym fragmentem kodu, np. poprzez dodanie przesunięcia.

Łącząc transformacje zawierające przesunięcie oraz obrót należy pamiętać, że wektor przesunięcia będzie dotyczył nowej bazy, tzn. jeśli ustawimy w transformacji przesunięcie o wektor (1, 0, 0) a następnie (lub wcześniej, kolejność nie ma znaczenia) dodamy (metodą mul) obrót o kąt 90 stopni wokół osi y, to obiekt znajdzie się w przestrzeni w miejscu (0, 0, 1). By zrozumieć ten problem, sugeruję szczegółowo zapoznać się z hierarchią transformacji w programie 3.

Program 2 – J3DRotatingCube – Wirujący sześcian

W tej części artykułu zapoznamy się m.in. z metodami tworzenia i dodawania własnych kształtów do świata 3D, poznamy szczegóły dotyczące działania oświetlenia oraz mechanizm nakładania tekstur na obiekty 3D.



Rysunek 3. *Układ współrzędnych w Javie 3D*

Najważniejsze klasy

Podstawowe obiekty, które wykorzystywać będziemy w tym programie, to:

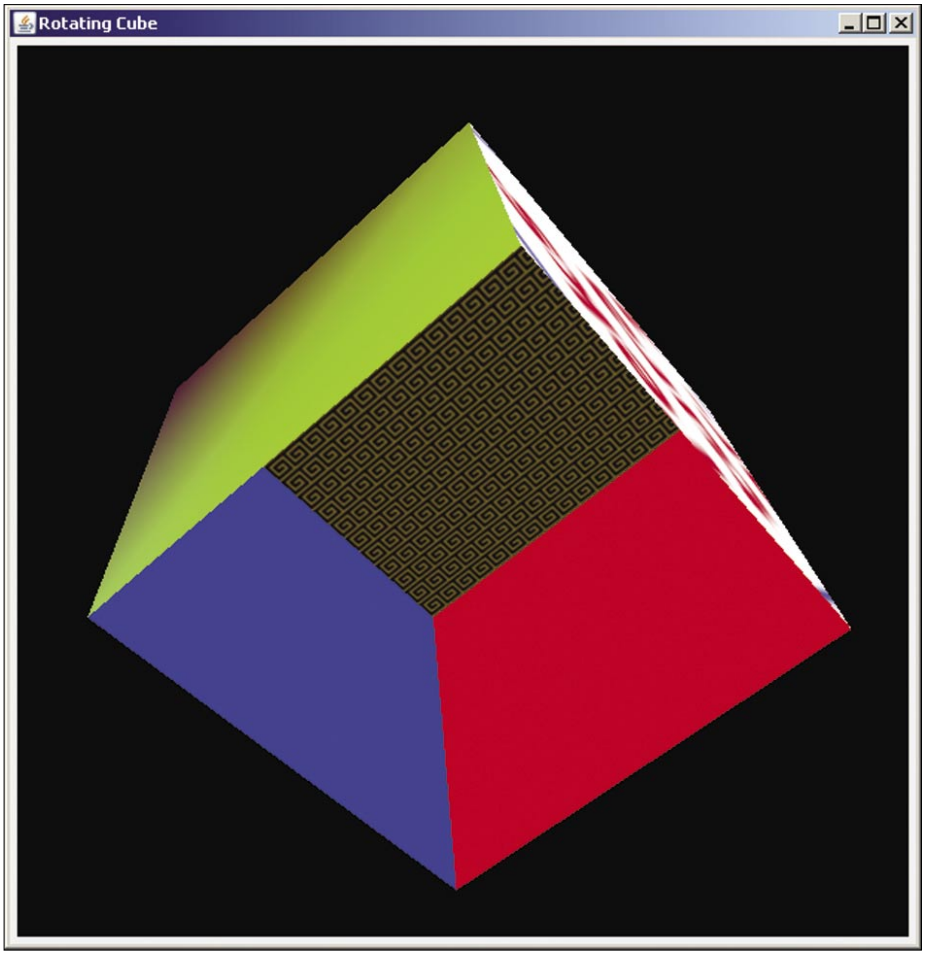
- `Appearance` – obiekt-kontener, zawierający wszystkie ustawienia związane z wyglądem danego obiektu, począwszy od koloru, przez sposób renderowania, aż po przezroczystość.
- `Shape3D` – podstawowa klasa opisująca obiekt 3D. By pojawić się na ekranie, musi posiadać swoją geometrię (Geometry), zaś jeśli chcemy, by pojawiała się w formie innej niż czarne plamy, musi także posiadać swój wygląd (Appearance).
- `Geometry` - klasa definiująca wierzchołki w przestrzeni 3D, a co za nimi idzie, polygony, a także ich normalne (wektory definiujące zorientowanie polygonu, potrzebne do obliczania wpływu oświetlenia), kolory dla wierzchołków oraz współrzędne dla tekstur.
  Istnieje wiele klas rozszerzających Geometry, zaś każda z nich ma właściwy dla siebie sposób zastosowania. W J3DRotatingCube wykorzystywana jest klasa QuadArray. W jej konstruktorze specyfikujemy, jakie informacje będzie zawierać – jest to ważne, gdyż brak odpowiedniej flagi będzie powodował rzucenie wyjątku przy próbie zapisu związanej z nią informacji.
- `Material` – opisuje właściwości świetlne powierzchni, musi być przypisana do klasy Appearance. Materiał opisywany jest przez:
  - `EmmisiveColor` - kolor, który jest emitowany przez obiekt,
  - `AmbientColor` – kolor, który jest emitowany przy oświetleniu światłem typu Ambient,
  - `DiffuseColor` – kolor kierunkowo oświetlonej części obiektu,
  - `SpecularColor` – kolor efektu odbicia światła od kierunkowo oświetlonego obiektu,
  - `Shininess` – liczba określająca rozmiar odbłysku SpecularColor,

By zrozumieć działanie poszczególnych kolorów (oraz wykorzystać dotychczas zdobytą wiedzę w praktyce) sugeruję stworzyć scenę 3D, w której znalazłaby się kula (Sphere-3D) oraz umiejscowione nad nią źródło światła, po czym poeksperymentować z różnymi materiałami.

Sześcian

Każda ze ścian sześcianu z programu J3DRotatingCube demonstruje oddzielne zagadnienie. Każda z nich tworzona jest jednak w ten sam sposób – poprzez ustalenie w obiekcie `QuadArray` wierzchołków opisujących powierzchnię. Wykorzystuje się do tego metodę `setCoordinate(int index, Point3f)`.

Na początek rozważmy *ścianę dolną*, czyli czerwoną. Po uruchomieniu programu można zauważyć, że widoczna jest ona tylko wtedy, kiedy patrzy się *od środka* sześcianu. Nie jest to przypadkowe zachowanie. Każdy polygon posiada nie tylko zdefiniowane położenie, ale ma także swoją stronę, czyli kierunek, z którego jest widoczny. Po szczegóły definiowania kierunków odsyłam do dokumentacji, dodam jednak, że są dwie metody na sprawienie, by polygon był widoczny z obu stron. Po pierwsze, można utworzyć drugi trójkąt, który będzie skierowany w przeciwnym kierunku. Po drugie, do wyglądu (klasa `Appearance`) obiektu 3D można przypisać obiekt



Rysunek 4. *Sześcian 3D – widok okna programu J3DRotatingCube*

Listing 4. *Definiowanie świateł*

```
// tworzymy światło - kierunkowe, świecące od góry
DirectionalLight sunlikeLight = new DirectionalLight(new Color3f(Color.RED), new Vector3f(0,-1,0));
sunlikeLight.setInfluencingBounds(new BoundingSphere());

// tworzymy światło - kierunkowe, świecące z lewej
DirectionalLight blueSideLight = new DirectionalLight(new Color3f(Color.BLUE), new Vector3f(1,0,0));
blueSideLight.setInfluencingBounds(new BoundingSphere());

// tworzymy światło - kierunkowe, świecące z tyłu
DirectionalLight greenBackLight = new DirectionalLight(new Color3f(Color.GREEN), new Vector3f(0,0,1));
greenBackLight.setInfluencingBounds(new BoundingSphere());
```

`PolygonAttributes` z ustawionym `CullFace` na `PolygonAttributes.CULL_NONE`. Warto jednak zauważyć, że większość obiektów 3D to obiekty zamknięte, wobec czego brak widoczności od środka wcale nie jest problemem, wręcz przeciwnie! Wyobraźmy sobie kamerę umiejscowioną w głowie renderowanej postaci 3D – widzielibyśmy wówczas, zamiast świata, wnętrze głowy. Widok niezbyt przydatny w grze.

Kolorowanie polygonów także może odbywać się na kilka sposobów. Istnieje obiekt `ColoringAttributes`, w którym można ustawić kolor, a następnie przypisać go do obiektu `Appearance` związanego z danym kształtem (`Shape3D`). Można także przydzielić kolor do konkretnego wierzchołka, co zrobione jest na *ścianie górnej*. Trzeba pamiętać, że funkcja wyznaczania koloru dla danego miejsca w polygonie jest w pełni modyfikowalna, może zależeć od światła, tekstury, koloru wierzchołka i koloru ustawionego w `ColoringAttributes`. Słowem, może być dość skomplikowana.

Wróćmy jednak do ściany górnej. Jej wygląd został zdefiniowany poprzez przypisanie kolorów do wierzchołków, a by to zrobić, do konstruktora `QuadArray` dodaliśmy flagę `GeometryArray.COLOR_3`, po czym przypisaliśmy kolory za pomocą metody `setColor(int index, Color3f color)`. Sposób obliczania koloru dla konkretnego miejsca na polygonie definiuje `shadeModel` w obiekcie `ColoringAttributes`, w naszym przypadku ma wartość `SHADE_GOURAUD`. Ściana górna jest widoczna z obu stron dzięki ustawieniu `PolygonAttributes.CULL_NONE`.

Ściana lewa zaś jest... po prostu biała. Tyle że ma zdefiniowane normalne – są to, w teorii, wektory prostopadłe do powierzchni. Cóż one zmieniają? Otóż kolor ściany lewej będzie zależeć od światła na nią padającego! Przyjrzyjmy się definicji świateł w naszej scenie (Listing 4)

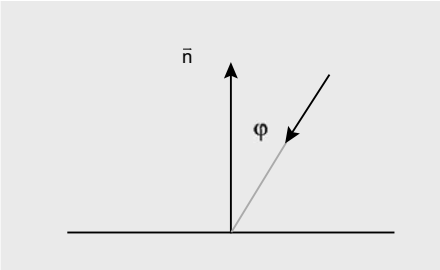
Widzimy trzy światła kierunkowe: padające z góry czerwone, z lewej niebieskie, oraz z tyłu zielone. Kierunek propagacji określony jest przez wektor podany w konstruktorze światła. Przykładowy kierunek normalnej i padania światła pokazuje ilustracja 5. Kąt φ określa wpływ światła na kolor, dla φ=0 jest on maksymalny, dla φ>PI/2 (90O) jest zerowy (powierzchnia nie jest oświetlona wcale). W naszym przykładzie mamy ścianę widoczną z obu stron, ale z uwagi na jedną normalną, będzie oświetla tylko wtedy, gdy będzie skierowana jedną stroną do światła. By uniknąć tego problemu, należy w `PolygonAttributes` przypisanych do danego Apparance i Shape-3D ustawić `setBackFaceNormalFlip(true)` . Wówczas powierzchnia będzie oświetlana z obu stron.

Przyjrzyjmy się teraz *ścianie tylnej*. Widzimy, że jej powierzchnia jest nieregularna – jest tak, ponieważ ściana ta pokryta jest teksturą. By nałożyć mapę bitową na powierzchnię, należy wykonać kilka kroków.

Po pierwsze, trzeba wczytać obrazek (do `BufferedImage`), po czym przypisać go do obiektu `Texture2D`. Po szczegóły dotyczące tej klasy odsyłam do dokumentacji J3D. Utworzoną w ten sposób teksturę poprzez obiekt `Appearance` dodajemy do `Shape3D`. Trzeba pamiętać o tym, że nie wszystkie systemy obsługują tekstury o dowolnych wymiarach – warto więc, dla bezpieczeństwa, upewniać się, by tworzone tekstury miały wymiary boków będące dowolną potęgą 2 (64, 128, 512, etc.). Wymiary boków nie muszą być równe (np. 128 x 512 także jest dozwolonym rozmiarem).

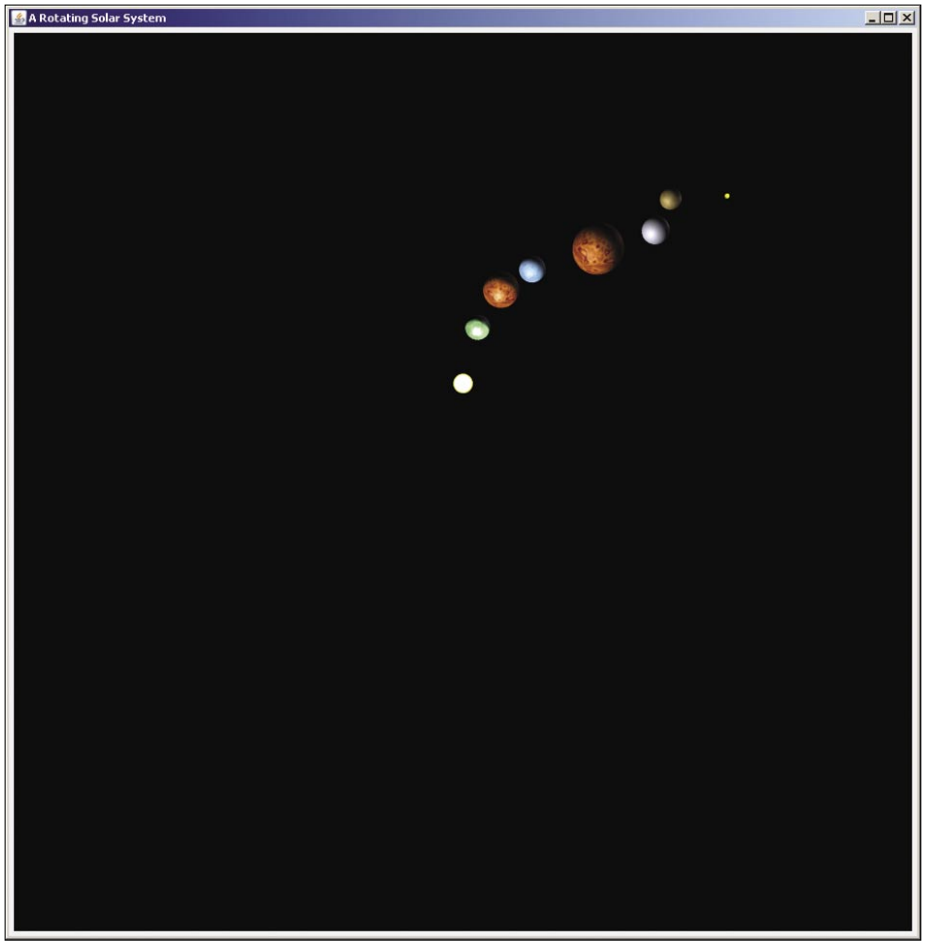
Po drugie, należy przypisać każdemu wierzchołkowi w geometrii, która ma być teksturowana, współrzędne tekstury. Pokazuje to Rysunek 6. Dla każdego punktu w świecie 3D przyporządkowujemy punkt 2D z zakresu 0 - 1. W naszym przypadku rozwiązanie jest dość oczywiste, po prostu lewy górny róg ściany to punkt (0, 1) na teksturze. Problem pojawia się dopiero w przypadku niepłaskich obiektów 3D, jak np. kula – tym jednak na razie przejmować się nie będziemy.

Dodatkowo, dla ściany tylnej ustawiliśmy normalne, dzięki czemu będzie ona mogła być oświetlana przez światła dodane do sceny. Nie jest to jednak jedyne, co trzeba zrobić. Niezbędny jest także wybór metody obliczania wartości koloru dla oświetlanej tekstury. Robi się to poprzez obiekt `TextureAttributes` przypisany do `Appearance`. Wywołujemy na nim metodę `setTextureMode(int mode)`. Po listę i opisy metod teksturowania odsyłam do dokumentacji J3D, gdyż jest ich naprawdę wiele; można nawet zdefiniować swoją własną.



Rysunek 5. *Wektory: normalna powierzchni i padające światło*

Rysunek 6. *Współrzędne tekstur w Javie 3D*



Rysunek 7. *Wirujący układ słoneczny – widok okna J3DSolarSystem*

Przy temacie teksturowania warto jeszcze wspomnieć o dwóch metodach klasy `Texture`: `setMagFilter` i `setMinFilter`. Definiują one sposób wyznaczania wartości koloru renderowanej powierzchni, jeśli na 1 piksel obrazka przypada więcej niż 1 piksel powierzchni (`magFilter`), lub na 1 piksel obrazka przypada mniej niż 1 piksel powierzchni (`minFilter`). Jeśli dysponujemy mocną maszyną, warto ustawić oba filtry na NICEST. Powierzchnie pokryte takimi teksturami staną się dużo ładniejsze.

Wirowanie

Oprócz tworzenia i konfigurowania zawartości sceny, bardzo ważnym elementem programu jest też dynamiczna jej modyfikacja. W J3DRotatingCube widzimy przykład zmiany orientacji obiektu – wirowanie. Jak tego dokonać? Najpierw należy cały obiekt, którego położenie mamy zmieniać, podpiąć do obiektu TransformGroup, któremu z kolei trzeba nadać właściwości umożliwiające tę zmianę. Są to tzw. Capabilities, których zastosowanie demonstruje fragment kodu w Listingu 5.

By móc odczytywać/zmieniać dowolny parametr dowolnego obiektu dziedziczącego po `SceneGraphObject` (nadrzędnej klasy dla wszystkich obiektów znajdujących się w drzewie świata 3D), należy nadać mu odpowiednie uprawnienia. Robi się to poprzez metodę `setCapability` – w naszym przypadku zezwalamy na zapisywanie transformacji do grupy, która odpowiada za obracanie sześcianu. W ramach nauki analizę metody `run()` zajmującej się obracaniem sześcianu pozostawiam czytelnikowi – wszelkie informacje dotyczące wykorzystywanych klas zostały już podane.

Program 3 – J3DSolarSystem – Wszechświat w Javie 3D

Program ten demonstruje wykorzystanie wszystkich poznanych dotychczas funkcjonalności oraz wprowadza kilka nowych: obracanie kamery oraz przezroczystość. Widok okna programu prezentuje Rysunek 7.

Klasą użytą w tym programie, która powinna wzbudzić największe zainteresowanie, jest `MouseRotate`. Po dodaniu do sceny oraz wskazaniu odpowiedniej `TransformGroupy`, pozwala ona na sterowanie jej transformacją poprzez przeciąganie myszą po odpowiednim obiekcie `Canvas3D`. Listing 6 przedstawia fragment kodu pokazujący tworzenie i konfigurację tej klasy.

Dodawanie tego obiektu jest bardzo proste i zarazem wydajne – sterowanie transformacją tą metodą jest szybsze niż samodzielne łapanie eventów `MouseListenerem`, dokonywanie obliczeń i aplikowanie ich w `TransformGroupie`. Wiem, bo sprawdzałem.

Oprócz `MouseRotate` warto także zapoznać się z klasami `MouseZoom`, `MouseWheelZoom` oraz `MouseTranslate`, które pozwalają na zmienianie skali transformacji oraz położenia kamery. Warto też pamiętać, o możliwości pobrania z `SimpleUniverse` `TransformGroupy` związanej z kamerą – wówczas, zamiast obracać światem, możemy obracać *głową*.

Przejdźmy teraz do rosnącej, czerwonej, przezroczystej sfery, która pojawia się w `J3DSolarSystem`. Jest to obiekt typu `Sphere`, z ustawionymi parametrami przezroczystości – czyli `TransparencyAttributes` przypisanymi do `Appearance`. Podobnie jak w przypadku wyboru metody obliczania koloru, przy inicjowaniu przezroczystości trzeba dokonać podobnego wyboru. Parametry, które ja podałem w konstruktorze `TransparencyAttributes` są dość uniwersalne i powinny działać na większości systemów – zachęcam jednak do zapoznania się z dokumentacją tej klasy i eksperymentowania, można uzyskać naprawdę ciekawe efekty.

Sugeruję dokładnie przeanalizować kod programu `J3DSolarSystem`, gdyż stanowi on swego rodzaju zestawienie wszystkiego, co zostało przedstawione w tym artykule.

Podsumowanie

Mam nadzieję, iż przekonałem Was, że Java 3D jest potężnym i zarazem łatwym w użyciu narzędziem. Przygotowanie zaprezentowanych tu programów zajęło mi może 6 godzin, przy czym starałem się robić to porządnie, pisać wyczerpujące komentarze, słowem, postępować dydaktycznie. Nie jest to bardzo długi czas.

Prawdopodobnie wielu z Was zastanawia się, czy Java 3D nadaje się do pisania gier. Moim zdaniem odpowiedź brzmi: tak. Osiągnięcie wydajności porównywalnej z językami z rodziny C jest raczej niemożliwe, wirtualna maszyna ma swoje obciążenia, jednak jestem w stanie sobie wyobrazić dynamicznego shootera 3D, działającego płynnie na współczesnych komputerach, napisanego z wykorzystaniem Javy 3D. Sęk w tym, że żadna większa firma takiego dzieła się nie podejmie – głównie dlatego, że nie ma to sensu. Istnieją dużo wydajniejsze i zapewne prostsze w użyciu platformy programistyczne.

Ewentualną komercyjną przyszłość Javy 3D widzę raczej w zaadaptowaniu biblioteki na potrzeby gier na komórki, w których platforma Java jest wszak bardzo popularna. Lecz czy producenci telefonów zaczną na poważnie rozwijać sprzęt w kierunku grafiki 3D? Czy pojawią się proste akceleratory, np. na poziomie pierwszego 3DFX? Na te pytania odpowiedzi nie znam. Ale wydaje mi się, że brzmi ona tak.

DARIUSZ WAWER

Student 4 roku Telekomunikacji na Politechnice Warszawskiej. Java Developer w firmie CC Otwarte Systemy Komputerowe.

Kontakt z autorem: dariusz.wawer@cc.com.pl

W Sieci

- *https://java3d.dev.java.net/* – oficjalna strona projektu
- *http://download.java.net/media/java3d/javadoc/1.5.0/index.html* – dokumentacja
- *http://www.j3d.org/* – portal społeczności Java 3D
- *http://pl.wikipedia.org/wiki/Grafika_3d* – podstawowe informacje o grafice 3D
- *http://pl.wikipedia.org/wiki/Mnożenie_macierzy* – mnożenie macierzy, przydatne przy do zrozumienia działania Transform3D

Listing 5. *Nadawanie uprawnień (Capabilities) do zapisywania i odczytywania transformacji*

```
// tworzymy grupę, w której będziemy zmieniać transformację, odpowiedzialną za
                    obracanie
rotateTransformGroup = new TransformGroup();

// pozwalamy na zapisywanie transformacji
rotateTransformGroup.setCapability( TransformGroup.ALLOW_TRANSFORM_WRITE);

// pozwalamy na odczytywanie transformacji
rotateTransformGroup.setCapability( TransformGroup.ALLOW_TRANSFORM_READ);
```

Listing 6. *Wykorzystanie klasy MouseRotate*

```
TransformGroup rotGr = new TransformGroup();
// dodajemy obiekt Behaviour z j3d pozwalający na obracanie sceny
MouseRotate wholeSceneMouseRotator = new MouseRotate();
// ustawiamy grupę, której transformację ma modyfikować
wholeSceneMouseRotator.setTransformGroup(rotGr);
wholeSceneMouseRotator.setSchedulingBounds(new BoundingSphere());
```