Podstawy WebGL/GLSL

May 20, 2019

1 Introduction

Celem jest podstawy WebGL/GLSL

W tym zajęciu zajmiemy się **WebGL, wersją OpenGL dla Internetu**. Three.js, omówione w poprzednich zajęciach, używa WebGL do grafiki 3D. Oczywiście trudniej jest używać WebGL bezpośrednio, ale dzięki temu masz pełną kontrolę nad sprzętem graficznym. A nauka tego będzie dobrym wprowadzeniem do programowania nowoczesnej grafiki.

Było wiele wersji OpenGL. WebGL 1.0 jest oparty na OpenGL ES 2.0, wersji przeznaczonej do użytku w systemach wbudowanych, takich jak smartfony i tablety. OpenGL ES 1.0 był bardzo podobny do OpenGL 1.1, który studiowaliśmy wcześniej. Jednak wersja 2.0 OpenGL ES wprowadziła poważne zmiany. W rzeczywistości jest to mniejszy, prostszy interfejs API, który nakłada większą odpowiedzialność na programistę. Na przykład funkcje do pracy z transformacjami, takie jak `glRotatef` i `glPushMatrix`, zostały wyeliminowane z API, dzięki czemu programista był odpowiedzialny za śledzenie transformacji. WebGL **nie używa** `glBegin / glEnd` do generowania geometrii i nie używa funkcji takich jak `glColor *` lub `glNormal *` do określania atrybutów wierzchołków. Niemniej jednak okaże się, że wiele z tego, czego nauczyliście się na poprzednich zajęciach, przeniesie się do WebGL.

Istnieją **dwie strony dowolnego programu WebGL**. Część programu napisana jest w **JavaScript**, języku programowania dla sieci. Druga część jest napisana w **GLSL**, języku do pisania programów „shader", które działają na GPU. Postaramy się zawsze jasno określić, o którym języku mówimy.

Na tym zajęciu wprowadzającym dotyczącym WebGL będziemy trzymać się podstawowej grafiki 2D. Dowiemy się o strukturze programów WebGL. Poznamy większość stron JavaScript API i nauczymy się pisać i używać prostych shaderów. Na następnym zajęciu przejdziemy do grafiki 3D, a dowiemy się znacznie więcej o GLSL.

2 Programowalny potok przetwarzania

OpenGL 1.1 używał potoku o stałej funkcji do przetwarzania grafiki (**fixed-function pipeline**). Dane są dostarczane przez program i przechodzą przez sz-

ereg etapów przetwarzania, które ostatecznie wytwarzają kolory pikseli widoczne na końcowym obrazie. Program może włączyć i wyłączyć niektóre etapy procesu, takie jak test głębokości i obliczenia oświetlenia. Ale nie ma sposobu, aby zmienić to, co dzieje się na każdym etapie. Funkcjonalność jest naprawiona.

OpenGL 2.0 wprowadził programowalny potok przetwarzania (**programmable pipeline**). Programista mógł zastąpić niektóre etapy w potoku własnymi programami. Daje to programistom pełną kontrolę nad tym, co dzieje się na tym etapie. W OpenGL 2.0 programowalność była opcjonalna; kompletny potok przetwarzania o stałej funkcji był nadal dostępny dla programów, które nie wymagały elastyczności programowania. WebGL używa programowalnego potoku i jest obowiązkowy. Nie ma możliwości używania WebGL bez pisania programów do implementacji części potoku przetwarzania grafiki.

Programy napisane jako część potoku nazywane są **shaderami**. W przypadku WebGL musisz napisać moduł cieniujący wierzchołków, który jest wywoływany raz dla każdego wierzchołka w prymitywie i fragmentujący moduł cieniujący, który jest wywoływany raz dla każdego piksela w prymitywie. Oprócz tych dwóch programowalnych etapów, potok WebGL zawiera również kilka etapów od oryginalnego potoku o stałej funkcji. Na przykład test głębi jest nadal częścią stałej funkcjonalności i może być włączony lub wyłączony w WebGL w taki sam sposób, jak w OpenGL 1.1.

Tu omówimy podstawową strukturę programu WebGL oraz sposób, w jaki dane przepływają od strony JavaScript programu do potoku graficznego i przez shadery wierzchołków i fragmentów.

Zajęcie obejmuje WebGL 1.0. Wersja 2.0 jest zgodna z wersją 1.0. WebGL 2.0 jest dostępny w przeglądarkach, w tym w Chrome i Firefox, ale jego nowymi funkcjami są rzeczy, których nie opisałbym w tej książce. Ponadto zauważam, że późniejsze wersje OpenGL wprowadziły dodatkowe etapy programowalne do potoku, oprócz shaderów wierzchołków i fragmentów, ale nie są one częścią WebGL i nie są omówione tutaj.

2.1 Kontekst graficzny WebGL

Aby korzystać z WebGL, potrzebujesz kontekstu graficznego WebGL. Kontekst graficzny to obiekt JavaScript, którego metody implementują stronę JavaScript interfejsu API WebGL. WebGL rysuje swoje obrazy w kanwasie HTML, tym samym rodzaju elementu `<canvas>`, który jest używany w interfejsie API 2D, który został opisany wcześniej. Kontekst graficzny jest powiązany z określonym kanwasem i można go uzyskać wywołując funkcję `canvas.getContext (,,webgl'')`, gdzie `canvas` jest obiektem DOM reprezentującym płótno. Kilka przeglądarek (zwłaszcza Internet Explorer i Edge) wymaga `,,experimental-webgl''` jako parametru `getContext`, więc kod do tworzenia kontekstu WebGL często wygląda mniej więcej tak:

```
canvas = document.getElementById("webglcanvas");
gl = canvas.getContext("webgl") || canvas.getContext("experimental-webgl");
```

Tutaj `gl` jest kontekstem graficznym WebGL. Ten kod może wymagać rozpakowania. To jest kod JavaScript, który wystąpiłby jako część skryptu w kodzie źródłowym strony internetowej. Pierwszy wiersz zakłada, że źródło HTML strony internetowej zawiera element canvas z id = "webglcanvas", taki jak

```
<canvas width = "800" height = "600" id = "webglcanvas"> </canvas>
```

W drugim wierszu `canvas.getContext (,,webgl'')` zwróci wartość `null`, jeśli przeglądarka internetowa nie obsługuje `,,webgl''` jako parametru `getContext`; w tym przypadku drugi operand operatora `||` zostanie oceniony. To użycie `||` to idiom JavaScript, który używa faktu, że wartość `null` jest uważana za fałszywą, gdy jest używana w kontekście boolowskim. Tak więc druga linia, w której tworzony jest kontekst WebGL, jest równoważna:

```
gl = canvas.getContext("webgl");
if ( ! gl ) {
    gl = canvas.getContext("experimental-webgl");
}
```

Jest możliwe, że `canvas.getContext (,,experimental-webgl'')` jest również `null`, jeśli przeglądarka obsługuje API kanwasu 2D, ale nie obsługuje WebGL. Co więcej, jeśli przeglądarka nie ma żadnej obsługi dla `<canvas>`, kod wyśle wyjątek. Tak więc używam funkcji podobnej do poniższej do inicjalizacji programów WebGL:

```
function init() {
    try {
        canvas = document.getElementById("webglcanvas");
        gl = canvas.getContext("webgl") ||
                        canvas.getContext("experimental-webgl");
        if ( ! gl ) {
            throw "Browser does not support WebGL";
        }
    }
    catch (e) {
          .
          .  // report the error
          .
        return;
    }
      .
      .  // other JavaScript initialization
      .
    initGL();  // a function that initializes the WebGL graphics context
}
```

W tej funkcji `canvas` i `gl` są zmiennymi globalnymi. A `initGL()` to funkcja zdefiniowana gdzie indziej w skrypcie, który inicjuje kontekst graficzny, w tym

tworzenie i instalowanie programów shaderów. Funkcję `init()` można wywołać na przykład przez program obsługi zdarzeń `onload` dla elementu `<body>` strony internetowej:

```
<body onload = "init ()">
```

Po utworzeniu kontekstu graficznego, `gl`, można go użyć do wywołania funkcji w interfejsie API WebGL. Na przykład polecenie włączenia testu głębi, który został napisany jako `glEnable (GL_DEPTH_TEST)` w OpenGL, staje się

```
gl.enable (gl.DEPTH_TEST);
```

Należy zauważyć, że zarówno funkcje, jak i stałe w interfejsie API są wywoływane w kontekście graficznym. Nazwa ,,gl'' dla kontekstu graficznego jest konwencjonalna, ale pamiętajmy, że jest to zwykła zmienna JavaScript, której nazwa zależy od programisty.

Chociaż używamy `canvas.getContext (,,experimental-webgl'')` w przykładowych programach, na ogół nie włączamy go do przykładów kodu w tekście.

2.2 Program Shadera

Rysowanie za pomocą WebGL wymaga programu shadera, który składa się z shadera wierzchołków i shadera fragmentów. Shadery są napisane w języku GLSL ES 1.0 (OpenGL Shader Language for Embedded Systems, wersja 1.0). GLSL jest oparty na języku programowania C. Moduł shadera **vertex** i moduł shadera **fragment**u to oddzielne programy, każdy z własną funkcją `main()`. Dwa shadery są kompilowane osobno, a następnie „łączone", aby stworzyć kompletny program shadera. JavaScript API dla WebGL zawiera funkcje do kompilowania shaderów, a następnie ich łączenia. Aby korzystać z funkcji, kod źródłowy shaderów musi być wierszami JavaScript. Zobaczmy, jak to działa. Utworzenie modułu **shadera wierzchołków** wymaga trzech kroków.

```
var vertexShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertexShader, vertexShaderSource);
gl.compileShader(vertexShader);
```

Funkcje, które są tutaj używane, są częścią kontekstu graficznego WebGL, `gl`, a `vertexShaderSource` to łańcuch zawierający kod źródłowy modułu shadera. Błędy w kodzie źródłowym spowodują awarię kompilacji. Aby sprawdzić błędy kompilacji, wywołaj funkcję

```
gl.getShaderParameter (vertexShader, gl.COMPILE_STATUS)
```

która zwraca wartość logiczną, aby wskazać, czy kompilacja się powiodła. W przypadku wystąpienia błędu można pobrać komunikat o błędzie za pomocą

```
gl.getShaderInfoLog (vsh)
```

który zwraca wiersz zawierający wynik kompilacji. (Dokładny format łańcucha nie jest określony przez standard WebGL. On ma być czytelny dla człowieka).

Shader fragmentu można utworzyć w ten sam sposób. Z obydwoma shaderami w ręku możesz utworzyć i połączyć program. Moduły shadera muszą być „dołączone" do obiektu programu przed połączeniem. Kod ma postać:

```
var prog = gl.createProgram ();
gl.attachShader (prog, vertexShader);
gl.attachShader (prog, fragmentShader);
gl.linkProgram (prog);
```

Nawet jeśli shadery zostały pomyślnie skompilowane, mogą wystąpić błędy, gdy zostaną połączone w kompletny program. Na przykład moduł shadera wierzchołków i fragmentów może współdzielić pewne rodzaje zmiennych. *Jeśli oba programy deklarują takie zmienne o tej samej nazwie, ale z różnymi typami, w czasie połączenia wystąpi błąd.* Sprawdzanie błędów łącza jest podobne do sprawdzania błędów kompilacji w modułach shadera.

Kod do tworzenia programu shadera jest zawsze taki sam, więc wygodnie jest spakować go do funkcji wielokrotnego użytku. Oto funkcja, której używamy w przykładach:

```
/**
 * Creates a program for use in the WebGL context gl, and returns the
 * identifier for that program.  If an error occurs while compiling or
 * linking the program, an exception of type String is thrown.  The error
 * string contains the compilation or linking error.
 */
function createProgram(gl, vertexShaderSource, fragmentShaderSource) {
   var vsh = gl.createShader( gl.VERTEX_SHADER );
   gl.shaderSource( vsh, vertexShaderSource );
   gl.compileShader( vsh );
   if ( ! gl.getShaderParameter(vsh, gl.COMPILE_STATUS) ) {
      throw "Error in vertex shader:  " + gl.getShaderInfoLog(vsh);
   }
   var fsh = gl.createShader( gl.FRAGMENT_SHADER );
   gl.shaderSource( fsh, fragmentShaderSource );
   gl.compileShader( fsh );
   if ( ! gl.getShaderParameter(fsh, gl.COMPILE_STATUS) ) {
      throw "Error in fragment shader:  " + gl.getShaderInfoLog(fsh);
   }
   var prog = gl.createProgram();
   gl.attachShader( prog, vsh );
   gl.attachShader( prog, fsh );
   gl.linkProgram( prog );
   if ( ! gl.getProgramParameter( prog, gl.LINK_STATUS) ) {
      throw "Link error in program:  " + gl.getProgramInfoLog(prog);
   }
```

```
    return prog;
}
```

Jest jeszcze jeden krok: musisz powiedzieć kontekstowi WebGL, aby użyć programu. Jeśli `prog` jest identyfikatorem programu zwracanym przez powyższą funkcję, odbywa się to przez wywołanie

```
gl.useProgram (prog);
```

Możliwe jest utworzenie kilku programów shaderów. Następnie możesz w dowolnym momencie przełączyć się z jednego programu na inny, wywołując `gl.useProgram`, nawet w środku renderowania obrazu. (Na przykład Three.js używa innego programu dla każdego typu materiału).

Zaleca się tworzenie dowolnych programów shadera, które są potrzebne w ramach inicjalizacji. Chociaż `gl.useProgram` to szybka operacja, kompilacja i łączenie są raczej powolne, dlatego lepiej jest unikać tworzenia nowych programów podczas rysowania obrazu.

Shadery i programy, które nie są już potrzebne, można usunąć, aby zwolnić zasoby, które konsumują. Użyj funkcji `gl.deleteShader(shader)` i `gl.deleteProgram (program)`.

2.3 Przepływ danych w potoku przetwarzania

Potok graficzny WebGL renderuje obraz. Dane definiujące obraz pochodzą z JavaScript. Podczas przechodzenia przez potok dane są przetwarzane przez bieżący moduł shadera wierzchołków i shader fragmentów, a także przez etapy o ustalonej funkcji potoku. Musisz zrozumieć, w jaki sposób dane są umieszczane przez JavaScript w potoku i jak dane są przetwarzane podczas przechodzenia przez potok.

Podstawową operacją w WebGL jest narysowanie prymitywu geometrycznego. WebGL używa tylko **siedmiu prymitywów** OpenGL. Prymitywy do rysowania quadów i wielokątów zostały usunięte. Pozostałe prymitywy rysują **punkty, segmenty linii i trójkąty**. W WegGL siedem typów prymitywów jest identyfikowanych przez stałe `gl.POINTS, gl.LINES, gl.LINE_STRIP, gl.LINE_LOOP, gl.TRIANGLES, gl.TRIANGLE_STRIP i gl.TRIANGLE_FAN`, gdzie `gl` jest kontekstem grafiki WebGL.

Gdy WebGL jest używany do rysowania prymitywu, istnieją dwie ogólne kategorie danych, które można dostarczyć dla prymitywu. Dwa rodzaje danych określa się jako **zmienne atrybutów** (lub po prostu „atrybuty") i **zmienne jednolite** (lub po prostu „uniforms"). Prymityw jest zdefiniowany przez jego typ i listę wierzchołków. Różnica między atrybutami i uniformami polega na tym, że **zmienna jednolita ma pojedynczą wartość, która jest taka sama dla całego prymitywu**, podczas gdy **wartość zmiennej atrybutu może być różna dla różnych wierzchołków**.

Jeden atrybut, który jest zawsze określony, to **współrzędne wierzchołka**. Współrzędne wierzchołków muszą być atrybutem, ponieważ każdy wierzchołek w prymitywie będzie miał swój własny zestaw współrzędnych. Innym możliwym

atrybutem jest **kolor**. Widzieliśmy, że OpenGL pozwala określić inny kolor dla każdego wierzchołka prymitywu. Możesz zrobić to samo w WebGL i w tym przypadku kolor będzie atrybutem. Z drugiej strony, może chcesz, aby cały prymityw miał taki sam „jednolity" kolor; w takim przypadku **kolor może być zmienną jednolitą**. Inne wartości, które mogą być atrybutami lub jednolitymi, w zależności od potrzeb, obejmują **normalne wektory** i **właściwości materiału**. **Współrzędne tekstury**, jeśli są używane, prawie na pewno będą atrybutem, ponieważ nie ma sensu, aby wszystkie wierzchołki w prymitywie miały takie same współrzędne tekstury. Jeśli **transformacja geometryczna** ma być zastosowana do prymitywu, naturalnie byłaby reprezentowana jako zmienna jednolita.

Ważne jest jednak, aby zrozumieć, że WebGL nie ma żadnych predefiniowanych atrybutów, nawet jednego dla współrzędnych wierzchołków. W programowalnym potoku używane atrybuty i uniforms zależą wyłącznie od programisty. Jeśli chodzi o WebGL, **atrybuty są tylko wartościami, które są przekazywane do modułu shadera wierzchołków**. Uniforms mogą być przekazywane do modułu shadera wierzchołków, modułu shadera fragmentu lub obu. WebGL nie przypisuje znaczenia wartościom. Znaczenie jest całkowicie zdeterminowane przez to, co shadery robią z wartościami. Zestaw atrybutów i uniforms używanych do rysowania prymitywu jest określony przez kod źródłowy shaderów używanych podczas rysowania prymitywu.

Aby to zrozumieć, musimy przyjrzeć się, co dzieje się w potoku nieco bardziej szczegółowo. Podczas rysowania prymitywu program JavaScript określa wartości dla dowolnych atrybutów i uniforms w programie shadera. Dla każdego atrybutu określa tablicę wartości, po jednej dla każdego wierzchołka. Dla każdego uniforms określi pojedynczą wartość. Wszystkie wartości zostaną wysłane do GPU przed narysowaniem prymitywu. Podczas rysowania prymitywu GPU wywołuje shader wierzchołków dla każdego wierzchołka. Wartości atrybutów dla przetwarzanego wierzchołka są przekazywane jako dane wejściowe do modułu shadera wierzchołków. Wartości zmiennych jednolitych są również przekazywane do modułu shadera wierzchołków. Zarówno atrybuty, jak i zmienne jednolite są reprezentowane jako zmienne globalne w module shadera, których wartości są ustawiane przed wywołaniem modułu shadera.

Jako jedno z wyjść, moduł shadera wierzchołków musi określić współrzędne wierzchołka w układzie współrzędnych klipu. Robi to, przypisując wartość do specjalnej zmiennej o nazwie `gl_Position`. Pozycja jest często obliczana przez zastosowanie transformacji do atrybutu, który reprezentuje współrzędne w układzie współrzędnych obiektu, ale dokładnie, jak obliczana jest pozycja, zależy od programisty.

Po obliczeniu pozycji wszystkich wierzchołków w prymitywie, faza o ustalonej funkcji w potoku wycina części prymitywu, którego współrzędne znajdują się poza zakresem ważnych współrzędnych klipu (-1 do 1 wzdłuż każdej osi współrzędnych). Prymityw jest następnie rasteryzowany; to znaczy, określa się, które piksele leżą wewnątrz prymitywu. Shader fragmentu jest następnie wywoływany dla każdego piksela, który leży w prymitywie. Moduł shadera fragmentu ma dostęp do zmiennych jednolitych (ale nie do atrybutów). Może również użyć specjal-

nej zmiennej o nazwie `gl_FragCoord`, która zawiera współrzędne piksela klipu. Współrzędne pikseli są obliczane przez interpolację wartości `gl_Position`, które zostały określone przez moduł shadera wierzchołków. Interpolacja jest wykonywana przez inny etap o stałej funkcji, który występuje między modułem shadera wierzchołka a modułem shadera fragmentu.

Inne wartości oprócz współrzędnych mogą działać w ten sam sposób. Oznacza to, że moduł shadera wierzchołków oblicza wartość w każdym wierzchołku prymitywu. Interpolator pobiera wartości w wierzchołkach i oblicza wartość dla każdego piksela w prymitywie. Wartość danego piksela jest następnie wprowadzana do modułu shadera fragmentu, gdy wywoływany jest moduł shadera w celu przetworzenia tego piksela. Na przykład **kolor** w OpenGL jest następujący: kolor wewnętrznego piksela prymitywu jest obliczany przez interpolację koloru na wierzchołkach. W GLSL ten wzorzec jest implementowany przy użyciu zmieniających się zmiennych (**varying variables**).

Zmieniająca się zmienna jest zadeklarowana zarówno w shaderze wierzchołku, jak i w shaderze fragmentu. Moduł shadera wierzchołków jest odpowiedzialny za przypisanie wartości zmieniającej się zmiennej. Interpolator pobiera wartości z modułu shadera wierzchołków i oblicza wartość dla każdego piksela. Gdy shader fragmentu jest wykonywany dla piksela, wartością zmieniającej się zmiennej jest interpolowana wartość tego piksela. Moduł shadera fragmentu może wykorzystać tę wartość w swoich własnych obliczeniach. (W nowszych wersjach GLSL termin „zmieniająca się zmienna" został zastąpiony przez **„zmienną wyjściową"** w shaderze wierzchołka i **„zmiennej wyjściowej"** w shaderze fragmentu.)

Istnieją różne zmienne do przekazywania danych z modułu shadera wierzchołków do modułu shadera fragmentu. Są one zdefiniowane w kodzie źródłowym modułu shadera. Nie są używane ani nie są wymieniane po stronie JavaScript interfejsu API. Zauważ, że to zależy od programisty, aby zdecydować, jakie zmienne zdefiniować i co z nimi zrobić.

Prawie dotarliśmy do końca potoku przetwarzania. Po tym wszystkim, **zadaniem modułu shadera fragmentu jest po prostu określenie koloru piksela**. Robi to, przypisując wartość do specjalnej zmiennej o nazwie `gl_FragColor`. Ta wartość zostanie następnie użyta na pozostałych etapach stałej funkcji potoku przetwarzania.

Podsumowując: strona JavaScript programu wysyła wartości atrybutów i zmiennych jednolitych do GPU, a następnie wydaje polecenie narysowania prymitywu. GPU wykonuje shader wierzchołków dla każdego wierzchołka. Moduł shadera wierzchołków może używać wartości atrybutów i zmiennych jednolitych. Przypisuje wartości do `gl_Position` i do wszelkich zmiennych, które istnieją w module shadera. Po przycięciu, rasteryzacji i interpolacji GPU wykonuje shader fragmentu na każdy piksel w prymitywie. Moduł shadera fragmentu może używać wartości zmieniających się zmiennych, zmiennych jednolitych i `gl_FragCoord`. Oblicza wartość dla `gl_FragColor`. Ten diagram podsumowuje przepływ danych:

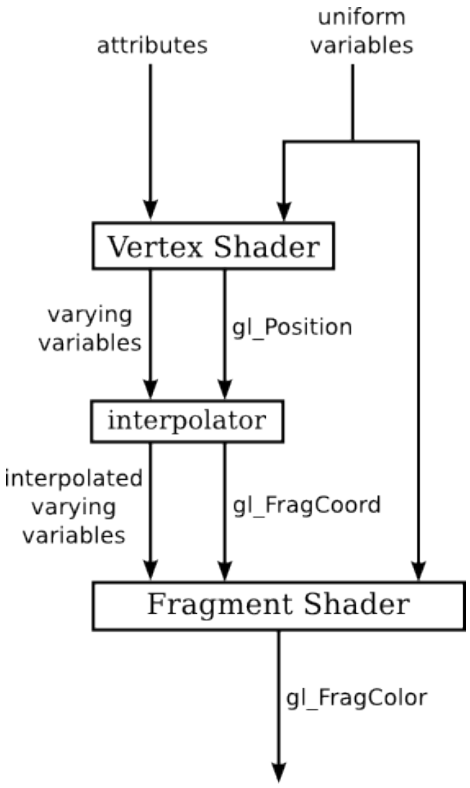

Figure 1: Diagram nie jest kompletny. Jest jeszcze kilka specjalnych zmiennych, o których nie wspomniałem. I jest ważne pytanie, jak używane są tekstury. Ale jeśli zrozumiesz diagram, dobrze zaczniesz rozumieć WebGL.

Literatura

Uwaga!
W języku angielskim

- książka interakcyjna: WebGL/GLSL `http://math.hws.edu/graphicsbook/c6/index.html` (rozdział 6)

3 Zadanie

Program w `lab11.html` pokazuje wiele ruchomych czerwonych kwadratów, które odbijają się od krawędzi płótna. Płótno wypełnia cały obszar zawartości przeglądarki internetowej. Kwadraty odpowiadają również myszą: Jeśli klikniesz lewym przyciskiem myszy lub klikniesz lewym przyciskiem myszy i przeciągniesz na płótnie, cały kwadrat będzie kierowany w stronę pozycji myszy. Jeśli klikniesz lewym przyciskiem myszy, dane punktów zostaną ponownie zainicjowane, więc zaczną się od środka. Możesz wstrzymać i ponownie uruchomić animację, naciskając spację.

Kwadraty są w rzeczywistości częścią jednego prymitywu WebGL typu `gl.POINTS`. Każdy kwadrat odpowiada jednemu z wierzchołków pierwotnego. Oczywiście renderowanie jest wykonywane przez moduł shadera wierzchołków i moduł shadera fragmentu. Kod źródłowy shaderów jest w dwóch fałszywych „skryptach" w górnej części pliku html.

Będziesz modyfikował kod modułu shadera i kod JavaScript, aby zaimplementować kilka różnych stylów dla prymitywu punktu. Na przykład możliwe będzie rysowanie kwadratów w różnych kolorach, rysowanie wielokątów zamiast kwadratów i tak dalej. Użytkownik będzie kontrolował program, naciskając klawisze na klawiaturze. Do ciebie należy decyzja, których klawiszy użyć, ale proszę udokumentować interfejs w odpowiednim komentarzu do funkcji `doKey()` lub na górze programu.

Program ma dwie funkcje, nad którymi będziesz musiał pracować: Funkcja `initGL()` jest wywoływana, gdy program jest uruchamiany po raz pierwszy, a funkcje `updateForFrame()` i `render()` są wywoływane dla każdej ramki animacji. Ten sam zestaw poleceń byłby legalny we wszystkich tych poleceniach, ale `initGL()` jest najlepszym miejscem do ustawiania rzeczy, które nie zmienią się w trakcie działania programu, takich jak położenie zmiennych i zmiennych atrybutów w module shadera; `updateForFrame()` jest przeznaczony do aktualizacji zmiennych JavaScript, które zmieniają się z ramki na ramkę; i `render()` ma na celu wykonanie rzeczywistego rysunku WebGL ramki.

Atrybut koloru

W oryginalnej wersji programu wszystkie kwadraty są czerwone. Pierwsze ćwiczenie polega na umożliwieniu przypisania innego koloru do każdego kwadratu. Ponieważ kwadraty są naprawdę wierzchołkami w pojedynczym prymitywie typu `gl.POINTS`, można użyć zmiennej atrybutu dla koloru. Atrybut może mieć inną wartość dla każdego wierzchołka.

Pierwszym zadaniem jest dodanie zmiennej kolorowej typu `vec3` do modułu shadera wierzchołka i użycie wartości atrybutu do pokolorowania kwadratów. Będziesz także musiał pracować po stronie JavaScript. Będziesz potrzebował Float32Array do przechowywania wartości kolorów po stronie JavaScript, a będziesz potrzebował bufora WebGL dla tego atrybutu. Program ma już jeden atrybut, który jest używany do współrzędnych wierzchołków. Będziesz robił coś podobnego do atrybutu `color` (poza tym, że możesz to zrobić w `initGL()`, ponieważ wartości kolorów nie zmienią się po ich utworzeniu). Można użyć losowych wartości w zakresie od 0,0 do 1,0 dla składników koloru.

Po uruchomieniu wielokolorowych kwadratów powinieneś ustawić kolory jako opcjonalne. Możesz włączać i wyłączać użycie tablicy wartości atrybutów za pomocą następujących poleceń, gdzie `a_color_loc` to identyfikator atrybutu `color` w programie shader:

```
gl.enableVertexAttribArray (a_color_loc); // użyj bufora atrybutów kolorów
gl.disableVertexAttribArray (a_color_loc); // nie używaj bufora
```

Gdy tablica atrybutów jest włączona, każdy wierzchołek otrzymuje swój własny kolor z buforu atrybutów. Gdy tablica atrybutów jest wyłączona, wszystkie wierzchołki otrzymują ten sam kolor, a tę wartość można ustawić za pomocą rodziny funkcji `gl.vertexAttrib *`. Na przykład, aby ustawić wartość używaną, gdy tablica atrybutów kolorów jest wyłączona, można użyć

```
gl.vertexAttrib3f (a_color_loc, 1, 0, 0); // ustaw kolor attrbute na czerwony
```

Pozwól użytkownikowi na naciśnięcie określonego klawisza, aby włączyć lub wyłączyć losowe kolory. Program ma funkcję `doKey()`, która jest już skonfigurowana do reagowania na wprowadzanie z klawiatury. Będziesz dodawać do programu kilka typów interakcji z klawiaturą. Aby odpowiedzieć na klawiszę, musisz znać numeryczny kod klawiszy. Funkcja `doKey()` wysyła kod do konsoli za każdym razem, gdy użytkownik uderza klawisz, i możesz użyć tej funkcji, aby odkryć wszystkie inne kody klawiszy, których potrzebujesz.

Styl punktów

Powinieneś **dodać opcję używania stylu wyświetlania dla punktów w postaci wielokąta**. Pozwól użytkownikowi wybrać styl za pomocą klawiatury; na przykład, naciskając klawisze numeryczne.

Style będą musiały zostać zaimplementowane w shaderze fragmentu, a będziesz potrzebował nowej zmiennej jednolitej, aby powiedzieć modułowi shadera fragmentu, którego stylu użyć. Dodaj jednolitą zmienną typu `int` do shadera fragmentu, aby kontrolować styl punktu, i dodaj kod do modułu cieniującego fragmentu, aby zaimplementować różne style. Będziesz także musiał dodać zmienną po stronie JavaScript dla lokalizacji zmiennej jednolitej, a będziesz musiał wywołać `glUniform1i`, gdy chcesz zmienić styl.

Naprzykład, żeby narysować punkt jako dysk, odrzucając niektóre piksele:

```
float dist = distance( vec2(0.5), gl_PointCoord );
if (dist > 0.5) {
    discard;
}
```

Powinieneś również wykorzystać przezroczystość alfa w niektórych stylach. Aby umożliwić korzystanie ze składnika alpha, musisz dodać następujące linie do funkcji initGL ():

```
gl.enable(gl.BLEND);
gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
```

Dzięki tym ustawieniom wartość alfa koloru będzie używana do przezroczystości w zwykły sposób. W szczególności jeden z twoich stylów powinien pokazywać punkt jako wielokąt, który zanika z całkowicie nieprzezroczystego w środku wielokątu do całkowicie przezroczystego na krawędzi.