Wstęp do WebGL,
obsługa klawiatury i myszy, transformacje układu współrzędnych, kamera

1. Wstęp:

WebGL to wieloplatformowy, otwarty standard niskopoziomowego interfejsu API grafiki 3D opartego na OpenGL ES wykorzystujący element HTML5 - Canvas. WebGL (Web Graphics Library) to API języka JavaScript służące do renderowania grafiki 3D i 2D poprzez kompatybilną przeglądarkę bez używania wtyczek. WebGL został oparty na API OpenGL ES 2.0.

WebGL wprowadza do sieci 3D bez wtyczek, jest zaimplementowane bezpośrednio w przeglądarce. Główni dostawcy przeglądarek Apple (Safari), Google (Chrome), Microsoft (Edge) i Mozilla (Firefox) są członkami grupy roboczej WebGL.

2. Oprogramowanie:

WebGL aktualnie jest wspierany przez Firefox, Google Chrome, Opera, Safari i Internet Explorer. Urządzenie na którym uruchamiany jest program musi spełniać wymagania sprzętowe oraz obsługiwać OpenGL w wersji nie niższej niż 2.0.

3. Ćwiczenie:

Należy przygotować projekt pozwalający na rysowanie sceny z wykorzystaniem potoku renderowania WebGL. Do utworzenia sceny i zapewnienia interakcji z użytkownikiem należy wykorzystać komponent canvas w szablonie strony HTML5.

4. Wykonanie zadania:

1. Przygotować szablon strony HTML5 oraz obiekt canvas z określeniem identyfikatora, wymiarów oraz funkcji startowej JavaScript (plik `start.html`)

```
<!DOCTYPE html>
<html>
   <head>
      <title>Wstęp WebGL w HTML5</title>
      <script type="text/javascript" src="start.js"> </script>
      <link rel="stylesheet" href="style.css"/>
   </head>

<body onload="start();">
   <canvas width="1920" height="1080" id="my_canvas" ></canvas>
</body>

</html>
```

2. **W pliku** `style.css` **wyłączyć marginesy strony, ustawić rozmiar obiektu canvas na 100%,**

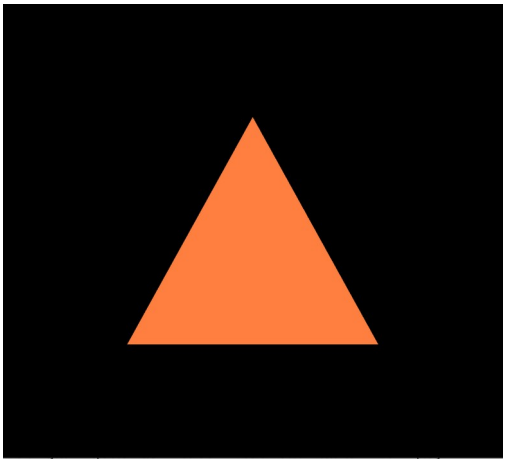
```
body
{
   margin: 0;
   padding: 0;
}

canvas
{
   position: absolute;
   width: 100%;
   height: 100%;
   overflow: hidden
}
```

3. **W pliku** `start.js` **w funkcji** `start()` **umieścić kod odpowiedzialny za**
    1. zainicjowanie kontekstu renderowania,
    2. utworzenie shaderów, ich skompilowania,
    3. utworzenia programu i dołączenia shaderów
    4. zlinkowania programu

```
function start() {
    const canvas = document.getElementById("my_canvas");
//Inicialize the GL contex
    const gl = canvas.getContext("webgl2");
    if (gl === null) {
    alert("Unable to initialize WebGL. Your browser or machine may not support it.");
    return;
}

console.log("WebGL version: " + gl.getParameter(gl.VERSION));
console.log("GLSL version: " + gl.getParameter(gl.SHADING_LANGUAGE_VERSION));
console.log("Vendor: " + gl.getParameter(gl.VENDOR));

const vs = gl.createShader(gl.VERTEX_SHADER);
const fs = gl.createShader(gl.FRAGMENT_SHADER);
const program = gl.createProgram();


    const vsSource =
          `#version 300 es
          precision highp float;
          in vec2 position;
          void main(void)
          {
             gl_Position = vec4(position, 0.0, 1.0);
          }
          `;

          const fsSource =
          `#version 300 es
          precision highp float;
          out vec4 frag_color;
          void main(void)
          {
             frag_color = vec4(1.0, 0.5, 0.25, 1.0);
          }
          `;


//compilation vs
      gl.shaderSource(vs, vsSource);
      gl.compileShader(vs);
      if(!gl.getShaderParameter(vs, gl.COMPILE_STATUS))
                {
                    alert(gl.getShaderInfoLog(vs));
                }
```

```
//compilation fs
      gl.shaderSource(fs, fsSource);
      gl.compileShader(fs);
      if(!gl.getShaderParameter(fs, gl.COMPILE_STATUS))
                {
                    alert(gl.getShaderInfoLog(fs));
                }

   if(!gl.getShaderParameter(fs, gl.COMPILE_STATUS))
   {
      alert(gl.getShaderInfoLog(fs));
   }

   gl.attachShader(program,vs);
   gl.attachShader(program,fs);
   gl.linkProgram(program);

   if(!gl.getProgramParameter(program, gl.LINK_STATUS))
   {
      alert(gl.getProgramInfoLog(program));
   }

const vertices =
[
-0.5, -0.5,
0.0, 0.5,
0.5, -0.5
];


const buffer = gl.createBuffer();
   gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
   gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
   const position = gl.getAttribLocation(program, "position");
   gl.enableVertexAttribArray(position);
   gl.vertexAttribPointer(position, 2, gl.FLOAT, false, 0, 0);


function draw(){
gl.clearColor(0, 0, 0, 1);
gl.clear(gl.COLOR_BUFFER_BIT);
gl.useProgram(program);
gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
gl.drawArrays(gl.TRIANGLES, 0, 3);
}
window.requestAnimationFrame(draw);
}
```

4. **Po prawidłowym wykonaniu po uruchomieniu pliku start.html zostanie wyświetlone okno:**



5. **Obsługa myszy, można użyć:**
MouseUp lub MouseDown:

```
// Add the event listeners for mousedown, mousemove, and mouseup
window.addEventListener('mousedown', e => {
  x = e.offsetX;
  y = e.offsetY;
   alert("x ="+x);
   alert("y ="+y);
});
```

6. **Obsługa klawiatury:**

```
// Add the event listeners for keydown, keyup
window.addEventListener('keydown', function(event) {
  switch (event.keyCode) {
    case 37: // Left
      alert('Lewo');
    break;

    case 38: // Up
      alert('Góra');
    break;

    case 39: // Right
      alert('Prawo');
    break;

    case 40: // Down
      alert('Dół');
    break;
  }
}, false);
```

7. **Transformacje układu współrzędnych, kamera**

WebGL nie obsługuje pojęcia kamery, ale możliwe jest jej zasymulowanie, np. dla translacji poprzez przesunięcie obiektów w przeciwnym kierunku. W tym celu należy skonfigurować kamerę np. FPS, która pozwala swobodnie poruszać się po scenie 3D. Konieczne jest również odpowiednie sprawdzanie stanu klawiszy sterujących.

8. **Widok 3D:**

W celu uzyskania ostatecznego położenia punktów konieczne jest przekształcenie ich współrzędnych z uwzględnieniem macierzy modelu, widoku i projekcji, co w shaderze może wyglądać następująco:

```
//vs Deklaracja zmiennych:
        uniform mat4 model;
        uniform mat4 view;
        uniform mat4 proj;

//W funkcji main określenie położenia punktów:

        gl_Position = proj * view * model * vec4(position, 1.0);
```

9. **Biblioteki pomocnicze:**

W języku Javascript dostępnie są biblioteki pomocnicze ułatwiające pracę z macierzami i wektorami.

Pierwsza z nich to np. gl-matrix

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/gl-matrix/2.8.1/gl-matrix-min.js"></script
```

Druga to eksperymentalna implementacja JavaScript biblioteki OpenGL Mathematics (GLM) C++

```
<script src='https://git.io/glm-js.min.js'></script>
```

Powyższe kody należy umieścić w pliku html w sekcji `<head>`

10. **Macierz modelu:**

Aby rozpocząć rysowanie w 3D, należy stworzyć macierz modelu. Macierz modelu, może składać się z przekształcenia translacji, skalowania, rotacji, które mają zostać wykonane w celu przekształcenia wszystkich wierzchołków obiektu w globalną przestrzeń świata. Macierz taka może wyglądać następująco:

```
const model = mat4.create();
const kat_obrotu = -25 * Math.PI / 180;   // in radians
mat4.rotate(model,  model,  kat_obrotu,   [0, 0, 1]);
```

Wysłanie do shadera:

```
let uniModel = gl.getUniformLocation(program, 'model');
gl.uniformMatrix4fv( uniModel, false, model);
```

11. Kamera / Widok (Macierz widoku):

Kiedy mówimy o przestrzeni kamery/widoku, mówimy o wszystkich współrzędnych wierzchołków, widzianych z perspektywy kamery, traktowanej jako punkt początkowy sceny (centrum projekcji). Macierz widoku przekształca wszystkie współrzędne świata na współrzędne widoku, które są zależne od położenia kamery i jej kierunku. Aby zdefiniować kamerę, trzeba znać jej pozycję w przestrzeni świata, kierunek w który jest zwrócona i wektor jej pionu.

Biblioteka glMatrix umożliwia wygenerowanie macierzy widoku wykorzystując funkcję LookAt.

```
const view = mat4.create();
mat4.lookAt(view, [0,0,3], [0,0,-1], [0,1,0]);
```

Aby kamera mogła się poruszać trzeba zdefiniować zmienne ją opisujące:

```
let cameraPos = glm.vec3(0,0,3);
let cameraFront = glm.vec3(0,0,-1);
let cameraUp = glm.vec3(0,1,0);
let obrot=0.0;
```

Sposób tworzenia macierzy widoku:

```
let cameraFront_tmp = glm.vec3(1,1,1);
```

Dla wciśniętego klawisza np. w górę, zmienne opisujące położenie kamery należy zaktualizować:

```
cameraPos.x+=cameraSpeed * cameraFront.x;
cameraPos.y+=cameraSpeed * cameraFront.y;
cameraPos.z+=cameraSpeed * cameraFront.z;
```

Dla klawisza obrotu np. w lewo:

```
obrot -= cameraSpeed;
cameraFront.x =  Math.sin(obrot);
cameraFront.z = -Math.cos(obrot);
```

Ostatecznie należy wysłać macierz do karty graficznej, aby uwzględnić ją w shaderach:

```
cameraFront_tmp.x = cameraPos.x+cameraFront.x;
cameraFront_tmp.y = cameraPos.y+cameraFront.y;
cameraFront_tmp.z = cameraPos.z+cameraFront.z;

mat4.lookAt(view, cameraPos, cameraFront_tmp, cameraUp);
gl.uniformMatrix4fv( uniView, false, view);
```

12. Macierz projekcji:

Macierz projekcji wykonuje przekształcenie rzutujące, które przekształca wierzchołki do tzw. układu współrzędnych przycinania (clip coordinates). W tym przypadku dla rzutowania perspektywicznego pomocna jest biblioteka glMatrix i funkcja perspective. Gdzie parametry to:

```
//macierz mnożona przez macierz przekształcenia,
//kąt rozwarcia kamery,
//stosunek szerokości do wysokości okna,
//odległość przedniej płaszczyzny obcinającej,
//odległość tylnej płaszczyzny obcinającej,
```

Kod dla macierzy:

```
const proj = mat4.create();
mat4.perspective(proj, 60 * Math.PI / 180, gl.canvas.clientWidth / gl.canvas.clientHeight, 0.1,
100.0 );
```

Wysłanie macierzy do karty graficznej:

```
let uniProj = gl.getUniformLocation(program, 'proj');
gl.uniformMatrix4fv( uniProj, false, proj);
```

13. Sprawdzanie stanu klawiszy:

W celu sprawdzania stanu klawiszy w każdej klatce animacji, konieczne jest wykorzystanie innego podejścia niż metody `window.addEventListener('keydown', function(event)...` Można zdefiniować funkcję dla obsługi zdarzeń keydown w oknie:

```
var pressedKey = {};
window.onkeyup = function(e) { pressedKey[e.keyCode] = false; }
window.onkeydown = function(e) { pressedKey[e.keyCode] = true; }
```

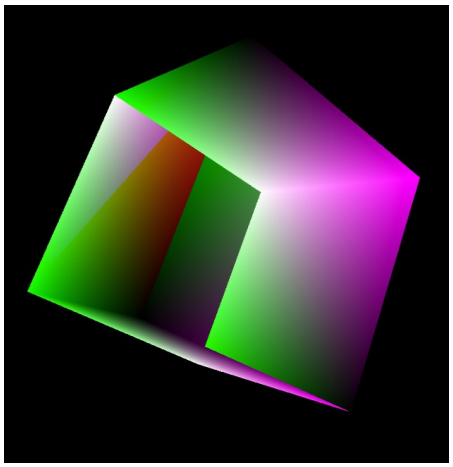
co pozwala na umieszczenie funkcji sprawdzającej klawisze w pętli głównej programu:

```
function ustaw_kamere() {

let cameraSpeed = 0.02;


        if (pressedKey["38"]) //Up
        {
        cameraPos.x+=cameraSpeed * cameraFront.x;
        cameraPos.y+=cameraSpeed * cameraFront.y;
        cameraPos.z+=cameraSpeed * cameraFront.z;
        }
.
.       //reszta klawiszy
.
.
.
        //wyślij macierz do karty
}
```

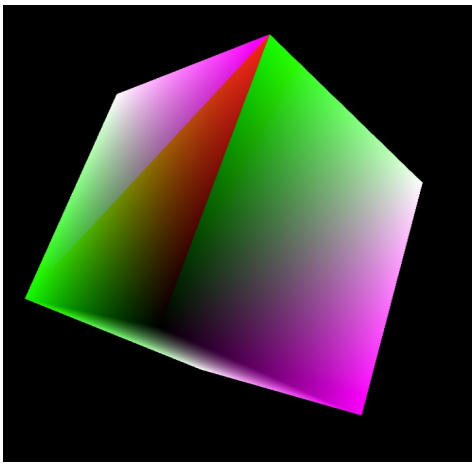
14. Bufor głębokości:

Należy umożliwić włączanie i wyłączanie z-bufora za pomocą wybranego klawisza w celu prawidłowego wyświetlania brył. Obiekty bliższe zasłaniają dalsze.

```
if (gl.isEnabled(gl.DEPTH_TEST))
        gl.disable(gl.DEPTH_TEST);
else
        gl.enable(gl.DEPTH_TEST);
```



Wyłączony z-bufor



Włączony z-bufor

15. Tablica wierzchołków przedstawiająca sześcian wraz z kolorami:

W przedstawionym przypadku dane wierzchołkowe zawierają współrzędne wierzchołków oraz odpowiadające im kolory RGB. Struktura każdej linii to (x,y,z,R,G,B) dla każdego vertexa.

```
function kostka() {

        let punkty_ = 36;

        var vertices = [
        -0.5, -0.5, -0.5,  0.0, 0.0, 0.0,
         0.5, -0.5, -0.5,  0.0, 0.0, 1.0,
         0.5,  0.5, -0.5,  0.0, 1.0, 1.0,
         0.5,  0.5, -0.5,  0.0, 1.0, 1.0,
        -0.5,  0.5, -0.5,  0.0, 1.0, 0.0,
        -0.5, -0.5, -0.5,  0.0, 0.0, 0.0,

        -0.5, -0.5,  0.5,  0.0, 0.0, 0.0,
         0.5, -0.5,  0.5,  1.0, 0.0, 1.0,
```

```
             0.5,  0.5,  0.5,  1.0, 1.0, 1.0,
             0.5,  0.5,  0.5,  1.0, 1.0, 1.0,
            -0.5,  0.5,  0.5,  0.0, 1.0, 0.0,
            -0.5, -0.5,  0.5,  0.0, 0.0, 0.0,

            -0.5,  0.5,  0.5,  1.0, 0.0, 1.0,
            -0.5,  0.5, -0.5,  1.0, 1.0, 1.0,
            -0.5, -0.5, -0.5,  0.0, 1.0, 0.0,
            -0.5, -0.5, -0.5,  0.0, 1.0, 0.0,
            -0.5, -0.5,  0.5,  0.0, 0.0, 0.0,
            -0.5,  0.5,  0.5,  1.0, 0.0, 1.0,

             0.5,  0.5,  0.5,  1.0, 0.0, 1.0,
             0.5,  0.5, -0.5,  1.0, 1.0, 1.0,
             0.5, -0.5, -0.5,  0.0, 1.0, 0.0,
             0.5, -0.5, -0.5,  0.0, 1.0, 0.0,
             0.5, -0.5,  0.5,  0.0, 0.0, 0.0,
             0.5,  0.5,  0.5,  1.0, 0.0, 1.0,

            -0.5, -0.5, -0.5,  0.0, 1.0, 0.0,
             0.5, -0.5, -0.5,  1.0, 1.0, 1.0,
             0.5, -0.5,  0.5,  1.0, 0.0, 1.0,
             0.5, -0.5,  0.5,  1.0, 0.0, 1.0,
            -0.5, -0.5,  0.5,  0.0, 0.0, 0.0,
            -0.5, -0.5, -0.5,  0.0, 1.0, 0.0,

            -0.5,  0.5, -0.5,  0.0, 1.0, 0.0,
             0.5,  0.5, -0.5,  1.0, 1.0, 1.0,
             0.5,  0.5,  0.5,  1.0, 0.0, 1.0,
             0.5,  0.5,  0.5,  1.0, 0.0, 1.0,
            -0.5,  0.5,  0.5,  0.0, 0.0, 0.0,
            -0.5,  0.5, -0.5,  0.0, 1.0, 0.0
            ];

            gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
            n_draw=punkty_;
}
```

16. Dane wierzchołkowe:

Ponieważ, zmienił się układ danych w buforze wierzchołków, dlatego konieczne jest określenie ich kolejności i przypisanie do odpowiednich wektorów w shaderach:

```
// dane wierzchołkowe ---------------------------------------------------------------------------

               const positionAttrib = gl.getAttribLocation(program, "position");
               gl.enableVertexAttribArray(positionAttrib);
               gl.vertexAttribPointer(positionAttrib, 3, gl.FLOAT, false, 6*4, 0);

               const colorAttrib = gl.getAttribLocation(program, "color");
               gl.enableVertexAttribArray(colorAttrib);
               gl.vertexAttribPointer(colorAttrib, 3, gl.FLOAT, false, 6*4, 3*4);

//-------------------------------------------------------------------------------------------------
```

17. Ćwiczenie:

Na podstawie poprzedniego kodu przygotować projekt, uwzględniając macierze projekcji, widoku i modelu umożliwiający poruszanie się po scenie w widoku FPS z widocznym kolorowym sześcianem. Klawiszem esc zatknąć aktualną zakładkę z programem po uprzednim monicie potwierdzającym.