WebGL, Free look camera, szybkość działania, teksturowanie

Część 1

1. Cel ćwiczenia:

Zapoznanie z programowaniem grafiki z użyciem WebGL, zastosowanie swobodnego poruszania kamery oraz kontroli szybkości działania pętli głównej, wyświetlanie FPS.

2. Wstęp:

W celu przeniesienia ruchu myszy na świat trójwymiarowy konieczne jest określenie o ile kursor myszy przemieścił się w każdej klatce animacji w kierunkach x i y. Kolejną rzeczą jest możliwość zmiany ilości klatek generowanych na sekundę oraz ustalenie stałej szybkości poruszania obiektów, niezależnie od ilości klatek animacji na sekundę.

3. Free look:

Pierwszą rzeczą którą trzeba wykonać to przechwycenie kursora myszy przez obszar canvas. W tym celu można wykorzystać `Pointer Lock API`. Blokada wskaźnika umożliwia dostęp do zdarzeń myszy, nawet gdy kursor wykracza poza granice przeglądarki lub ekranu. Na przykład użytkownicy mogą nadal obracać lub manipulować modelem 3D, przesuwając mysz bez końca. Bez blokady wskaźnika obrót lub manipulacja zatrzymuje się w momencie, gdy wskaźnik dotrze do krawędzi przeglądarki lub ekranu.

Przechwycenie kursora myszy uzyskuje się poprzez kliknięcie w obszarze canvas, natomiast uwolnienie poprzez klawisz ESC.
Przykładowy kod:

```
//*****************pointer lock object forking for cross browser*********************

canvas.requestPointerLock = canvas.requestPointerLock ||
                            canvas.mozRequestPointerLock;

document.exitPointerLock = document.exitPointerLock ||
                           document.mozExitPointerLock;

canvas.onclick = function() {
 canvas.requestPointerLock();
};

// Hook pointer lock state change events for different browsers
document.addEventListener('pointerlockchange', lockChangeAlert, false);
document.addEventListener('mozpointerlockchange', lockChangeAlert, false);

function lockChangeAlert() {
  if (document.pointerLockElement === canvas ||
      document.mozPointerLockElement === canvas) {
    console.log('The pointer lock status is now locked');
    document.addEventListener("mousemove", ustaw_kamere_mysz, false);
  } else {
    console.log('The pointer lock status is now unlocked');
    document.removeEventListener("mousemove", ustaw_kamere_mysz, false);
  }
}

//************************************************************
```

Do aktualizacji położenia wektora front konieczna jest osobna funkcja do aktualizacji za pomocą myszy – `ustaw_kamere_mysz.`

Przykładowy kod:

```
let x = 50; //zmiana położenia w kierunku x
let y = 50; //zmiana położenia w kierunku y
```

W celu kontroli aktualnego nachylenia kamery potrzebne będą dwa kąty:

```
let yaw =-90;  //obrót względem osi X
let pitch=0;   //obrót względem osi Y


function ustaw_kamere_mysz(e) {
```

//Wyznaczyć zmianę pozycji myszy względem ostatniej klatki

```
        let xoffset = e.movementX;
        let yoffset = e.movementY;

        let sensitivity = 0.1;
        let cameraSpeed = 0.05*elapsedTime;

        xoffset *= sensitivity;
        yoffset *= sensitivity;
```

//Uaktualnić kąty

```
        yaw += xoffset * cameraSpeed;
         pitch -= yoffset*cameraSpeed;
```

//Nałożyć ograniczenia co do ruchy kamery

```
        if (pitch > 89.0)
                pitch = 89.0;
        if (pitch < -89.0)
                pitch = -89.0;


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

//Wyznaczenie wektora kierunku na podstawie kątów Eulera

```
        front.x = Math.cos(glm.radians(yaw))*Math.cos(glm.radians(pitch));
        front.y = Math.sin(glm.radians(pitch));
        front.z = Math.sin(glm.radians(yaw)) * Math.cos(glm.radians(pitch));
        cameraFront = glm.normalize(front);
}
```

Obsługa klawiszy również ulegnie zmianie. Klawisze strzałek w lewo i w prawo nie będą już służyły do obrotu kamery, ale do ruchu prostopadłego do wektora widoku tzw. strafe. Aby to uzyskać należy kreślić wektor prostopadły do wektora widoku i pionu kamery. Można wykorzystać iloczyn wektorowy:

```
                let cameraPos_tmp = glm.normalize(glm.cross(cameraFront, cameraUp));
                cameraPos.x+=cameraPos_tmp.x * cameraSpeed;
                cameraPos.y+=cameraPos_tmp.y * cameraSpeed;
                cameraPos.z+=cameraPos_tmp.z * cameraSpeed;
```

4. Szybkość poruszania kamery:

Aby uzyskać stałą szybkość poruszania obiektów konieczne jest obliczenie współczynnika, mnożonego przez wektory ruchu obiektów (również kamery).

```
let startTime=0;
let elapsedTime=0;

            function draw()
            {

                elapsedTime = performance.now() - startTime;
```

Uzyskany współczynnik należy przemnożyć przez szybkość kamery dla klawiszy i myszy.

5. Zmiana liczby FPS.

Maksymalna liczba FPS to około 60.
Aby to zmienić można wykorzystać funkcję:

```
setTimeout(() => { requestAnimationFrame(draw);}, 1000 / FPS);
```

w pętli głównej programu.

6. Wyświetlenie liczby FPS na ekranie.

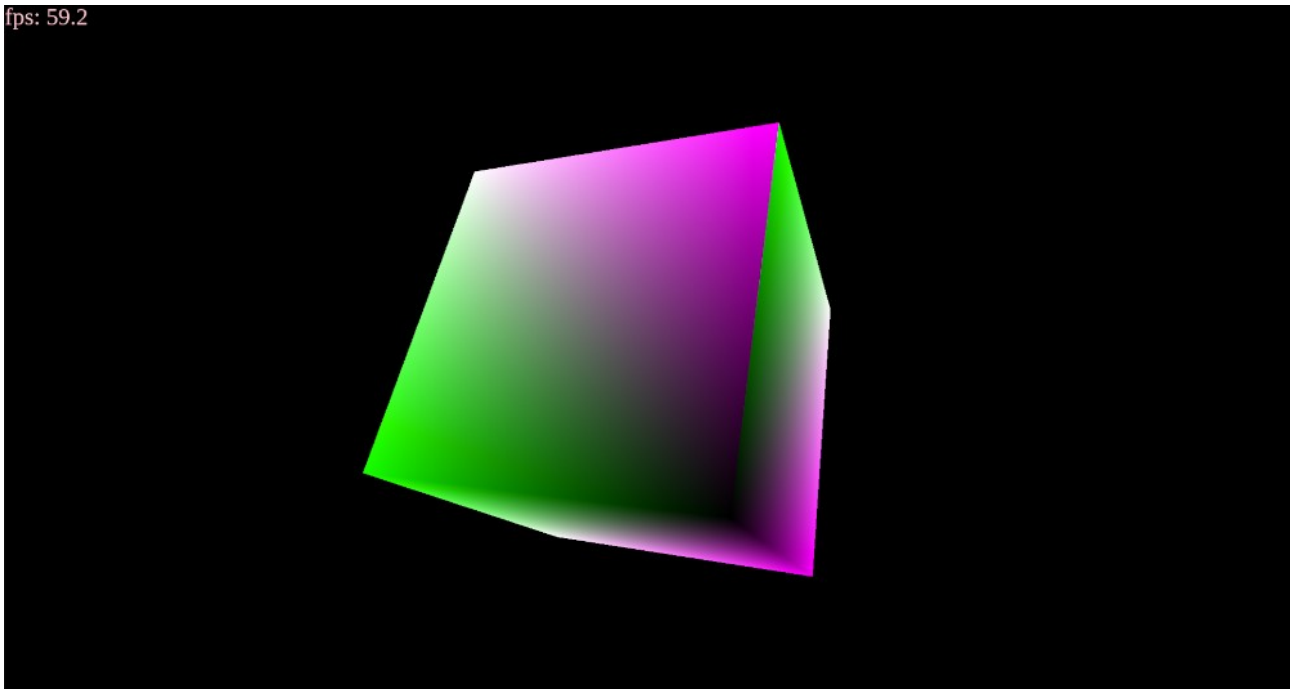
W celu wyświetlenia liczby FPS konieczne jest wyznaczenie czasu wykonywania pojedynczej klatki w milisekundach.

```
let licznik=0;
const fpsElem = document.querySelector("#fps");
```

```
let startTime=0;
let elapsedTime=0;

            function draw()
            {
                        elapsedTime = performance.now() - startTime;

                        startTime = performance.now();

                        licznik++;
                        let fFps = 1000 / elapsedTime;

// ograniczenie częstotliwości odświeżania napisu do ok 1/s
                        if(licznik > fFps){
                        fpsElem.textContent = fFps.toFixed(1);

                        licznik = 0;

                        }
```

7. Ćwiczenie:

Należy przygotować projekt pozwalający na swobodne poruszanie kamery po scenie z użyciem klawiatury i myszy. Uniezależnić szybkość działania programu od szybkości komputera (można to sprawdzić poprzez ograniczenie ilości klatek na sekundę). Wyświetlić liczbę klatek na sekundę na ekranie w obszarze canvas.



Część 2 teksturowanie

1. Cel ćwiczenia:

Zapoznanie z funkcjami realizującymi wczytywanie tekstur oraz mechanizmami nakładania tekstur na wielokąty.

2. Wstęp:

Biblioteka WebGL umożliwia zastosowanie tekstur:

1D – (jednowymiarowa) jest to obraz nieposiadający wysokości lub szerokości. Są to obrazy w postaci paska o szerokości lub wysokości jednego piksela,

2D – (dwuwymiarowa) to obraz szerszy i wyższy niż jeden piksel. Zwykle jest ładowany z pliku *.BMP ale również innych formatów. Tekstury 2D są używane w celu nadanie obiektom

realistycznego wyglądu poprzez nakładanie na siatki wielokątów. Są używane również do tworzenia realistycznego tła.
3D – (trójwymiarowe, wolumetryczne) jest to tekstura opisana w trzech wymiarach . Można ją utworzyć ze stosu tekstur bitmapowych 2D lub jako teksturę proceduralną 3D. Zaletą jest teksturowanie obiektów w całej ich objętości.

3. Wprowadzenie:

Pierwszą rzeczą którą trzeba wykonać, aby użyć tekstur jest załadowanie pliku graficznego do aplikacji. Obrazy tekstur mogą być przechowywane w różnych formatach, każdy z własną strukturą i kolejnością danych. Jednym z rozwiązań mogło by być wybranie konkretnego formatu pliku np. *.PNG i napisanie własnego programu ładującego obrazy. Innym rozwiązaniem jest użycie biblioteki ładującej obraz lub dostępnych rozwiązań HTML i Java Script.

Przykładowy kod ładujący plik graficzny i tworzący teksturę:

```
//texture1   ***************************************************************************
  const texture1 = gl.createTexture();
  gl.bindTexture(gl.TEXTURE_2D, texture1);

  const level = 0;
  const internalFormat = gl.RGBA;
  const width = 1;
  const height = 1;
  const border = 0;
  const srcFormat = gl.RGBA;
  const srcType = gl.UNSIGNED_BYTE;
  const pixel = new Uint8Array([0, 0, 255, 255]);
  gl.texImage2D(gl.TEXTURE_2D, level, internalFormat,
                width, height, border, srcFormat, srcType,
                pixel);

  const image = new Image();

  image.onload = function() {
    gl.bindTexture(gl.TEXTURE_2D, texture1);
    gl.texImage2D(gl.TEXTURE_2D, level, internalFormat,srcFormat, srcType, image);

       gl.generateMipmap(gl.TEXTURE_2D);
       gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.REPEAT);
       gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.REPEAT);
       gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
       gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
  };
  image.crossOrigin = "";
  image.src = "https://cdn.pixabay.com/photo/2013/09/22/19/14/brick-wall-185081_960_720.jpg";

//************************************************************
```

4. Użycie tekstury:

Aby tekstura mogła zostać prawidłowo nałożona na wielokąt konieczne jest podanie koordynat tekstury. Konieczna jest modyfikacja tablicy z danymi o wierzchołkach:
Przykładowo dla jednej ściany sześcianu:

```
function kostka() {
        var vertices = [
        -0.5, -0.5, -0.5,  0.0, 0.0, 0.0,  0.0, 0.0,
         0.5, -0.5, -0.5,  0.0, 0.0, 1.0,  1.0, 0.0,
         0.5,  0.5, -0.5,  0.0, 1.0, 1.0,  1.0, 1.0,
         0.5,  0.5, -0.5,  0.0, 1.0, 1.0,  1.0, 1.0,
        -0.5,  0.5, -0.5,  0.0, 1.0, 0.0,  0.0, 1.0,
        -0.5, -0.5, -0.5,  0.0, 0.0, 0.0,  0.0, 0.0
        ];
```

Dwie ostatnie wartości w wierszach to koordynaty tekstury. Tablicę wysyłamy do karty graficznej w sposób standardowy:

```
        gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
```

Następnie aby była możliwość przekazania tych informacji do shadera konieczne jest określenie pozycji tych danych w buforze:

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

```
            .
            .
            .

            const texCoord = gl.getAttribLocation(program, "aTexCoord");
            gl.enableVertexAttribArray(texCoord);
            gl.vertexAttribPointer(texCoord, 2, gl.FLOAT, false, 8*4, 6*4);

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

Następnie konieczna jest zmiana kodu vertex i fragment shadera, aby przyjąć współrzędne tekstury jako atrybut wierzchołka, a następnie przekazać je do modułu vertex shadera.
Linijki do uzupełnienia:

```
in vec2 aTexCoord;
out vec2 TexCoord;
```

A w funkcji main() przekazać dane do fragment shadera:

```
TexCoord = aTexCoord;
```

Fragment shader przyjmuje te dane jako wejściowe:

```
 in vec2 TexCoord;
```

Do shadera trzeba przekazać jeszcze teksturę. Można to zrobić z użyciem tzw. samplera:

```
uniform sampler2D texture1;
```

Ustawienie kolorów fragmentów zgodnie z kolorami pobranymi z tekstury zgodnie z koordynatami można uzyskać wywołaniem w funkcji main():

```
frag_color = texture(texture1, TexCoord);
```

W przypadku wykorzystania wielu tekstur, należy określić jakich tekstur ma użyć każdy sampler np.

```
gl.uniform1i(gl.getUniformLocation(program, "texture1"), 0);
gl.uniform1i(gl.getUniformLocation(program, "texture2"), 1);
```

W pętli głównej konieczne jest powiązanie i uaktywnienie tekstur na czas rysowania obiektu np. w celu mieszania tekstur.

```
        gl.activeTexture(gl.TEXTURE0);
        gl.bindTexture(gl.TEXTURE_2D, texture1);
        gl.activeTexture(gl.TEXTURE1);
        gl.bindTexture(gl.TEXTURE_2D, texture2);
         gl.drawArrays(prymityw, 0, 36);
```

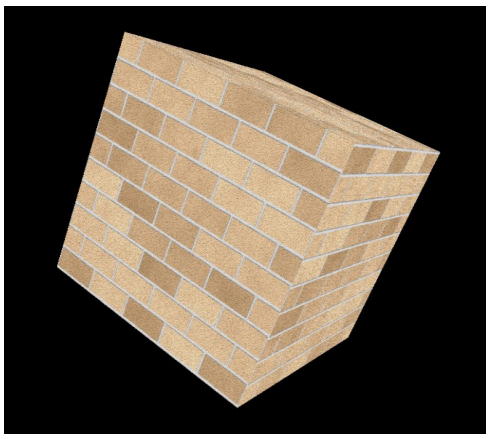
W przypadku korzystania z więcej niż jednej tekstury ale nie w tym samym czasie można pozwiązywać teksturę z jednym samplerem na czas rysowania poszczególnych ścian.

```
        gl.activeTexture(gl.TEXTURE0);
        gl.bindTexture(gl.TEXTURE_2D, texture2);
         gl.drawArrays(prymityw, 0, 12);

        gl.activeTexture(gl.TEXTURE0);
        gl.bindTexture(gl.TEXTURE_2D, texture1);

         gl.drawArrays(prymityw, 12, 24);
```

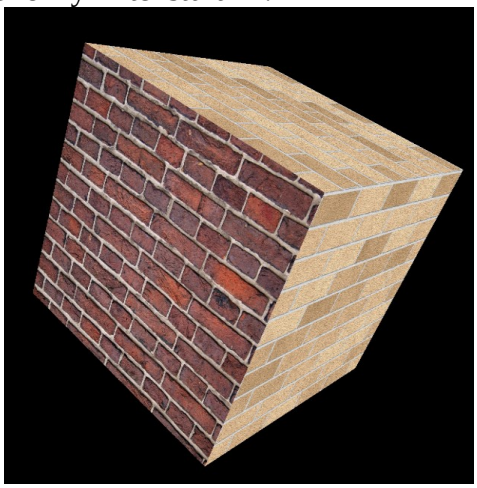
5. Ćwiczenie:

Należy przygotować projekt przedstawiający teksturowany sześcian w trzech wersjach.

1. Cały sześcian oteksturowany jedną teksturą:



2. Ściany oteskturowane różnymi teksturami:



3. Ściany oteskturowane połączeniem dwóch tekstur (mix).