# Grafika interaktywna

Krzysztof Gdawiec

UNIWERSYTET ŚLĄSKI
INSTYTUT INFORMATYKI

# Rysowanie cd

Wiele formatów zapisu obiektów 3D używa ściankowej reprezentacji tzn. mamy tablicę z wierzchołkami obiektu oraz tablicę ze ściankami w postaci indeksów do wierzchołków tworzących daną ściankę.

Używając VBO do rysowania możemy użyć tablic wierzchołków (`GL_ARRAY_BUFFER`) oraz indeksowanych tablic wierzchołków (`GL_ELEMENT_ARRAY_BUFFER`).

W takim przypadku tworzymy dwa obiekty VBO jeden `GL_ARRAY_BUFFER`, w którym umieszczamy współrzędne wierzchołków przekazywane do shadera wierzchołków jako atrybut. W drugim obiekcie `GL_ELEMENT_ARRAY_BUFFER` umieszczamy tablicę indeksów.

Mając dane o wierzchołkach i indeksach załadowane do VBO możemy je wyrysować za pomocą funkcji:

```
gl.drawElements(mode, count, type, offset);
```

Parametr `mode` oznacza rodzaj prymitywu do wyrysowania (`gl.LINES`, `gl.TRIANGLES` itp.), parametr `count` oznacza liczbę indeksów, które chcemy wyrysować, parametr `type` określa typ indeksów (`gl.UNSIGNED_BYTE`, `gl.UNSIGNED_SHORT`), zaś `offset` to przesunięcie (w bajtach) na element od którego chcemy zacząć rysować (np. `2 * n`, gdzie `n` to numer elementu).

Przy bardziej skomplikowanym modelu typ `gl.UNSIGNED_SHORT` może być za mały. Wówczas możemy skorzystać z rozszerzenia `OES_element_index_uint`, które daje nam typ `gl.UNSIGNED_INT`, a które włączamy poleceniem:

```
gl.getExtension("OES_element_index_uint");
```

Należy pamiętać, że model mogą tworzyć różne obiekty VBO reprezentujące różne atrybuty (współrzędne, normale, kolory itp.), ale tylko jeden obiekt VBO z indeksami.

Należy pamiętać, że model mogą tworzyć różne obiekty VBO reprezentujące różne atrybuty (współrzędne, normale, kolory itp.), ale tylko jeden obiekt VBO z indeksami.

Zmiana grubości linii:

```
gl.lineWidth(width);
```

Parametr `width` określa grubość linii, która nie może być mniejsza niż 0. Domyślnie grubość linii wynosi 1.

W zależności od systemu operacyjnego i ustawień przeglądarki funkcja `lineWidth` może działać lub nie!

Każdy wielokąt (ściana) w WebGL ma dwie strony (przednią i tylną). Domyślnie przednią ścianą jest ta, której wierzchołki uporządkowane są przeciwnie do ruchu wskazówek zegara.

Można to zmienić funkcją:

```
gl.frontFace(mode);
```

Parametr `mode` przyjmuje jedną z dwóch wartości:

- `gl.CW` – przednia strona wielokąta z wierzchołkami uporządkowanymi zgodnie z ruchem wskazówek zegara,
- `gl.CCW` – przednia strona wielokąta z wierzchołkami uporządkowanymi odwrotnie do ruchu wskazówek zegara.

Standardowo w WebGL rysowane są obie strony wielokąta. Aby móc wyłączyć rysowanie którejkolwiek ze stron lub obu najpierw należy wywołać funkcję:

```
gl.enable(gl.CULL_FACE);
```

Następnie możemy wyłączać rysowanie strony wielokąta za pomocą funkcji:

```
gl.cullFace(mode);
```

Parametr `mode` przyjmuje jedną z wartości:

- `gl.FRONT` – nie rysowana jest przednia strona,
- `gl.BACK` – nie rysowana jest tylna strona,
- `gl.FRONT_AND_BACK` – nie rysowane są obie strony.

# Definiowanie sceny 3D

```
gl.viewport(x, y, width, height);
```

funkcja zmieniająca obszar renderingu w elemencie canvas, gdzie

- `x`, `y` – współrzędne lewego dolnego narożnika obszaru renderingu względem lewego dolnego narożnika elementu canvas,
- `width`, `height` – szerokość, wysokość obszaru renderingu.

# Definiowanie sceny 3D

```
gl.viewport(x, y, width, height);
```

funkcja zmieniająca obszar renderingu w elemencie canvas, gdzie

- `x, y` – współrzędne lewego dolnego narożnika obszaru renderingu względem lewego dolnego narożnika elementu canvas,
- `width, height` – szerokość, wysokość obszaru renderingu.

Do zmniejszenia obszaru renderingu możemy określić prostokąt okrawający (ang. scissor rectangle). Wówczas obraz będzie renderowany wyłącznie w podanym obszarze pomijając rendering z pozostałej części obszaru renderingu elementu canvas.

Włączenia funkcji okrawania dokonujemy za pomocą `gl.enable(gl.SCISSOR_TEST)`. Do określenia prostokąta okrawającego służy funkcja:

```
gl.scissor(x, y, width, height);
```

`x`, `y` – lewy dolny róg prostokąta, `width`, `height` – szerokość i wysokość prostokąta. Wszystkie wielkości podawane są w pikselach w elemencie canvas.

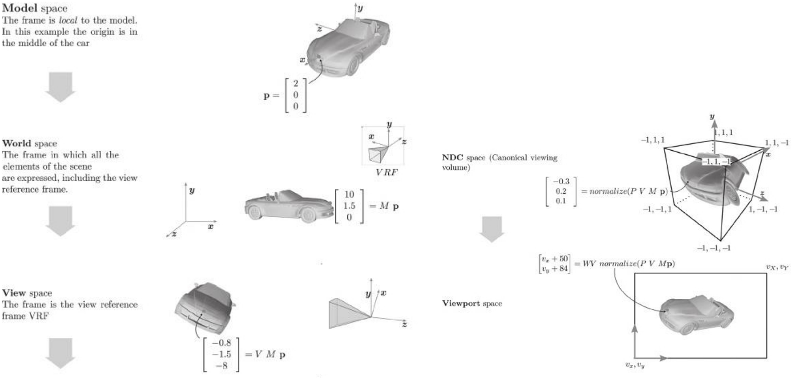
Włączenia funkcji okrawania dokonujemy za pomocą `gl.enable(gl.SCISSOR_TEST)`. Do określenia prostokąta okrawającego służy funkcja:

```
gl.scissor(x, y, width, height);
```

`x`, `y` – lewy dolny róg prostokąta, `width`, `height` – szerokość i wysokość prostokąta. Wszystkie wielkości podawane są w pikselach w elemencie canvas.

W grafice 3D scena zazwyczaj określona jest za pomocą kilku układów współrzędnych:

- modelu,
- świata,
- oka lub kamery,
- przycięcia,
- ekranu.



Model space
The frame is local to the model. In this example the origin is in the middle of the car
y
z
x
p = [2 0 0]
World space
The frame in which all the elements of the scene are expressed, including the view reference frame.
y
x
z
VRF
[10 1.5 0] = M p
View space
The frame is the view reference frame VRF
[−0.8 −1.5 −8] = V M p
NDC space (Canonical viewing volume)
1, 1, 1
−1, 1, 1
1, 1, −1
−1, 1, −1
−1, −1, 1
1, −1, −1
−1, −1, −1
[−0.3 0.2 0.1] = normalize(P V M p)
[v_x + 50 v_y + 84] = WV normalize(P V M p)
Viewport space
v_X, v_Y
v_x, v_y

Z algebry liniowej wiadomo, że przechodzenie pomiędzy poszczególnymi układami dokonywane jest za pomocą mnożenia przez odpowiednią macierz przejścia.

Wyróżniamy kilka takich macierzy:

- modelu $M$ – przejście z układu modelu do układu świata,
- widoku $V$ – przejście z układu świata do układu oka,
- projekcji $P$ – przejście z układu oka do układu przycięcia.

Zadaniem shadera wierzchołków jest przeliczenie współrzędnych wierzchołka $p$ z układu modelu do układu przycięcia czyli wymnożeniu przez macierz modelu, widoku i projekcji, tzn. $PVMp$. Brakuje nam jeszcze macierzy przejścia z układu przycięcia do układu ekranu. Macierz ta tworzona jest przez WebGL i przejście realizowane jest automatycznie.

Macierz modelu mówi nam w jaki sposób umieścić model w świecie. Zatem przechowuje ona informacje m.in. o translacji, rotacji i skalowaniu modelu.

Dla przypomnienia macierze podstawowych transformacji w 3D wyglądają następująco:

- translacja

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

gdzie $[t_x, t_y, t_z]^T$ to wektor translacji

- skalowanie

$$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

gdzie $s_x, s_y, s_z$ to współczynniki skali

- rotacja o kąt $\theta$ względem osi $X$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- rotacja o kąt $\theta$ względem osi $Y$

$$\begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- rotacja o kąt $\theta$ względem osi $Z$

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Za pomocą macierzy rotacji względem osi $X$, $Y$, $Z$ możemy wyrazić dowolną rotację w 3D poprzez wymnożenie tych macierzy.

Rotacja względem osi $Y$ nazywana jest też odchyleniem (ang. yaw), względem osi $X$ pochyleniem (ang. pitch), względem osi $Z$ przechyleniem (ang. roll).

Model, w którym podajemy kąty obrotu względem osi $X$, $Y$, $Z$ nazywany jest modelem kątów Eulera.

Za pomocą macierzy rotacji względem osi $X$, $Y$, $Z$ możemy wyrazić dowolną rotację w 3D poprzez wymnożenie tych macierzy.

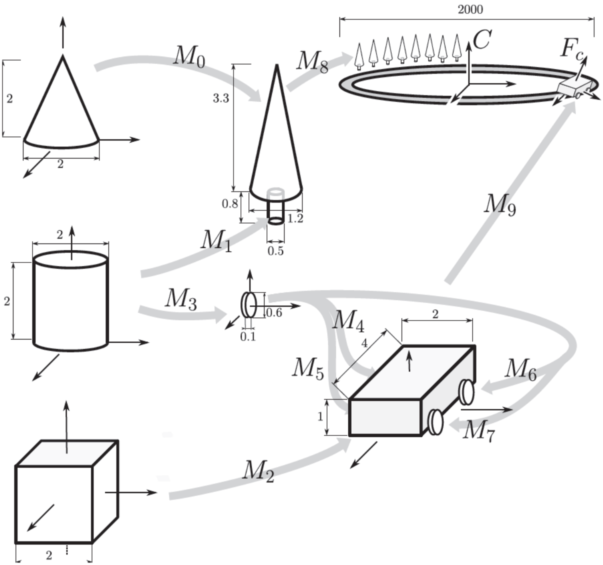
Rotacja względem osi $Y$ nazywana jest też odchyleniem (ang. yaw), względem osi $X$ pochyleniem (ang. pitch), względem osi $Z$ przechyleniem (ang. roll).

Model, w którym podajemy kąty obrotu względem osi $X$, $Y$, $Z$ nazywany jest modelem kątów Eulera.

- rotacja względem dowolnej osi

$$\begin{pmatrix} x^2(1-c)+c & xy(1-c)-zs & xz(1-c)+ys & 0 \\ yx(1-c)+zs & y^2(1-c)+c & yz(1-c)-xs & 0 \\ xz(1-c)-ys & yz(1-c)+xs & z^2(1-c)+c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

gdzie $[x, y, z]^T$ to wektor reprezentujący oś wokół której dokonujemy obrotu, $\theta$ – kąt obrotu oraz $c = \cos\theta$, $s = \sin\theta$.



2000
$C$
$F_c$
$M_0$
$M_8$
$M_1$
$M_3$
$M_4$
$M_5$
$M_6$
$M_7$
$M_2$
$M_9$
2
2
3.3
0.8
1.2
0.5
2
2
0.6
0.1
2
4
1
2

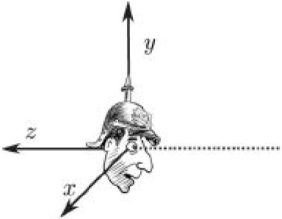
Macierz widoku określa w jaki sposób wyrazić obiekty w świecie względem położenia obserwatora.
Niech $e \in \mathbb{R}^3$ – położenie oka, $c \in \mathbb{R}^3$ – punkt na który patrzymy, $\mathrm{p} \in \mathbb{R}^3$ – wektor do góry.

$$\mathrm{z} = \frac{e - c}{\|e - c\|} \qquad \mathrm{x} = \frac{\mathrm{p} \times \mathrm{z}}{\|\mathrm{p} \times \mathrm{z}\|} \qquad \mathrm{y} = \mathrm{z} \times \mathrm{x},$$

gdzie $\times$ oznacza iloczyn wektorowy.

Macierz widoku ma wówczas postać:

$$\begin{pmatrix} \mathrm{x}_x & \mathrm{x}_y & \mathrm{x}_z & -\mathrm{x} \cdot e \\ \mathrm{y}_x & \mathrm{y}_y & \mathrm{y}_z & -\mathrm{y} \cdot e \\ \mathrm{z}_x & \mathrm{z}_y & \mathrm{z}_z & -\mathrm{z} \cdot e \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

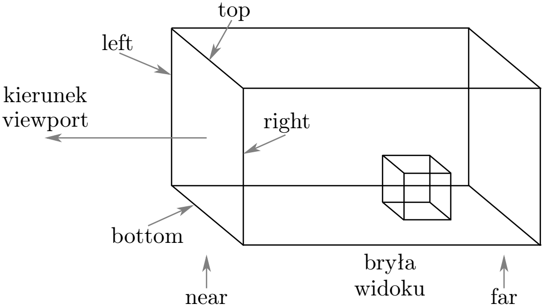



gdzie $\cdot$ oznacza iloczyn skalarny.

W celu przekształcenia sceny 3D w 2D musimy dokonać między innymi rzutowania. Wyróżniamy dwa podstawowe typu rzutowania: prostopadłe (ortogonalne), perspektywiczne.

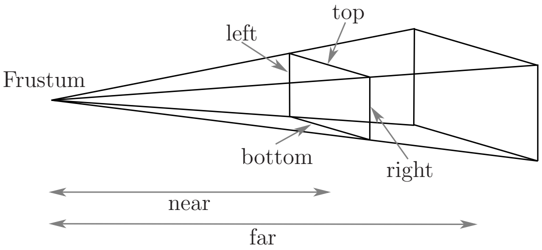
Oba typy rzutowania są definiowane przez tzw. bryłę widzenia, która w przypadku rzutowania ortogonalnego jest prostopadłościanem, a w przypadku rzutowania perspektywicznego ściętym ostrosłupem o podstawie prostokąta.

## Rzutowanie ortogonalne

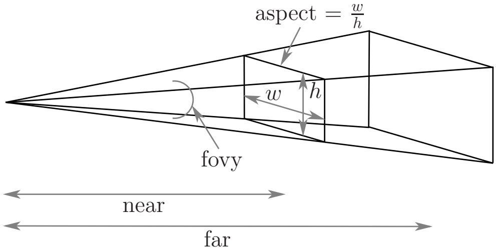



$$\begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

## Rzutowanie perspektywiczne




$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$




W macierzy z poprzedniego slajdu przyjmujemy:

$$t = n \cdot \mathrm{tg}\,\frac{fovy}{2} \qquad b = -n \cdot \mathrm{tg}\,\frac{fovy}{2}$$

$$r = aspect \cdot t \qquad l = aspect \cdot b$$

Dla każdej z macierzy (model, widok, rzutowania) tworzymy zmienną reprezentującą bieżącą macierz i inicjujemy ją macierzą jednostkową. Do tej macierzy domnażamy (z prawej) macierze transformacji. W ten sposób transformacja, która ma być wykonana jako pierwsza jest domnażana jako ostatnia.

W przypadku modeli hierarchicznych bardzo przydatny jest stos macierzy przekształceń. Ma on za zadanie zapamiętanie bieżącej macierzy na szczycie i późniejsze szybkie jej przywrócenie zdejmując szczyt stosu.

W WebGL nie mamy żadnych funkcji związanych z obsługą macierzy, które widzieliśmy przed chwilą. Wszystko musimy napisać sami lub korzystamy z bibliotek.

# Biblioteka glMatrix

Biblioteka glMatrix udostępnia funkcje tworzące i operujące na podstawowych macierzach ($2 \times 2$, $3 \times 3$, $4 \times 4$) i wektorach (2D, 3D, 4D) używanych w grafice komputerowej.

W celu utworzenia macierzy jednostkowej $4 \times 4$ wywołujemy funkcję:

```
glMatrix.mat4.create();
```

Jeśli zamiast `mat4` użyjemy `mat3` lub `mat2`, to utworzymy macierz $3 \times 3$ lub $2 \times 2$ (odpowiednio).
W podobny sposób tworzymy wektory zerowe, tzn. zastępujemy `mat4` przez `vec2`, `vec3` lub `vec4`.

Macierze podstawowych transformacji (we wszystkich funkcjach `out` to macierz wynikowa):

- translacja – mnożymy macierz `a` przez macierz translacji o wektor `v`

  ```
  glMatrix.mat4.translate(out, a, v);
  ```

- skalowanie – mnożymy macierz `a` przez macierz skalowania o skalach `v`

  ```
  glMatrix.mat4.scale(out, a, v);
  ```

- rotacja względem osi $X$ – mnożymy macierz `a` przez macierz rotacji o kąt `rad` względem osi X

  ```
  glMatrix.mat4.rotateX(out, a, rad);
  ```

- rotacja względem osi $Y$ – mnożymy macierz `a` przez macierz rotacji o kąt `rad` względem osi Y

  ```
  glMatrix.mat4.rotateY(out, a, rad);
  ```

- rotacja względem osi $Z$ – mnożymy macierz `a` przez macierz rotacji o kąt `rad` względem osi Z

  ```
  glMatrix.mat4.rotateZ(out, a, rad);
  ```

- rotacja względem dowolnej osi – mnożymy macierz `a` przez macierz rotacji o kąt `rad` względem osi `axis`

  ```
  glMatrix.mat4.rotate(out, a, rad, axis);
  ```

- macierz widoku – `eye` położenie oka, `center` punkt na który patrzymy, `up` wektor do góry

  ```
  glMatrix.mat4.lookAt(out, eye, center, up);
  ```

- macierz rzutowania ortogonalnego

  ```
  glMatrix.mat4.ortho(out, left, right, bottom, top, near,
  far);
  ```

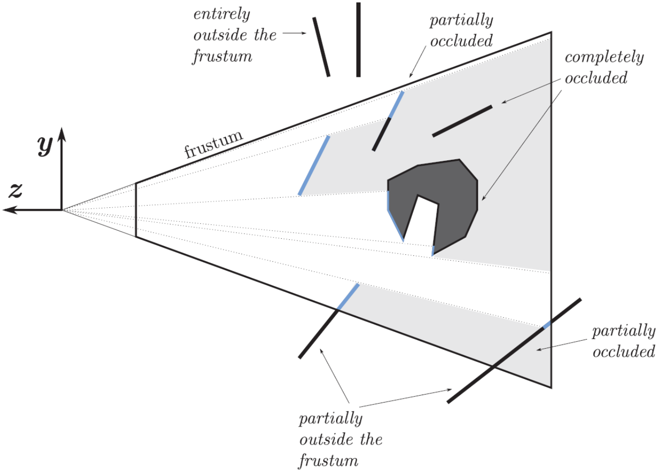
- macierz rzutowania perspektywicznego 1

  ```
  glMatrix.mat4.frustum(out, left, right, bottom, top, near,
  far);
  ```

- macierz rzutowania perspektywicznego 2

  ```
  glMatrix.mat4.perspective(out, fovy, aspect, near, far);
  ```

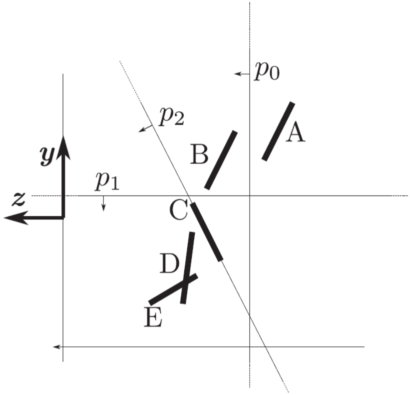
# Usuwanie powierzchni niewidocznych



**Sortowanie głębi**
Jest to modyfikacja algorytmu malarza, którego idea polega na tym, że prymitywy są sortowane względem odległości od obserwatora, a następnie rysowane są w porządku od najdalszego do najbliższego (ang. back-to-front drawing).

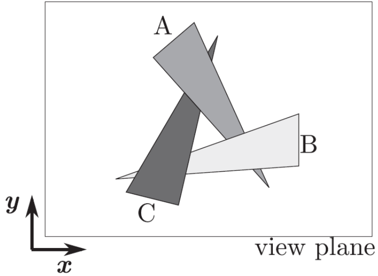
W pierwszych podejściach użycia algorytmu malarza jako odległość prymitywu od obserwatora brano odległość barycentrum prymitywu od obserwatora. Takie podejście nie gwarantuje, że jeśli barycentrum prymitywu $A$ jest bliżej niż barycentrum prymitywu $B$, to całe $A$ jest bliżej niż $B$.

W sortowaniu głębi próbujemy znaleźć odpowiedni porządek (od najdalszego do najbliższego) przez patrzenie na płaszczyznę rozdzielającą czyli taką płaszczyznę, która rozdziela prymitywy na dwie grupy (leżące po dodatniej i ujemnej stronie płaszczyzny).



Niestety, jeśli prymitywy przecinają się, to nie jesteśmy w stanie ich uporządkować a co za tym idzie narysować w odpowiedniej kolejności.

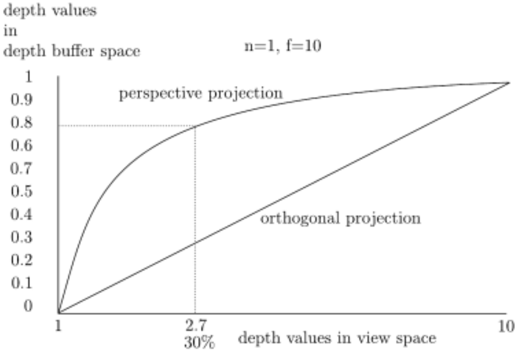
Nawet jeśli prymitywy nie przecinają się może zajść przypadek, w którym nie będziemy w stanie znaleźć odpowiedniego porządku, np. część $A$ jest za $B$, część $B$ jest za $C$ i część $C$ jest za $A$.



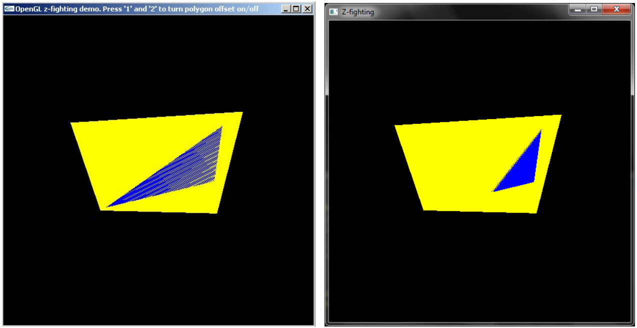
**Bufor głębi (ang. depth bufer) lub bufor $z$ (ang. $z$-buffer)**

Na początku pracy algorytmu bufor $z$ jest wypełniany maksymalną wartością głębi (domyślnie 1). Jednocześnie wszystkie piksele obrazu przyjmują barwę tła.
Następnie rysowane są wielokąty (w dowolnej kolejności) – to znaczy wypełniane są ich rzuty odpowiednią barwą. Podczas wypełniania zwykła procedura wypełniająca jest poszerzona o sprawdzenie głębi odpowiadającej danemu pikselowi. Piksel jest wypełniony tylko wtedy, kiedy jego głębia jest mniejsza niż wartość zapisana w buforze.

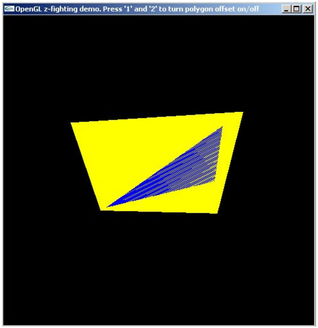
Precyzja działania bufora głębi zależy od wartości: $r = \frac{far}{near}$. Im większa wartość $r$, tym mniej efektywne jest działanie bufora głębi ze względu, że tracimy $\log_2 r$ bitów precyzji.

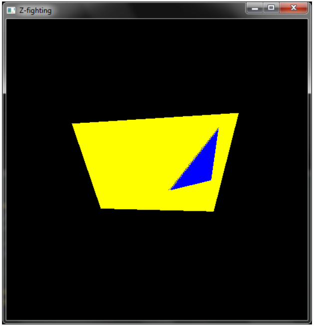


Z buforem głębi związany jest również tzw. *z*-fighting.



Z buforem głębi związany jest również tzw. *z*-fighting.




Włączenia bufora głębi (domyślnie bufor ten jest wyłączony) dokonujemy za pomocą funkcji `gl.enable` z parametrem `gl.DEPTH_TEST`.

Wyłączenie/włączenie zapisu do bufora głębi:

`gl.depthMask(flag);`

`flag = gl.TRUE` – włączenie zapisu, `flag = gl.FALSE` – wyłączenie zapisu.

Wyłączenie/włączenie zapisu do bufora głębi:

`gl.depthMask(flag);`

`flag = gl.TRUE` – włączenie zapisu, `flag = gl.FALSE` – wyłączenie zapisu.

Czyszczenie bufora głębi dokonywane jest przez dodanie stałej `gl.DEPTH_BUFFER_BIT` przy wywołaniu funkcji `gl.clear`.

Standardowo bufor głębi zawiera liczby z $[0, 1]$ i jest czyszczony wartością 1. Do zmiany wartości, którą czyszczony jest bufor $z$ służy funkcja:

`gl.clearDepth(depth);`

Wyłączenie/włączenie zapisu do bufora głębi:

`gl.depthMask(flag);`

`flag = gl.TRUE` – włączenie zapisu, `flag = gl.FALSE` – wyłączenie zapisu.

Czyszczenie bufora głębi dokonywane jest przez dodanie stałej `gl.DEPTH_BUFFER_BIT` przy wywołaniu funkcji `gl.clear`.

Standardowo bufor głębi zawiera liczby z $[0, 1]$ i jest czyszczony wartością 1. Do zmiany wartości, którą czyszczony jest bufor $z$ służy funkcja:

`gl.clearDepth(depth);`

Do zmiany zakresu wartości bufora głębi służy funkcja:

`gl.depthRange(zNear, zFar);`

`zNear` – minimalna wartość, `zFar` – maksymalna wartość.

Wybór testu bufora głębi dokonujemy za pomocą funkcji:

```
gl.depthFunc(func);
```

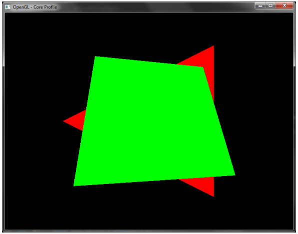
gdzie `func` przyjmuje jedną z wartości ($z$ – testowana wartość, $z_b$ – wartość w buforze):

- `gl.NEVER` – testu zawsze negatywny,
- `gl.LESS` – test pozytywny, gdy $z < z_b$ (wartość domyślna),
- `gl.EQUAL` – test pozytywny, gdy $z = z_b$,
- `gl.LEQUAL` – test pozytywny, gdy $z \leqslant z_b$,
- `gl.GREATER` – test pozytywny, gdy $z > z_b$,
- `gl.NOTEQUAL` – test pozytywny, gdy $z \neq z_b$,
- `gl.GEQUAL` – test pozytywny, gdy $z \geqslant z_b$,
- `gl.ALWAYS` – test zawsze pozytywny.

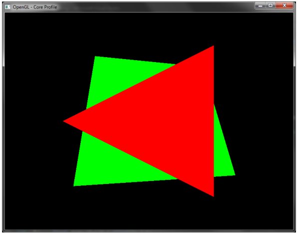
```
var triangleVerts =
[
      -0.8, 0.0, -0.2, 1.0,
      0.6, -0.7, -0.2, 1.0,
      0.6, 0.7, -0.2, 1.0
];

var quadVerts =
[
      -0.5, 0.6, -0.5, 1.0,
      -0.7, -0.6, -0.5, 1.0,
      0.5, 0.5, -0.5, 1.0,
      0.8, -0.5, -0.5, 1.0
];
```

Kolejność rysowania: trójkąt, czworokąt.

Wyłączony test głębi




Włączony test głębi



Jednym ze sposobów walki z $z$-fighting jest przesunięcie wartości głębi. Jest to mechanizm pozwalający na przesuwanie wartości głębi pikseli przy rysowaniu wielokątów. Przesunięcie obliczane jest według wzoru:

$$m \cdot \mathit{factor} + r \cdot \mathit{units},$$

gdzie $m$ – maksymalne nachylenie głębokości wielokąta, $r$ – zależna od implementacji najmniejsza różnica wartości przechowywanych w buforze głębi.

Pozostałe dwa współczynniki skalowania ustawiane są za pomocą funkcji:

```
gl.polygonOffset(factor, units);
```

Wartości początkowe obu współczynników wynoszą 0, najczęściej ustawia się je na wartość 1.

Pozostałe dwa współczynniki skalowania ustawiane są za pomocą funkcji:

```
gl.polygonOffset(factor, units);
```

Wartości początkowe obu współczynników wynoszą 0, najczęściej ustawia się je na wartość 1.

Domyślnie przesuwanie wartości głębi jest wyłączone. W celu włączenia wywołujemy funkcję `glEnable` z parametrem `gl.POLYGON_OFFSET_FILL`.

# Elementy GLSL ES

## Struktury

- Struktury służą do łączenia innych zdefiniowanych wcześniej typów w jedną całość. Do tego celu podobnie jak w C używamy słowa `struct`.
- Struktura musi posiadać co najmniej jedną składową.
- Każda składowa musi być wcześniej zdefiniowanym typem.
- Definicja struktury nie może zawierać wewnętrznych definicji innych struktur oraz struktur anonimowych.
- Struktury mogą być inicjalizowane przy użyciu konstruktorów, które nazywają się tak samo jak nazwa struktury.
- Argumenty konstruktora zawierają wartości lub konstruktory kolejnych jej pól.

- Dostęp do składowych struktury odbywa się poprzez operator kropki.
- Dla struktur zdefiniowane są operatory `==`, `!=`. Dwie struktury są sobie równe gdy wszystkie składowe są równe.
- Zdefiniowany jest również operator przypisania `=`.

```
struct Light
{
  float intensity;
  vec3 position;
} light1; // opcjonalna deklaracja zmiennej typu Light

uniform Light light2;

Light light3 = Light( 3.0, vec3( 1.0, 2.0, 3.0 ) );
vec3 pos = light3.position;
```

W przypadku gdy tworzymy zmienną typu strukturalnego, która równocześnie jest zmienną jednorodną lub atrybutem wierzchołka musimy przekazać dane do takiej zmiennej z aplikacji do shadera.

Zmienna typu strukturalnego nie ma swojej lokalizacji, ale każde jej pole ma. Zatem musimy pobrać lokalizację każdego z pól osobno i później używając tej lokalizacji przekazać dane. Przy pobieraniu lokalizacji jako nazwę podajemy nazwę zmiennej wraz z nazwą pola, np. `light2.position`.

## Tablice

- W GLSL ES dostępne są jedynie tablice jednowymiarowe.
- Przy definiowaniu tablicy nie musimy podawać jej rozmiaru, ale przed jej użyciem musimy określić rozmiar (ponowna definicja).
- Rozmiar tablicy podawany przy jej definiowaniu musi być dodatnim wyrażeniem stałym czyli nie możemy tworzyć tablic dynamicznych.
- Indeksowanie tablicy zaczyna się od 0.
- Tablica deklarowana jako parametr formalny funkcji musi posiadać rozmiar.
- Tablica może być typem zwracanym przez funkcję.
- Tablice mogą być każdego typu podstawowego oraz struktury.
- Na tablicy można wywołać funkcję `length`, która zwraca rozmiar tablicy.

- Dla tablic zdefiniowane są operatory `==`, `!=`. Dwie tablice są sobie równe gdy wszystkie elementy (o tych samych indeksach) są równe.

- Dla tablic zdefiniowane są operatory `==`, `!=`. Dwie tablice są sobie równe gdy wszystkie elementy (o tych samych indeksach) są równe.

```
float frequencies[3];
uniform vec4 lightPosition[4];
light lights[];
const int numLights = 2;
light lights[numLights];

float a[5];
float b[] = a;
float c[5] = a;

a.length(); // zwraca 5

float d[5] = float[5](3.4, 4.2, 5.0, 5.2, 1.1);
float e[5] = float[](3.4, 4.2, 5.0, 5.2, 1.1);

float[5] foo() {}
void foo(float[5] a);
```

W przypadku gdy tworzymy tablicę, która jest zmienną jednorodną lub atrybutem wierzchołka musimy przekazać dane do tablicy z aplikacji do shadera.

Dla zmiennych jednorodnych w tym celu używamy poznanych wcześniej funkcji `gl.uniform2fv`, ..., `gl.uniformMatrix4fv`, gdzie w ostatnim argumencie przekazujemy tablicę z wartościami.

Dla atrybutów nie ma specjalnej funkcji pozwalającej przesłać całą tablicę do shadera wierzchołków. W tym przypadku musimy pobrać lokalizację każdego elementu tablicy używając `gl.getAttribLocation`. Jako nazwę podajemy nazwę zmiennej wraz z indeksem, np. `aColor[1]`. A następnie przesyłamy dane tak jak w przypadku pojedynczego atrybutu.

Sposób przesyłania danych stosowany dla atrybutów można również użyć w przypadku zmiennych jednorodnych.

W przypadku atrybutu będącego tablicą kolejne elementy tej tablicy otrzymują kolejne lokalizacje, np. załóżmy, że mamy tablicę `foo` i że pierwszemu elementowi `foo[0]` przyporządkowano lokalizację 5, wówczas `foo[1]` ma lokalizację 6, `foo[2]` lokalizację 7 itd.

### Funkcje relacyjne na wektorach

Operatory `<`, `<=`, `>`, `>=`, `==`, `!=` zdefiniowane są jedynie dla wartości skalarnych. Jeśli chcemy dokonać porównania po składowych musimy skorzystać z funkcji relacyjnych:

- `bvec lessThan(vec v1, vec v2);`
- `bvec lessThanEqual(vec v1, vec v2);`
- `bvec greaterThan(vec v1, vec v2);`
- `bvec greaterThanEqual(vec v1, vec v2);`
- `bvec equal(vec v1, vec v2);`
- `bvec notEqual(vec v1, vec v2);`

Dla wektorów boolowskich w GLSL ES zdefiniowano funkcje:

- zwraca `true` jeśli którakolwiek składowa wektora jest `true`

  ```
  bool any(bvec v);
  ```

- zwraca `true` jeśli każda składowa jest `true`

  ```
  bool all(bvec v);
  ```

- zwraca zaprzeczenie każdej ze składowych

  ```
  bool not(bvec v);
  ```

# Animacja

Przed HTML5 w celu stworzenia animacji musieliśmy skorzystać z timerów w postaci funkcji:

```
setInterval(func, ms);
setTimeOut(func, ms);
```

gdzie `func` to funkcja jaką należy uruchomić po `ms` milisekundach.

Różnica pomiędzy `setInterval` i `setTimeOut` jest taka, że `setInterval` uruchamia w kółko `func` co `ms` milisekund, a `setTimeOut` uruchamia `func` raz po `ms` milisekundach.

W HTML5 wprowadzono specjalną funkcję służącą do obsługi animacji:

```
requestAnimationFrame(animationLoop);
```

gdzie `animationLoop` to funkcja, która zawiera naszą animację.

`requestAnimationFrame` mówi przeglądarce, że chcemy wykonać animację z prędkością 60 klatek/sek. Wywołuje ona `animationLoop` tylko raz, więc wewnątrz `animationLoop` musimy ponownie użyć `requestAnimationFrame`.