Wprowadzenie do shaderów

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.

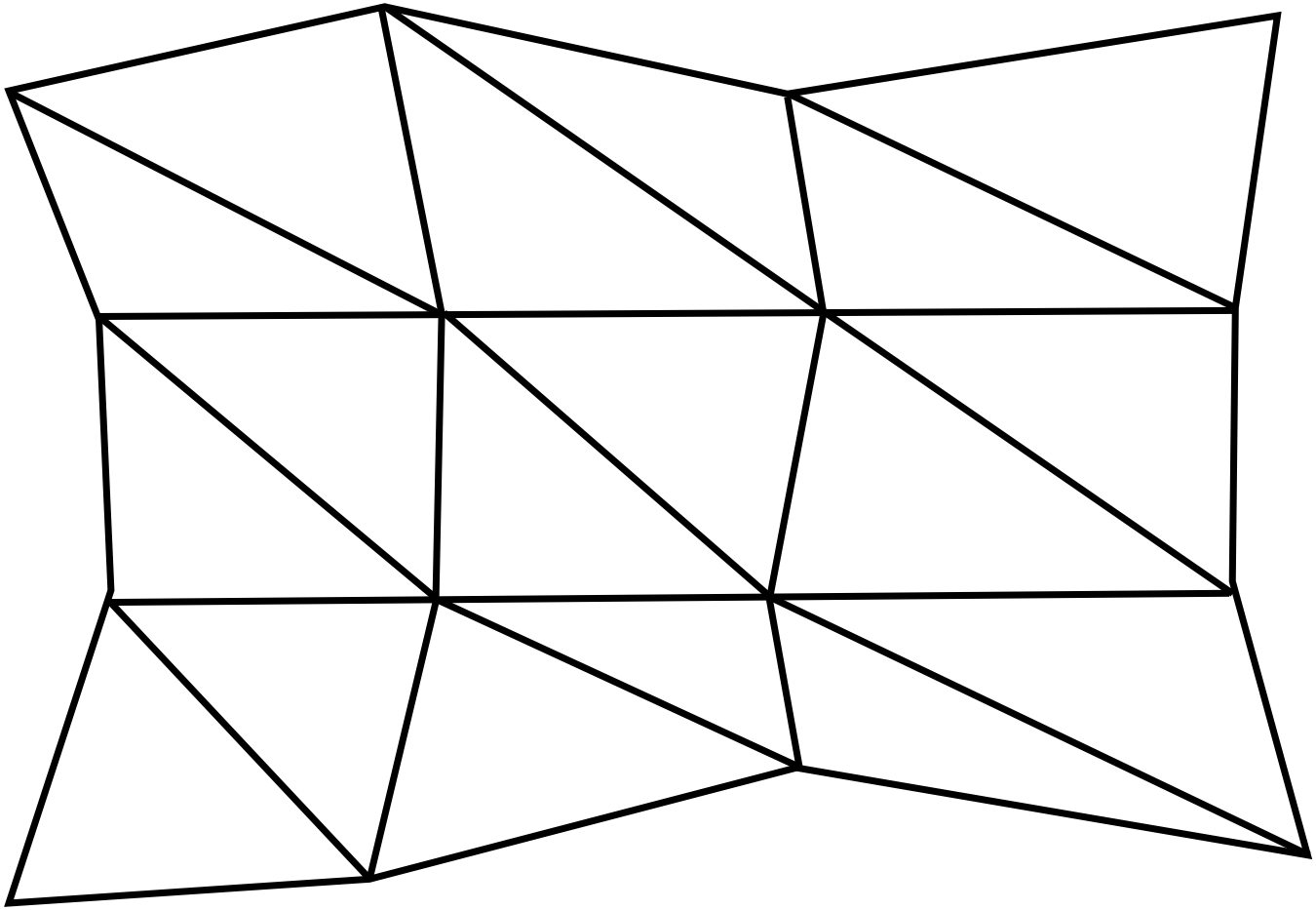
Mając dane o wierzchołkach i indeksach załadowane do VBO i VAO możemy je wyrysować za pomocą funkcji:

```
void glDrawElements(GLenum mode, GLsizei count, GLenum type, const
GLvoid* indices);
```

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`, `GL_UNSIGNED_INT`), zaś `indices` to wskaźnik na element od którego chcemy zacząć rysować (np. `(void*)(5 * sizeof(GLuint))`).

Należy pamiętać, że do obiektu VAO możemy dowiązać różne obiekty VBO reprezentujące różne atrybuty (współrzędne, normale, kolory itp.), ale tylko jeden obiekt VBO z indeksami.

Czy jesteśmy w stanie wyrysować poniższą siatkę używając pojedynczego wywołania `glDrawElements` i pasków trójkątów?



Możliwość wyrysowania siatki z poprzedniego slajdu za pomocą pojedynczego `glDrawElements` dodano w wersji 3.1 OpenGL-a.

W wersji tej wprowadzono tzw. resetowanie prymitywu. Polega ono na tym, że do tablicy indeksów po każdym prymitywie dodajemy indeks oznaczający koniec prymitywu.

Domyślnie resetowanie prymitywu jest wyłączone, więc musimy je włączyć funkcją `glEnable` z parametrem `GL_PRIMITIVE_RESTART`. Do ustalenia indeksu oznaczającego koniec prymitywu służy funkcja:

```
void glPrimitiveRestartIndex(GLuint index);
```

Zmiany sposobu rysownia wielokątów dokonujemy za pomocą funkcji:

```
void glPolygonMode(GLenum face, GLenum mode);
```

Parametr `face` musi przyjąć wartość `GL_FRONT_AND_BACK`, a parametr `mode` określa sposób rysowania:

- `GL_POINT` – rysowane są tylko wierzchołki wielokąta,
- `GL_LINE` – rysowane są tylko krawędzie wielokąta,
- `GL_FILL` – rysowany jest wypełniony wielokąt.

Zmiana wielkości punktu:

```
void glPointSize(GLfloat size);
```

Parametr `size` określa wielkość punktu, która nie może być mniejsza niż 0. Domyślna wielkość punktu wynosi 1.

Zmiana grubości linii:

```
void glLineWidth(GLfloat width);
```

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

W OpenGL mamy możliwość skorzystania z antialiasingu dla linii. Do włączenia/wyłączenia antialiasingu służą funkcje `glEnable`/`glDisable` z parametrem `GL_LINE_SMOOTH`.

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

Można to zmienić funkcją:

```
void glFrontFace(GLenum 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.

W shaderze fragmentów możemy sprawdzić czy aktualnie przetwarzany fragment należy do przedniej czy tylnej strony wielokąta. Do tego celu mamy zmienną typu `bool` o nazwie `gl_FrontFacing`. Wartość `true` oznacza, że fragment należy do przodu wielokąta.

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

```
glEnable(GL_CULL_FACE);
```

Następnie możemy wyłączać rysowanie strony wielokąta za pomocą funkcji:

```
void glCullFace(GLenum 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

```
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

funkcja zmieniająca obszar renderingu w oknie, gdzie

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

Definiowanie sceny 3D

```
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

funkcja zmieniająca obszar renderingu w oknie, gdzie

- `x, y` – współrzędne lewego dolnego narożnika obszaru renderingu względem lewego dolnego narożnika okna,
- `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 okna.

Włączenia funkcji okrawania dokonujemy za pomocą `glEnable(GL_SCISSOR_TEST)`. Do określenia prostokąta okrawającego służy funkcja:

```
void glScissor(GLint x, GLint y, GLsizei width, GLsizei height);
```

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

Włączenia funkcji okrawania dokonujemy za pomocą `glEnable(GL_SCISSOR_TEST)`. Do określenia prostokąta okrawającego służy funkcja:

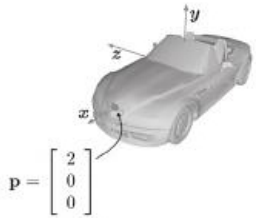
```
void glScissor(GLint x, GLint y, GLsizei width, GLsizei height);
```

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

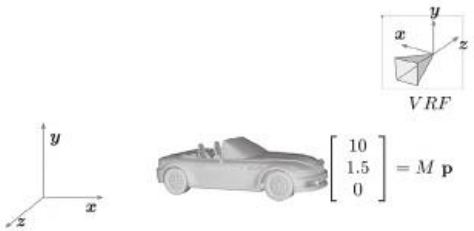
W grafice 3D scena zazwyczaj określona jest za pomocą kilku układów współrzędnych:

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

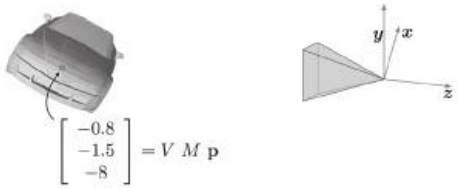
Model space
The frame is *local* to the model. In this example the origin is in the middle of the car



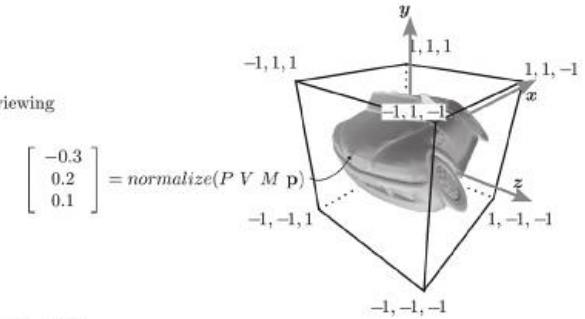
World space
The frame in which all the elements of the scene are expressed, including the view reference frame.



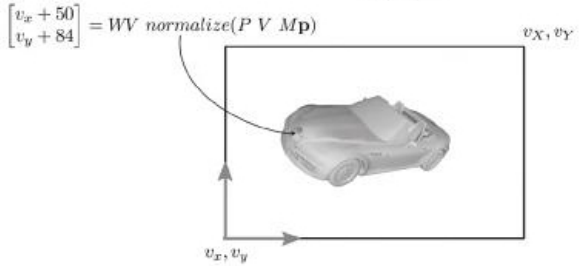
View space
The frame is the view reference frame VRF



NDC space (Canonical viewing volume)



Viewport space



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 OpenGL 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 θ 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 θ 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 θ 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.

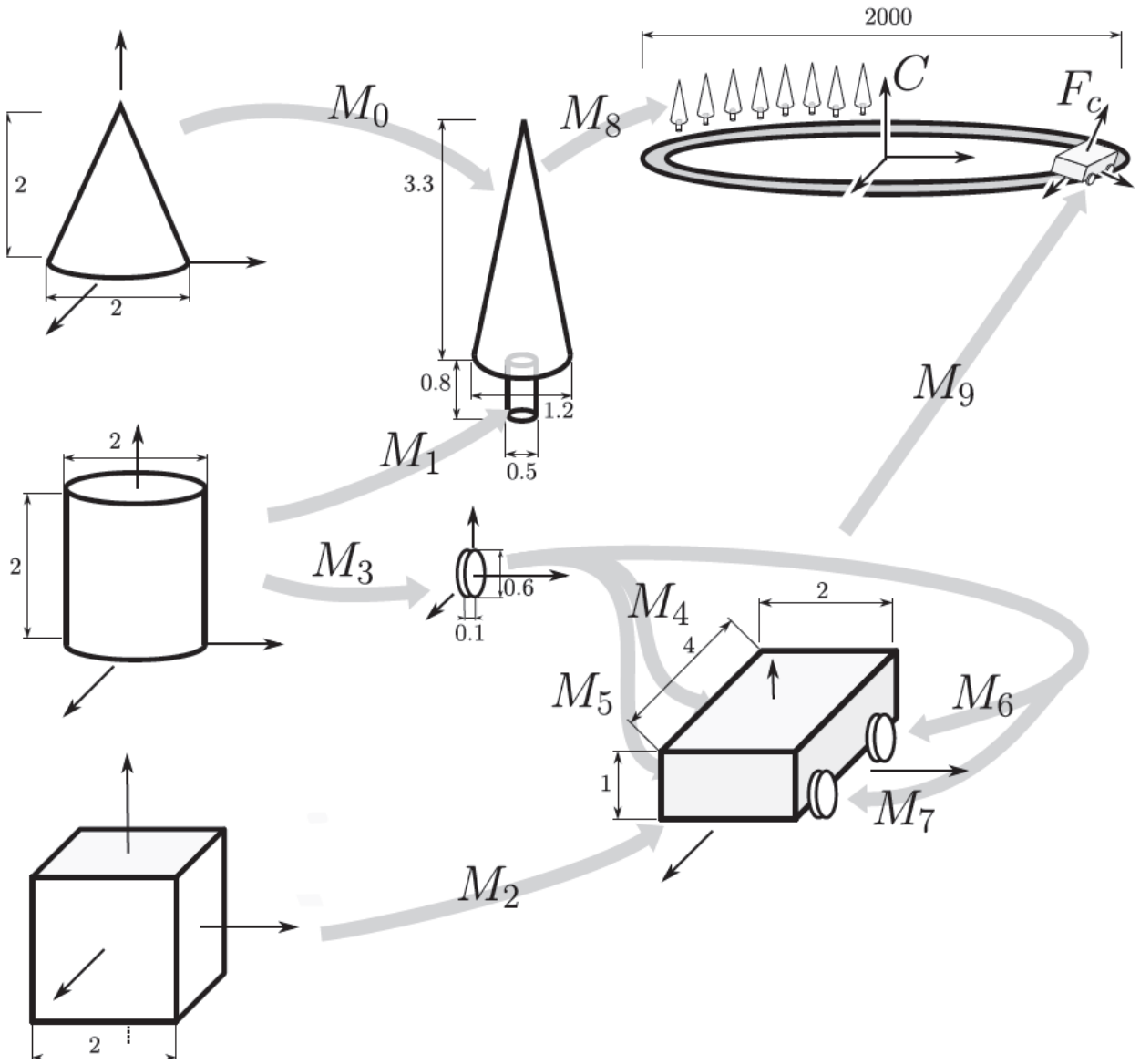
Rotacja względem osi Y nazywana jest też odchyleniem (ang. yaw), względem osi X pochyleniem (ang. pitch), względem osi Z przechyleniem (ang. roll).

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, θ – kąt obrotu oraz $c = \cos\theta$, $s = \sin\theta$.


2000
C
F_c
M_0
M_8
M_9
M_1
M_2
M_3
M_4
M_5
M_6
M_7
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.

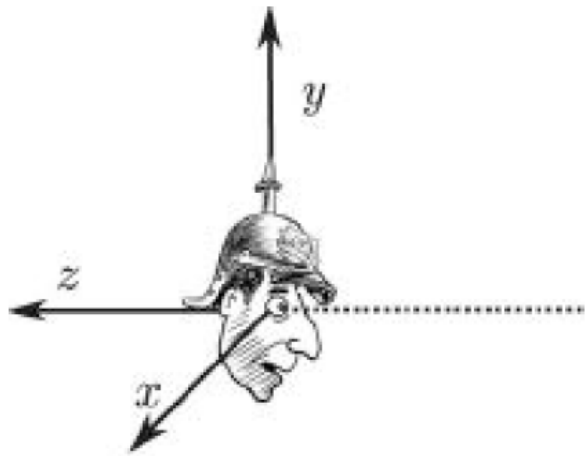
$$\mathrm{z} = \frac{e - c}{\|e - c\|} \qquad \mathrm{x} = \frac{\mathrm{p} \times \mathrm{z}}{\|\mathrm{p} \times \mathrm{z}\|} \qquad \mathrm{y} = \mathrm{z} \times \mathrm{x},$$

gdzie $\times$ oznacza iloczyn wektorowy.

Macierz widoku ma wówczas postać:

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

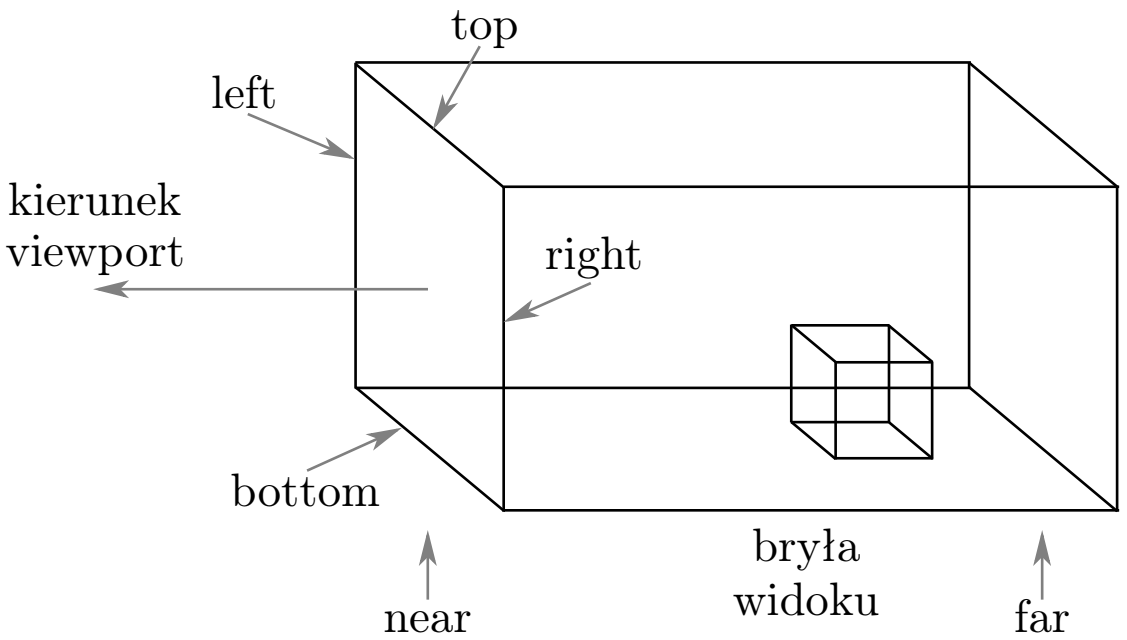
gdzie $\cdot$ oznacza iloczyn skalarny.



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

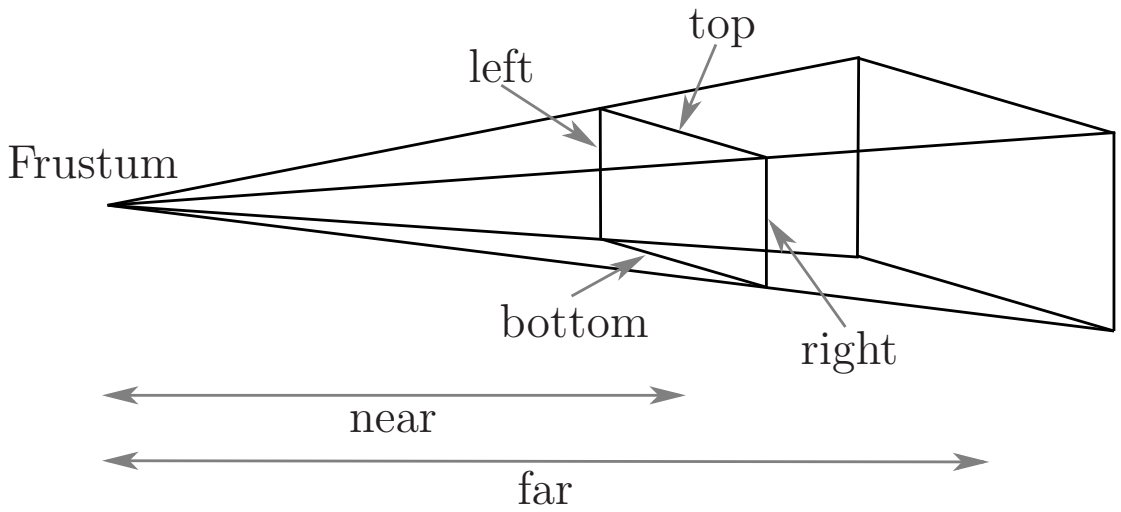
Oba typy rzutowania są definiowane przez tzw. bryłę widzenia, która w przypadku rzutowania ortogonalnego jest prostopadłościanem, a w przypadku rzutowania perspektywicznego ściętym ostrosłupem o podstawie prostokąta.

Rzutowanie ortogonalne

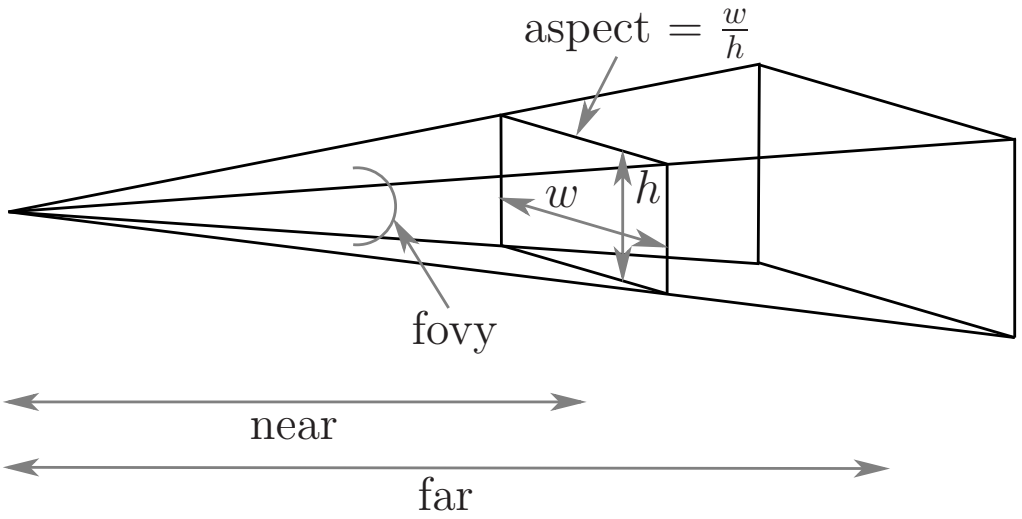


$$\begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

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

Biblioteka OpenGL Mathematics

Biblioteka OpenGL Mathematics (GLM) jest biblioteką składającą się wyłącznie z plików nagłówkowych, w której mamy zaimplementowane macierze oraz wiele innych przydatnych rzeczy związanych z matematyką. Stosuje ona konwencję nazewnictwa jaką mamy w GLSL.

Dostępne typy wektorowe: `vec2`, `vec3`, `vec4`, `dvec2`, `dvec3`, `dvec4`, `ivec2`, `ivec3`, `ivec4`, `uvec2`, `uvec3`, `uvec4`, `bvec2`, `bvec3`, `bvec4`.

Dostępne typy macierzowe: `mat2`, `mat3`, `mat4`, `mat2x2`, `mat2x3`, `mat2x4`, `mat3x2`, `mat3x3`, `mat3x4`, `mat4x2`, `mat4x3`, `mat4x4`, `dmat2`, `dmat3`, `dmat4`, `dmat2x2`, `dmat2x3`, `dmat2x4`, `dmat3x2`, `dmat3x3`, `dmat3x4`, `dmat4x2`, `dmat4x3`, `dmat4x4`.

Dla typów wektorowych i macierzowych przeciążone są podstawowe operatory arytmetyczne.

W rozszerzeniu `GLM_EXT_matrix_transform` znajdują się funkcje odpowiedzialne za tworzenie macierzy przekształceń:

- Translacja

  ```
  mat4 translate(mat4 const &m, vec3 const &t);
  ```

  `m` – macierz, do której domnażamy (z prawej) macierz translacji, `t` – wektor translacji

- Skalowanie

  ```
  mat4 scale(mat4 const &m, vec3 const &s);
  ```

  `m` – macierz, do której domnażamy (z prawej) macierz skalowania, `s` – wektor ze współczynnikami skali

- Rotacja

  ```
  mat4 rotate(mat4 const &m, float angle, vec3 const &axis);
  ```

  `m` – macierz, do której domnażamy (z prawej) macierz rotacji, `angle` – kąt obrotu (w radianach), `axis` – oś obrotu

- Macierz widoku

  ```
  mat4 lookAt(vec3 const &e, vec3 const &c, vec3 const &u);
  ```

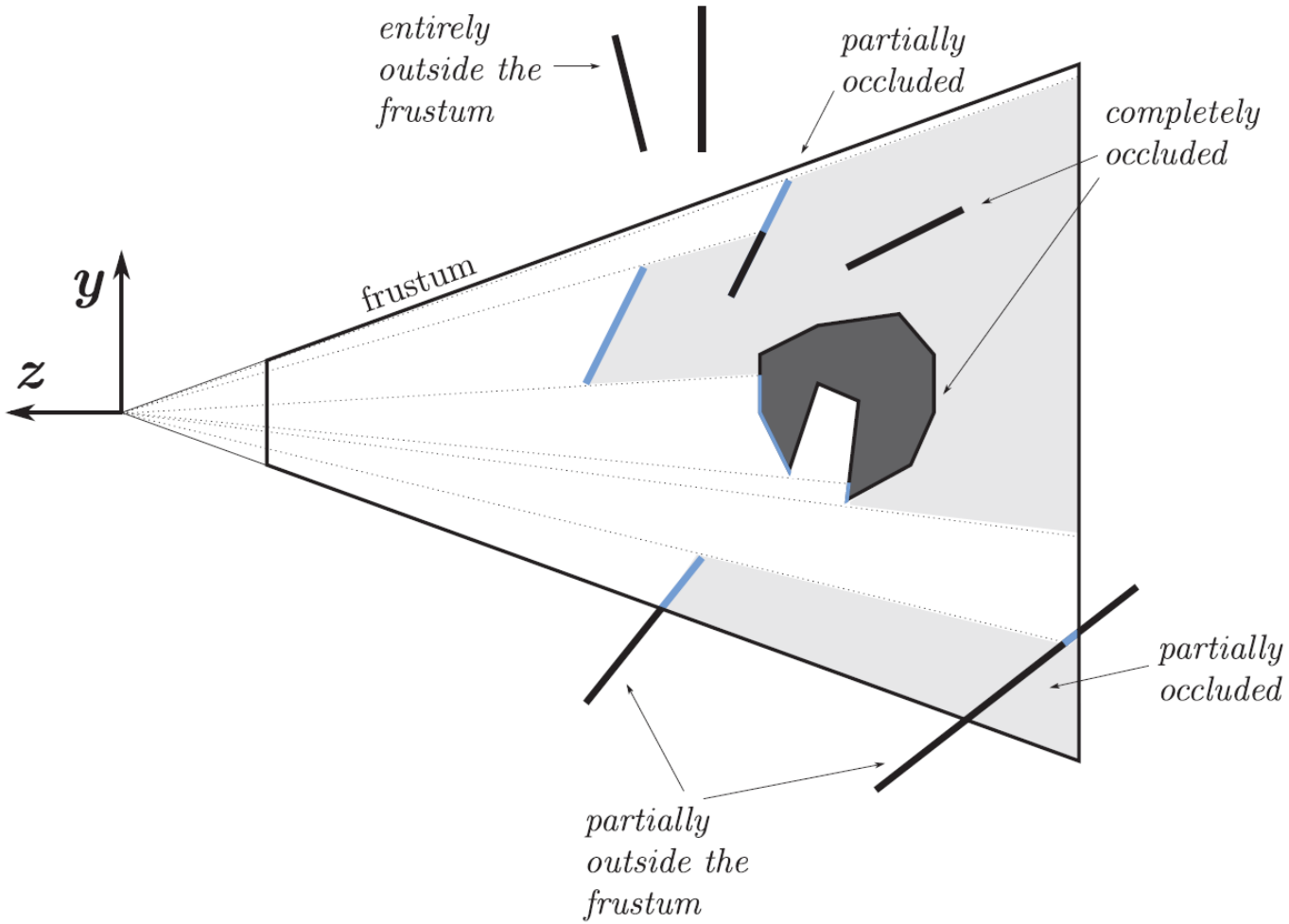
  `e` – położenie oka, `c` – punkt, na który patrzymy, `u` – wektor do góry
- Rzutowanie ortogonalne

  ```
  mat4 ortho(float left, float right, float bottom, float top, float
  zNear, float zFar);
  ```
- Rzutowanie perspektywiczne 1

  ```
  mat4 frustum(float left, float right, float bottom, float top,
  float near, float far);
  ```
- Rzutowanie perspektywiczne 2

  ```
  mat4 perspective(float fovy, float aspect, float near, float far);
  ```

Usuwanie powierzchni niewidocznych

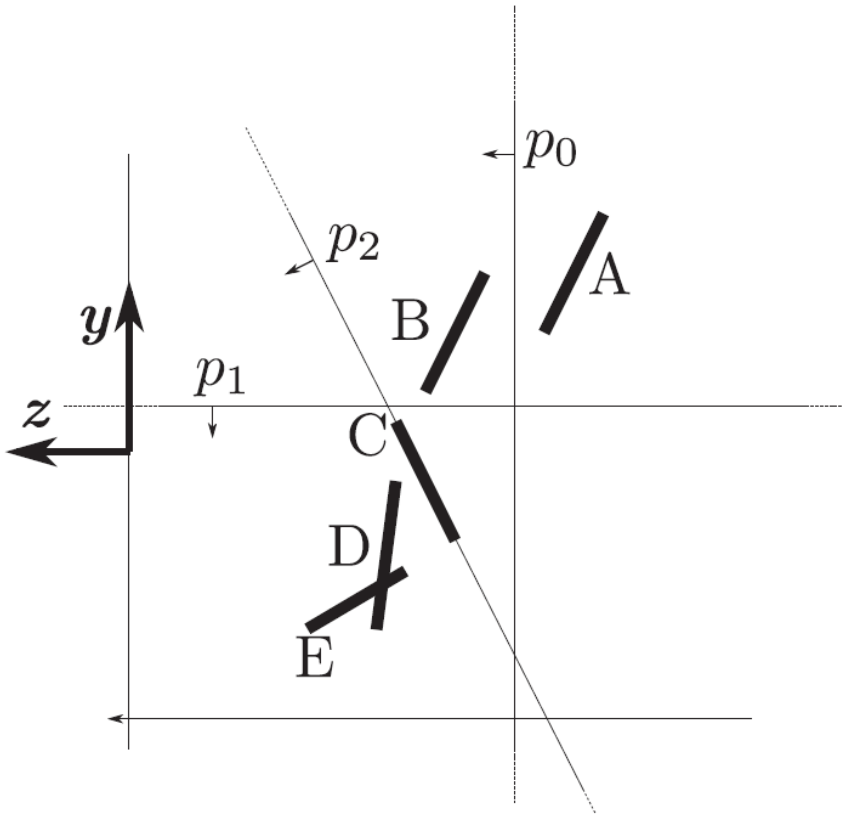


Sortowanie głębi

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).

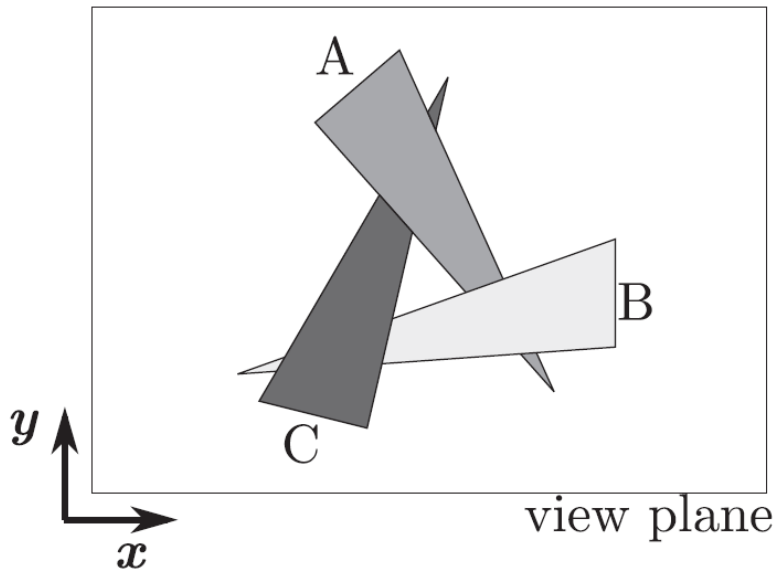
W pierwszych podejściach użycia algorytmu malarza jako odległość prymitywu od obserwatora brano odległość barycentrum prymitywu od obserwatora. Takie podejście nie gwarantuje, że jeśli barycentrum prymitywu A jest bliżej niż barycentrum prymitywu B, to całe A jest bliżej niż B.

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.

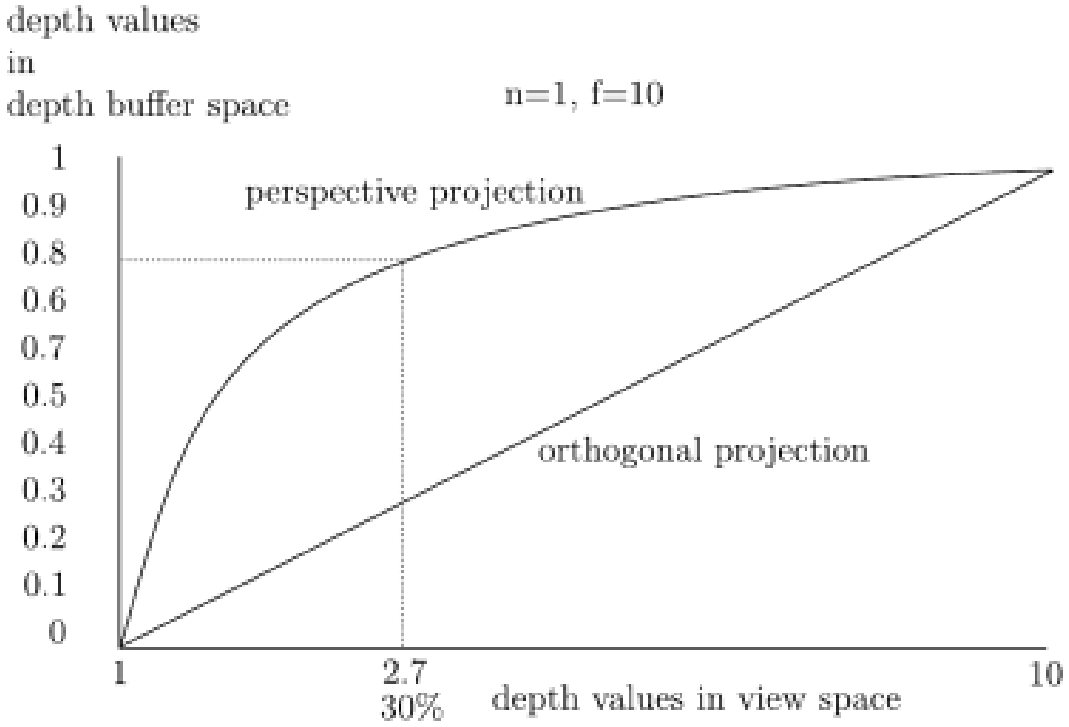
Nawet jeśli prymitywy nie przecinają się może zajść przypadek, w którym nie będziemy w stanie znaleźć odpowiedniego porządku, np. część A jest za B, część B jest za C i część C jest za A.



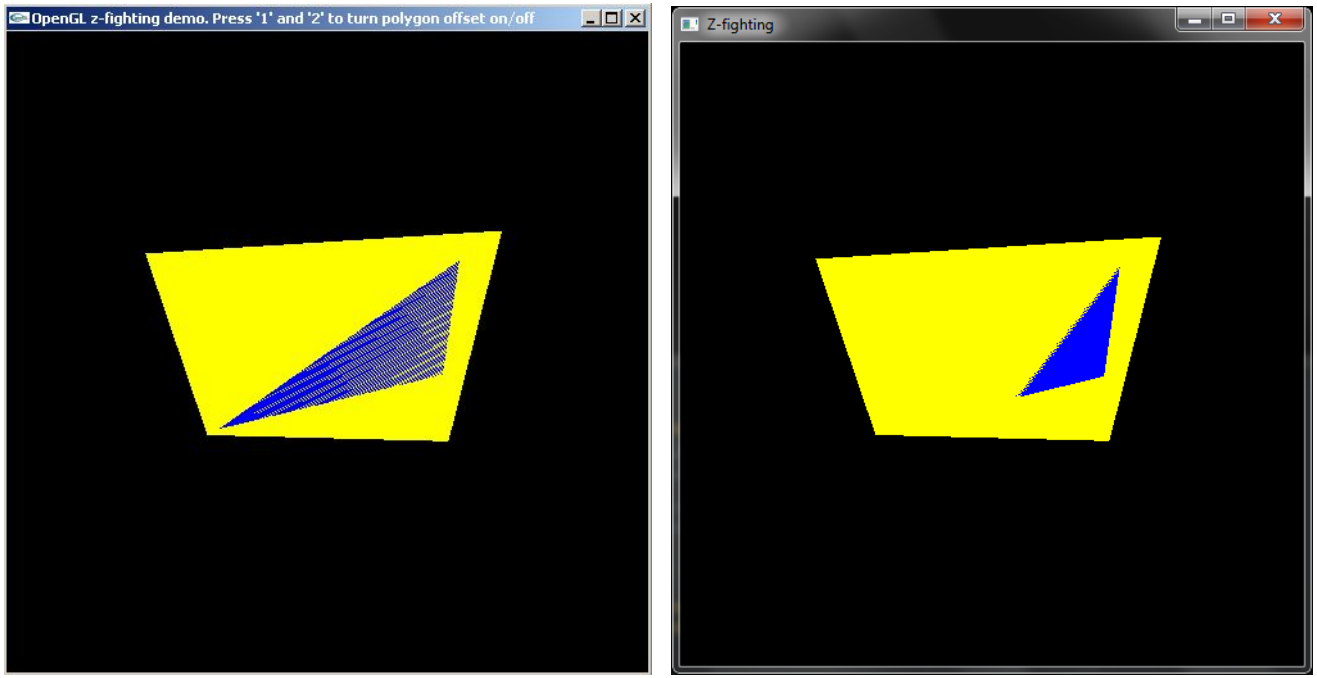
Bufor głębi (ang. depth bufer) lub bufor z (ang. z-buffer)

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.



Żeby korzystać z bufora głębi należy:

- włączyć bufor głębi (domyślnie bufor ten jest wyłączony) za pomocą funkcji `glEnable` z parametrem `GL_DEPTH_TEST`,
- w zależności od wykorzystywanej biblioteki do tworzenia okna może być konieczne włączenie bufora głębi, np. w bibliotece freeGLUT należy dodać stałą `GLUT_DEPTH` przy wywołaniu funkcji `glutInitDisplayMode`,

Wyłączenie/włączenie zapisu do bufora głębi:

```
void glDepthMask(GLboolean flag);
```

`flag = GL_TRUE` – włączenie zapisu, `flag = GL_FALSE` – wyłączenie zapisu.

Wyłączenie/włączenie zapisu do bufora głębi:

```
void glDepthMask(GLboolean 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 `glClear`.

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

```
void glClearDepth(GLclampd depth);
```

Wyłączenie/włączenie zapisu do bufora głębi:

```
void glDepthMask(GLboolean 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 `glClear`.

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

```
void glClearDepth(GLclampd depth);
```

Do zmiany zakresu wartości bufora głębi służy funkcja:

```
void glDepthRange(GLclampd zNear, GLclampd zFar);
```

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

Wybór testu bufora głębi dokonujemy za pomocą funkcji:

```
void glDepthFunc(GLenum func);
```

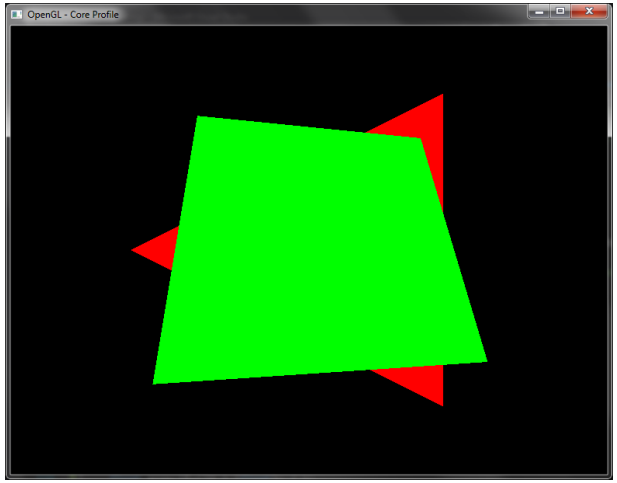
gdzie `func` przyjmuje jedną z wartości (z – testowana wartość, z_b – wartość w buforze):

- `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.

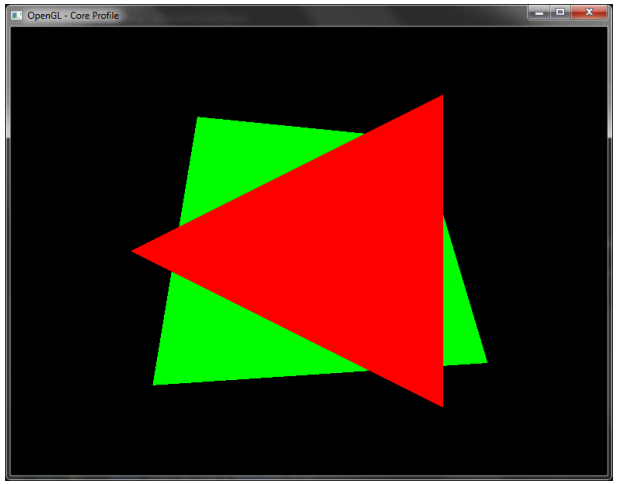
```
float triangleVertices[] =
{
  -0.8f, 0.0f, -0.2f, 1.0f,
  0.6f, -0.7f, -0.2f, 1.0f,
  0.6f, 0.7f, -0.2f, 1.0f
};

float quadVertices[] =
{
  -0.5f, 0.6f, -0.5f, 1.0f,
  -0.7f, -0.6f, -0.5f, 1.0f,
  0.5f, 0.5f, -0.5f, 1.0f,
  0.8f, -0.5f, -0.5f, 1.0f
};
```

Kolejność rysowania: trójkąt, czworokąt.

Wyłączony test głębi



Włączony test głębi



Jednym ze sposobów walki z *z*-fighting jest wprowadzone w OpenGL 1.1 przesunięcie wartości głębi. Jest to mechanizm pozwalający na przesuwanie wartości głębi pikseli przy rysowaniu wielokątów. Przesunięcie obliczane jest według wzoru:

$$m \cdot factor + r \cdot units,$$

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:

```
void glPolygonOffset(GLfloat factor, GLfloat units);
```

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

Pozostałe dwa współczynniki skalowania ustawiane są za pomocą funkcji:

```
void glPolygonOffset(GLfloat factor, GLfloat 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 jednym z parametrów:

- `GL_POLYGON_OFFSET_POINT`,
- `GL_POLYGON_OFFSET_LINE`,
- `GL_POLYGON_OFFSET_FILL`.

Oprócz przesunięcia głębi jakie oferuje OpenGL możemy również dokonać tego przesunięcia sami. Do tego celu w shaderze fragmentów mamy zmienną:

```
out float gl_FragDepth;
```

Używając zmiennej `gl_FragDepth`, to w jaki sposób obliczymy wartość głębi danego fragmentu zależy od nas i nie musi to być wzór, który był przedstawiony kilka slajdów wcześniej.

Elementy GLSL

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 do wersji 4.30 dostępne są jedynie tablice jednowymiarowe. W wersji 4.30 dodano tablice dwuwymiarowe.
- 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 `glUniform2fv`, `...`, `glUniformMatrix4fv`. Argument `count` tych funkcji oznacza liczbę elementów do przesłania, np. rozmiar tablicy.

Dla atrybutów nie ma specjalnej funkcji pozwalającej przesłać całą tablicę do shadera wierzchołków. W tym przypadku musimy pobrać lokalizację każdego elementu tablicy używając `glGetAttribLocation`. 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 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

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