Zachodniopomorski
Uniwersytet
Technologiczny
w Szczecinie

Bartosz Bazyluk

OpenGL

Współczesne podejście do programowania grafiki

Część II: Programy cieniujące (*shadery*)

Programowanie Gier Komputerowych, Informatyka S1, III Rok

Wydział
Informatyki

PLAN WYKŁADU

- **Transformacje** geometryczne
  - Pożegnanie z "dobrodziejstwami" OpenGL < 3
  - Macierze transformacji i układy współrzędnych
- Programowalny **potok renderowania** w OpenGL
- Tworzenie **programów cieniujących** z użyciem GLSL
  - Kompilacja i linkowanie
  - Dane wejściowe: atrybuty wierzchołków, uniforms, we/wy.

Przekształcenia geometryczne

- Przekształcenia geometryczne w przestrzeni 3D opisane są **macierzami 4x4**
  - Aby **poddać współrzędne** (x, y, z) **danemu przekształceniu** T, macierz T mnożona jest przez wektor (rozszerzony o dodatkową współrzędną $w = 1$), a cały wynik jest dzielony przez ostatnią współrzędną w'

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} T_{00} & T_{01} & T_{02} & T_{00} \\ T_{10} & T_{11} & T_{12} & T_{13} \\ T_{20} & T_{21} & T_{22} & T_{23} \\ T_{30} & T_{31} & T_{32} & T_{33} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Przekształcenia geometryczne

- Podstawowe transformacje geometryczne to:
  - Translacja
  - Rotacja
  - Skalowanie
  - Projekcja
- Macierze, które je opisują:

X-Rotation in 3D

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Z-Rotation in 3D

$$\begin{bmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scale in 3D

$$\begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Y-Rotation in 3D

$$\begin{bmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translation in 3D

$$\begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Przekształcenia geometryczne: **Przykład**

- Translacja o wektor (4, -6, 2):

$$M_T = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad T_{xyz} = (4, -6, 2)$$

Przekształcenia geometryczne: **Przykład**

- Translacja o wektor (4, -6, 2):

$$M_T = \begin{bmatrix} 1 & 0 & 0 & 4 \\ 0 & 1 & 0 & -6 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Niech *p* będzie punktem, który chcemy poddać translacji:

$p = (1, 2, 3)$ $\qquad$ $p' = ?$

(punkt po transformacji)

Przekształcenia geometryczne: **Przykład**

- Translacja o wektor (4, -6, 2):

Niech *p* będzie punktem, który chcemy poddać translacji:

$p = (1, 2, 3)$ $\qquad$ $p' = ?$

$$p' = \begin{bmatrix} 1 & 0 & 0 & 4 \\ 0 & 1 & 0 & -6 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 2 \\ 3 \\ 1 \end{bmatrix} = \ldots ?$$

Przekształcenia geometryczne: **Przykład**

- Translacja o wektor (4, -6, 2):

Niech *p* będzie punktem, który chcemy poddać translacji:

$p = (1, 2, 3)$ $\qquad$ $p' = ?$

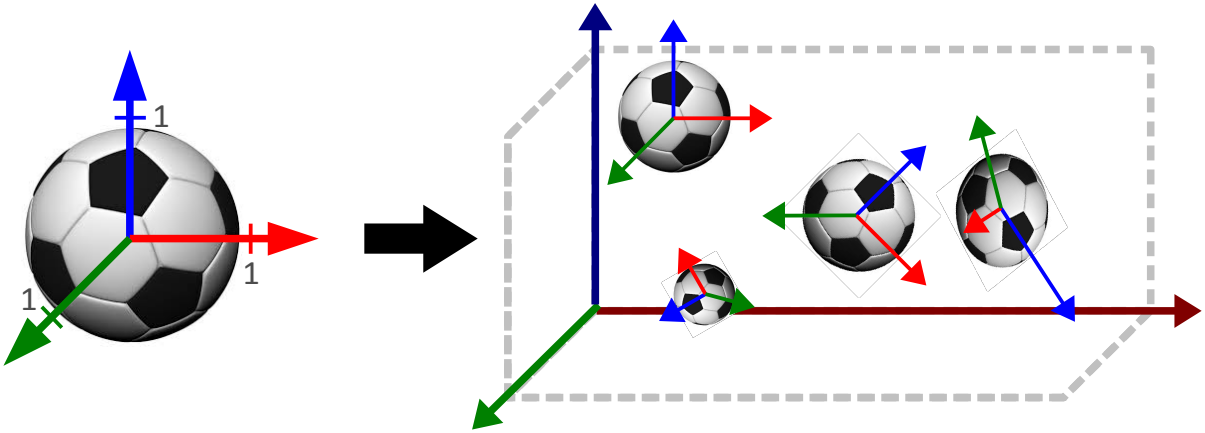
$$p' = \begin{bmatrix} 1 & 0 & 0 & 4 \\ 0 & 1 & 0 & -6 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 2 \\ 3 \\ 1 \end{bmatrix} =$$

$$= \begin{bmatrix} 1 \cdot 1 + 0 \cdot 2 + 0 \cdot 3 + 4 \cdot 1 \\ 0 \cdot 1 + 1 \cdot 2 + 0 \cdot 3 - 6 \cdot 1 \\ 0 \cdot 1 + 0 \cdot 2 + 1 \cdot 3 + 2 \cdot 1 \\ 0 \cdot 1 + 0 \cdot 2 + 0 \cdot 3 + 1 \cdot 1 \end{bmatrix} = \begin{bmatrix} 5 \\ -4 \\ 5 \\ 1 \end{bmatrix}$$

(aby **pozbyć się czwartej współrzędnej**, należy najpierw cały wektor **podzielić przez jej wartość**)

Przekształcenia geometryczne

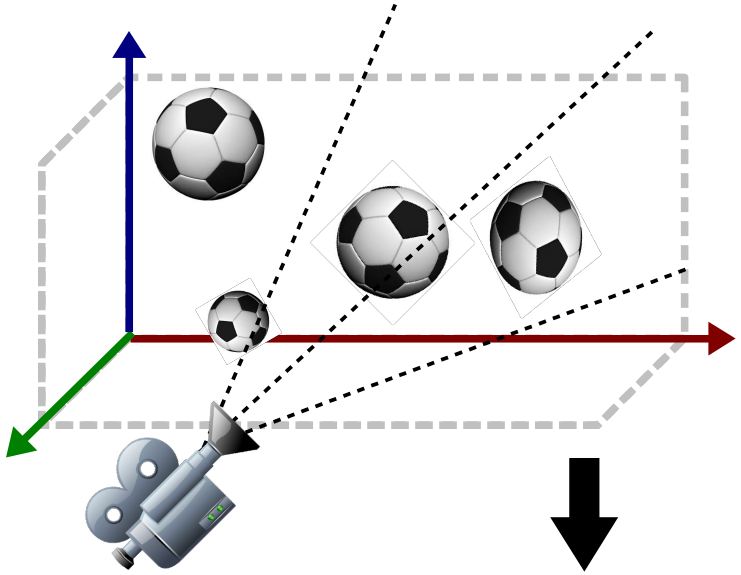
- **Macierz modelu** jest złożeniem odpowiednich translacji, rotacji i skalowań
  - Czyli **iloczynem macierzy** odpowiadających takim transformacjom
  - Ważna jest **kolejność** mnożenia!
- Służy do **umieszczenia naszego modelu**, opisanego lokalnymi współrzędnymi, **w pożądanym miejscu** wirtualnego **świata**
- Dzięki takiemu podejściu możliwe jest **wielokrotne używanie** dokładnie tego samego modelu, opisanego **tymi samymi współrzędnymi wierzchołków**



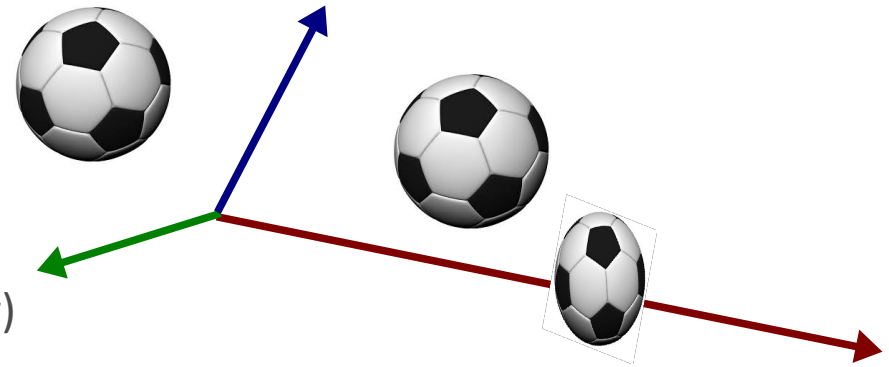
(współrzędne **modelu**) (współrzędne **świata**)

Przekształcenia geometryczne

- **Macierz widoku**, a więc macierz związana z transformacją kamery, jest złożeniem odpowiednich **translacji** i **rotacji**
  - Czyli **iloczynem macierzy** odpowiadających takim transformacjom



(współrzędne **świata**)

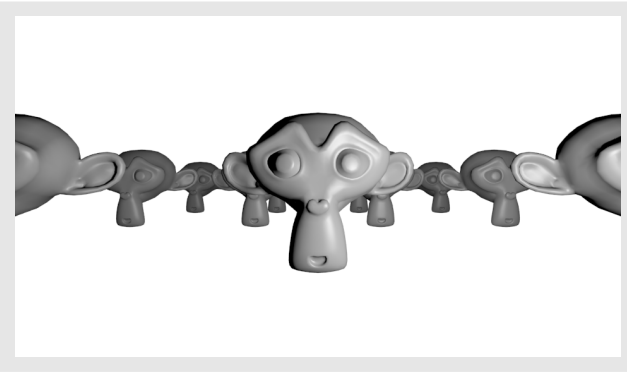


(współrzędne **kamery**)

Projekcja

- Jest to **sposób odwzorowania** wycinka trójwymiarowego świata na płaszczyznę ekranu
- Istnieją dwa podstawowe **rodzaje rzutowania** (projekcji):

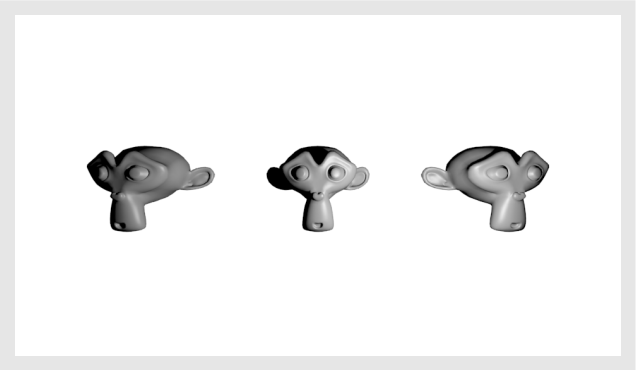
Rzut perspektywiczny

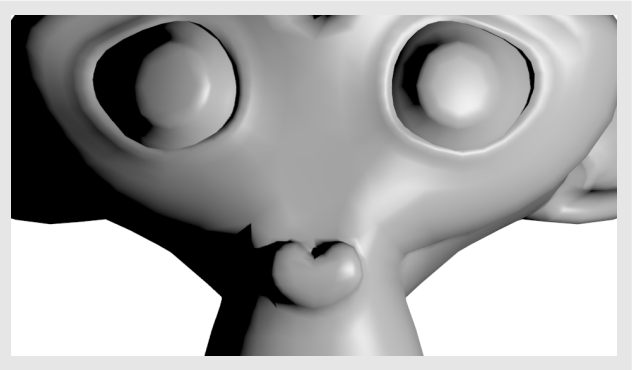


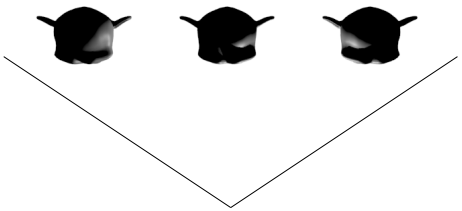
Rzut prostokątny
(ang. *orthographic*)

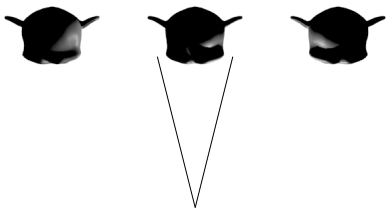
Kamera perspektywiczna

- Dodatkowo kamerę określają:
  - **Kąt widzenia** (najczęściej w stopniach) ***fov***
    - Jego zmiana odpowiada **zmianie ogniskowej obiektywu** (popularnie zwanej "*zoomem*" w aparacie fotograficznym)
    - Nie jest tożsamy z przybliżeniem/oddaleniem kamery!







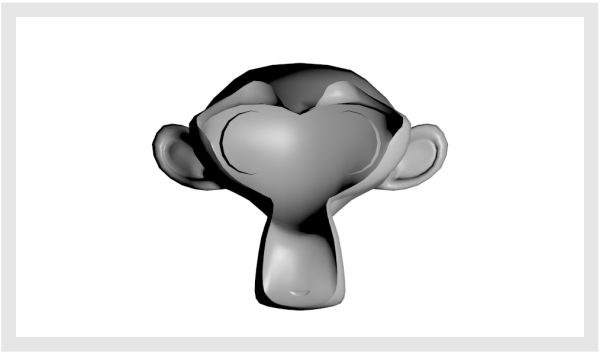


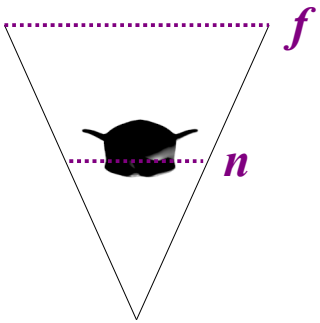
Szeroki kąt
(krótka ogniskowa)

Wąski kąt
(długa ogniskowa)

Kamera perspektywiczna

- Dodatkowo kamerę określają:
  - **Odległość płaszczyzn przycinania: bliskiej n i dalekiej f** (ang. *near/far clipping plane*)
    - Określają, od jakiej do jakiej odległości będzie renderowana scena
    - Ma to związek z **precyzją bufora głębokości**
      - Im większy zakres odległości jest renderowany, tym większa szansa powstania problemów typu *z-fighting* – wartości powinny zostać dobrane odpowiednio do charakterystyki danej sceny





Kamera perspektywiczna

- Dodatkowo kamerę określają:
  - **Proporcje** boków podstawy ściętego ostrosłupa (ang. *frustum*)
    - Stosunek szerokości do wysokości
      - 4:3, 16:9, 16:10, ...

Przekształcenia geometryczne

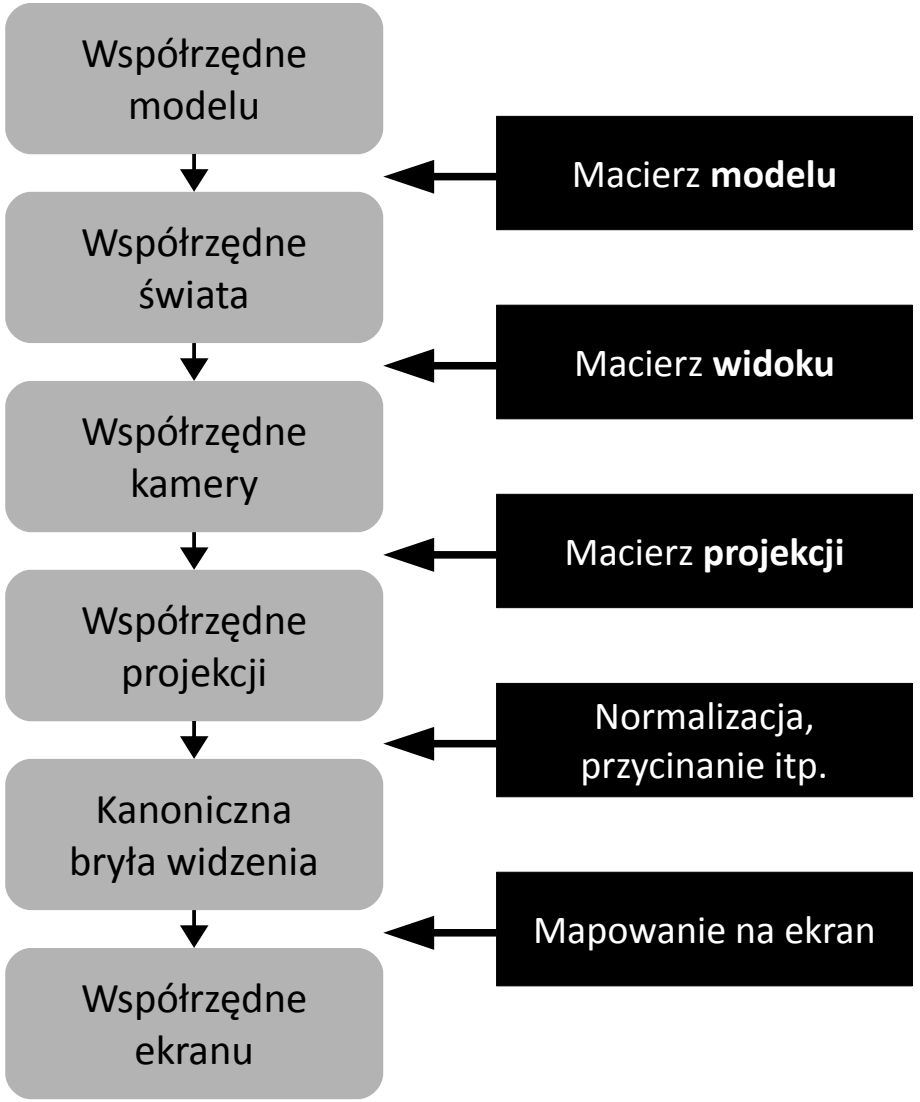
- Macierz **projekcji**, odpowiada za przejście do współrzędnych kanonicznej bryły widzenia
  - Zakres **od -1 do +1** w każdym z wymiarów (odwrócony *z*!)
  - Macierz **projekcji perspektywicznej** wygląda następująco:

$$\begin{bmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{zFar+zNear}{zNear-zFar} & \frac{2\times zFar\times zNear}{zNear-zFar} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

    - Gdzie:
      - $f = 1 / tan(fov / 2)$
      - ***aspect*** = *proporcje*
      - ***zNear*** = *odległość bliskiej płaszczyzny przycinania*
      - ***zFar*** = *odległość dalekiej płaszczyzny przycinania*

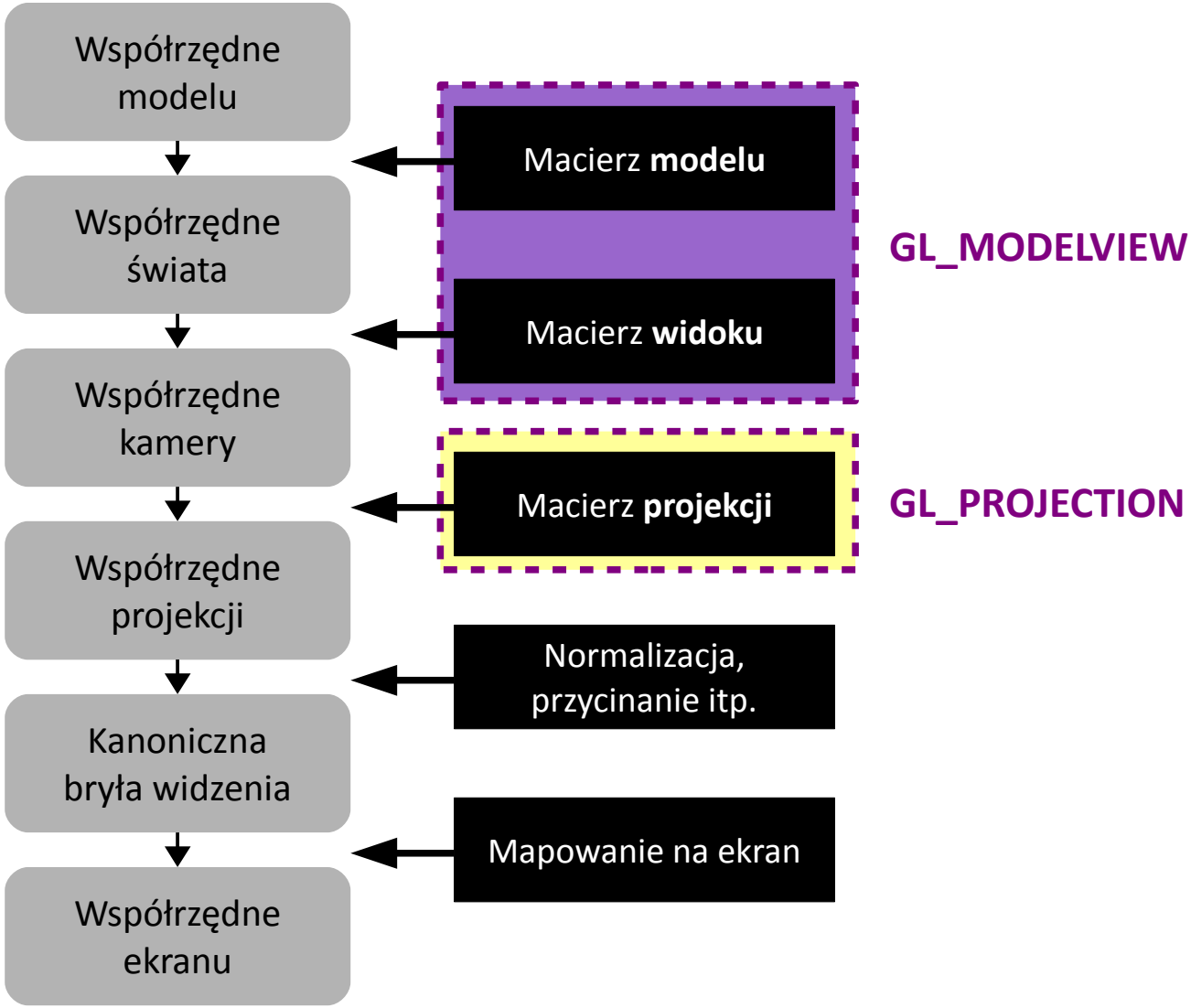
Transformacje w OpenGL

- Macierze transformacji i układy współrzędnych w **potoku renderowania**



Transformacje w OpenGL

- Macierze będące **częścią stanu** dawnego *OpenGL*



Transformacje w OpenGL

- Częścią stanu dawnego OpenGL są dwie podstawowe **macierze transformacji**:
  - Macierz **Modelu-Widoku** (**GL_MODELVIEW**)
    - Łączy w sobie przekształcenia związane zarówno z przejściem **ze współrzędnych modelu do współrzędnych świata**, jak i **ze współrzędnych świata do współrzędnych kamery**
      - Czyli uwzględnia zarówno **przekształcenie konkretnego obiektu na scenie**, jak i **położenie i obrót kamery**
  - Macierz **Projekcji** (**GL_PROJECTION**)
    - Uwzględnia **rodzaj projekcji** (perspektywiczna/prostokątna), **kąt widzenia**, **płaszczyzny przycinania**, **proporcje**.

- Tylko **jedna** z tych macierzy jest w danym momencie **edytowalna**
  - Wyboru aktualnie edytowanej macierzy dokonuje się za pomocą funkcji **glMatrixMode**(*matrix*)
    - *matrix* – **GL_MODELVIEW** lub **GL_PROJECTION**

Transformacje w OpenGL

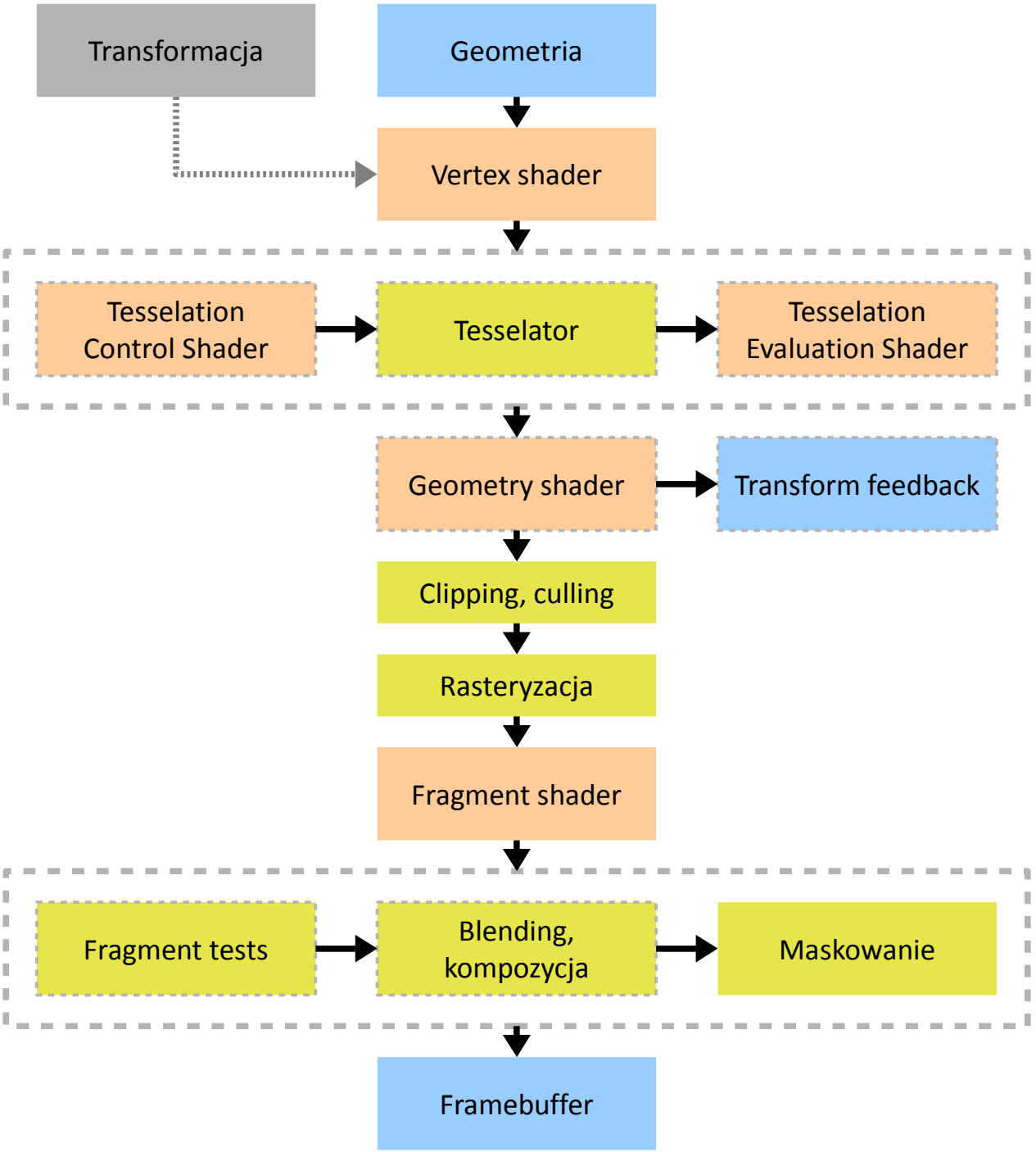
- Do tej pory w celu **modyfikacji zawartości macierzy transformacji**, wykorzystywało się funkcje jak np.:
  - *glTranslatef*(), *glRotatef*(), *glScalef*(), *glLoadIdentity*(), *glMultMatix*(), *glLoadMatrix*()...
  - *glPushMatrix*(), *glPopMatrix*()
- Począwszy od *OpenGL 3.0* są one uznane za *deprecated*, natomiast **w *OpenGL 3.1* zostały usunięte** (poza *compatibility profile*)
- Obecnie obsługa jak i przechowywanie transformacji **leżą w gestii aplikacji**, nie *OpenGL*.
- Wykorzystywane **shadery muszą obsługiwać** podawane na wejście **macierze transformacji**

Transformacje w OpenGL

- Wskazana jest własna implementacja, albo użycie gotowej biblioteki. Przydatne rozwiązania to np.:
  - ***GLM*** (*OpenGL Mathematics*)
    - Składnia mająca naśladować *GLSL*
    - http://glm.g-truc.net/
  - ***Armadillo***
    - Składnia mająca naśladować *MATLABa*
    - http://arma.sourceforge.net/
- Czego należy oczekiwać od rozwiązania obsługującego transformacje:
  - Użyteczne **typy danych** (wektor, macierz) i **operacje na nich**
  - Łatwe **tworzenie macierzy odpowiadających prostym przekształceniom**, jak np. translacja, rotacja, skalowanie
  - Łatwa możliwość uzyskania **wskaźnika do tablicy wartości**
  - Zapamiętywanie **stanu transformacji**
    - Wymaga przemyślanego zaprojektowania **architektury sceny**

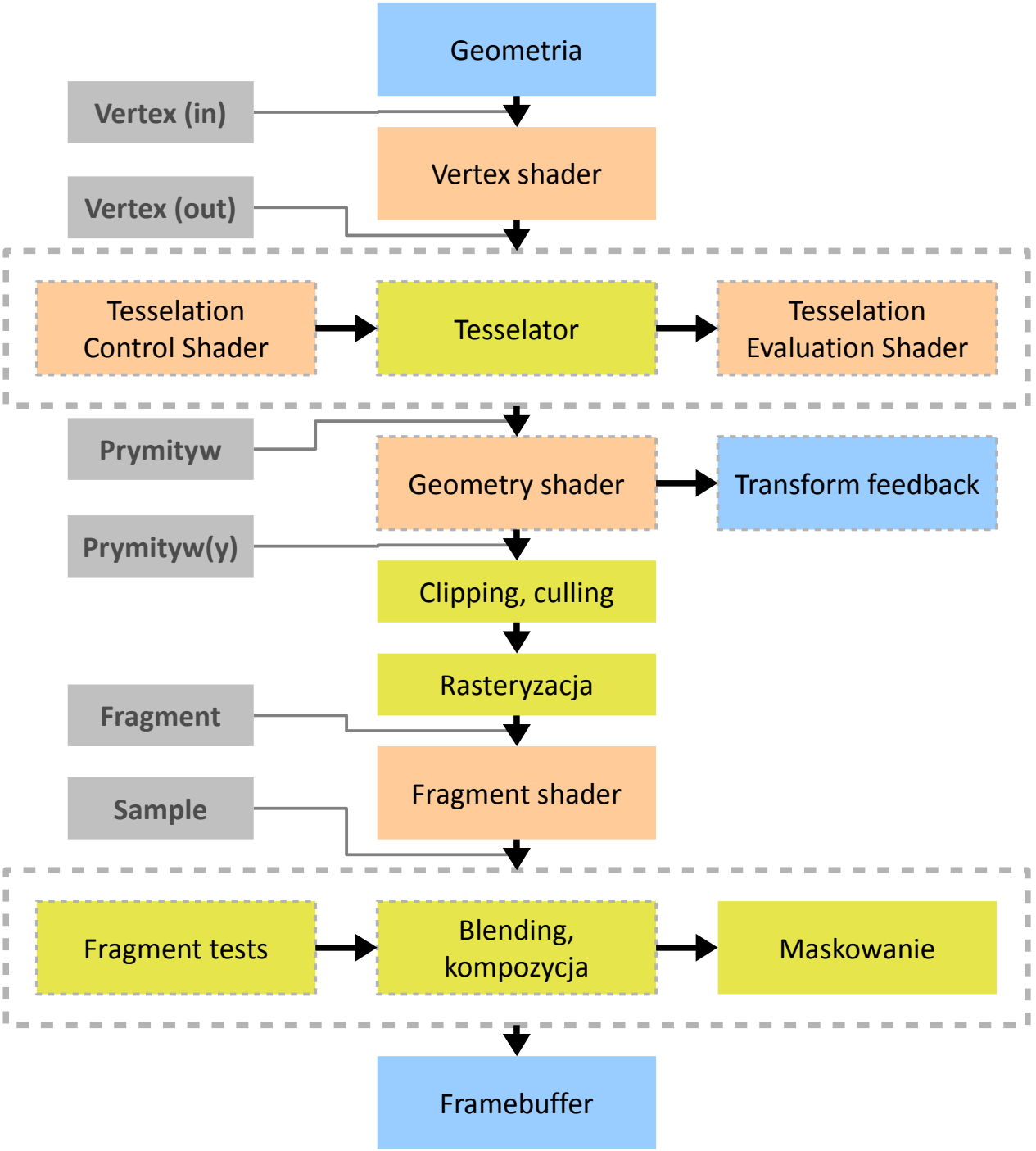
Programowalny potok renderowania

OpenGL 4



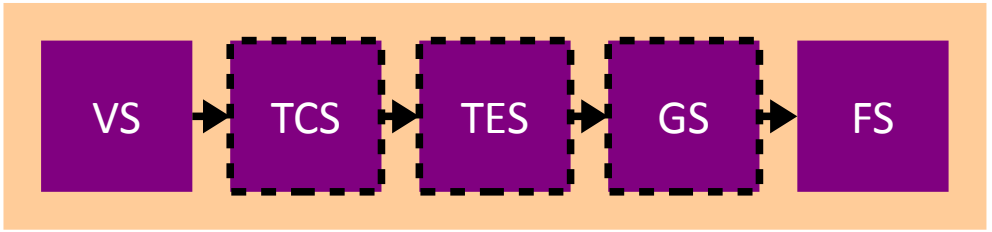
Programowalny potok renderowania

OpenGL 4



Programy cieniujące

- **Programem cieniującym** (ang. *shading program*) nazywamy **zbiór współdziałających programów** wykonywanych przez kartę graficzną w poszczególnych etapach potoku renderowania
- Program cieniujący musi składać się przynajmniej z:
  - **Vertex** shader (VS)
  - **Fragment** (Pixel) shader (FS/PS)
    - Poprawniejszą nazwą wydaje się FS, jako że operuje on na *fragmentach* – kandydatach na wartości wpływające na piksele, a nie na samych wyjściowych *pikselach*
- Opcjonalnie można podłączyć (ang. *attach*) także:
  - **Geometry** shader (GS) (*OpenGL 3.2+*)
  - **Tesselation** shaders (TCS + TES) (*OpenGL 4.0+*)



Programy cieniujące

- Programy wykonywane **w obrębie poszczególnych etapów** potoku będziemy nazywać ***shaderami***
- Kod shaderów w przypadku *OpenGL* pisany jest **z użyciem języka *GLSL*** *(GL Shading Language)*
  - Składniowo zbliżony jest do *C*
  - Język ten ewoluuje wraz z kolejnymi wersjami *OpenGL*
- Sterownik karty graficznej **kompiluje** kod shaderów do postaci wykonywalnej
  - Każdy z *shaderów* osobno
- Sterownik karty graficznej dokonuje **łączenia** (ang. *linking*) całego programu cieniującego
  - Powiązanie wyjść z wejściami
  - Przypisanie identyfikatorów (*locations*) zmiennym

Programy cieniujące

- **Proces utworzenia programu cieniującego** przebiega następująco:
  - Utworzenie programu

```
GLuint programId = glCreateProgram();
```

  - Utworzenie shaderów (z podaniem typu)

```
GLuint shaderId = glCreateShader(type);
```

  - Przekazanie kodów źródłowych shaderów

```
glShaderSource(shaderId, 1, &shaderCode, NULL)
```

  - Kompilacja shaderów

```
glCompileShader(shaderId);
```

  - Podłączenie do (aktywnego) programu

```
glAttachShader(programId, shaderId);
```

  - Linkowanie programu

```
glLinkProgram(programId);
```

Programy cieniujące

- W danym momencie tylko jeden program cieniujący jest **aktywny**
  - Aktywacja:

```
glUseProgram(programId);
```

  - Deaktywacja:

```
glUseProgram(0);
```

- Program posiada swój **aktualny stan**
  - Najistotniejszym elementem stanu są akutalne wartości zmiennych *uniform*, czyli współdzielonych pomiędzy shaderami danych wejściowych

Dane wejściowe shaderów

- Dane wejściowe do shaderów:
  - **Atrybuty** wierzchołków (*attributes*)
    - Inne dla każdego z przetwarzanych przez *VS* wierzchołków
    - Pobierane z *VBO* na podstawie zdefiniowanych adresów
    - Nie można modyfikować w *shaderach*
  - **Uniforms**
    - Jednakowe wartości na wszystkich etapach potoku
    - Nie można modyfikować w *shaderach*
  - **Wyjścia** z poprzednich etapów potoku
    - Ustawiane przez poprzedzający *shader*
    - W przypadku *VS* ich rolę pełnią atrybuty wierzchołków
    - Dla *FS* interpolowane w obrębie prymitywów

Vertex attributes

- Identyfikator atrybutu
  - Potrzebny przy wskazywaniu adresu
  - Sposoby ustawienia:
    - W kodzie shadera

```
layout(location = 0) in vec3 v3Position;
layout(location = 1) in vec3 v3Normal;
```

    - Przed linkowaniem (w kodzie programu)

```
glBindAttribLocation(programId, 0, ”v3Position”);
glBindAttribLocation(programId, 1, ”v3Normal”);
```

    - Automatycznie
      - Pobranie id na podstawie nazwy: **glGetAttribLocation**(*name*)
  - Dobra praktyka:
    - **Nie odpytywać o id za każdym razem**, gdy go potrzebujemy
      - Zapamiętujemy, albo sami ustalamy
    - Własnoręcznie ustalić id, ale w bezpieczny sposób
      - Tak, by uniknąć kolizji z przyszłymi, potencjalnymi id
      - Np. stałe numeryczne zgromadzone w jednym miejscu

Vertex attributes

- Ustalanie wartości
  - W *OpenGL 3+* nie ma już specjalnych tablic dla pozycji wierzchołków, wektorów normalnych czy współrzędnych tekstur. **Wszystkie tablice są zupełnie *generyczne* i ich interpretacja należy do programisty**
  - **glVertexAttribPointer**(*id*, *num*, *type*, *normalized*, *stride*, *offset*)
    - Ostatni parametr jest offsetem, a więc liczbą bajtów od początku definicji wierzchołka do początku danej wartości
      - Pracując na strukturach w C++ warto użyć makra *offsetof*
  - **glEnableVertexAttribArray**(*id*)
    - Początkowo wszystkie tablice są **wyłączone**
  - Stan tablic warto zapamiętać w *VAO*

Vertex Array Object (VAO)

- Pozwala na zapamiętanie **stanu definicji** wierzchołków
  - **Nie zapamiętuje** *danych* wierzchołków – te siedzą w *VBO*
  - **Zapamiętuje** m.in.:
    - wskazania ustawione z użyciem *glVertexAttribPointer()*
    - stan włączenia tablic atrybutów (*glEnableVertexAttribArray()*)
    - id użytego *VBO* i *IBO*
  - Co **zyskujemy**:
    (poza tym, że *VAO* **trzeba** używać we współczesnym *OpenGL*)
    - mniej odwołań do funkcji *OpenGL*,
      mniej przekazywanych danych
    - konieczność zapamiętania tylko jednego id,
      aby przełączyć się w renderowanie nowego obiektu
- Obsługa z użyciem funkcji:
  - **glGenVertexArrays**(*num*, *ids_out*)
  - **glDeleteVertexArrays**(*num*, *ids*)
  - **glBindVertexArray**(*id*)

Uniforms

- Identyfikator zmiennej *uniform*
  - Potrzebny przy ustawianiu wartości
  - Sposoby ustawienia identyfikatora:
    - W kodzie shadera

```
layout(location = 0) uniform mat4 m4ModelViewProjection;
layout(location = 1) uniform mat4 m4ModelView;
```

      - Wymaga **GLSL 4.30**! Lub użycia w shaderze rozszerzenia:

```
#extension GL_ARB_explicit_uniform_location : enable
```

    - Automatycznie
      - Pobranie id na podstawie nazwy: **glGetUniformLocation**(*name*)
  - Dobra praktyka:
    - **Nie odpytywać o id za każdym razem**, gdy go potrzebujemy
      - Zapamiętujemy, albo sami ustalamy
      - Zwykle zmieniamy wartości uniformów wielokrotnie w ciągu klatki – ciągłe odpytywanie jest koszmarnie złym rozwiązaniem!

Uniforms

- Identyfikator zmiennej *uniform*
  - Struktury
    - Każda składowa posiada swój własny identyfikator
    - Można go pobrać za pomocą:

```
glGetUniformLocation(”MyStruct.MyField”);
```

    - Kiedy ustalamy własne identyfikatory struktur, kolejne składowe przyjmują kolejne identyfikatory:

```
struct SS {
     vec3 MyField;
     mat4 MyField2;
};

layout(location = 32) uniform SS MyStruct;
```

```
glGetUniformLocation(”MyStruct.MyField”);  // 32
glGetUniformLocation(”MyStruct.MyField2”); // 33
```

Uniforms

- Identyfikator zmiennej *uniform*
  - Tablice
    - Każdy element tablicy posiada swój własny identyfikator
    - Można go pobrać za pomocą:

```
glGetUniformLocation(”MyArray[3]”);
```

    - Id zerowego elementu jest równy id całej tablicy
    - Kiedy ustalamy własne identyfikatory, kolejne elementy przyjmują kolejne identyfikatory:

```
layout(location = 10) uniform vec4 MyArray[8];
```

```
glGetUniformLocation(”MyArray”);    // 10
glGetUniformLocation(”MyArray[0]”); // 10
glGetUniformLocation(”MyArray[1]”); // 11
glGetUniformLocation(”MyArray[7]”); // 17
```

    - Analogicznie dla tablic struktur
    - Należy pamiętać, żeby **pozostawić odpowiednio wiele wolnych identyfikatorów dla elementów tablic**! Identyfikatory muszą być unikalne

Uniforms

- Ustawianie wartości
  - Wartości zmiennych *uniform* **są częścią stanu programu cieniującego**
  - **glUniform***(...)
    - Ustawienie wartości uniform dla aktywnego (*glUseProgram()*) programu cieniującego
    - Istnieje wiele odmian tej funkcji dla różnego rodzaju danych (macierze, wektory, skalary, ...)
  - **glProgramUniform***(...)
    - Ustawienie wartości uniform dla konkretnego programu (*OpenGL 4.1+* lub *ARB_separate_shader_objects*)
  - **Uniform Buffer Objects** (*UBO*)
    - Świat byłby zbyt przygnębiającym miejscem, gdyby i do tego nie było odpowiedniego rodzaju bufora
    - Wymagają organizacji uniformów w bloki (interface blocks)
    - Możliwość współdzielenia wartości pomiędzy programami (ten sam bufor dla kilku programów)
    - Szybkie uzupełnianie, szybkie przełączanie
    - **glUniformBlockBinding**(), **glBindBufferRange**(), **glBindBufferBase**()

Wyjścia z poszczególnych etapów

- **Wejścia** kolejnych etapów **muszą dokładnie odpowiadać wyjściom** poprzedzających etapów
  - Inaczej wystąpi błąd linkowania
- Vertex shader
  - Wyjściem są **przetworzone dane wierzchołka**
    - Każdemu jednemu wierzchołkowi wejściowemu odpowiada **dokładnie jeden** wierzchołek wyjściowy
    - Domyślnym działaniem jest **transformacja ze współrzędnych modelu do współrzędnych projekcji**
  - Często wygodne jest użycie tzw. ***interface blocks***, czyli zgrupowanie interfejsu we/wy w bloki
    - Nazwa bloku wyjściowego może być różna od nazwy bloku wejściowego – nie stanowi to problemu podczas linkowania

VS

```
out VertexData {
     vec3 v3Normal;
     vec3 v3Eye;
} VertexOut;
```

FS

```
in VertexData {
     vec3 v3Normal;
     vec3 v3Eye;
} VertexIn;
```

Wyjścia z poszczególnych etapów

- Fragment shader
  - Wyjściem są dane **fragmentu**
    - A więc dane, które potencjalnie mogą posłużyć do wpłynięcia na wartość wynikowego piksela bufora klatki
      - Jeśli nie zostaną odrzucone np. przez test głębokości albo w wyniku blendingu
  - Podstawowym wyjściem jest **sekwencja kolorów**
    - Kolejne zmienne wyjściowe odpowiadają różnym buforom będącym częścią stanu *framebuffera*
    - Przypisania konkretnych buforów można dokonać
      - W kodzie shadera:

```
layout(location = 0) out vec4 v4Color;
```

      - Przed linkowaniem:

```
glBindFragDataLocation(programId, bufferId, varName);
```

      - Pozostawić automatycznemu przydzieleniu
    - Bufory definiuje się przy użyciu:

```
glDrawBuffers(num, buffers);
```

Wyjścia z poszczególnych etapów

- Geometry shader
  - Wyjściem jest dowolna liczba wierzchołków
    - Nie ma tablicy na wyjściu, tylko wierzchołki są **emitowane**
      - Każdy z wierzchołków otrzymuje aktualne wartości:
        - **gl_Position**, **gl_PointSize**, **gl_ClipDistance**[], ...
      - Oprócz tego oczywiście można definiować własne wyjścia które będą miały zmienne wartości per-vertex
    - **EmitVertex**()
      - Po wywołaniu tej funkcji wszystkie wyjściowe zmienne mają nieustalone wartości
  - Wyjściem są wierzchołki zorganizowane w prymitywy
    - **EndPrimitive**()
      - Wywołanie nie emituje nowego wierzchołka
  - Layered rendering
    - Na etapie *GS* możemy określić, do której warstwy framebuffera powędruje nasz prymityw
      - Warstwę określa się za pomocą **gl_Layer**
      - Wszystkie wierzchołki prymitywu powinny posiadać ten sam **gl_Layer**
    - Można też określić **gl_ViewportIndex** (od OpenGL 4.1)

Zarządzanie shaderami

- Do zarządzania obsługą shaderów warto stworzyć odpowiedniego, wyspecjalizowanego **managera shaderów**
  - Ładowanie kodu źródłowego
  - Translacja kodu źródłowego
    - ***Übershader***
      - Warunkowe włączanie/wyłączanie bloków przed kompilacją
    - Include'owanie kodu z zewnętrznych plików
    - Interpretacja **stałych**
      - Pozwala np. na wprowadzenie konfiguracji zależnej od możliwości sprzętowych
  - Kompilacja, linkowanie i **obsługa błędów**
  - **Buforowanie** danych
    - Zapewnienie optymalnego transferu klient-serwer

Zarządzanie shaderami

- **Przykład** kodu *GLSL* wzbogaconego o instrukcje warunkowe interpretowane przed kompilacją:

```
#if SHADOWS
     m4ObjectToWorld = mat4(1.0);
     #include Shadows\PCF.body.frag
#else
     float fShadow = 1.0;
#endif
```

  - Stała SHADOWS może być ustawiona zależnie od konfiguracji gry, możliwości sprzętowych itp.
  - Część kodu źródłowego jest wydzielona do zewnętrznego pliku – możliwe jest jej ponowne wykorzystanie w innych shaderach

Zachodniopomorski
Uniwersytet
Technologiczny
w Szczecinie

Bartosz Bazyluk

OpenGL

Współczesne podejście do programowania grafiki

Część II: Programy cieniujące (*shadery*)

Programowanie Gier Komputerowych, Informatyka S1, III Rok

Wydział
Informatyki