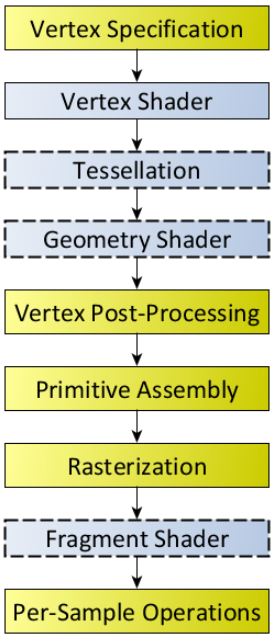
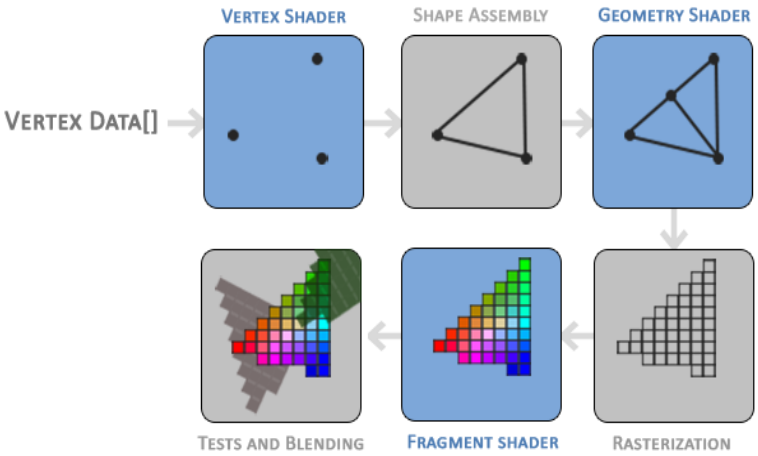
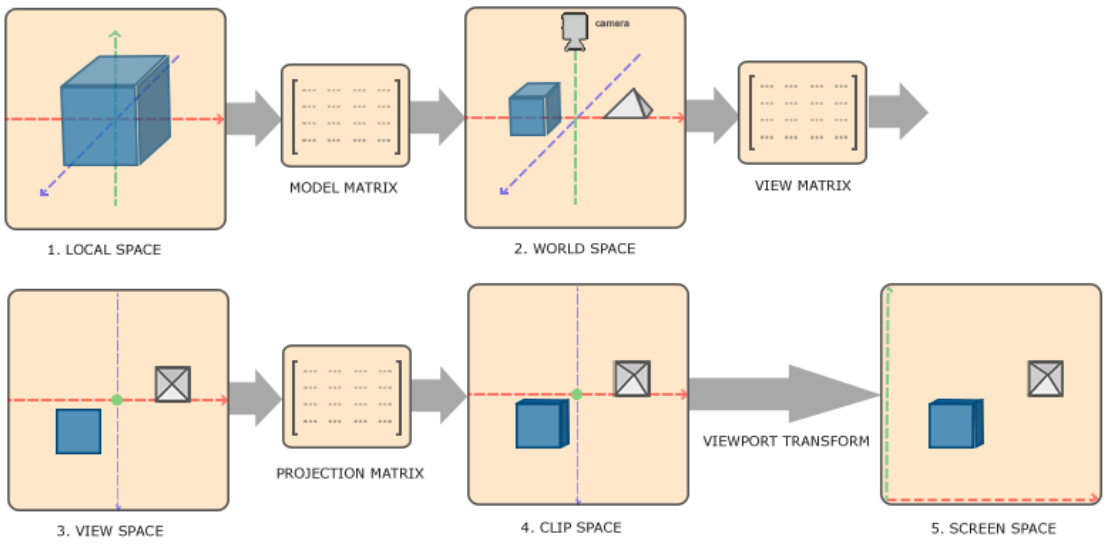
1. Cele kursu:

- Nowoczesne techniki OpenGL.
- Utworzenie okna oraz obsługa operacji wejściowych (klawiatura),
- **Vertex**, **fragment** oraz **Geometry Shader**,





- Rysowanie **obiektów 3D**,
- Używanie biblioteki **GLM** (OpenGL Maths),
- Przenoszenie (**Translate**), Obracanie (**Rotate**) oraz Skalowanie (**Scale**) modeli,
- Używanie interpolacji (**interpolation**) ~ używane do tekstur oraz światła.
- Używanie indeksowego rysowania (**indexed draws**) ~ pozwala używać wierzchołki, które już zostały wspomniane.
- Używanie różnych rodzajów projekcii/ rzutowania (**projection**) ~ ortograficzna dla 2D lub z perspektywą dla 3D.



- Kontrola **kamery** oraz poruszanie nią,
- **Mapowanie tekstur** ~ nakładanie tekstur na obiekty.
- **Phong** Model Oświetlania ~ najbardziej popularny.
- Kierunkowe (**Directional jak słońce**), Punktowe **(Point jak kula)** oraz Miejscowe **(Spot jak pochodnia)** oświetlenie.
- Importowanie wcześniej zrobionych modeli 3D.
- **Mapowanie cieni** (też z różnych źródeł światła).
- Implementacja SkyBox (iluzja dużego świata).

2. Wprowadzenie do GLEW, GLFW oraz SDL:

GLEW:

Co to jest GLEW? **(ROZSZERZENIA ORAZ STERUJE KARTA)**

- OpenGL Extension Wrangler ~ Obsługiwacz rozszerzeń OpenGL.
- Interfejs dla OpenGL wersji ponad 1.1
- Ładuje rozszerzenia OpenGL,
- Niektóre **rozszerzenia** są zależne od platformy, GLEW **może sprawdzić jeżeli one istnieją na tej platformie**.
- **Alternatywy:** GL3W, glLoadGen, glad, glsdk, glbinding, libepoxy, Glee,

Używanie GLEW?

- **#include <GL/glew.h>**
- Po zainicjowaniu kontekstu (GLFW) należy:
  - **glewExperimental = GL_TRUE; (pozwala używać bardziej zawansowane operacje przy pomocy GLEW).**
  - **glewInit(); (inicjalizacja GLEW)**
- Powinno zwrócić **GLEW_OK**. Jeżeli się nie uda to zwróci error.
- Można odczytać error używając **glewGetErrorString (result);**
- Może sprawdzić czy rozszerzenia istnieją (niektóre rozszerzenia są zależne od platformy):
  - **if(!GLEW_EXT_framebuffer_object)**{}
- **wglew.h** tylko dla Windows tylko z funkcjami.

GLFW:

Co to jest GLFW? (**TWORZY KONTEKST ORAZ INPUT) ~ KREATOR KONTEKSTU**

GLFW oraz SDL służą do tworzenia okien oraz kontekstu. **Kontekst** jest **zasadniczą maszyną stanu**, która przechowuje wszystkie dane związane z wyświetlaniem aplikacji. Gdy aplikacja jest zamykana, kontekst OpenGL jest niszczony.

- OpenGL FrameWork ~ budowa/ struktura/ ramka.

- **Obsługuje utworzenie okna (kontekstu) oraz jego kontrole (położenie, rozmiar),**
- Obsługę operacji wejściowych z klawiatury, myszy, joysticka oraz kontrolera.
- Obsługuję obsługę wielu monitorów,
- Używa OpenGL kontekst dla okien, czyli inaczej **tworzy okna** a GLEW je **wypełnia zawartością**.

Alternatywą GLFW, który służy do tworzenia kontekstu oraz okna **jest SDL**:

SDL:

- Simple DirectMedia Layer ~ prosta warstwa bezpośrednich mediów.
- Potrafi zrobić prawie wszystko co GLFW **i więcej !!!!**

Np.: Potrafi obsługiwać:

  - Audio,
  - Wątkowość,
  - System plików,
  - itp.,
- **Inaczej mówiąc SDL umożliwia więcej rzeczy do robienia niż tylko tworzenie kontekstu, okna i obsługę operacji wejściowych (GLFW) ale również potrafi obsługiwać audio, wątkowość oraz system plików.**
- Używane w: FTL, Amnesia, Starbound oraz Dying Light,
- Używane w edytorach poziomów dla Source Engine oraz Cryengine.

Alternatywy dla GLFW oraz SDL:

- **SFML** (Simple and Fast Multimedia Library): Prawie jak SDL ale zawiera więcej możliwości. Niestety kontekst OpenGL jest bardzo słaby, ponieważ bazuje na grafice 2D.
- **GLUT** (OpenGL Utility Toolkit): Należy go unikać!
- **Win32 API:** GLFW, SDL, SFML, GLUT używają tego w tle. Tylko dla osób, które wiedzą co robią. Najniższy poziom do tworzenie kontekstu/ okien. Inne kreatory kontekstu używają tego w tle.

Podsumowanie:

- **GLEW** (OpenGL Extension Wrangler) ~ zapewnia nam interfejs/ ***połączenie z nowoczesną wersją*** OpenGL oraz ***obsługuję rozszerzenia*** zależne platformowo (bezpiecznie).

- **GLFW** pozwala nam ***utworzyć okna*** oraz OpenGL ***kontekst*** również pozwala ***obsługę operacji wejściowych*** od użytkownika (klawiatura, myszka, gamepad).
- **SDL** umożliwia wiele więcej rzeczy niż GFLW (np.: obsługę audio).

Czyli:
GLFW służy do **tworzenia okna oraz kontekstu** (maszyny stanu, która przechowuje wszystkie danego związane z wyświetlaniem aplikacji).

Natomiast **GLEW** służy do **korzystania z nowoczesnej wersji** OpenGL oraz do **ładowania i korzystania z dostępnych rozszerzeń**. Dzięki niemu możemy w sposób nowoczesny korzystać z maszyn stanu. Umożliwia korzystanie z OpenGL.

GLEW umożliwia nam rysowanie kontekstu wewnątrz okna ale za to GLFW umożliwia załączenie tego kontekstu.

Etapy załączania GLFW:

1. Załączamy plik nagłówkowy.
2. Inicjalizujemy GLFW.
3. Ustawiamy parametry okna.

3. Shadery oraz Rendering Pipeline (strumień renderowania).

Rendering pipeline ~ zestaw operacji, które są wykonywane za każdym razem przez kartę graficzną.

1. **Co to jest strumień renderowania?**

- Strumień renderowania (rendering pipeline) jest to zestaw etapów, które muszą się wykonać w celu wyrenderowania obrazku na ekranie.
- **Cztery etapy** są programowalne przez „Shadery”:
  - **Vertex Shader** (Najważniejszy),
  - **Fragment Shader** (Najważniejszy),
  - **Geometry Shader,**
  - **Testalation shader,**
- **Shadery** są to **kawałki kodu napisane** w ***GLSL*** (OpenGL Shading Language ~ Jezyki shaderów) albo *HLSL* (High-Level Shading Language) jeżeli używamy Direct3D.
- **GLSL** jest napisany w języku C.

2. Etapy renderowania (The Rendering Pipeline Stages):

1. **Vertex Specifacation (Specyfikacja wierzchołka)** ~ Specyfikacja wierzchołków.
    - Wierzchołek (vertex) jest to punkt w przestrzeni, zazwyczaj zdefiniowany przez koordynaty x, y oraz z.
    - Prymityw jest prosty kształt używający jeden lub więcej wierzchołków.
    - Zazwyczaj używamy trójkątów, ale możemy również używać punktów, linii oraz czworokątów.
    - **Specyfikacja wierzchołka:** ***Ustawianie danych wierzchołków dla prymitywa, który chcemy wyrenderować (narysować na ekranie).***
    - Sporządzone w aplikacji przez siebie.
    - Używają **VAOs** (Vertex Array Objects) oraz **VBOs** (Vertex Buffer Objects).
    - **VAO** definują jakie dane wierzchołek ma np.: pozycja, kolor, tekstura, normalne, itp.:). Po prostu określają ich cechy.
    - **VBO** określają już dane. Po prostu określają je.
    - Wskaźniki atrybutów definiują określają gdzie oraz jak shadery mogą otrzymywać dane o wierzchołkach
    - **Są jeszcze IBO** (Index Buffer Objects).

Tworzenie VAO/VBO:

1. Utwórz VAO identyfikator (id vertex array object).

2. Powiąż (bind) VAO z tym ID (bind).

3. Utwórz VBO identyfikator (id vertex buffer object).

4. Powiąż (bind) VBO z tym ID (teraz pracujemy na wybranym VBO z załączonym do niego VAO).

OpenGL się domyśla, że wcześniej zbindowane VAO jest tym na którym, będziemy pracowali kiedy będzie używali VBO.

5. Dołącz dane wierzchołków do tego VBO.

6. Zdefiniuj formatowanie wskaźnika atrybutu.

7. Aktywuj wskaźnik atrybutu.

8. Odwiąż (unbind) VAO oraz VBO, gotowe do przywiązania nowego obiektu.

Inicjalizacja Rysowania:

1. **Aktywacja programu** z **shaderem** (Shader Program) tego, którego chcemy użyć.

   Shader program, może zawierać kod dotyczący vertex shader, fragment shader oraz geometry shader. Dlatego jest to nazywane programem.

2. Powiązanie/ **bind VAO** obiektu, który chcemy narysować.
3. Wywołanie funkcji ***glDrawArrays*** , która zainicjalizuje resztę strumienia renderowania.

   Proste oraz wygodne!

2. Vertex Shader (programowalny).

Cechy:

- Obsługuję wierzchołki indywidualnie.
- Nie jest opcjonalny.
- Musi zawierać coś w ***gl_Position***, ponieważ będzie to później używane przez późniejsze etapy strumienia renderowania.
- Może określić dodatkowe wartości wyjściowe, które mogą zostać podniesione oraz użyte przez shadery zdefiniowane przez użytkownika, które później występują w strumieniu renderowania.
- Dane wejściowe składają się danych wierzchołków w sobie (pozycja, texture cordinates).

Przykład:

```
#version 330

layout (location = 0) in vec3 pos;

void main()
{
        gl_Position = vec4(pos, 1.0);
}
```

layout ~ definiuje pozycje w shaderze (każdy input ma swoje id).

in ~ znaczy, że jest to input.

vec3 ~ znaczy, że jest to wektor, która składa się z trzech wartości (x, y, z).
pos ~ nazwa zmiennej.

gl_Position (finalna pozycja wierzchołka).

3. Tessellation (programowalny).

- Pozwala podzielić dane na kilka mniejszych prymitywów (grupa wierzchołków ~ prymityw).
- Relatywnie nowy typ shadera, pojawił się w OpenGL 4.0.
- Może być użyty do wyższego poziomu szczegółowości dynamicznie.

4. Geometry Shader (programowalny).

- Vertex shader obsługiwał wierzchołki, Geometry shader **obsługuję prymitywy** (***grupy wierzchołków np. trójkąt (3 wierzchołki)***),
- Bierze prymitywy potem emituje (outputs) jej wierzchołki do utworzenia prymitywu albo nowych prymitywów.
- Może **przerabiać podane** dane do przerobionych danych prymitywów albo nawet tworzyć nowe,
- Może nawet zmienić prymitywne typy (punkty, linie, trójkąty,…).

Na przykład możemy dać grupę wierzchołków taką jak trójkąt a następnie geometry shader może nam to przerobić i utworzyć nowy prymityw lub przesunąć na przykład o 3 wartości w bok pozycje. Rożne takie bajery. Zatem vertex shader obsługuję każdy wierzchołek indywidualnie zaś geometry shader obsługuję grupę wierzchołków razem czyli na przykład taką grupę, która reprezentuje trójkąt. Proste 😊.

5. Vertex Post Processing.

- Przekształca informację zwrotną (jeżeli jest to włączone):
  - Wynik vertex oraz geometry etapów jest zapisany do buforów dla późniejszego użycia.
- Obkrajanie/ Wykrajanie (Clipping):

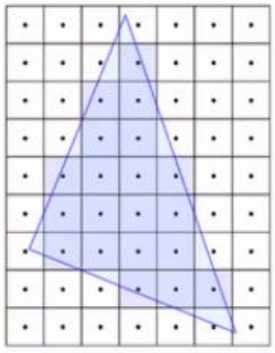
- Prymitywy, które nie są widziane są usuwane (nie chcemy rysować rzeczy, których nie widać).
- **Pozycje przekonwertowane z *przestrzeni obkrajania* („clip space”) do *przestrzeni okna* („window space”).**

6. Primitive Assembly (łączenie prymitywów (grup wierzchołków)):

- Wierzchołki **są konwertowane do serii prymitywów**.
- Wiec jeżeli mamy trójkąty**... 6 wierzchołków to z nich zostanie utworzone 2 trójkąty** (3 wierzchołki każdy).
- Face culling ~ usuwanie twarzy.
- Face culling jest to proces usuwania prymitywów, które nie mogą być widziane albo są patrzone z bardzo dalekiej odległości. **Nie chcemy rysować czegoś czego nie widzimy**.

7. Rasteryzacja.

- Zamiana prymitywów do „fragmentów”.
- Fragmenty są kawałki danych dla każdego pixela, uzyskane z procesu rasteryzacji.
- Dane fragmentu będą interpolowane na podstawie ich relatywnej pozycji dla każdego wierzchołka.



8. Fragment Shader (programowalny).

- **Obsługuje dane dla każdego fragmentu oraz wykonuje operacje na nim.**
- Jest opcjonalny ale rzadko kto go nie używa. Wyjątkami są przypadki gdzie głębia albo matryca/ szablon dane są wymagane.
- Najważniejszą **wartością wyjściową jaką jest kolor piksela**, który fragment obejmuje.
- Najprostszy OpenGL program obejmuje zazwyczaj Vertex Shader oraz Fragment Shader.
- Będzie obsługiwał ***oświetlenie oraz teksturowanie, cieniowanie.***

Przykład:

```
#version 330

out vec4 colour;

void main()
{
        colour = vec4(1.0, 0.0, 0.0, 1.0);
}
```

9. Per-Sample Operations.

- Seria testów sprawdzających czy pixel/ fragment powinien być namalowany/ narysowany.
- Najważniejszym testem: Test głębokości (**Depth Test**). Determinuje jeżeli coś jest naprzeciwko punktu, który ma być narysowany.
- **Mieszanie kolorów (Colour Blending)**: Używa zdefiniowanych operacji, kolory fragmentów są wymieszane razem z nachodzącymi fragmentami. Zazwyczaj używane do obsługi przezroczystych obiektów.
- Dane fragmentów **są wpisane do obecnie zajmowanego bufora ramki (Framebuffer) (zazwyczaj podstawowego bufora).**
- Ostatecznie, w kodzie aplikacji użytkownik zazwyczaj definiuj zamianę buforów tutaj, kładąc nowo zaktualizowany bufor ramki do przodu.

Framebuffor to jest na którym pracujemy, rysujemy.

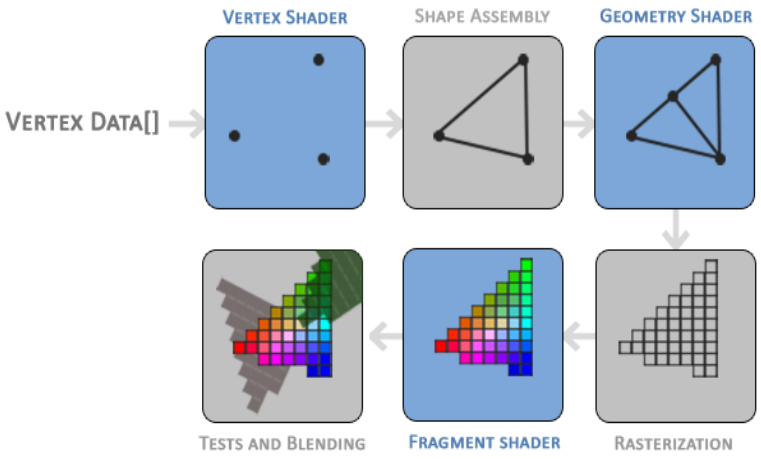
Oryginalny widzi użytkownik.

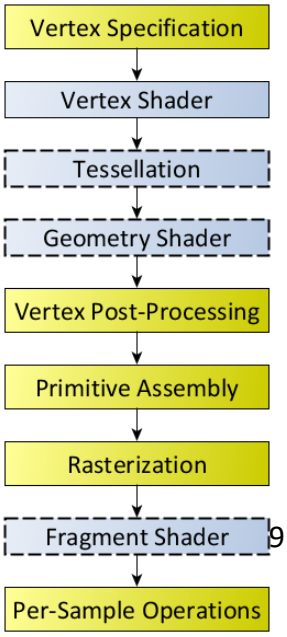
Na koniec oryginalny jest zamieniany z framebuffer, stary framebuffer staje się oryginalnym a stary oryginalny staje się nowym framebufferem

Na zakończenie zamieniamy oryginalny na framebuffer.

Możemy mieć tyle frame bufforow ile chcemy na przykład dla rożnych scen.

- **Strumień renderowania jest zakończony 😊.**





Jak używać ich oraz jak się tworzy Shadery?

O pochodzeniu Shaderów:

- Programy Shaderowe (Shaders Programs) ***są grupą shaderów*** (Vertex, Tessellation, Geometry, Fragment…) powiązane ze sobą.
- Są one tworzone w OpenGL przez serie funkcji.

Tworzenie programu z shaderami:

1. Utworzenie pustego programu.
2. Utworzenie pustych shaderów np.: Vertex Shader, Fragment Shader.
3. Załączenie shaderu kodu źródłowego do shaderów.
4. Kompilacja shaderów.
5. Załączenie shaderów do programu.
6. Załączenie/ Powiązanie programu (tworzy plik wykonawczy z shaderów oraz łączy je w całość).
7. Walidacja programu (opcjonalne ale bardzo sugerowane, ponieważ debugowanie shaderów jest straszne).

Używanie programu z shaderami:

- Kiedy utworzymy shader to dostajemy identyfikator (jak w przypadku VAO oraz VBO).
- Po prostu wywołujemy funkcje ***glUseProgram(shaderID)***,
- Wszystkie wywołania rysowania od teraz będą używały tego shadera, ***glUseProgram*** *jest używane na nowym identyfikatorze shadera albo 0 (brak shadera).*

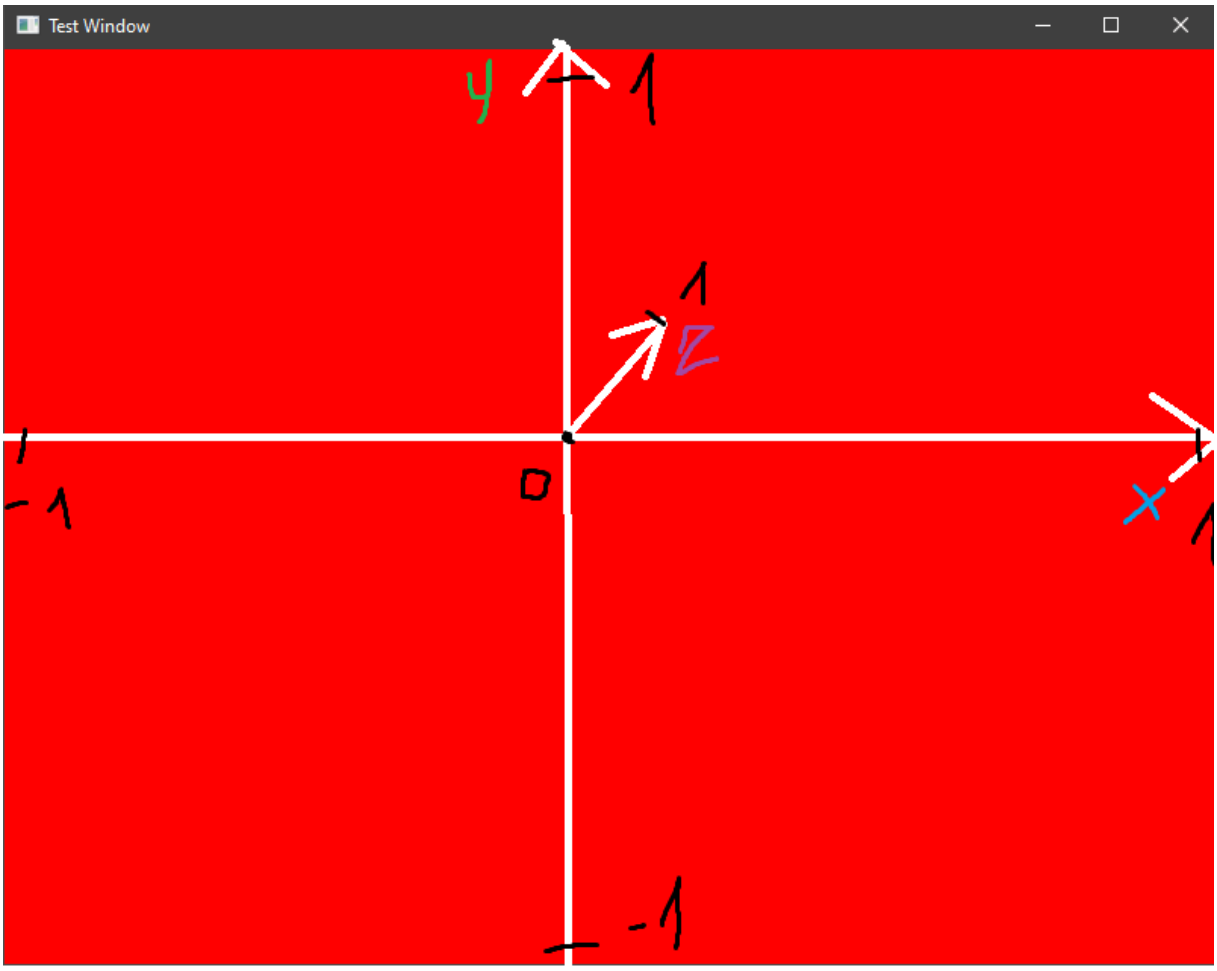
Podsumowanie:

- Strumień renderowania (Rendering Pipeline) składa się z kilku etapów.
- **Cztery etapy** są **programowalne** poprzez shadery (Vertex, Tessellation, Geometry, Fragment).
- **Vertex Shader jest** obligatoryjny,
- **Wierzchołki** (Vertices): Punkty zdefiniowane przez użytkownika, które znajdują się w przestrzeni.
- **Prymitywy: Grupy wierzchołków**, które tworzą prosty kształt (zazwyczaj jest to trójkąt).

- **Fragmenty:** Dane każdego piksela stworzone przez prymitywy.
- **Vertex Array Object** (VAO): Definiuje jakie dane zawiera wierzchołek.
- **Vertex Buffer Object** (VBO): Wierzchołek samy w sobie.
- **Programy z shaderami** są tworzone z przynajmniej Vertex Shader (Shaderem Wierzchołka) a potem aktywowane przed użyciem.
- Wierzchołki są obsługiwane przez Vertex Shader, Prymitywy są obsługiwane przez Geometry Shader a fragmenty są obsługiwane przez Fragment Shader.

Dane przesyłamy do Vertex Shader natomiast Fragment Shader później podnosi wynik po ostatnich operacjach. Itp…

Początkowe ustawienie okna to:



1. Załączone shadery (Shaders Added).
2. Utworzenie trojkata (Triangle created).
3. Wywolanie funkcji rysowania.

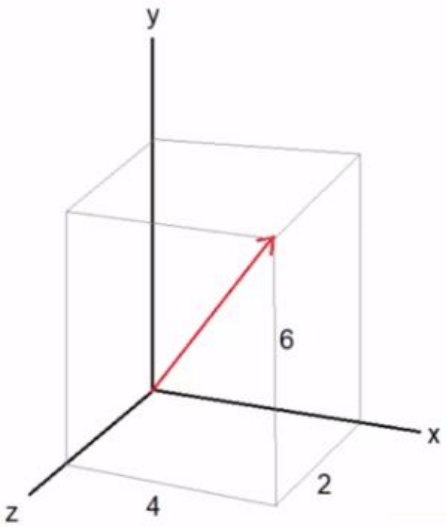
4. Wektory, macierze oraz jednolite zmienne (uniform variables):

Omówienie wektorów:

Co to znaczy:

- Wielkość, która ma długość (magnitude) oraz kierunek.
- Inaczej mówiąc określa jak daleko jest dany obiekt oraz w jakim kierunku jest on skierowany.
- Może być użyty do wielu rzeczy, normalnie do reprezentacji kierunku albo pozycji (jak daleko jest oraz w jakim kierunku jest on skierowany, relatywnie do określonego punktu).
- **x = 4, y = 6, z = 2**
- **v = [4, 6, 2]**

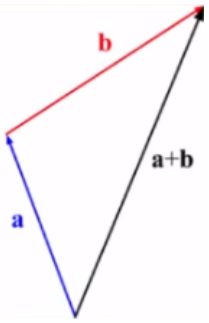
$$v = \begin{bmatrix} 4 \\ 6 \\ 2 \end{bmatrix}$$



Dozwolone operacje:

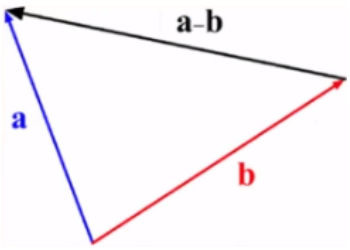
- ***Dodawanie*:**

[1, 2, 3] **+** [2, 4, 6] = [1+2, 2+4, 6+3] = [3, 6, 9]



- ***Odejmowanie*:**

[1, 2, 3] **–** [2, 4, 6] = [1-2, 2-4, 3-6] = [-1, -2, -3]



- ***Mnożenie przez skalar (poj. wartość)***

[1, 2, 3] ***** 2 = [1*2, 2*2, 3*2] = [2, 4, 6]

- Mnożenie przez wektor?
- Trudne do zdefiniowania i nie używane.
- Zamiast tego używamy **Dot Product (iloczyn skalarny).**

Magnitude/ Długość:

- Wektory z prostego kąta trójkątów.
- Więc możemy obliczyć długość z wariacji twierdzenia Pitagoras'a.
- In 3D, to jest po prostu: **$|v| = sqrt(v_x^2 + v_y^2 + v_z^2)$.**
- v = [1, 2, 2]
  |v| = sqrt(1+4+4) = sqrt(9) = 3.

Dot product:

- Również nazywane Scalar Product, ponieważ zwraca pojedynczą/ skalarną wartość w przeciwieństwie do wektora.
- **Może być użyte na dwa sposoby:**
  - [a, b, c] * [d, e, f] = a*d + b*e + c*f

    [1, 2, 3] * [4, 5, 6] =1*4 + 2*5 + 3*6 = 32

$$v_1 \bullet v_2 = |v_1| * |v_2| * \cos(\varphi)$$

$|v_1|$ jest to **długość**/ magnitude wektora v_1.

φ jest to **kąt między wektorami** v_1 oraz v_2.

Pozwala robić refleksje oraz detekcje kolizji (możemy użyć jednego wektora aby rzutować na drugi).

- To pozwala na kilka ciekawych scenariuszy…
- **v1 • v2 = |v1| * |v2| * cos(φ)**
- Jeżeli wiemy v1 • v2 z alternatywnej metody…
- I obliczymy dwie długości wektorów…
- **(v1 • v2)/ (|v1| * |v2|) = cos(φ)**

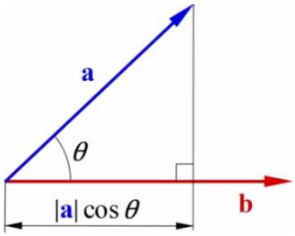
$$\cos^{-1}((v1 \bullet v2)/(|v1| * |v2|)) = \varphi$$

- Więcej o tym kiedy przejdziemy do oświetlania.
- Jest to **skalarna projekcja (rzutowanie).**

- Dot product z założeniem, że ‘b’ jest wektorem jednostkowym,
- Wektor jednostkowy jest to wektor o długości 1,
- Jeżeli a oraz b są pod właściwymi kątami to długość rzutowania będzie 0.
- Ma to sens, ponieważ:

|a|*cos(90) = |a| x 0 = 0

- Ważne w kontekście światła.



Wektor jednostkowy:

- Czasami chcemy wiedzieć tylko kierunek oraz jak iść w tym kierunku.
- Wektor jednostkowy jest to **wektor o długości 1**.

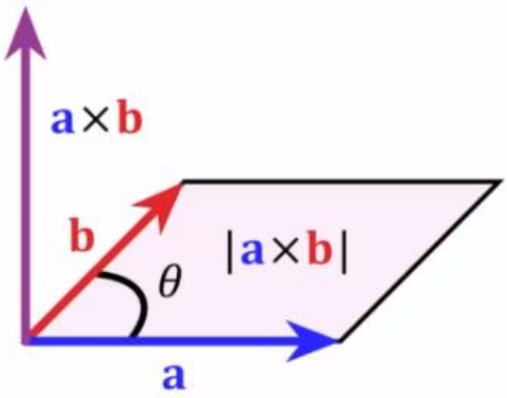
$$u = \frac{V}{|V|}$$

- $v = [1, 2, 2]$
- $|v| = sqrt(1^2 + 2^2 + 2^3) = sqrt(1 + 4 + 4) = sqrt(9) = 3$
- $u = \frac{[1,2,2]}{3} = [\frac{1}{3}, \frac{2}{3}, \frac{2}{3}]$
- u ma ten sam kierunek co v ale wartość jego długości to 1.

Cross product (produkt krzyżowy):

- Wprawdzie tylko działa w 3D.
- Tworzy wektor, który jest prostopadły do dwóch pozostałych.
- Kolejność ma znaczenie.

$$\begin{pmatrix} A_x \\ A_y \\ A_z \end{pmatrix} \times \begin{pmatrix} B_x \\ B_y \\ B_z \end{pmatrix} = \begin{pmatrix} A_y \cdot B_z - A_z \cdot B_y \\ A_z \cdot B_x - A_x \cdot B_z \\ A_x \cdot B_y - A_y \cdot B_x \end{pmatrix}$$



Omówienie Macierzy:

Co to jest:

- Grupa wartości umieszczona w siatce o rozmiarach i x j.
- Przykładem jest 2x3 macierz.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

i ~ wiersze,

j ~ kolumny,

- Może być użyte dla wielu rzeczy poprzez grafikę, tworzenie gier oraz pola naukowe.
- Będziemy ich używać to ***obsługi:***
  - **modelów transformacji** (translacji, rotacji oraz skalowania) ~ poruszanie obiektu, obracanie oraz skalowanie obiektu.
  - **projekcji/ rzutowania** (projections) ~ jak widzimy rzeczy np.: poprzez kamerę (np.: ortogonalna projekcja).
  - **widoków** (views) jest to pozycja oraz orientacja kamery.

Dodawanie oraz odejmowanie macierzy:

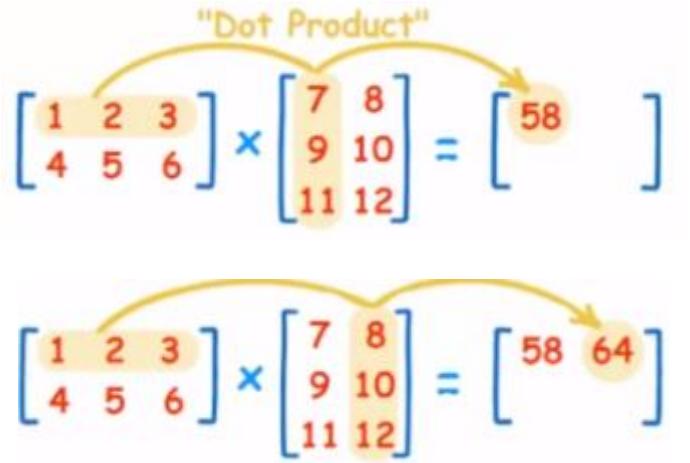
- Skalar: Po prostu dodaje/ odejmuje wartość z każdego elementu, podobnie jak w przypadku wektorów.
- Macierz: Dodaje wartości w przeliczeniu na element. Każdy musi pasować swoją pozycją do innej macierzy.
- To znaczy, że **rozmiary macierzy**, które chcemy dodać lub odjąć muszą być **takie same**.

$$\begin{bmatrix} 1 & 3 \\ 1 & 0 \\ 1 & 2 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 7 & 5 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1+0 & 3+0 \\ 1+7 & 0+5 \\ 1+2 & 2+1 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 8 & 5 \\ 3 & 3 \end{bmatrix}$$

Mnożenie macierzy:

- **Po przez skalar**: Po prostu mnożymy wartość z każdym elementem, tak samo jak w przypadku wektorów.
- **Po przez macierz**:
  - **Kolejność ma znaczenie**.

- **Ilość kolumn** macierzy po lewej stronie musi zawsze się **równać ilości wierszy** macierzy po prawej stronie.



- Biblioteka ***GLM*** będzie to obsługiwała z nas.

Związek Macierzy z Wektorami:

- Jak Macierze pracują z wektorami?
- **Wektory są to macierze**, które mają tylko **jedną kolumnę**.
- Mnożenie wektora przez macierz da nam **zmodyfikowaną postać tego wektora**.
- **WEKTOR ZAWSZE BĘDZIE PO PRAWEJ STRONIE.**

$$v = \begin{bmatrix} 4 \\ 6 \\ 2 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 2 & 0 \\ 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 0 \\ 6 & 0 & 0 & 7 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 5 \\ 1 \\ 8 \end{bmatrix} = \begin{bmatrix} 4 \\ 47 \\ 5 \\ 68 \end{bmatrix}$$

Macierze transformacji:

- Macierze mogą być użyte z wektorami, żeby zaaplikować transformację do nich (translacja, rotacja oraz skalowanie).
- Najbardziej podstawową jest **macierz jednostkowa**.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- Po prostu zwraca dany wektor.
- Zachowuje się jak **punkt startowy** do **aplikacji innych transformacji.**

Macierz Translacji:

- Translacja przemieszcza wektor.
- Używają jej żeby zmienić pozycję czegoś.

$$\begin{bmatrix} 1 & 0 & 0 & X \\ 0 & 1 & 0 & Y \\ 0 & 0 & 1 & Z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x+X \\ y+Y \\ z+Z \\ 1 \end{bmatrix}$$

Macierz skalowania:

- Skalowanie zmienia/ rozszerza wektor.
- Może być użyty jeżeli chcemy poszerzyć dystans o jakiś czynnik, albo częściej żeby zrobić obiekt większy.

$$\begin{bmatrix} SX & 0 & 0 & 0 \\ 0 & SY & 0 & 0 \\ 0 & 0 & SZ & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} SX \cdot x \\ SY \cdot y \\ SZ \cdot z \\ 1 \end{bmatrix}$$

Macierz rotacji:

- Macierz rotacji obraca wektor.
- Powinna być nauczana jako rotacja wokół jego pochodzenia (Origin).
- Zatem wybranie punktu rotacji, translacja wektora, wiec punkt, do którego będziemy się obracać jest pochodzeniem (Origin).
- Istnieją trzy różne macierze do obsługi rotacji.

Rodzaje macierzy rotacji:

- Rotacja wokół osi **X:**

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ \cos\theta \cdot y - \sin\theta \cdot z \\ \sin\theta \cdot y + \cos\theta \cdot z \\ 1 \end{bmatrix}$$

- Rotacja wokół osi **Y:**

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta \cdot x + \sin\theta \cdot z \\ y \\ -\sin\theta \cdot x + \cos\theta \cdot z \\ 1 \end{bmatrix}$$

- Rotacja wokół osi Z:

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta \cdot x - \sin\theta \cdot y \\ \sin\theta \cdot x + \cos\theta \cdot y \\ z \\ 1 \end{bmatrix}$$

UWAGA: Kąt musi być w *radianach* nie w stopniach!!!!

Nie trzeba ich pamiętać bo GLM (OpenGL Mathematics) zrobi większość operacji na macierzach zamiast nas.

Łączenie macierzy transformacji:

1. Żeby połączyć macierze transformacji należy je połączyć.

np.: Najpierw jest macierz translacji a potem skalowania.

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 2 & 0 & 2 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2. Potem zaaplikowanie tej macierzy do wektora (mnożenie).

$$\begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 2 & 0 & 2 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 2x+1 \\ 2y+2 \\ 2z+3 \\ 1 \end{bmatrix}$$

- Kolejność ma znaczenie.

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 2 & 0 & 2 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- **Transformacje zachodzą w *odwrotnej kolejności***: Skalowanie jest aplikowane pierwsze a potem translacja (chociaż w zapisie macierzy jest najpierw macierz translacji a potem skalowania).
- Jeżeli zamienimy je miejscami, że macierz skalowania będzie pierwsza a macierz translacji druga to będzie najpierw zaaplikowana operacja translacji a potem skalowania.
- Więc skalowanie również będzie skalowało transformację.

GLM:

- GLM jest darmową biblioteką do obsługi powszechnie używanych operacji w OpenGL.
- Najważniejsze: Wektory oraz Macierze.
- Używa vec4 (vector z 4 wartosciami) oraz mat4 (4x4 macierz) typy.
- Prosty kod:

glm::mat 4 trans;

trans = glm::translate(trans, glm::vec3(1.0f, 2.0f, 3.0f));

Uniform Zmienne:

- Rodzaj zmiennej w shader'ze.
- Uniform są wartościami globalnymi dla shadera, który nie jest powiązany z danym wierzchołkiem.

```
#version 330

in vec3 pos;

uniform mat4 model;

void main()
{
    gl_Position = model * vec4(pos, 1.0);
}
```

- Każdy uniform ma swój identyfikator lokalizacji w shaderze.
- Należy znaleźć lokalizacje gdzie możemy powiązać wartość do niego.

int location = glGetUniformLocation(shaderID, „uniformVarName”);

- Teraz możemy powiązać wartość z tą lokalizacją.

glUniform1f(location, 3.5f);

- Upewnij się, że ustawiłeś odpowiedni shader, które będzie używany.
- Różne typy danych:
    - glUniform1f ~ pojedynczy typ float.
    - glUniform1i ~ pojedynczy typ całkowity.
    - glUniform4f ~ vec4 z wartości float.
    - glUniform4fv ~ vec4 z wartości float, wartości określone przez wskaźnik.
    - glUniformMatrix4fv ~ mat4 utworzony z wartości float, wartości określone przez wskaźnik.

Podsumowanie:

- Wektory są kierunkami oraz pozycjami w przestrzeni.
- Macierze są dwu wymiarowymi tablicami z danymi, które są używane do obliczania transformacji oraz rożnych rodzaju funkcji (projection matrixes oraz views matrixes).
- Wektor jest typem macierzy oraz może mieć zastosowane te operacje na nim.
- Kolejność wykonywania transformacji ma znaczenie!
- Ostatnia wykonana operacja na macierzy jest pierwsza.
- GLM jest używany do obsługi macierzowych obliczeń.
- Uniform zmienne przepuszczają dane globalne do shaderów.
- Potrzebujemy znać lokalizacji uniformu a potem powiązać daną z nim..