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.

1. Etapy renderowania (The Rendering Pipeline Stages):
2. **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.

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

Proste oraz wygodne!

1. 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:**

****

**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).**

1. 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.

1. 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 😊.

1. 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”).**

1. 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**.

1. 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.

1. 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:

****

1. 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:

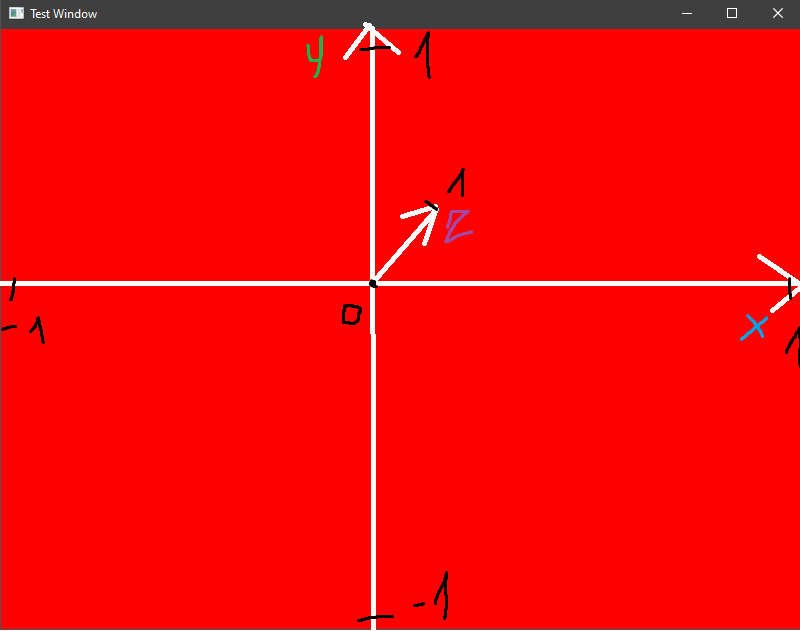
* Kiedy utworzymy shader to dostajemy identyfikator (jak w przypadku VAO oraz VBO).
* Po prostu wywołujemy funkcje ***glUseProgram(shaderID)****,*
* Wszystkie wywołania rysowania od teraz będą używały tego shadera, ***glUseProgram*** *jest używane na nowym identyfikatorze shadera albo 0 (brak shadera).*

**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).

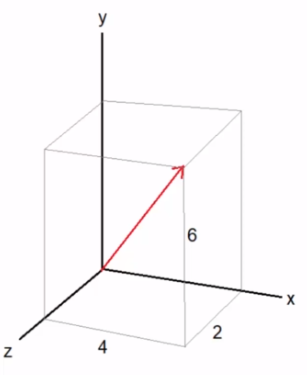
2. Utworzenie trojkata (Triangle created).

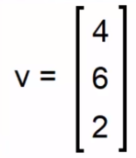
3. Wywolanie funkcji rysowania.

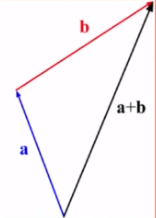
4. Wektory, macierze oraz jednolite zmienne (uniform variables):

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]**



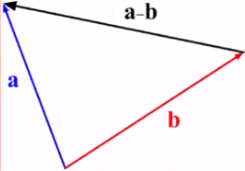
****

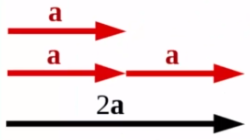
Dozwolone operacje:

* ***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(vx­­2+ vy­2+ vz­­2).**
* 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

**v1 • v2 = |v1| \* |v­2|\* cos(φ)**

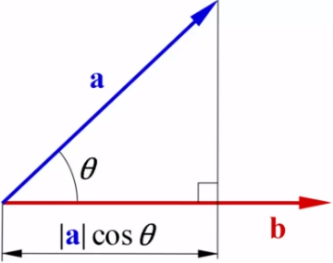
**|v1|** jest to **długość**/ magnitude wektora v1­.

**φ** jest to **kąt między wektorami** v1 oraz v2.

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•v2)/(|v1|\*|v2)) = φ

* 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**

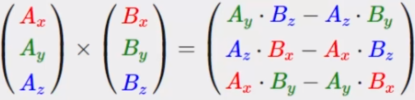
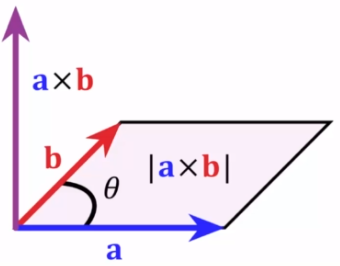
* Ważne w kontekście światła.

**Wektor jednostkowy:**

* Czasami chcemy wiedzieć tylko kierunek oraz jak iść w tym kierunku.
* Wektor jednostkowy jest to **wektor o długości 1**.
* u ma ten sam kierunek co v ale wartość jego długości to 1.

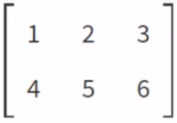
**Cross product** (produkt krzyżowy):

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



**Omówienie Macierzy:**

Co to jest:

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

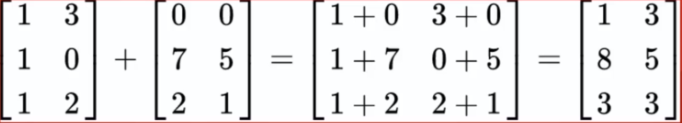
**i ~ wiersze,**

**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.

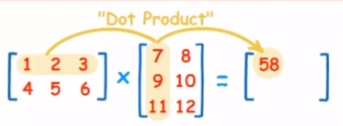
Dodawanie oraz odejmowanie macierzy:

* 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**.



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.

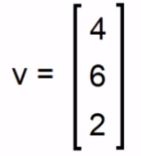
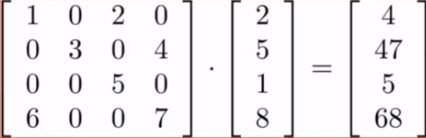




* Biblioteka ***GLM*** będzie to obsługiwała z nas.

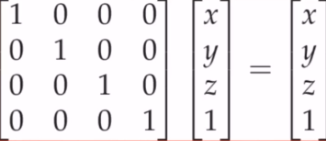
Związek Macierzy z Wektorami:

* 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.**



**Macierze transformacji:**

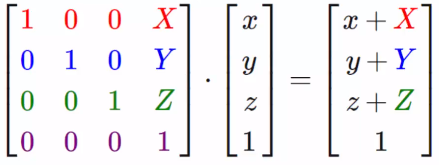
* Macierze mogą być użyte z wektorami, żeby zaaplikować transformację do nich (translacja, rotacja oraz skalowanie).
* Najbardziej podstawową jest **macierz jednostkowa**.



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

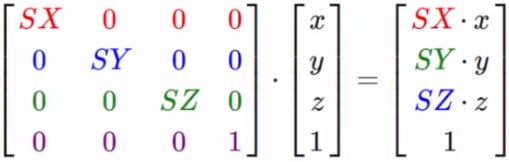
**Macierz Translacji:**

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



**Macierz skalowania:**

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

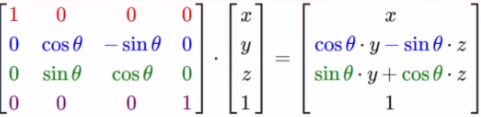


**Macierz rotacji:**

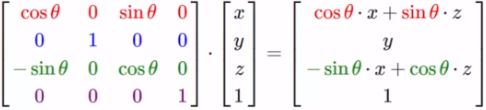
* Macierz rotacji obraca wektor.
* Powinna być nauczana jako rotacja wokół jego pochodzenia (Origin).
* Zatem wybranie punktu rotacji, translacja wektora, wiec punkt, do którego będziemy się obracać jest pochodzeniem (Origin).
* Istnieją trzy różne macierze do obsługi rotacji.

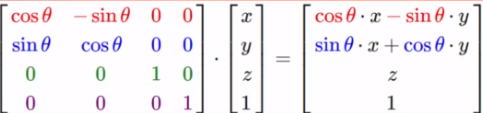
**Rodzaje macierzy rotacji:**

* Rotacja wokół osi **X:**



* Rotacja wokół osi **Y:**



* Rotacja wokół osi Z:

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

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

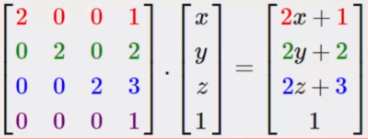
**Łączenie macierzy transformacji:**

1. Żeby połączyć macierze transformacji należy je połączyć.

np.: Najpierw jest macierz translacji a potem skalowania.



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



* Kolejność ma znaczenie.
* **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.

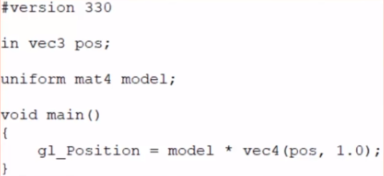
**GLM:**

* 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.
* 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:**

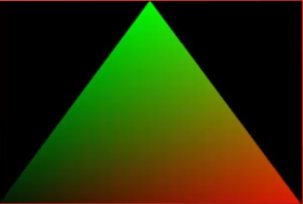
* Wektory są kierunkami oraz pozycjami w przestrzeni.
* Macierze są dwu wymiarowymi tablicami z danymi, które są używane do obliczania transformacji oraz rożnych rodzaju funkcji (projection matrixes oraz views matrixes).
* Wektor jest typem macierzy oraz może mieć zastosowane te operacje na nim.
* Kolejność 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..

**Trzy główne macierze:**

* Widok,
* Modelu (translacji, rotacji, skalowania)
* Projekcji (rzutowania),

System koordynatów:

* Model ~ każdy koordynat idzie od 0, 0 (zbudowany wokół pochodzenia). Jeżeli obiekt jest przesunięty to musimy użyć macierzy modelu żeby przejść do świata.

**Macierz projekcji ~** koordynaty świata (world cordinates) do ekranu (screen cordinates).

**Interpolacja:**

* Każde atrybuty wierzchołka przesłanego są interpolowane używając innych wartości na prymitywie.
* Inaczej mówiąc: ***Średnia ważona trzech wierzchołków trójkąta*** jest przesłana.
* Fragment Shader podnosi tą interpolowaną wartość a następnie używa jej.
* Wartość jest efektywnie oszacowanie jaka wartość powinna być na tej pozycji, jaką my wcześniej określiliśmy.
* Klasyczny przykład: Używając kordynatów pozycji jako wartości RGB.
* Góra trójkąta jest zielona, ponieważ (x, y, z) y jest wysokościa.
* Zamiana RGB, potem G jest wysokością.
* W połowie drogi czerwone oraz zielone, kolory są mieszane, ale nie określiśmy tych pozycji tego wierzchołka.
* Wartość została zinterpolowana.
* Interpolacja jest używana do szybkiego oraz dokładnego oszacowania wartości bez ich określania.
* Może zostać użyte do interpolowania kordynatów tekstury kiedy mapujemy tekstury.
* Może zostać do interpolacji normalnych wektorów kiedy obsługujemy oświetlenie.
* Specjalnie używane w **Phong Shading** do stworzenie iluzji okrągłej smukłej powierzchni.

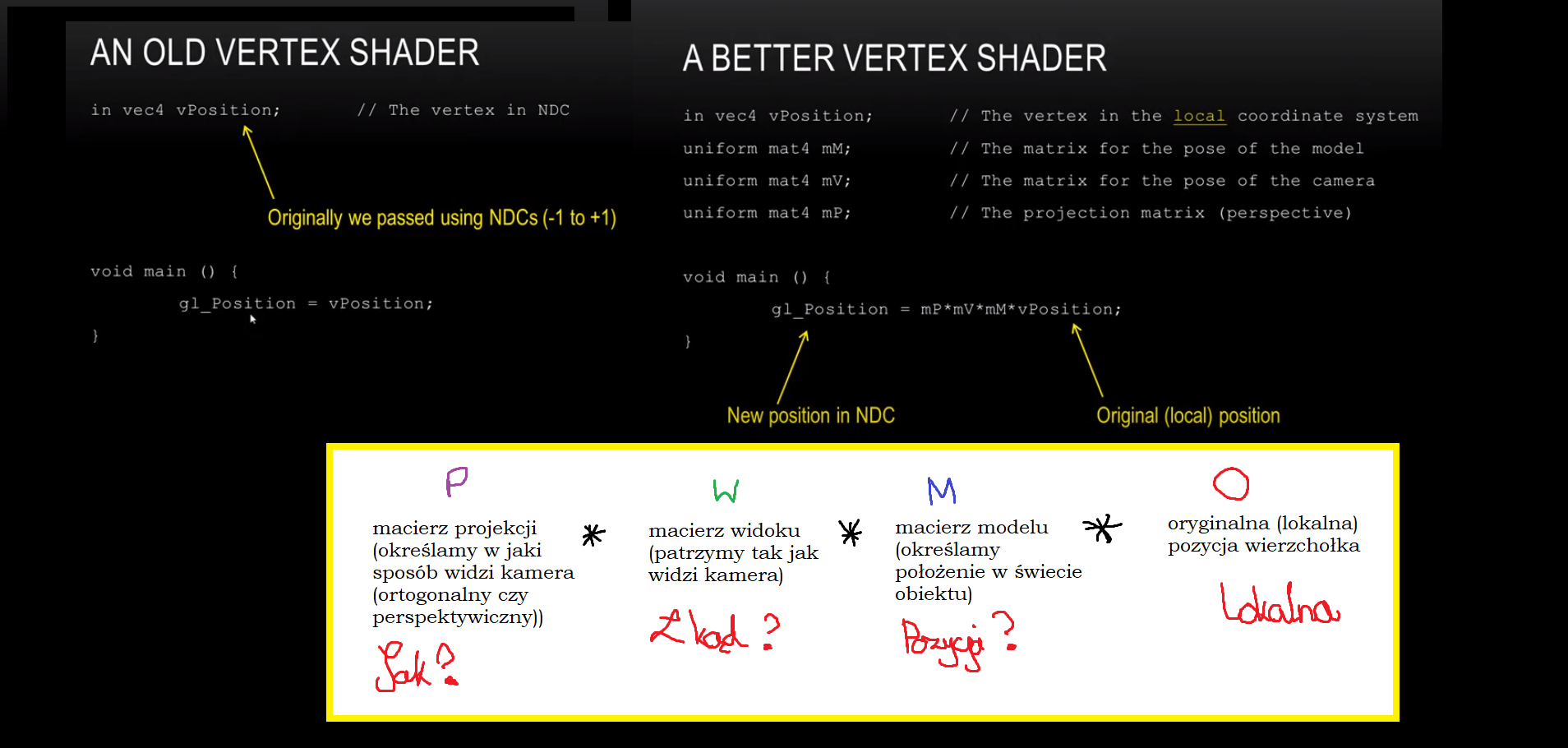
**Indeksowane rysowanie (ang.: Indexed Draws ~ IBO):**

* Zdefiniowanie wierzchołków do narysowania sześcianu.
* Sześcian składa się z 12 trójkątów (dwa na każdą stronę).
* 12 x 3 wierzchołki na trójkąt = 36 wierzchołków.
* Tylko przecież sześcian ma 8 wierzchołków.
* Niektóre zostaną określone kilka razy, kiepskie.
* Zdefiniujmy tylko 8 wierzchołków sześcianu.
* Ponumerujemy jest od 1 do 8 (albo 0 do 7 w C++).
* Oraz nawiążmy do nich poprzez liczbę.

Po prostu nawiąż do nich oraz ***ELEMENT ARRAY BUFFER*** w VAO.:

***glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, IBO);***

* Czasami są one nazywane elementami zamiast indeksami. Mają te same znaczenie.
* Może być dalej trochę trudno.
* Rozwiązanie: Oprogramowanie do tworzenia modeli 3D.
* Później będziemy ładować modele.



**Projekcje (ang.: Projections):**

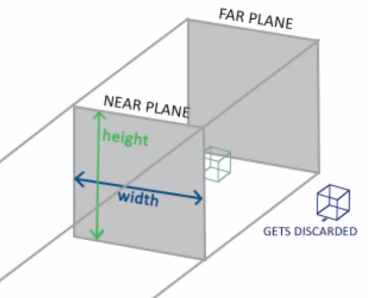
* Używane do konwersji z „Przestrzeni Widoku (ang.: View Space)” do „Przestrzeni Ujęcia (ang.:Clip Space).
* Może być użyte do ujęcia sceny z perspektywy 3D.
* Alternatywnie może być użyte do stworzenia stylu 2D.
* Należy rozumieć system kordynatów.

**Systemy koordynatów:**

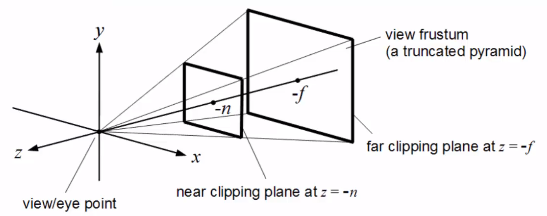
* **Przestrzeń lokalna** (ang.: Local Space): Czysta/ surowa pozycja każdego wierzchołka relatywnie do pochodzenia (ang.: origin). **Pomnożona przez macierz modelu (ang.: Model Matrix)** żeby dostać przestrzeń świata (**ang.: World Space**).
* **Przestrzeń świata** (ang.: World Space): Pozycja każdego wierzchołka w świecie sama w sobie jeżeli kamera jest przypuszczana mieć pozycję w pochodzeniu (ang.: origin). **Pomnożona przez macierz widoku (ang.: View Matrix) aby uzyskać przestrzeń widoku (ang.: View Space).**
* **Przestrzeń widoku** (ang.: View Space): Pozycja każdego wierzchołka w świecie, która jest relatywnie do kamery pozycji oraz orientacji. **Pomnożona przez macierz Projekcji (ang.: Projection Matrix) żeby dostać przestrzeń ujęcia (ang.: Clip Space).**
* **Przestrzeń ujęcia** (ang.: Clip Space): Pozycja każdego wierzchołka w świecie, relatywnie do kamery pozycji oraz orientacji, widziana w obszarze nie do przecięcia z finalnego wyjścia.
* **Przestrzeń ekranu** (ang.: Screen Space): Po wycinaniu (clipping) ma miejsce, finalny obraz jest tworzony oraz umiejscowiany w systemie koordynatów okienka samego w sobie (ten standardowy).

**Projekcje:**

* Żeby stworzyć **ujęcie przestrzeni** (ang.: Clip Space) z określonym obszarem (frustum ~ stożek ścięty) z tego co nie ma być wycięte za pomocą macierzy projekcji.
* Dwa powszechnie używane projekcji:
  + **Ortograficzna** (używana do gier 2D),
    - Stożkiem ściętym ortograficznej projekcji jest prostopadłościan.
    - Wszystko co jest pomiędzy bliskiej oraz dalekiej płaszczyzny jest zachowane, reszta nie.
    - Równoległa natura ortograficznej znaczy 3D głębia nie istnieje.
    - Poruszanie obiektem bliżej/ dalej nie zmieni jego rozmiaru na ekranie.

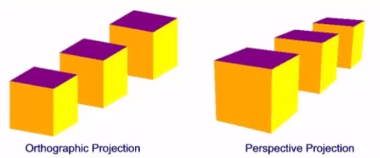


* + **Perspektywiczna** (używana do gier 3D):
    - Stożkiem ściętym dla perspektywicznej projekcji jest kadłubowa piramida.
    - Każdy piksel na piksel na bliskiej płaszczyźnie rozbiega się pod kątem żebym połączyć się z pasującym punktem na dużej płaszczyźnie.
    - Dzięki niej mam efekt głębokości.



**Porównanie projekcji:**

* Ortogonalna: Ten ostatni z tyłu ma taki sam rozmiar jak ten z przodu, sugerując że jest większy.
* Perspektywiczna: Ten ostatni z tyłu wygląda na mniejszego od te z przodu, przez to że jest bardziej odległy, tak jak powinno.



**Projekcje z GLM oraz OpenGL:**

* **glm::mat4 proj = glm::perspective(fov, aspect, near, far);**
* **fov =** field-of-view (pole widzenia), kąt widzenia stożka ściętego.
* **aspect =** aspect ratio of the viewport (proporcje portu widzenia (zazwyczaj dlugosc podzielona przez wysokość).
* **near** = odległość najbliższej płaszczyzny.
* **far** = odległość najdalszej płaszczyzny.
* **Powiąż otrzymaną macierz z uniformem w shaderze.**

**gl\_Position = projection \* view \* model \* vec4(pos, 1.0);**

clip space<-view space<- world space <-local space

* Znaczenie mnożenia macierzy ma znaczenie!

**Podsumowanie:**

* Interpolacja oblicza wartości ważone pomiędzy wierzchołkami podczas rasteryzacji.
* Rysowanie indeksowane pozwala nam raz zdefiniować wierzchołki potem do nich nawiązywać podczas rysowania.
* Macierze projekcji zamieniają przestrzeń widoku (ang.: View Space) do przestrzeni ujęcia (ang.: Clip Space).
* Ortogonalna projekcja jest używana do 2D aplikacji i nie pozwala na dostrzeganie głębi.
* Perspektywiczna projekcja jest używana do 3D aplikacji oraz tworzy iluzje głębi.
* GLM mas **glm::perspective** funkcje, która tworzy macierze perspektywicznej projekcji (macierz projekcji typu persepktywiczna).

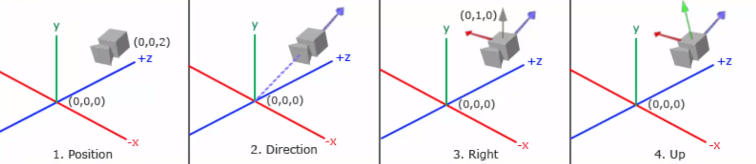
😊

**Kamera / Przestrzeń Widoku (ang.: View Space):**

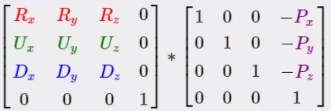
* Kamera przetwarza scenę tak jak jest widziana w „Przestrzeni Widoku”.
* Przestrzeń widoku jest system kordynatów gdzie każdy wierzchołek jest widziany z kamery.
* Używamy macierzy widoku do konwersji z Przestrzeni świata do Przestrzeni widoku.
* **Macierz widoku wymaga 4 wartości:** Pozycji kamery, Kierunku, Prawa oraz Góra.

Jeżeli chcemy kamerę przesunąć do tyłu to tak naprawdę cały świat przesuniemy do tyłu, ponieważ kamera jest cały czas w tym samym miejscu!

* **Pozycja kamery**: Po prostu pozycja kamery.
* **Kierunek:** Kierunek kamery w, który ona patrzy.
* Wektor kierunku tak naprawdę jest skierowany w przeciwną stronę niż intuitywnie „kierunek”.
* **Prawa:** Wektor, który jest **skierowany na prawo** od kamery, definiuje płaszczyznę x. Może być obliczony przez zrobienie **produktu krzyżowego** (ang.: **Cross Product**) na kierunku oraz „Góra” (nie relatywny do kamery!!!) wektorze [0, 1, 0].
* **Góra:** Jest **skierowany w górę** relatywnie gdzie kamera jest skierowana. Może uzyskać ten wektor przez zrobienie **produktu krzyżowego** (ang.: Cross Product) na kierunku oraz Prawym wektorze.



* Połóż wartości w macierzy, żeby obliczyć macierz widoku.
* Macierz widok jest **aplikowana do każdego wierzchołka** **przekształci go do przestrzeni widoku**.



* Na szczęście, GLM ma funkcję, która zrobi to wszystko.

**glm::mat4 viewMatrix = glm::lookAt(position, target, up);**

* glm::lookAt(position, target, up);
* position ~ pozycja kamery.
* target ~ punkt na który kamera patrzy. Np. Zazwyczaj dodajmy dany wektor do pozycji kamery. Jak chcemy patrzeć w dół rogu pokoju to bierzemy kierunek do niego (pozycja kamery – tego rogu) a potem ten kierunek dodajemy do pozycji kamery 😊.
* Target (ang.: cel) jest zazwyczaj zdefiniowany jako pozycja kamery z kierunkiem dodanym do niego. Efektywnie mówiąc „spójrz przed siebie”.
* UP ~ Kierunek skierowany ku górze **ŚWIATA**, nie kamery. **lookAt** używa tego do obliczenia „prawa” oraz „góra” relatywnie do kamery.

**Używanie macierzy widoku:**

* Powiąż macierz widoku z uniformem na shaderze.
* Zastosuj go pomiędzy macierzą projekcji oraz macierzą modelu.

**gl\_Postion = projection \* view \* model \* vec4(pos, 1.0) (**initial position ~ początkowa/ lokalna pozycja**);**

* Zapamiętaj: **MNOŻENIA MACIERZY MA ZNACZENIE**.
* Mnożenie ich w różnych kolejnościach nie zadziała!

**Input: Poruszanie kamerą**

* Potrzebne żeby zmienić pozycję kamery.
* **GLFW: *glfwGetKey(****window, GLFW\_KEY\_W);*
* **SDL:** Sprawdza zdarzenie, jeżeli jest to zdarzenie KEYDOWN, sprawdza, który klawisz został wciśnięty.
* Potem dodajemy wartość do pozycji kamery kiedy przycisk jest wciśnięty.

**Input: Przyrost czasu** **(ang.: Delta Time)**

* Szeroka koncepcja,
* Podstawowa idea: Sprawdza jak dużo czasu minęło od ostatniej pętli, aplikuje matematykę, żeby utrzymać stałą prędkość.
* **deltaTime = currentTime – last Time;**

**lastTime = currentTime;**

* Potem należy pomnożyć szybkość poruszania kamerą przez przyrost czasu.

gafferongames.com/post/fix\_your\_timestep/

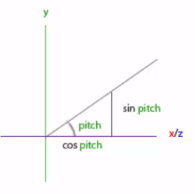
**Input: Obracanie**

* Trzy typy kątów.
* **Pitch:** Patrzenie w górę lub w dół.
* **Yaw:** Patrzenie w prawo lub lewo.
* **Roll:** Jak samolot robi beczkę.

Pitching żeby obracać się górę oraz dół osi relatywnie do yaw.

Yaw tylko nas obróci wokół naszej osi (y-axis).

1. **Pith (pochylanie):**

Oś pochylania jest zależna od osi odchylania (yaw)… musi zaktualizować x, y oraz z.

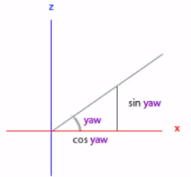
y = sin(pitch)

x = cos(pitch)

z = cos(pitch)

**Pamiętać:** My aktualizujemy **x** oraz **z** ponieważ oś odchylania mogłaby mieć kamerę patrzącą łącznie z ich kombinacją.

1. **Yaw (odchylanie):**

Możemy bazować osią odchylenia na osi pochylenia też, ale będzie to nie realistyczne!

Zatem tylko aktualizujemy **x** oraz **z**.

x = cos(yaw)

z = sin(yaw)

**Input: Obracanie ~ Pochylenie oraz odchylenie:**

* Połączenie tych wartości z pitch oraz yaw żeby dostać wektor kierunkowy z tymi atrybutami/ właściwościami.

x = cos(pitch) x cos(yaw)

y = sin(pitch)

z = cos(pitch) x sin(yaw)

* Vektor [x, y, z] będzie miał dany pitch (nachylenie) oraz yaw (odchylenie).
* **Aktualizujemy kierunek patrzenia kamery z tym nowym uzyskanym wektorem.**
* **GLFW: *glfwSetCurorPosCallBack****(window, callback);*

Przechowujemy starą pozycję, porównujemy ją do nowej pozycji. Używamy różnicy żeby zdecydować zmianę pitch/ yaw.

Jeżeli x zostało zwiększone to chcemy zwiekszyc pitch.

Jeżeli y zostało zwiększone to chcemy zwiększy yaw.

* **SDL:** Sprawdzaj dla SDL\_MOUSEMOTION zdarzenie.

Wywołaj SDL\_GetMouseState(&x, &y);

Potem zrób to samo co powyżej.

**Podsumowanie:**

* Macierz widoku wymaga **pozycji,** **kierunku** patrzenia (direction) oraz **prawa**, **gora** wektorów.
* **glm::lookAt** zrobi to dla nas.
* Żeby poruszyć kamerą, żeby zmieniać pozycję kiedy klawisz został wciśnięty.
* **Przyrost czasu** (delta time) pozwala nam **zachować spójność** prędkości pomiędzy systemami.
* **Obracanie używa Pitch (nachylenia) oraz Yaw (odchylenia)** (i czasem Roll w pewnych przypadkach).
* Używamy Pitch oraz Yaw żeby obliczyć nowe wektory kierunkowe.
* **Porównujemy ostatnią oraz obecną** pozycję myszki żeby zadecydować jak pitch (nachylenie) oraz yaw (odchylenie) zmienić.