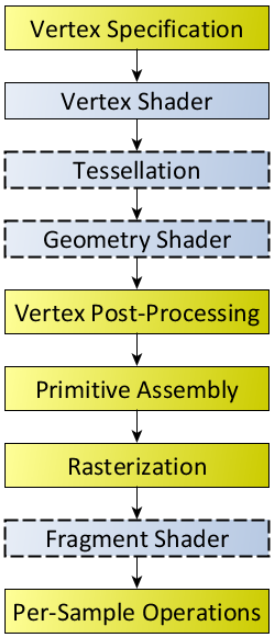
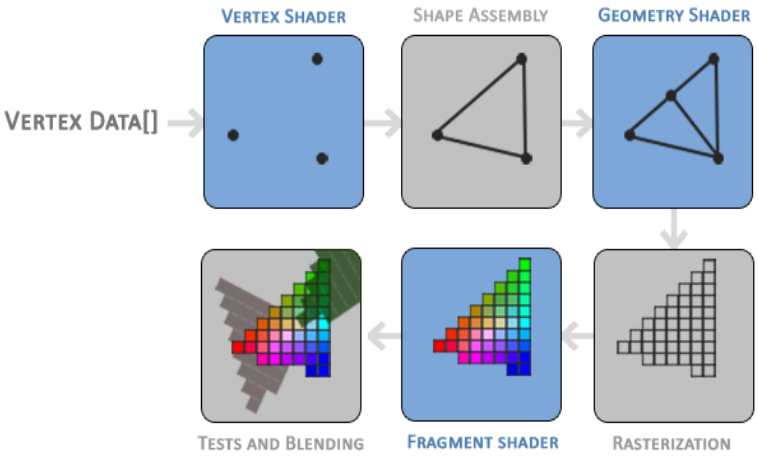
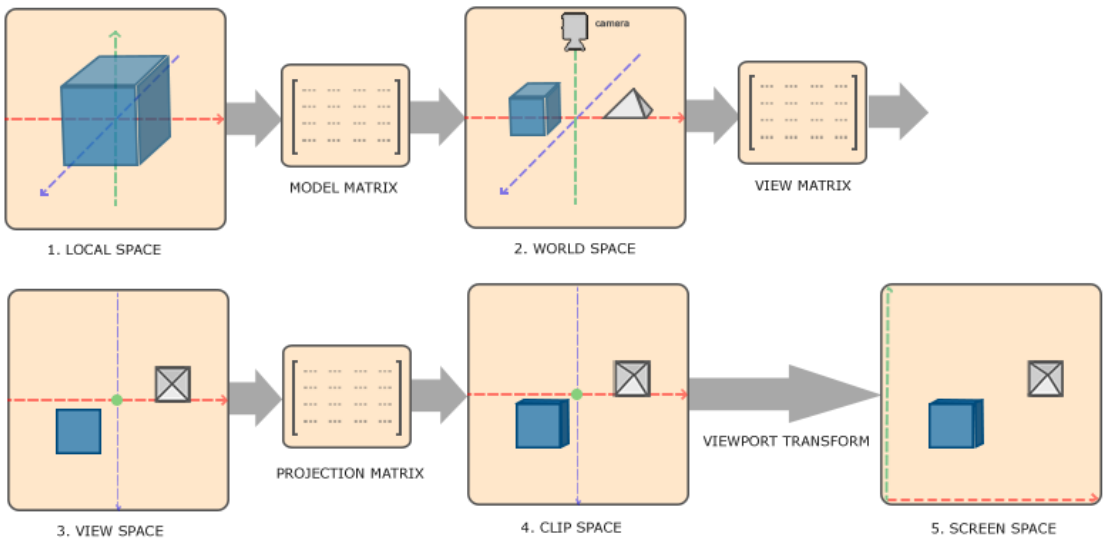
1. Cele kursu:

- 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):

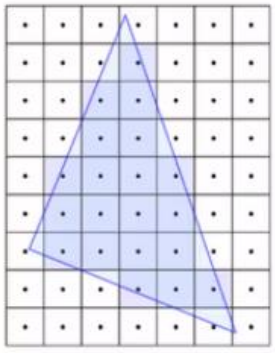
- Prymitywy, które nie są widziane są usuwane (nie chcemy rysować rzeczy, których nie widać).
- **Pozycje przekonwertowane z *przestrzeni obkrajania* („clip space”) do *przestrzeni okna* („window space”).**

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.

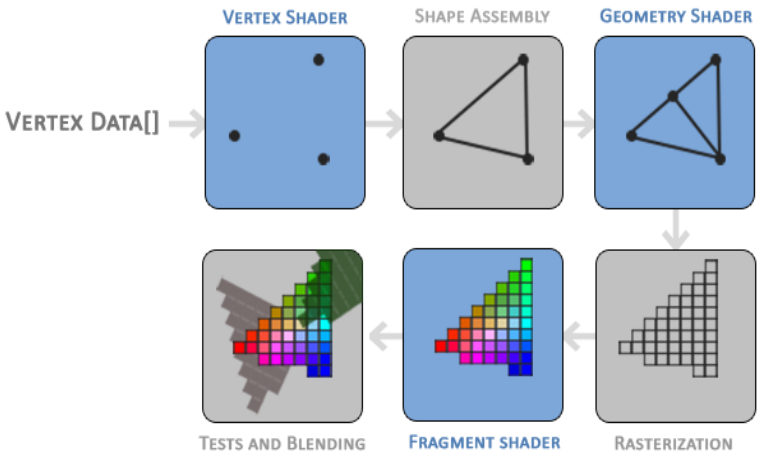
Oryginalny widzi użytkownik.

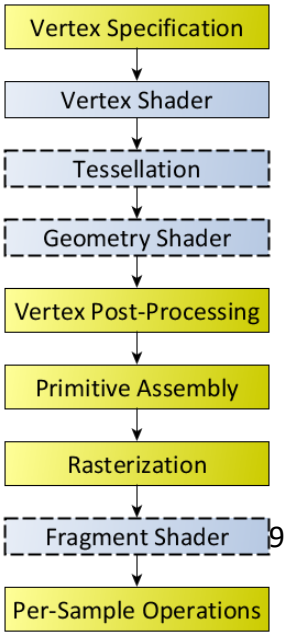
Na koniec oryginalny jest zamieniany z framebuffer, stary framebuffer staje się oryginalnym a stary oryginalny staje się nowym framebufferem

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).*

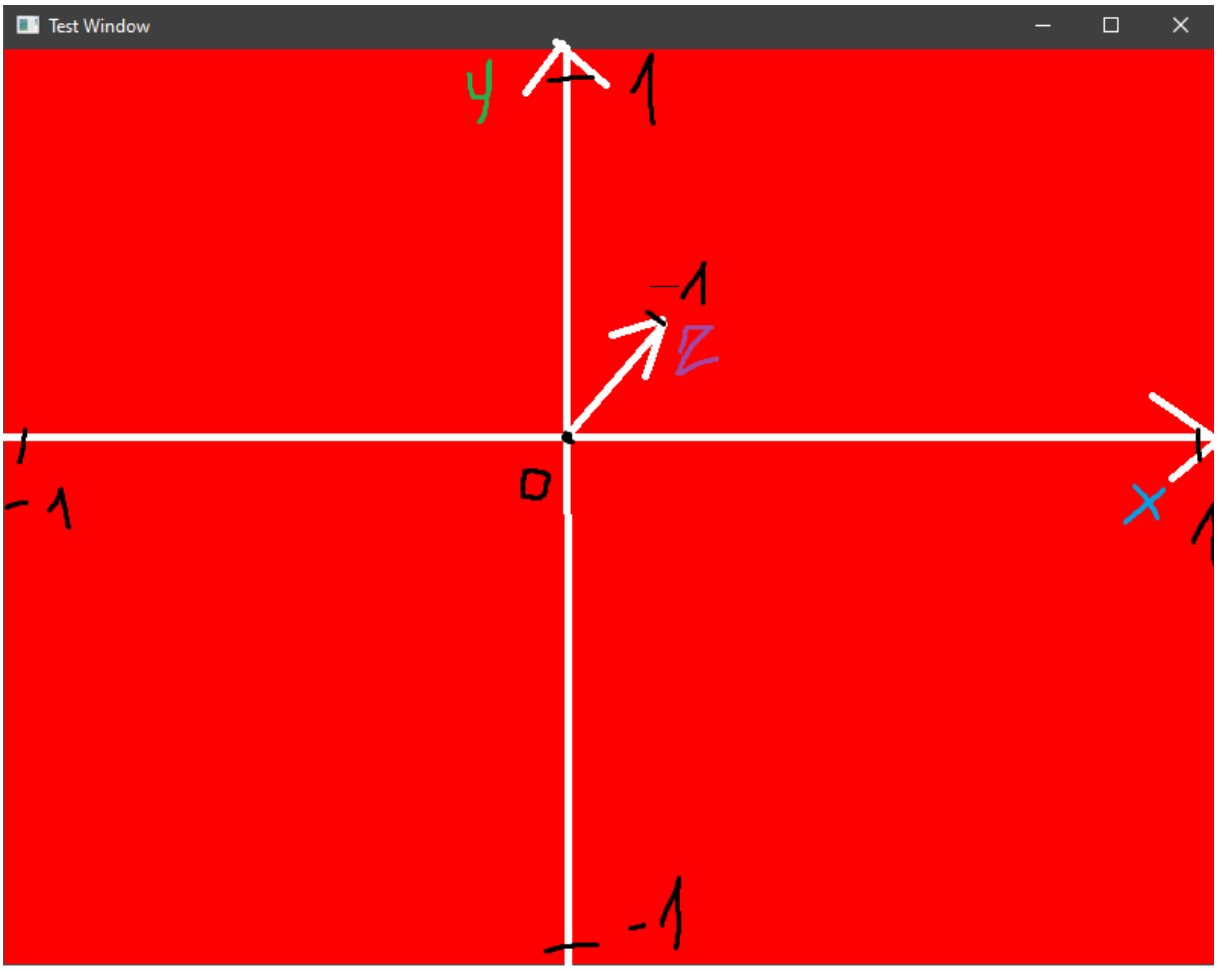
Podsumowanie:

- 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.

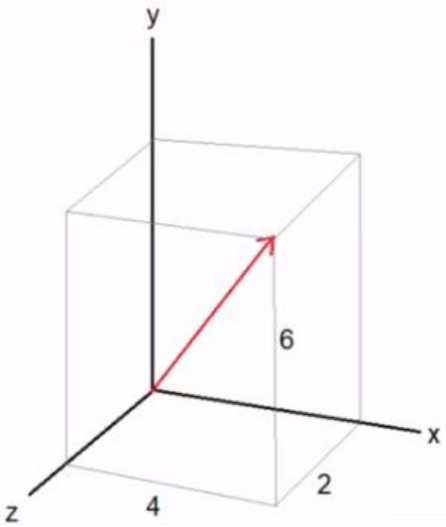
4. Wektory, macierze oraz jednolite zmienne (uniform variables):

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]**

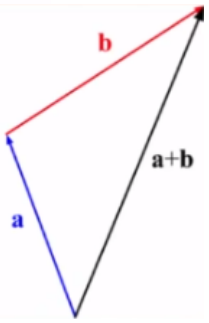
$$v = \begin{bmatrix} 4 \\ 6 \\ 2 \end{bmatrix}$$



Dozwolone operacje:

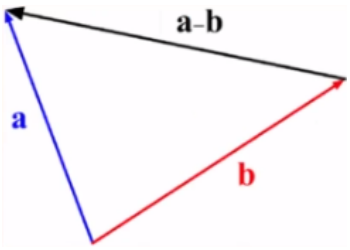
- ***Dodawanie*:**

[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)**

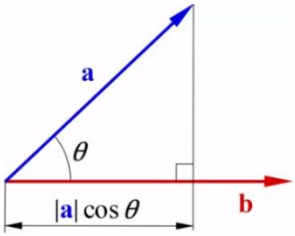
$$\cos^{-1}((v1 \bullet v2)/(|v1| * |v2|)) = \varphi$$

- 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**.

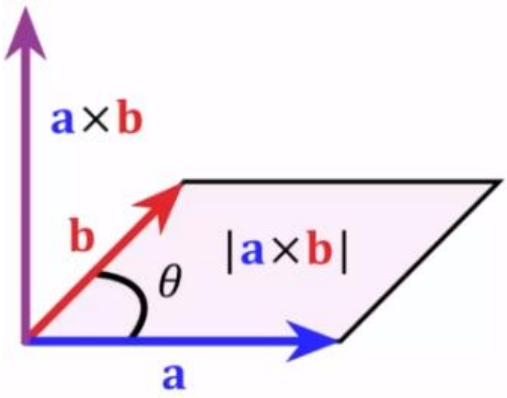
$$u = \frac{V}{|V|}$$

- $v = [1, 2, 2]$
- $|v| = sqrt(1^2 + 2^2 + 2^3) = sqrt(1 + 4 + 4) = sqrt(9) = 3$
- $u = \frac{[1,2,2]}{3} = [\frac{1}{3}, \frac{2}{3}, \frac{2}{3}]$
- 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:

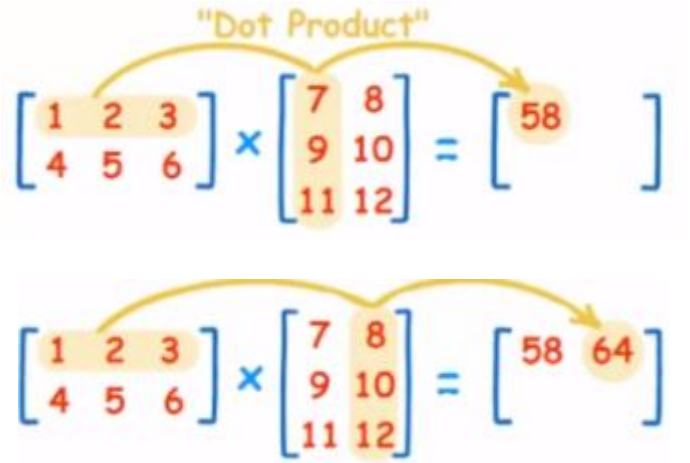
- Skalar: Po prostu dodaje/ odejmuje wartość z każdego elementu, podobnie jak w przypadku wektorów.
- Macierz: Dodaje wartości w przeliczeniu na element. Każdy musi pasować swoją pozycją do innej macierzy.
- To znaczy, że **rozmiary macierzy**, które chcemy dodać lub odjąć muszą być **takie same**.

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

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..

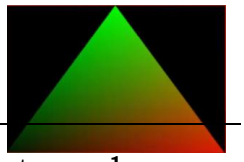
Trzy główne macierze:

- 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):

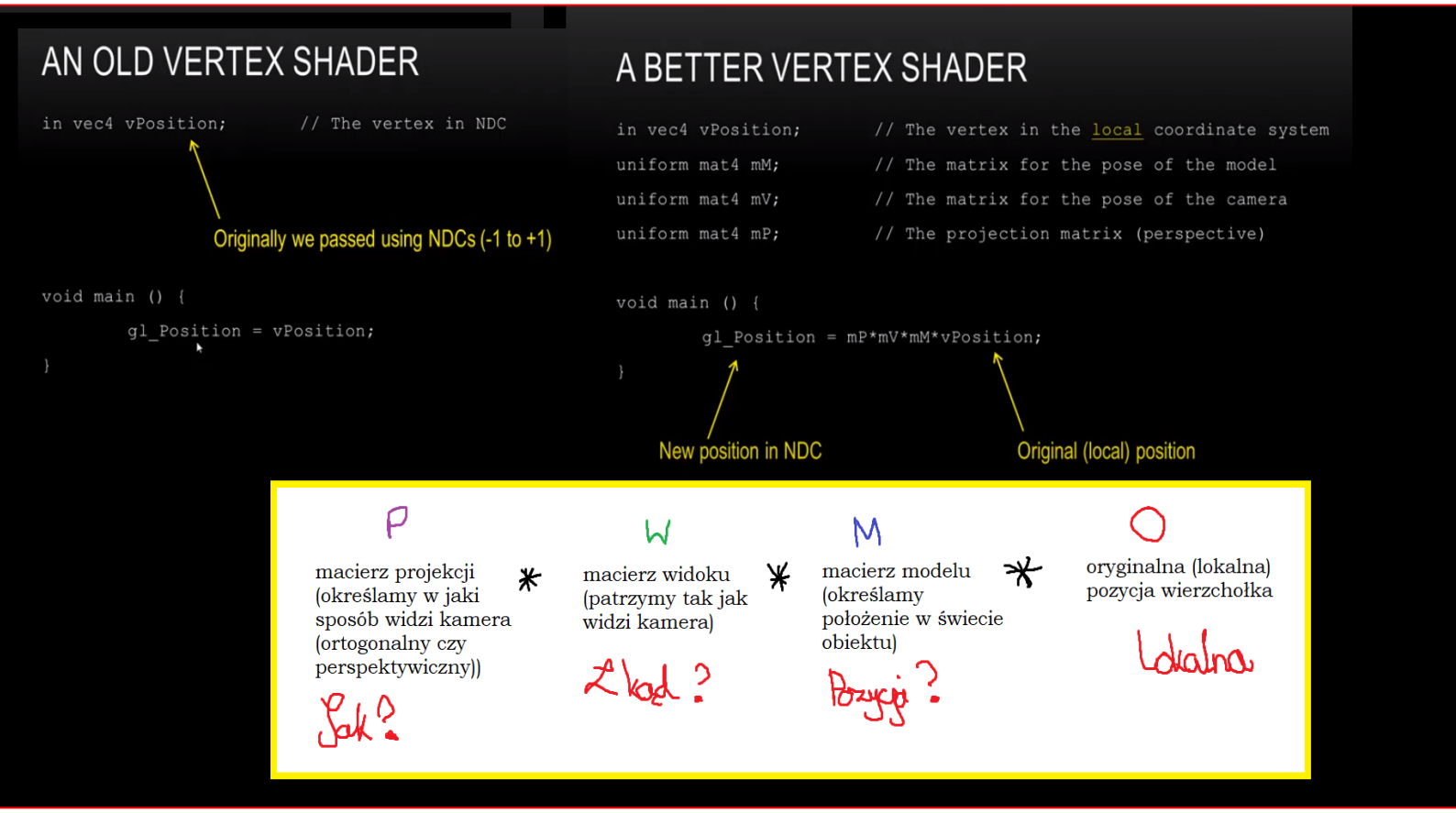
- Zdefiniowanie wierzchołków do narysowania sześcianu.
- Sześcian składa się z 12 trójkątów (dwa na każdą stronę).
- 12 x 3 wierzchołki na trójkąt = 36 wierzchołków.
- Tylko przecież sześcian ma 8 wierzchołków.

- 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.

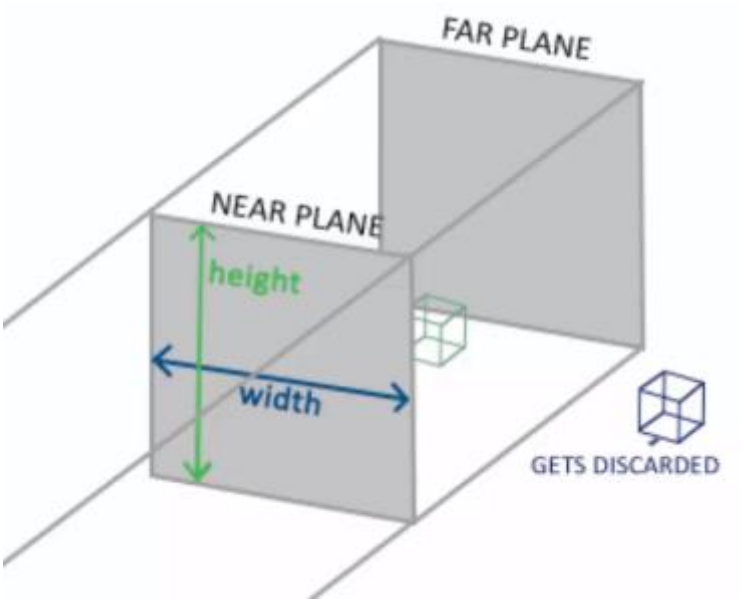
Systemy koordynatów:

- **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).

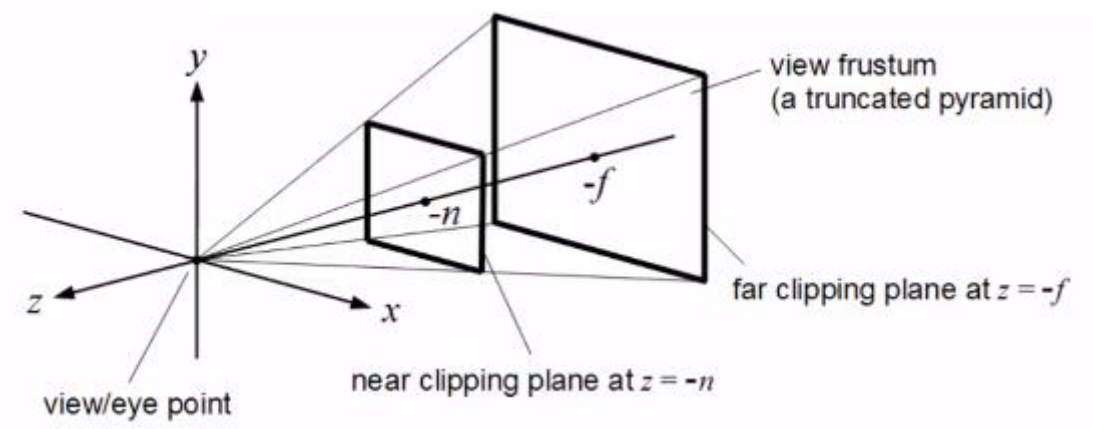
Projekcje:

- 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.

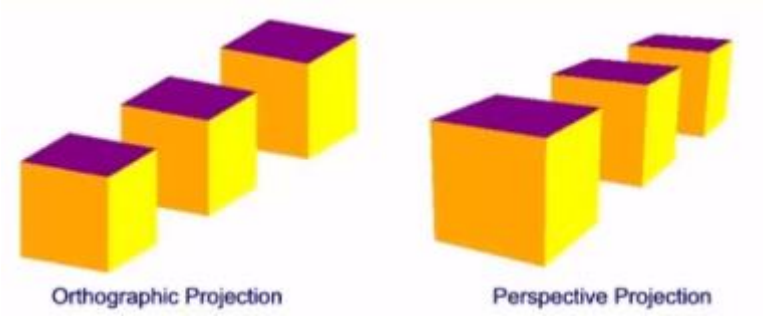


- **Perspektywiczna** (używana do gier 3D):
  - Stożkiem ściętym dla perspektywicznej projekcji jest kadłubowa piramida.
  - Każdy piksel na piksel na bliskiej płaszczyźnie rozbiega się pod kątem żebym połączyć się z pasującym punktem na dużej płaszczyźnie.
  - Dzięki niej mam efekt głębokości.



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).

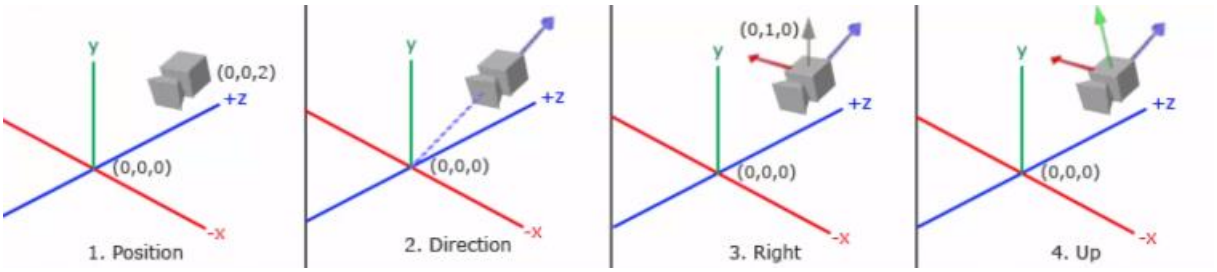
😊

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**.

$$\begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- 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

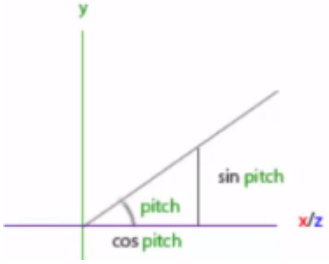
- Trzy typy kątów.
- **Pitch:** Patrzenie w górę lub w dół.
- **Yaw:** Patrzenie w prawo lub lewo.
- **Roll:** Jak samolot robi beczkę.

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

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

a) 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.

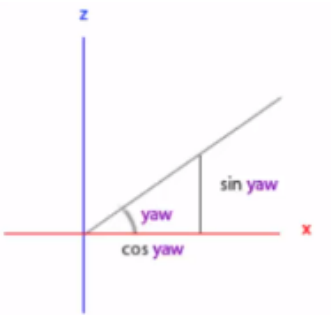
b) Yaw (odchylanie):

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.

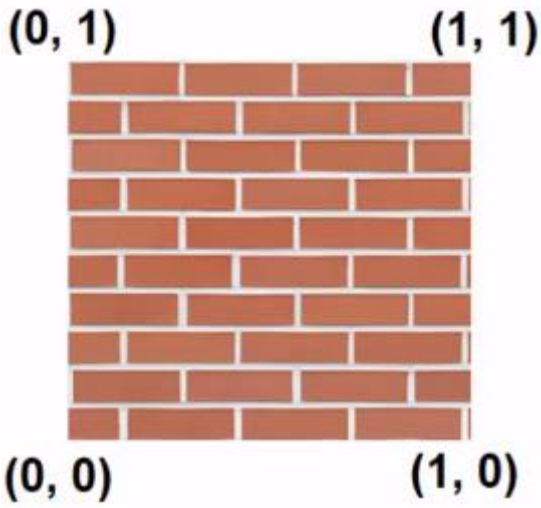
Podsumowanie:

- 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.

Tekstury:

- Tekstury są obrazki używane, żeby dodać ekstra detale do obiektu.
- Tekstura jest to **macierz tekseli**.
- Tekstury mogą być również używane żeby trzymać ogólne dane.
- Zazwyczaj 2D ale mogą być również 1D albo 3D tekstury. Są też cubmap textures ale są tablice 2D tekstur (np.: do wolumetrycznych map).
- Punkty na teksturach są **„tekselami”** (ang.: texels) a nie pikselami.
- Teksele są zdefiniowane **pomiędzy 0 oraz 1**.
- Zatem żeby spróbkować punkt w górnym-środku nawiązujesz do teksela (0.5, 1).
- **Zmapuj teksele do wierzchołków.**
- **Interpolacja między każdym fragmentem obliczy odpowiedni teksel pomiędzy przypisanymi tekselami**.



Obiekty tekstur:

- Utworzenie tekstury działa **podobnie** jak tworzenie VBO oraz VAO.
- **glGenTextures(1, &texture)**

  glBindTexture(GL_TEXTURE_2D, texture)
- Są różne rodzaje tekstur, takie jak:
  - GL_TEXTURE_1D,
  - GL_TEXTURE_3D,
  - GL_TEXTURE_CUBE_MAP
- **glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);**

1 argument: Cel na którego nakładamy teksturę (ang.: Texture Target).

2 argument: Mipmap Level

3 argument: Format przechowywanych danych na karcie. RGB są czerwony, zielony oraz niebieskie wartości. Jest też RGBA, która ma wartość odpowiadającą za przezroczystość (Alpha).

4 argument: Długość tekstury (piksele).

5 argument: Wysokość tekstury (piksele).

6 argument: To powinno być zawsze 0. Stary koncept obsługiwania krawędzi tekstury, który już nie jest używany przez OpenGL.

7 argument: Format danych jakie przesyłamy do karty (w przeciwieństwie do zachowanego w trzecim argumencie.)

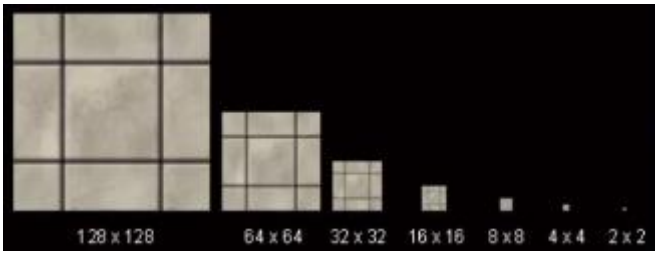
8 argument: Typ danych tych wartości (int, float, byte, itp.).

9 argument: Dane same w sobie.

Mipmaps (mip mapy):

- Limitacje rozdzieloczości tekstury.
- Im bliżej się zbliżamy się, tym bardziej pikselowany tekstura staje się. Im dalej, tym bardziej się stara renderowania kilka tekseli na jednym pikselu.

- Rozwiązanie: Stwórz kilka wersji obrazka o różnej rozdzielczości i zamieniaj je między sobą bazują na dystansie.



Parametry tekstury ~ Filtry:

- Co jeśli chcemy wyrenderować środek tekseli?
- Mamy dwie możliwości:
  - **Najbliższa:** Używa teksela, który najwięcej zasłania (tworzy efekty pikselowania).
  - **Liniowa:** Używa średniej ważonej z otaczającej jej tekseli (miesza granice pikseli).
- **Liniowa jest bardziej popularna.**
- **Najbliższą użyj jeżeli chcesz mieć efekty pikselozy (takie jak w grach retro).**

Żeby ustawić filtry należy:

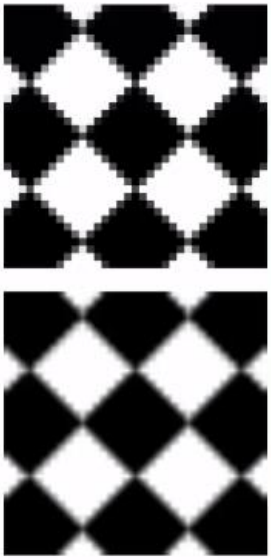
- **glTexParameter:** Używany aby ustawić tekstury renderowania parametry.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

- **GL_TEXTURE_MIN_FILTER:** Filtr, który będzie aplikowany gdy tekstura robi się mniejsza (jest dalej).
- **GL_TEXTURE_MAG_FILTER:** Filtr, który będzie aplikowany gdy tekstura robi się większa (jest bliżej).
- **GL_LINEAR:** Filtr liniowany (miesza otaczające teksele).
- **GL_NEAREST:** Filtr najbliższy (wybiera najbliższy teksel to prókowania punktu).

Porównanie:

- Górny obrazek: Używa najbliższego, ma bardziej spikselozowany wygląd, dobry dla pewnych stylów.
- Dolny obrazek: Używa liniowego, miesza teksele razem żeby stworzyć łagodniejszy wygląd. Dobrze działa na zawansowanych teksturach a mniej na prostszych.



Parametry tekstury ~ Owijanie (Wrap)

- Co jeśli chcemy spróbkować punkt powyżej 0,1 zasięgu?
- Wiele opcji to obsłużenia takiej sytuacji:
  - Powtarzamy teksturę.
  - Powtarzamy odbitą lustrzanie formę tekstury.
  - Poszerzamy piksele na krawędziach.
  - Aplikujemy kolorowe krawędzie.
- Możemy użyć **glTexParameter** żeby określić jak to obsłużyć.

Jak to ustawić?

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

GL_TEXTURE_WRAP_S : Jak obsłużymy owijanie na „s-osi” (x-axis).

GL_TEXTURE_WRAP_T : Jak obsłużymy owijanie na „t-osi” (y-axis).

- **GL_REPEAT:** Powtarzamy teksturę.
- **GL_MIRRORED_REPEAT:** Powtarzamy oraz odbijamy lustrzanie teksturę.
- **GL_CLAMP_TO_EDGE:** Rozszerza piksele na krawędziach.
- **GL_CLAMP_TO_BORDER:** Aplikuje kolory na krawędziach.

Ładowanie obrazków jako tekstury:

- Możemy napisać własny ładownik obrazków (ang.: image loader).
- Będzie to trochę trudne, ponieważ musimy obsłużyć wiele typów obrazków (bmp, jpg, png, gif, tga, itp.).
- Biblioteki służące do ładowania obrazków zrobią to za nas.
- Popularna bibliotek: **Simple OpenGL Image Library (SOIL).**
- My będziemy używali dla uproszczenia mniejszej biblioteki: **stb_image**.

Używanie stb_image:

- Wymaga tylko pliku nagłówkowego, zatem jest lekka.
- **Początek** projektu **musi zaczynać** się z:

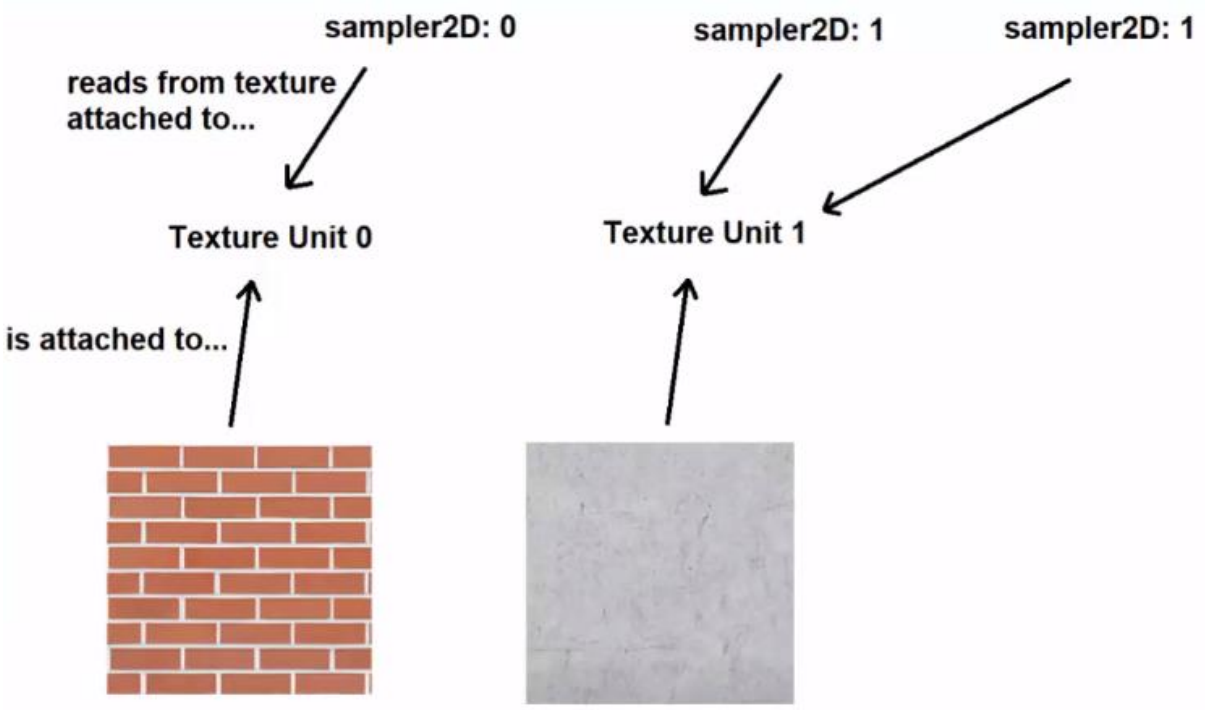
  #define STB_IMAGE_IMPLEMENTATION

- **unsigned char* data = stbi_load(''image.jpg'', &width, &height, &bitDepth, 0);**
- Będziemy musieli obrócić obrazek.
- **stbi_set_flip_vertically_on_load(true);**

Próbki tekstury:

- Tekstury w Shaderach są dostępne poprzez próbki (ang.: Samplers).
- Tekstury są załączane/ powiązane do jednostek tekstury. (ang.: Texture Unit).
- Próbki mają dostęp do tekstur załączonych do jednostek tekstur.
- W shaderze używamy **sampler2D** typu.
- Żeby dostać wartość tekselu, używamy GLSL **„texture"** funkcji.
- **texture(textureSampler, TexCoord);**
- **textureSampler:** Obiekt sampler2D.

- **TexCoord:** Interpolowany koordynat tekselu w fragment shaderze.



Jednostki tekstur:

- Ustaw/ powiąż z pożądaną jednostką tekstury:

  glActiveTexture(GL_TEXTURE0);

  glBindTexture(GL_TEXTURE_2D, textureID);

- Zapewnij, że zmienna typu sampler2D wie, do której jednostki tekstury ma dostęp:
  glUniform1i(uniformTextureSampler, 0);
- Wartość załączona do uniformu jest liczbą jednostki tekstury.

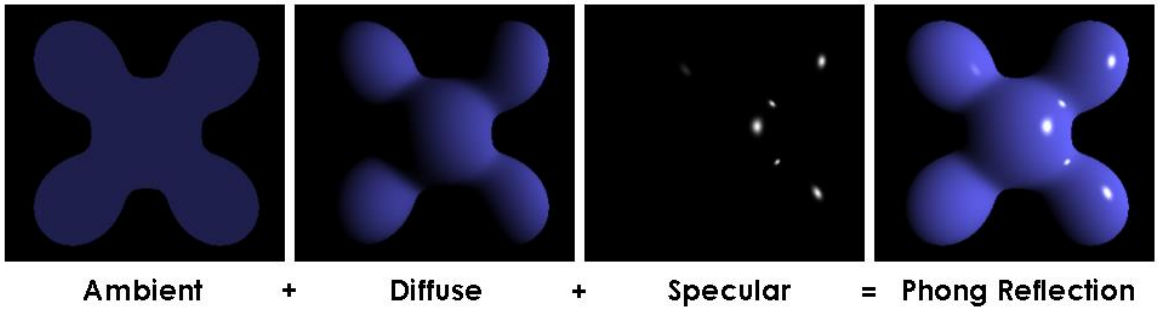
Podsumowanie:

- Tekstury używają tekseli pomiędzy 0 oraz 1.
- Tekstury są powiązane z wierzchołkami a wartości są interpolowane.
- Mimaps obsługują poziom szczegółowości bardziej zgrabnie.
- Filtrowanie tekstur określają jak teksele są zmieszane (lub nie) bazując na rozmiarze na ekranie.

- Owijanie tekstur zmienia jak tekstury są obsługiwane dla wartości tekselów powyżej 0 oraz 1 zasięgu.
- Filtrowanie oraz owijanie są definiowane używając **glTexParameteri** funkcji.
- Ładowanie obrazków z trzeciego rzędu biblioteką żeby było prościej.
- SQIL jest to popularna biblioteka ale stb_image jest bardziej lekka dla tego projektu.
- Tekstury są załączane do jednostek tekstury, próbki odczytują tekstury załączone do jednostek tekstury.

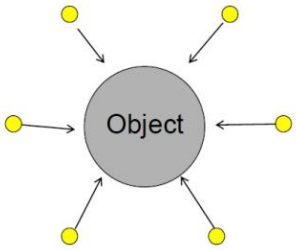
Phong Lighting oraz Directional Lights:

Phong Lighting ~ model oświetlenia (ang.: lighting model). Jest to sposób jak możemy ująć obliczenia żeby stworzyć efekt oświetlenia sceny np.: Światło kierunkowe (ang.: directional light).



Phong lighting składa się z trzech części:

- **Ambient Lighting** (światło otoczenia): Światło, które zawsze jest obecne, nawet jeżeli słońce światła jest bezpośrednio zablokowane.



  - Najprostszy koncept.
  - Symuluje odbicia światła od obiektów.
  - Na przykład: Tworzy cień na ziemi z twoją ręką, używając słońca. Dalej możesz zobaczyć kolor w cieniu! Dalej jest oświetlony.
  - Globalna iluminacja symuluje to ~ zawansowany.

Jak go uzyskać?

Proces tworzenia czynnika oświetlenia otoczenia (ang.: ambient lighting factor):

ambient = lightColour * ambientStrength;

Ten czynnik pokazuje jak **dużo koloru fragmentów** te oświetlenie otoczenia (ang.: light's ambient) pokazuje.

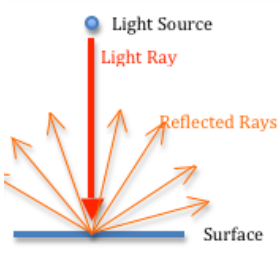
fragColour = objectColour * ambient;

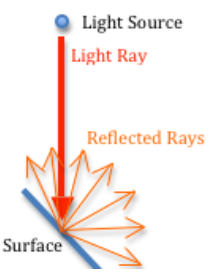
Jeżeli **ambient = 1** (pełna moc) to znaczy, że fragment jest cały oświetlony.

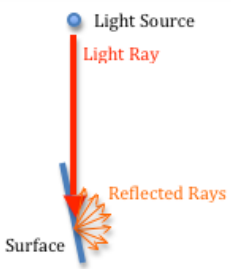
Jeżeli **ambient = 0.5** (pół mocy) to znaczy, że fragment jest w połowie swojego koloru.

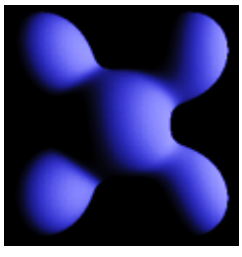
Jeżeli **ambient = 0** (brak mocy) to znaczy, że fragment jest zawsze czarny.

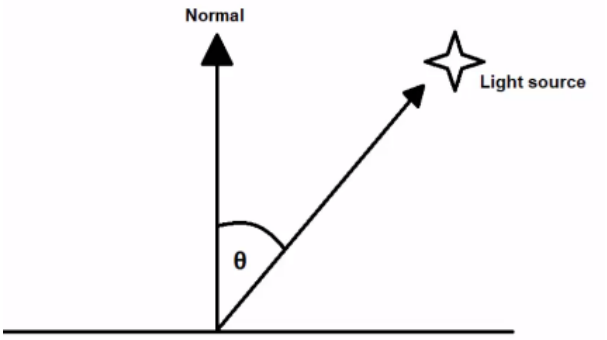
- **Diffuse Lighting** (światło rozproszone): Światło zdeterminowane przez kierunek źródła światła. Tworzy efekt przyciemnienia dalej od światła.
  - Bardziej zawansowane.
  - Symuluje spadanie światła jak kąt padania światła staję się bardziej powierzchniowy.
  - Cześć skierowana bezpośrednio do światłą jest bardziej oświetlona.
  - Cześć skierowana pod kątem jest bardziej ciemna.
  - Możemy użyć kąta pomiędzy wektorem łączącym źródło światła z fragmentem oraz wektora, który jest prostopadły do powierzchni (surface „normal") (ang.: face).











- Używa Φ do określenia współczynnika rozproszenia.
- Mniejszy Φ tym **więcej** światła.
- Większy Φ tym **mniej** światła.

Powtórzenie z Wektor: Dot produkt.

v1 * v2 = |v1| x |v2| x cos(Φ)

Jeżeli oba wektory są **znormalizowane** (są wektorami jednostkowymi):

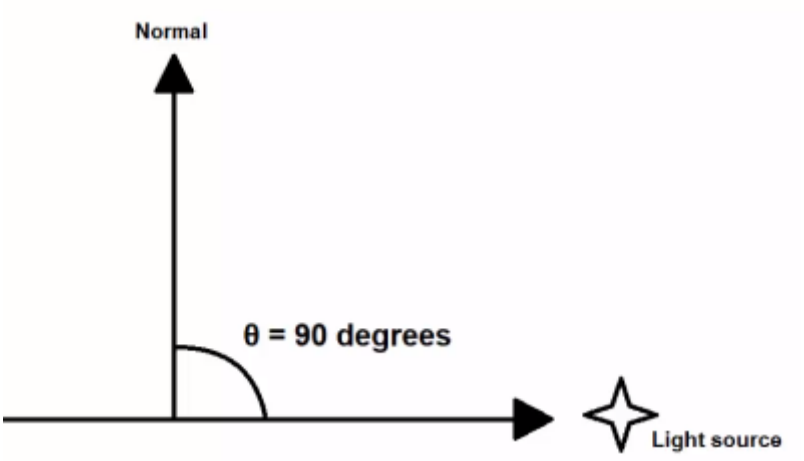
Wtedy: ***|v1|= |v2| = 1***

Zatem: ***v1 * v2 = cos(Φ)***

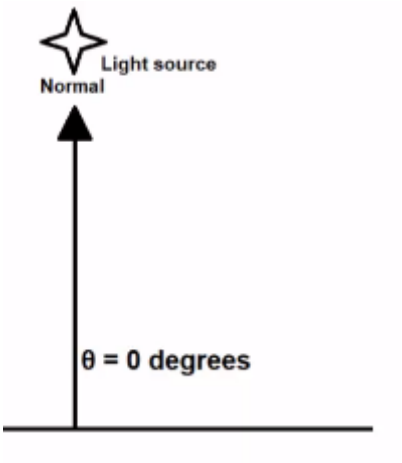
Od kiedy **cos(0 deg) = 1**, oraz **cos(90 deg) = 0**

Możemy użyć wartości wyjściowej z **v1 * v2** żeby zdeterminować **współczynnik rozproszenia** 😊.

Np.:



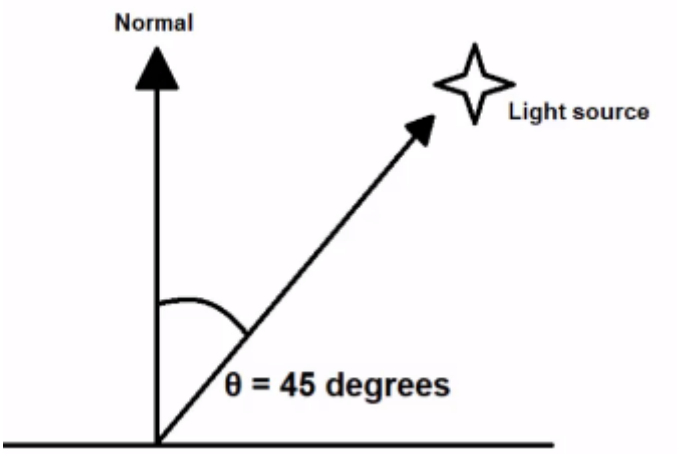
- Znormalizowane wektor normalny oraz światła.
- v1 * v2 = cos(**Φ**) = cos(90 deg) = 0
- Współczynnik rozproszenia wynosi 0 czyli zero rozproszenia światła.



Znormalizowany wektor normalny oraz światła.

v1 * v2 = cos(**Φ**) = cos(0 deg) = 1.

Współczynnik rozproszenia będzie wynosił 1 (100% rozproszenia światła).



Znormalizowany wektor normalny oraz światła.

v1 * v2 = cos(**Φ**) = cos(45 deg) = 0.71

$$v1 * v2 = \cos(\varphi) = \cos(45\ deg) = 0.71$$

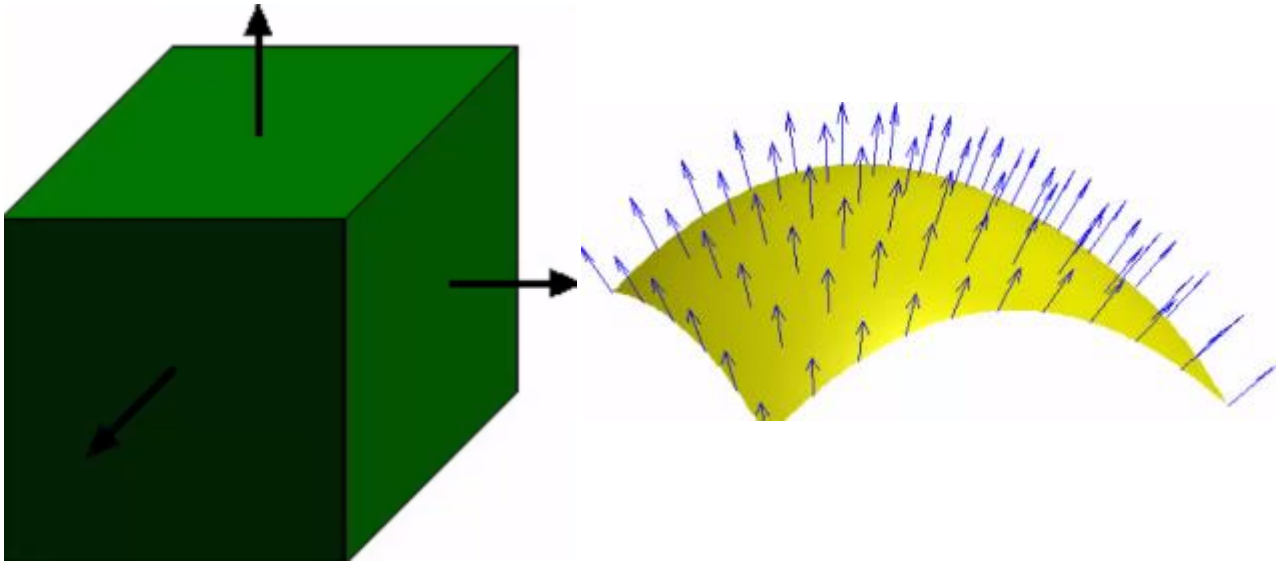
Współczynnik rozproszenia 0.71 czyli 71 oświetlenia rozproszonego.

- **Jeżeli współczynnik jest negatywny (mniej niż 0) to znaczy, że światło jest za powierzchnią wiec normalnie do zera.**
- **Aplikacja** współczynnika rozproszenia z otoczeniem:

fragColour = objectColour * (ambient + diffuse).

Normalne:

- Normalne są to wektory, które są prostopadłe do punktu na powierzchni.



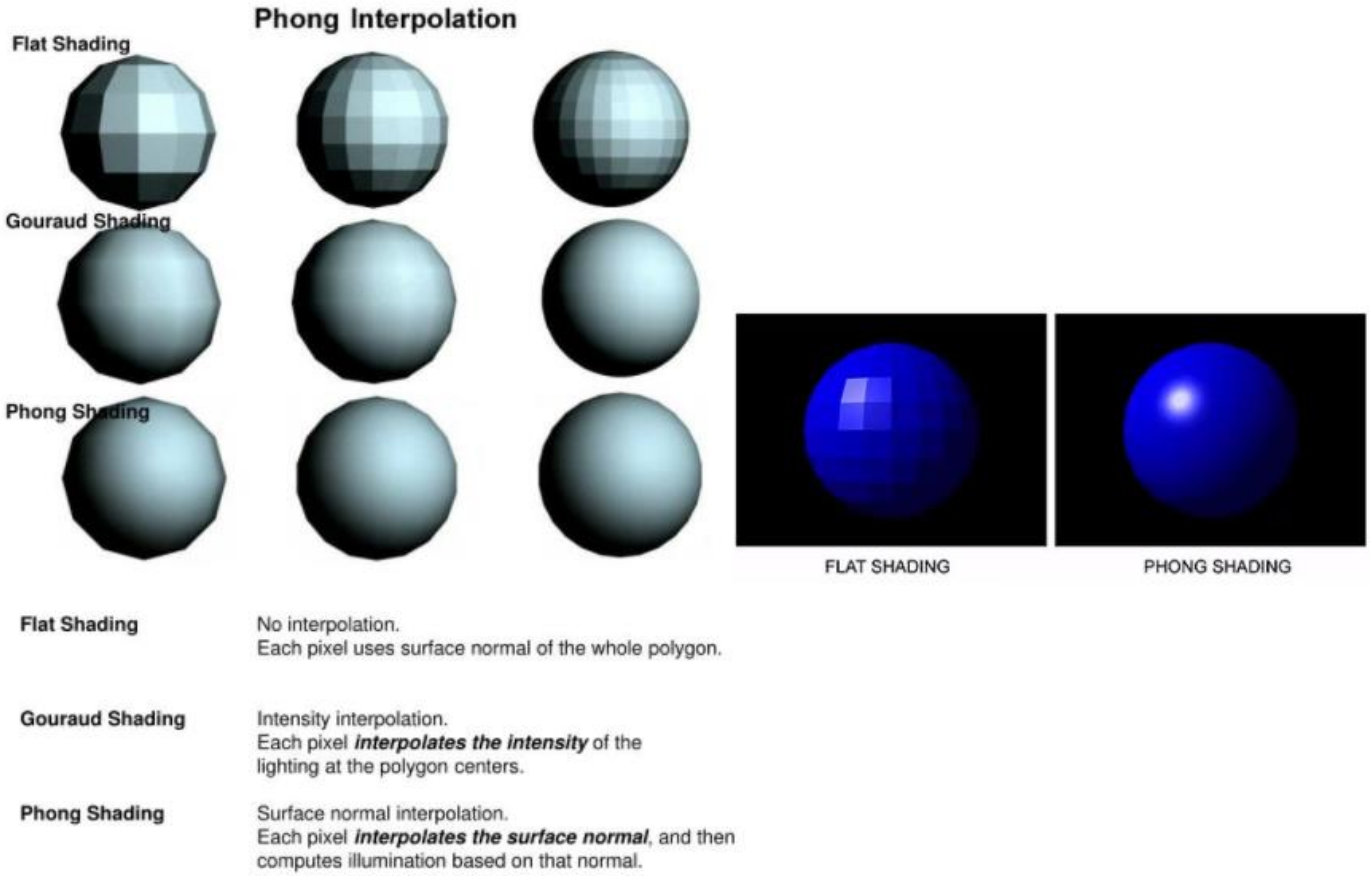
- Możemy je manualnie zdefiniować dla każdej strony.
- Dla każdego wierzchołka (Vertex shader) możemy zdefiniować wiele normalnych, jedna dla każdej strony, której jest częścią.
- Dobre dla „płaskiego cieniowania” (ang.: flat shading), nie dobre dla realistyczne gładkiego cieniowania.
- Również nie dobrze działa dla rysowania indeksowanego: Definiujemy tylko jeden wierzchołek na każdą stronę.

Alternatywa: Cieniowanie Phong (ang.: **Phong Shading**) (nie Phong Lighting). **Cieniowanie Phong jest to interpolacja** zaś **Oświetlanie Phong to jest nazwa całego modelu**.

Każdy wierzchołek ma średnią normalnych wszystkich powierzchni, których jest częścią.

Interpolacja pomiędzy tymi średnimi w shaderze do stworzenia gładkiego efektu.

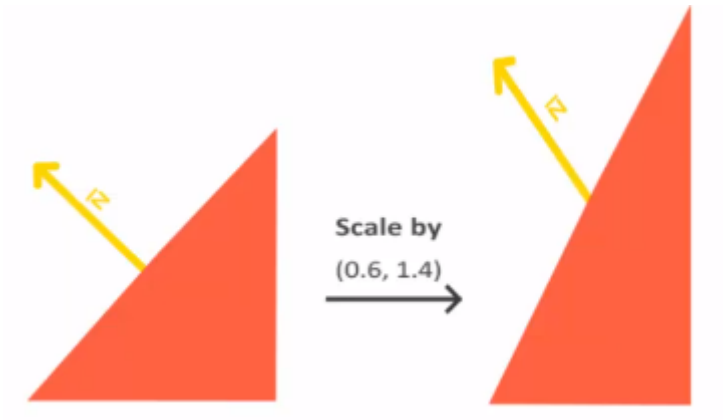
Dobre dla zaawansowanych modeli ale nie dla prostych z ostrymi krawędziami (jeżeli nie używasz dobrych technik modelowania).



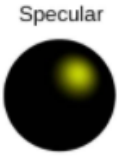
- Phong ścieniowane sfera jest tak samo zdefiniowana as płasko ścieniowana. Tylko w przypadku płaskiej każdy trójkąt ma swój normal, który nie jest średnią ważona z danej powierzchni.
- Gładkość jest iluzją stworzoną poprzez interpolację oraz efektywnie „faking" powierzchni normalne żeby wyglądała na zaokrągloną (ang.: curved).

Oświetlenie rozproszone – Normalne:

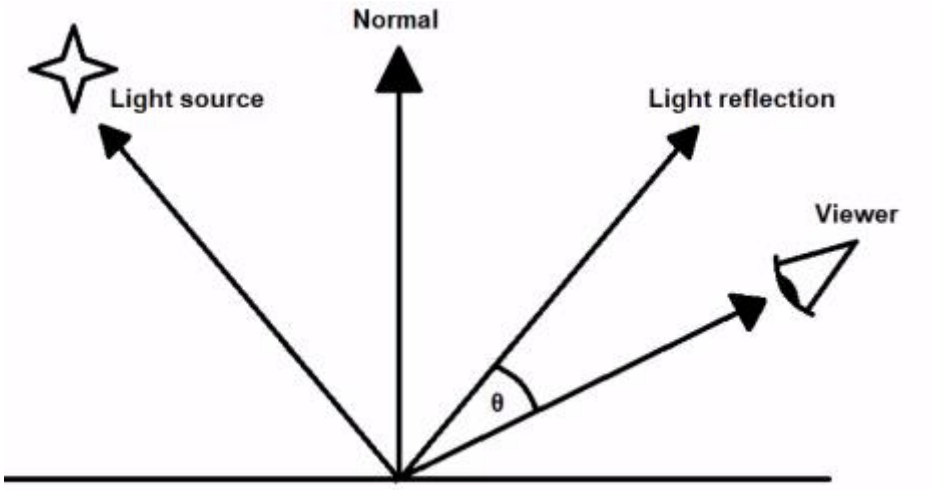
- Problem z non-uniform skalą.
- Źle przekrzyżowanie normale.
- Może być naprawione poprzez tworzenie „**normalnej macierzy**" **z macierzy modelu.**
- Transform normals z: ***mat3(transpose(inverse(model)))***
- Pełne objaśnienie:
  - lighthouse3d.com/tutorials/glsl-tutorial/the-normal-matrix/



- **Specular lighting** (światło odbicia/ odbicie lustrzane światła): Światło odbite bezpośrednio ze źródła do oka oglądacza. Efektywnie odbicie światła. Bardziej widoczne na błyszczącym przedmiocie.



  - Uwzględnia pozycje oglądacza.
  - Jest to bezpośrednia odbicie źródła światła, które trafia w oczy oglądacza.
  - Chodzenie wokół sprawi efekt pozornej pozycji odbicia lustrzanego na powierzchni.
  - **Żeby to zrobić potrzebujemy cztery rzeczy:**
    - Wektor **ś**wiatła,
    - Wektor **n**ormalny,
    - Wektor **o**dbicia (Wektor światła odbity wokół normalnego).
    - Wektor **w**idoku (Wektor od oglądacza do fragmentu).



- Jest potrzebny kąt pomiędzy oglądaczem oraz odbiciem/ refleksją.
- **Mniejszy** kąt = **Więcej światła.**
- **Większy** kąt = **Mniej światła.**

Wektor widoku (ang.: Viewer/ Camera) ~ jest to po prostu **różnica pomiędzy pozycją fragmentu oraz oglądaczem** (kamery).

Wektor odbicia (ang.: Light Reflection) ~ może być uzyskany z wbudowanej w GLSL funkcji: ***reflect(incident, normal)***

- Incident ~ wektor do obicia (light source)
- Normal ~ Normal wektor do odbicia wokół (normal)

Tak jak ze światłem rozproszonym, używamy **dot produkt pomiędzy znormalizowanymi formą wektora widoku** oraz **wektora odbicia**, żeby **uzyskać współczynnik światła odbicia** (**ang.: specular factor**).

współczynnik odbicia = dot product pomiędzy miedzy wektorem odbicia oraz widoku/ kamery i z tego uzyskujemy kąt.

$$specularFactor = view \bullet reflection$$

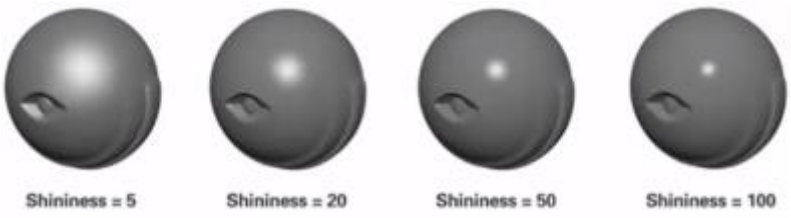
Wszystko to daje nam łącznie model oświetlenia phong.

Odbicie Phong jest to kolejny termin do określenia modelu oświetlenia phong.

Lustrzane odbicie (ang.: Specular Lighting) ~ Jasność (ang.: Shininess):

- Ostatni krok żeby zmienić współczynnik odbicia jest: Jasność.
- Jasność tworzy bardziej dokładne odbicie.
- Wyższa jasność: Mniej gęste odbicie.
- Mniejsza jasność: Większe, ściemnione odbicie.
- Po prostu wcześniej obliczone współczynnik odbicia do potęgi wartości jasności.
- Drewno będzie miało mały współczynnik zaś metal duże.

$$specularFactor = (view \bullet reflection)^{shininess}$$

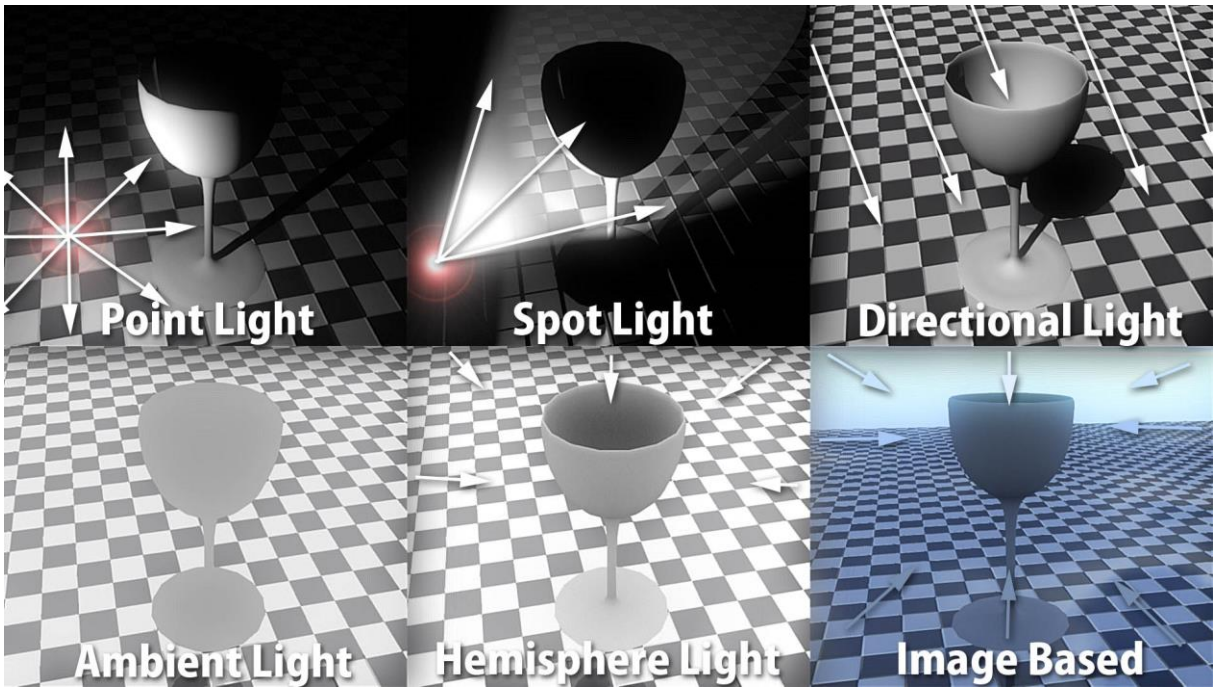


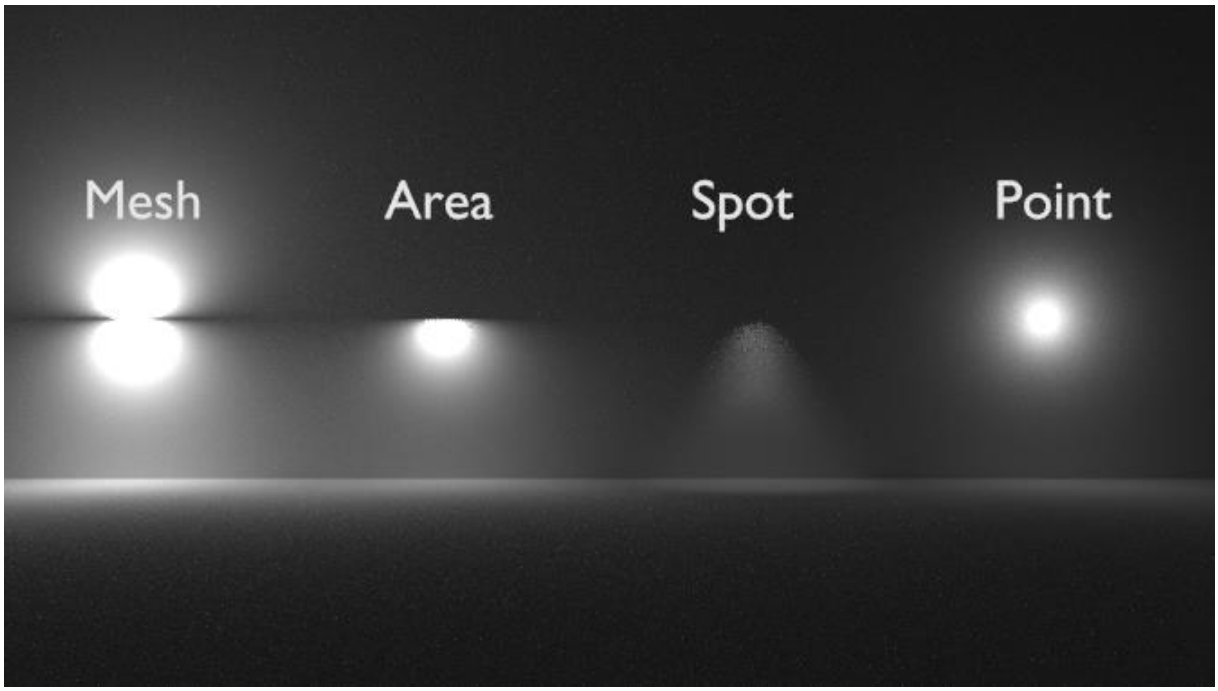
Czyli:

Żeby zaaplikować współczynnik odbicia (specular) z otoczeniem (ambient) oraz rozproszeniem (diffuse):

$$fragColour = objectColour * (ambient + diffuse + specular)$$

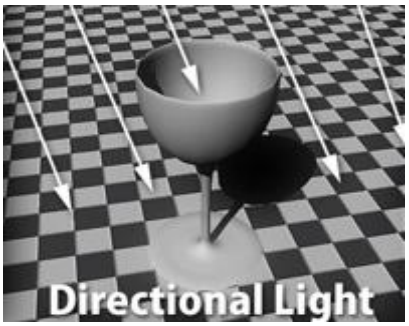
Ale skąd pochodzi światło?



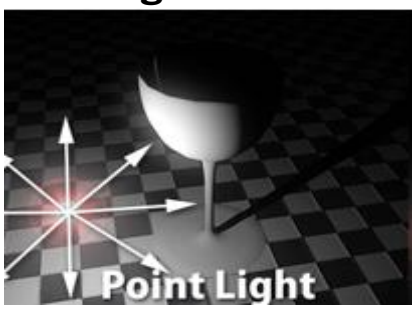


Rodzaje świateł:

- **Światło kierunkowe** (ang.: directional light): Światło bez pozycji oraz pochodzenia. Całe światło nadchodzi jako równoległe promienie z widocznie nieskończonego dystansu. Na przykład **słońce**.



  - Najprostsza forma światła.
  - Wymaga tylko podstawowych informacji (kolor, współczynnika otoczenia, współczynnika rozproszenia, współczynnika odbicia) oraz kierunku.
  - Wszystkie obliczenia wykorzystując **ten sam kierunek wektora światła** (**ang.: light vector**).
  - Nie potrzeba obliczać wektora światła!
- **Światło punktowe** (ang.: point light): Światło z pozycją, która świeci we wszystkie kierunki.
  Na przykład żarówka lub **ognisko**.



- **Reflektor** (ang.: Spot light): Światło podobne do światła punktowego ale skrócone do emitowania w określonym zasięgu pod określonym kątem.
  Na przykład **latarka**.

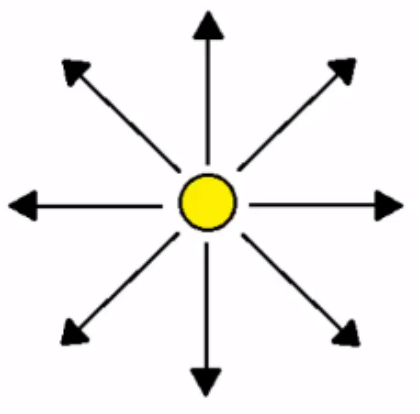
- **Światło obszarowe** (ang.: Area light): Emituje światło z powierzchni. Na przykład **panel świecący ze ściany** lub **sufitu**.

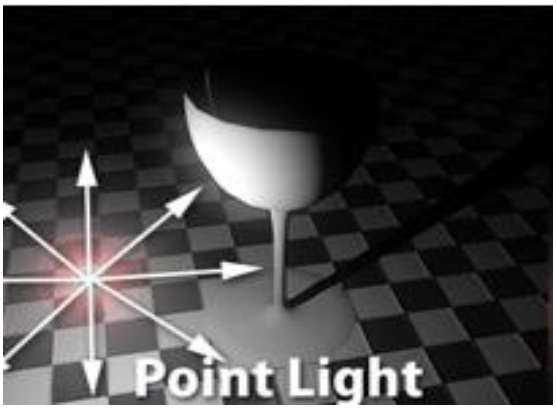
Podsumowanie:

- Phong model oświetlenia **składa się z współczynnika otoczenia**, **rozproszenia** oraz **odbicia światła**.
- Rozproszenie oraz odbicie wymagają normalnych wektorów.
- Używamy **dot produkt** oraz normalnych żeby zdeterminować **światło rozproszenia** (ang.: diffuse lighting).
- Używamy **dot produkt** ora odbitego światła (ang.: light reflected) wokół normalnych żeby zdeterminować **światło odbicia lustrzanego** (ang.: specular lighting).
- Cieniowanie Phong (ang.: Phong Shading) wykonuje interpolacje średnich wektorów normalnych żeby stworzyć gładkie/ nieostre powierzchnie w przeciwieństwie do płaskiego cieniowania (ang.: Flat Shading).
- Wyróżniamy główne cztery typy oświetlenia: **kierunkowe** (ang.: Directional Light ~ **słońce**), **punktowe** (ang.: Point light ~ **ognisko**), **reflektorowe** (ang.: Spot Light ~ **latarka**) oraz **obszarowe** (ang.: Area Light ~ **panele**).
- Światło kierunkowe jest najprostsze, ponieważ wymaga tylko kierunku oraz pozwala nam obliczyć wektora kierunkowego światła!

14.09.2021: Światła

1. **Światła punktowe** (ang.: Point Lights):

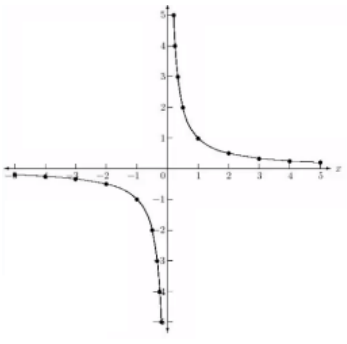




- Światła z pozycją, które emitują we **wszystkie** kierunki.
- Należy zdeterminować wektor kierunku manualnie/ dynamicznie poprzez uzyskanie różnicy pomiędzy pozycją światła oraz fragmentu.
- Zastosuj matematykę światła kierunkowego aby obliczyć wektor kierunku.

Współczynnik tłumienia (ang.: Attenuation):

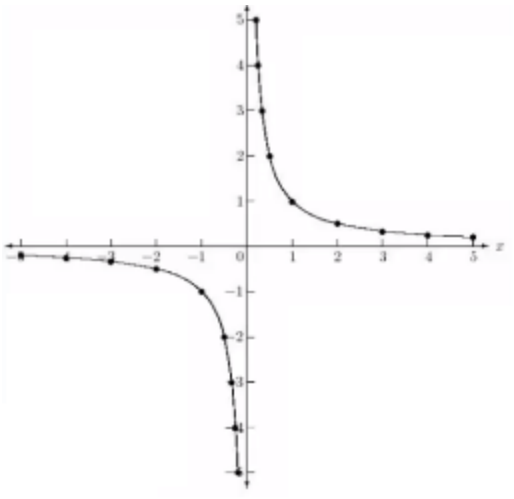
- Światła kierunkowe symulują nieskończony dystans, więc dystans nie wpływa na siłę światła.
- Światła punktowe moją pozycję, dystans z punktu, który świeci zmienia moc oświetlenia ~ dystans wpływa na moc oświetlenia obiektu przez światło punktowe.
- Jedna możliwe rozwiązanie: Liniowe opadanie.
- Miej moc oświetlenia opadającą w bezpośredniej proporcji z odległości do źródła światła.
- Proste ale nie realistyczne.
- W rzeczywistości, moc oświetlenia inicjalnie opada szybciej z dystansem.
- Im dalej jesteśmy, tym dłużej opada/ zmniejsza się.
- Dla dodatnich wartości, odwrotność funkcji kwadratowej może dać nam ten efekt.



$$f(x) = \frac{1}{ax^2+bx+c}$$

gdzie:

x ~ jest do dystans pomiędzy źródłem światła oraz fragment.



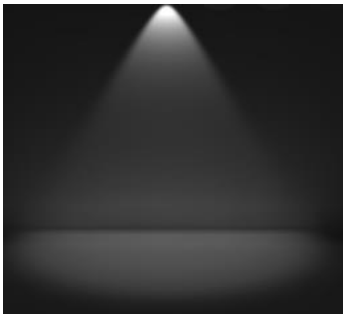
Współczynnik tłumienia

$$= \frac{1.0}{quadratic * distance^2 + linear * distance + constant}$$

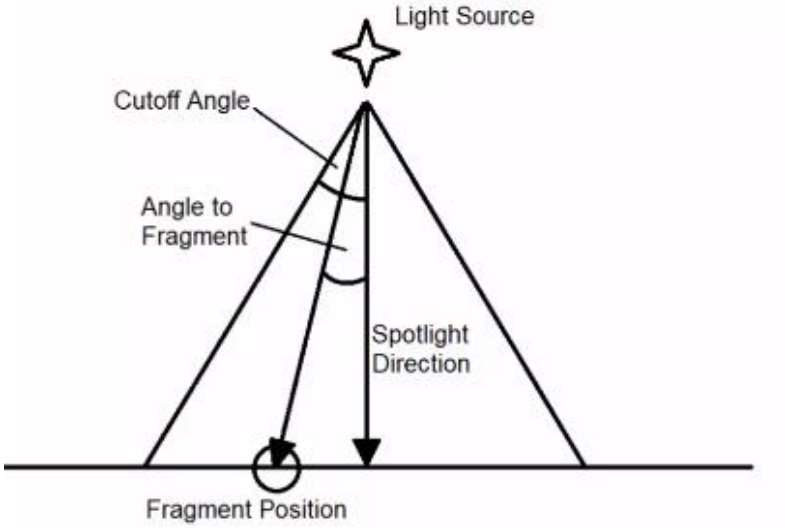
$$\text{Attenuation Factor} = \frac{1.0}{\text{quadratic} * \text{distance}^2 + \text{linear} * \text{distance} + \text{constant}}$$

- **distance** (pol.: dystans) ~ Odległość pomiędzy światłem oraz fragmentem.
- **quadratic** (pol.: kwadratowy) ~ Wartość zdefiniowana przez użytkownika, zazwyczaj najniższa z trzech.
- **linear** (pol.: liniowy) ~ Wartość zdefiniowana przez użytkownika, mniejsza niż stała (ang.: constant).
- **constant** (pol.: stała) ~ Zazwyczaj 1.0 żeby zapewnić, że mianownik zawsze jest większy niż 1. Na przykład jeśli mianownik jest 0.5 to 1.0/0.5 = 2.0, zatem współczynnik tłumienia będzie miał dwa razy większą moc świecenia poza wartością ustawioną.
- **Przydatne wartości:** wiki.ogre3d.org/tiki-index.php?page=-Point+Light+Attenauation.
- **Aplikacja: Zastosuj współczynnik tłumienia do otoczenia, rozproszenia oraz odbicia.**

2. Światła reflektorowe (ang.: Spot Lights):



- Działa podobnie jak światła Punktowe w teorii.
- Mają pozycję, używają współczynnika tłumienia, itp…
- Również mają **kierunek oraz odcięty kąt.**
- **Kierunek** (ang.: direction) **~ Tam gdzie światła jest skierowane** (ang.: facing).
- **Odcięty kąt** (ang.: Cut-off angle): ~ Kąt opisuję krawędzie światła z wektora kierunku.



- Potrzebujemy sposobu żeby móc porównać „Kąt do fragmentu” (ang.: **Angle to Fragment**) do kąta odcięcia (ang.: **Cut off Angle**).
- Do tego użyjemy **Dot Product** znowu.

$$angleToFragment = lightVector * lightDirection$$

gdzie:

lightVector (wektor światła) ~ Wektor od światła do fragmentu.

lightDirection (wektor kierunku) ~ Kierunek, które oświetlenie reflektorowe jest skierowane.

- Zatem **angleToFragment** (**kąt do fragmentu**) będzie miał wartość pomiędzy 0 oraz 1, reprezentując wartość pomiędzy dwoma.
- Po prostu cos(cutOffAngle) dla Cut Off kąta (kąta odcięcia).
- Im większa wartość: Mniejszy kąt.
- Mniejsza wartość: Większy kąt.
- Jeżeli wartość kąta do fragmentu jest większa od cos(cutOffAngle), to ze spotem: Aplikujemy światło.
- Jeżeli wartość kąta do fragmentu jest mniejsza, ma większy kąt niż odcięcia: Nie aplikujemy światła.

Gładkie Krawędzie:

- Obecne podejście da nam ostry kąt odcięcia na krawędziach reflektora.
- Tworzy nierealistyczny reflektor.
- Potrzebujemy sposób do łagodzenia kiedy dochodzimy do krawędzi do zasięgu kąta odcięcia.
- Użyjemy wynik wcześniejszego działania dot product jako współczynnik.
- Problem: W związku wybranym zasięgiem, dot product nie będzie skalował dobrze.

Na przykład: Jeśli kąt odcięcia jest 10 deg.:

  - Minimum dot product jest cos(10 deg) = 0.98
  - Zasięg dot product będzie w 0.98 – 1.00
  - Używając dot product do złagodzenia krawędzi będzie niezauważalne.
- Rozwiązanie: Skaluj zasięg dot produkt do 0 – 1.

Formuła do skalowania pomiędzy zasięgami:

$$newValue = \frac{(newRangeMax - newRangeMin)(originalValue - originalRangeMin)}{originalRangeMax - originalRangeMin}$$

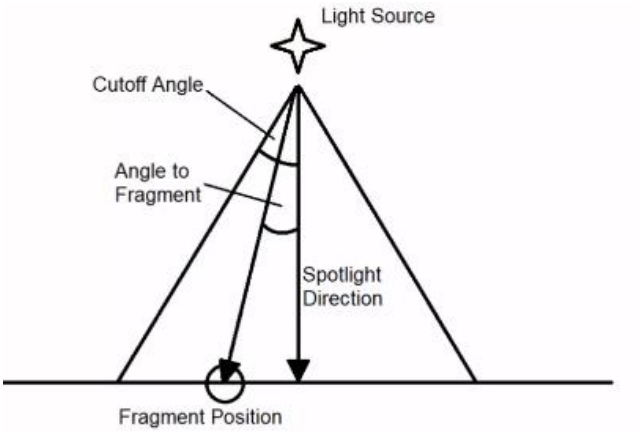
- **newRangeMin** jest 0, **newRangeMax** jest 1, zatem licznik jest **originalValue – originalRangeMin**.
- **originalRangeMax** jest 1.

- **Zatem** po kilku zamianach min oraz max wartości:

$$spotLightFade = 1 - \frac{1 - angleToFragment}{1 - cutOffAngle}$$

- O wiele prostsze!
- Po obliczeniu Oświetlenia Reflektorowego (ang.: Spot Light lighting).
- Pomnożony przez **spotLightFade** efekt.

Uzyskujemy:



1. $angleToFragment = lightVector \bullet lightDirection$
2. $spotLightFade = 1 - \frac{1 - angleToFragment}{1 - cutOffAngle}$
3. $colour = spotLightColour * spotLightFade$

Podsumowanie:

- Światła punktowe emitują światło we wszystkich kierunkach.
- Używamy algorytm światła kierunkowego z wektorem światła.
- Zanikanie światła w związku z dystansem za pomocą wartości współczynnika.
- Światła reflektorowe są światłami punktowymi z kierunkiem oraz zasięgiem odcięcia.
- Porównywanie kąta wektora światła z kątem odcięcia.
- Łagodniejsze krawędzi z/ze/dzięki skalowaną formą kąta światła wektora.