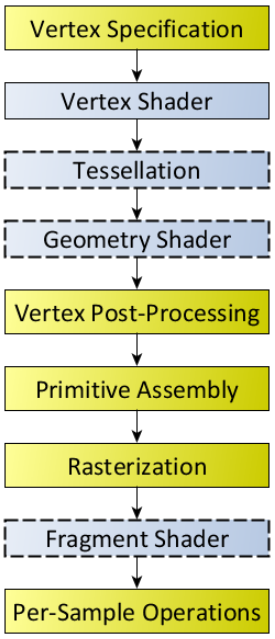
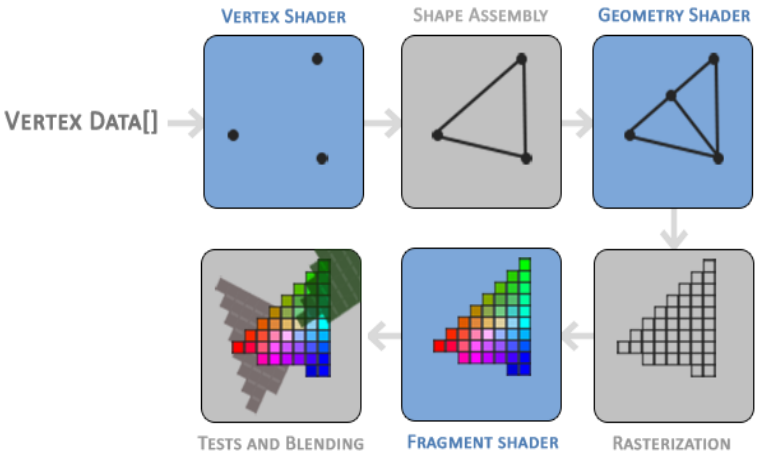
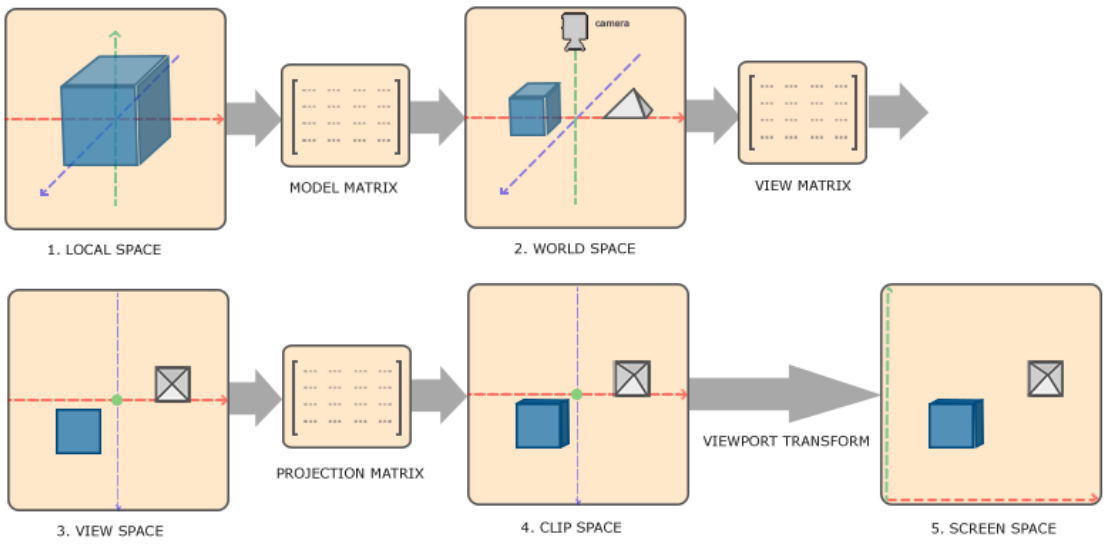
1. Cele kursu:

- 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):

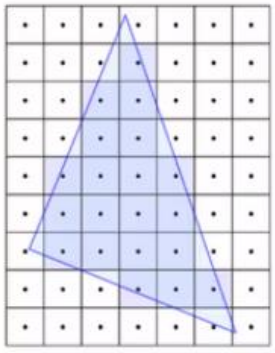
- Prymitywy, które nie są widziane są usuwane (nie chcemy rysować rzeczy, których nie widać).
- **Pozycje przekonwertowane z *przestrzeni obkrajania* („clip space”) do *przestrzeni okna* („window space”).**

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.

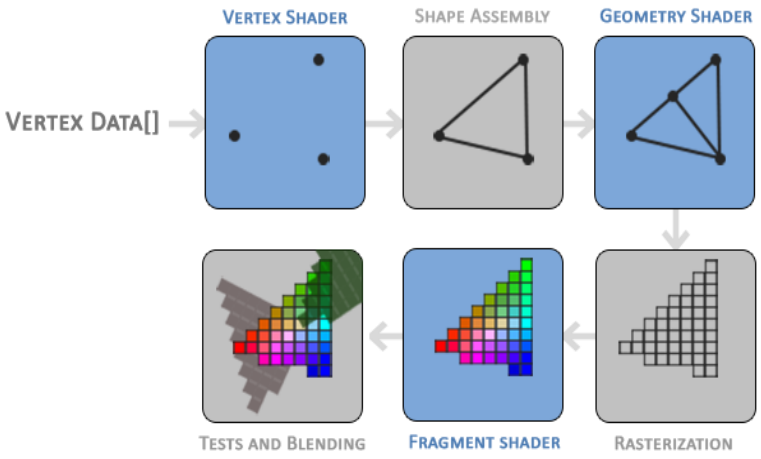
Oryginalny widzi użytkownik.

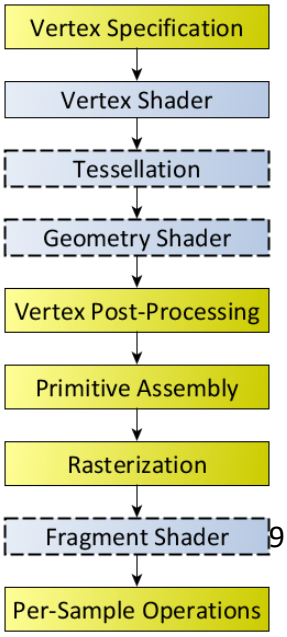
Na koniec oryginalny jest zamieniany z framebuffer, stary framebuffer staje się oryginalnym a stary oryginalny staje się nowym framebufferem

Na zakończenie zamieniamy oryginalny na framebuffer.

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

- **Strumień renderowania jest zakończony 😊.**





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.
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).