Uczenie maszynowe w animacjach

Wykład nr 2

Podstawy współczesnej grafiki komputerowej – jak to działa?

Szymon Datko

szymon.datko@pwr.edu.pl

Wydział Informatyki i Telekomunikacji,
Politechnika Wrocławska

semestr letni 2021/2022

Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Rozwoju Regionalnego Program Operacyjny Polska Cyfrowa na lata 2014-2020,
Oś Priorytetowa nr 3 "Cyfrowe kompetencje społeczeństwa" Działanie nr 3.2 "Innowacyjne rozwiązania na rzecz aktywizacji cyfrowej"
Tytuł projektu: „Akademia Innowacyjnych Zastosowań Technologii Cyfrowych (AI Tech)"
Porozumienie nr POPC.03.02.00-00-0001/20-00

Czym jest OpenGL?

- Zapoczątkowany przez firmę Silicon Graphics Inc. (SGI):
  - producenta wysokowydajnych stacji graficznych,
  - początkowo jako ich własnościowy system IRIS GL,
    - *Integrated Raster Imaging System Graphics Library*,
  - ostatecznie udostępniony jako **otwarty standard**.
- Formalnie, wbrew nazwie, nie jest to biblioteka:
  - Interfejs Programowania Aplikacji (ang. *API*),
  - **specyfikacja** w jaki sposób mają być realizowane instrukcje,
  - odpowiednią implementację dostarczają **sterowniki** sprzętu.
- Obecnie nad rozwojem standardu czuwa Khronos group,
  - konkretnie **OpenGL Architectural Review Board** (ARB).

Wersje OpenGL

- **OpenGL 1.0** – I 1992,
- OpenGL 1.1 – I 1997,
- OpenGL 1.2 – III 1998,
- OpenGL 1.2.1 – X 1998,
- OpenGL 1.3 – VIII 2001,
- OpenGL 1.4 – VII 2002,
- OpenGL 1.5 – VII 2003,
- **OpenGL 2.0** – IX 2004,
- OpenGL 2.1 – VII 2006,
- **OpenGL 3.0** – VIII 2008,
- OpenGL 3.1 – III 2009,
- OpenGL 3.2 – VIII 2009,
- **OpenGL 3.3** – III 2010,
- **OpenGL 4.0** – III 2010,
- OpenGL 4.1 – VII 2010,
- OpenGL 4.2 – VIII 2011,
- OpenGL 4.3 – VIII 2012,
- OpenGL 4.4 – VII 2013,
- OpenGL 4.5 – VIII 2014,
- **OpenGL 4.6** – II 2017,

Warto odwiedzić: https://www.khronos.org/opengl/wiki/History_of_OpenGL

Rozszerzenia i rodzaje specyfikacji

Kolejne wersje OpenGL definiują wymagane funkcje, jakie muszą zostać zapewnione przez producentów sprzętu.

Niektórzy producenci sprzętu oferują własne zaawansowane funkcje w ramach standardu OpenGL. Są one dostępne jako **rozszerzenia**.
Z czasem mogą one zostać włączone do oficjalnej specyfikacji.
Wyróżnia się rozszerzenia: **dostawców**, **EXT** i **ARB**.
Przykład: **ARB_get_program_binary**.

W 2008 roku specyfikacja OpenGL została podzielona na:

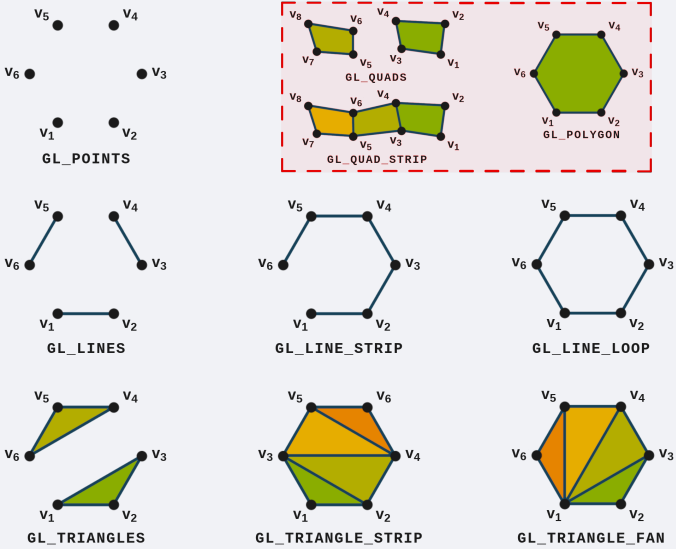
- **profil główny** (ang. *core profile*),
- **profil zgodności** (ang. *compatibility profile*).

Podstawowe pojęcia

- **Wierzchołek** – punkt w przestrzeni, używany w prymitywach.
- **Prymityw** – podstawowa jednostka renderingu w OpenGL,
  - obecnie to między innymi: punkt, linia i trójkąt;
  - dawniej także: czworokąt i wielobok.
- **Renderowanie** – proces budowania obrazu w komputerze.
- **Rasteryzacja** – przekształcenie prymitywów w zbiór pikseli[1].
- **Piksel** – najmniejsza jednostka wizualna na wyświetlaczu.
- **Ramka** – zbiór pikseli, pojedynczy pełen obraz.
- **Teselacja** – podział złożonego obiektu na zbiór prymitywów.
- **Shader** – program wykonywany przez sprzęt graficzny.

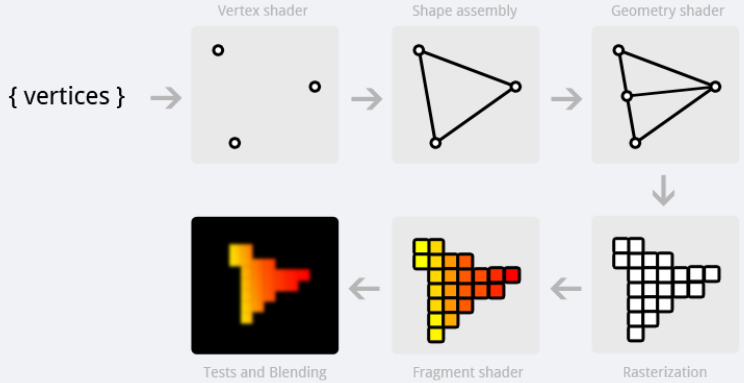
[1] Formalnie w OpenGLu mówimy tutaj o **fragmentach**: https://www.khronos.org/opengl/wiki/Fragment

Rodzaje prymitywów



Powstawanie grafiki w komputerze

To wszystko to po prostu matematyka! ;-)



Źródło obrazka: https://open.gl/drawing

Potok graficzny

Etapy przetwarzania w aktualnej wersji OpenGL:

- Pobranie wierzchołków.
- **Shader wierzchołków**.
- **Shader sterowania teselacją**.
- Teselacja.
- **Shader wyliczenia teselacji**.
- **Shader geometrii**.
- Składanie prymitywów.
- Rasteryzacja.
- **Shader fragmentów**.
- Działania na buforze ramki.

Pogrubione elementy listy reprezentują etapy **programowalne**.

Shadery

- Programy uruchamiane równolegle na procesorze graficznym.
- Zwykle definiuje się je przy pomocy języka **GLSL**.
  - Inne języki mogą być dostępne jako rozszerzenia.
  - Wersja OpenGL 4.6 dopuściła także język **SPIR-V**.
- Obowiązują zasady podobne, jak przy typowych programach:
  - kod źródłowy umieszcza się w **obiekcie shadera** i kompiluje,
  - skompilowane kody łączy się w **obiekt programu** (linking),
  - domyślnie cały ten proces odbywa się w locie.
- Minimalny użyteczny potok musi zawierać:
  - shader wierzchołków lub shader obliczeniowy,
  - shader fragmentów[1].

[1] Tylko jeśli coś ma zostać wyświetlone na ekranie.

Rodzaje shaderów (1/2)

- **Shader wierzchołków**:
  - uruchamiany dla każdego wierzchołka wejściowego,
  - zazwyczaj służy do transformowania położenia wierzchołków.
- **Shader sterowania teselacją** / shader sterujący:
  - przyjmuje dane z shadera wierzchołków,
  - określa poziomy podziałów dla mechanizmu teselacji,
  - wytwarza nowy zbiór wierzchołków i współczynniki teselacji.
- **Shader wyliczenia teselacji** / shader wyliczenia:
  - zostaje uruchomiony dla każdego powstałego wierzchołka,
  - pozwala określić docelową pozycję tych wierzchołków.

Rodzaje shaderów (2/2)

- **Shader geometrii**:
  - uruchamiany raz dla każdego prymitywu,
  - pozwala tworzyć nowe prymitywy i zmieniać istniejące.
- **Shader fragmentów**:
  - uruchamiany dla każdego fragmentu (wyniku rasteryzacji),
  - stosowany do określenia wynikowego koloru piksela.
- **Shader obliczeniowy**:
  - element specjalnego, niezależnego potoku,
  - nie ma określonych wejść, wyjść, ani miejsca,
  - zazwyczaj stosowany do zadań niezwiązanych z rysowaniem.

Język GLSL (1/2)

- *OpenGL Shading Language*.
- Język o składni i działaniu podobnym do języka **C**.
- Dostosowany do potrzeb Grafiki Komputerowej:
  - wbudowane wektorowe i macierzowe typy danych,
    - elementy dostępne jak w zwykłej tablicy,
    - można odwoływać się także jak do pól w strukturach,
    - nazwy pól można sklejać, aby wydobyć nowy wektor,
    - kolejność i powtórzenia pól – bez znaczenia,
    - `vec4 color; return color.grb; // zwraca vec3`
  - wbudowane funkcje matematyczne i pomocnicze,
  - ...

Język GLSL (2/2)

- Dostosowany do potrzeb Grafiki Komputerowej:
  - ...
  - zaprojektowany na potrzeby wysokiego zrównoleglenia,
  - brak rekurencji i ograniczona różnorodność typów,
    - 32- i 64-bitowe liczby zmiennoprzecinkowe,
    - 32 bitowe całkowite (ze znakiem i bez),
    - wartości logiczne.
  - struktury definiuje się bezpośrednio je tworząc,
    - w języku nie ma słowa kluczowego `typedef`,
  - tablice mają wbudowaną metodę `length()`,
  - rozmiar tablicy można zapisać obok jej typu:
    - `float[5] var = float[5](1.0, 2.0, 3.0, 4.0, 5.0);`

Język GLSL – przykład shadera wierzchołków

```
#version 330 core

uniform mat4 mvp;
uniform float offset;

out MY_BLOCK {
    vec2 tc;
} vs_out;

void main(void) {
    const vec2[4] position = vec2[4](
        vec2(-0.5, -0.5),
        vec2( 0.5, -0.5),
        vec2(-0.5,  0.5),
        vec2( 0.5,  0.5)
    );

    vs_out.tc = (position[gl_VertexID].xy + vec2(offset, 0.5))
                * vec2(30.0, 1.0);

    gl_Position = mvp * vec4(position[gl_VertexID], 0.0, 1.0);
}
```

Dane w OpenGL (1/2)

- Część danych można zapisać bezpośrednio w shaderach,
  - nie są problemem także wyliczenia wewnątrz shaderów,
  - jest to jednak rozwiązanie dosyć ograniczone.
- Najczęściej dane do shaderów przekazuje się z aplikacji.
  - W tym celu wykorzystuje się **bufory** oraz **tekstury**.
- **Bufory** stanowią ciągły fragment zaalokowanej pamięci.
  - Najpierw należy zadeklarować **nazwę**, jako odnośnik bufora.
  - Następnie utworzyć **magazyn danych** – zarezerwować pamięć.
  - Później należy zapisać w buforze dane, np. przez **mapowanie**.
  - Dalej następuje **dowiązanie** bufora do kontekstu OpenGL.
  - Miejsce dowiązania określa się fachowo jako **cel** (*target*), który opisuje w jaki sposób dane z bufora będą wykorzystane.

Dane w OpenGL (2/2)

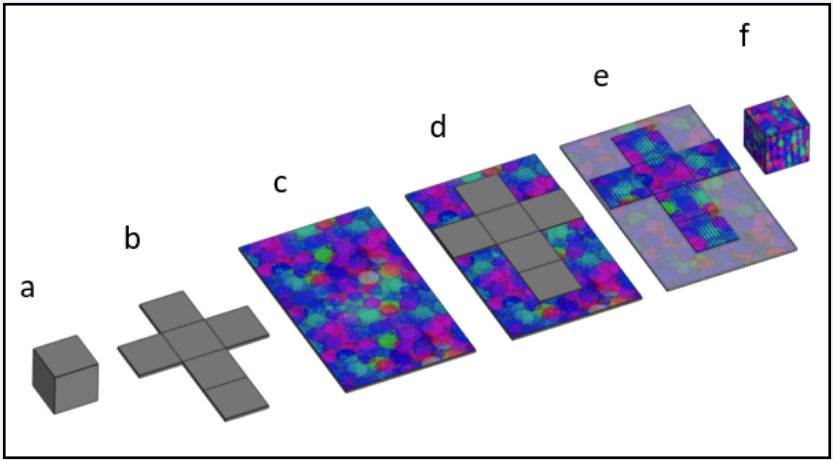
- Dane można przekazać bezpośrednio do shadera wierzchołków za pośrednictwem tak zwanego **Vertex Array Object** (*VAO*).
  - W ramach *VAO* określa się **artrybuty wierzchołków**.
  - Przechowuje referencje do obiektów bufora (*VBO*).
- Kolejne etapy potoku mogą otrzymać dane, jeśli zostaną one przekazane w odpowiednio dalej z shadera wierzchołków.
  - Stosuje się w tym celu słowa kluczowe `in` i `out` w GLSL.
  - Nazwy przekazywanych zmiennych lub bloków muszą być takie same w sąsiadujących shaderach w potoku.
- Alternatywnie można skorzystać z danych typu **uniform**.
  - Takie dane dostępne są od razu we wszystkich shaderach.
  - Umożliwiają jednak wyłącznie odczyt danych, bez ich zmian.

Tekstury

- Bufory ze zdefiniowanymi mapami kolorów.
- Intuicyjnie: obrazki rozciągane pomiędzy wierzchołkami.
- Mogą stanowić dane wejściowe, jak i miejsce na zapis wyniku prac.
- Najczęściej wykorzystywane w shaderze fragmentów, poprzez:
  - obiekty typu `uniform sampler` (odniesienie do bufora),
  - określenie koloru fragmentu przez współrzędne w teksturze.
- Podstawowe rodzaje tekstur:
  - 1-wymiarowe,
  - 2-wymiarowe,
  - 3-wymiarowe.
- Kiedyś ograniczone: kwadratowe, rozmiary boków jako potęgi liczby 2,
  - wciąż jednak lepiej jest trzymać się tych założeń co do tekstur.

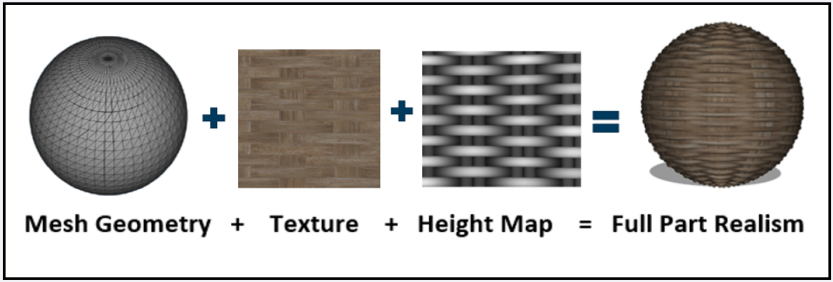
Teksturowanie – idea

- Do wierzchołków modelu mapuje się punkty z przestrzeni UV tekstury.



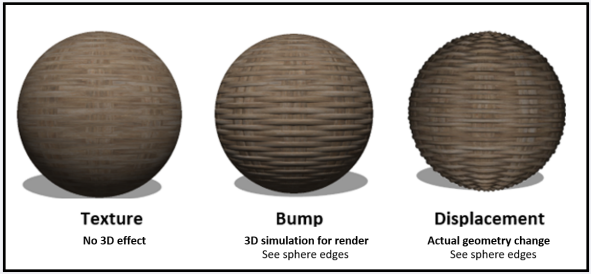
Źródło: https://grabcad.com/tutorials/texture-bump-and-displacement-how-to-make-photorealistic-models

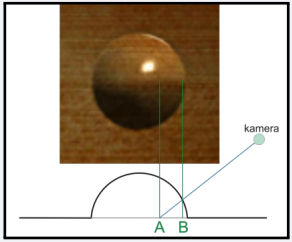
Tekstury to nie tylko mapy kolorów!

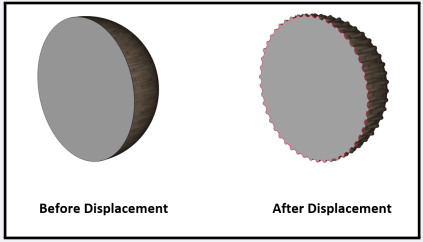


Źródło: https://grabcad.com/tutorials/texture-bump-and-displacement-how-to-make-photorealistic-models

Praca na fragmentach, a praca na geometrii



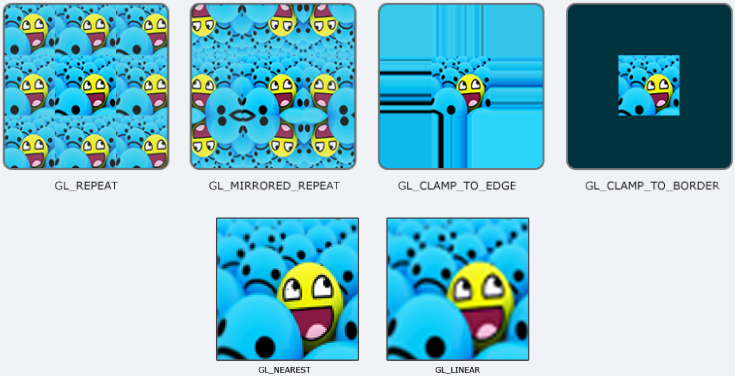




Źródło: Dr inż. Rafał Wcisło "Rendering czasu rzeczywistego" (AGH) [Mapowanie paralaksy],
https://grabcad.com/tutorials/texture-bump-and-displacement-how-to-make-photorealistic-models

Zawijanie i filtrowanie tekstur

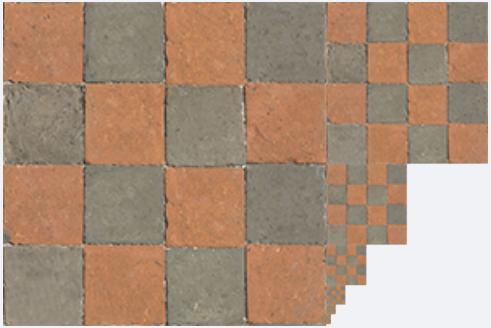
- Domyślnie każda tekstura, niezależnie od rozmiaru w pikselach, jest mapowana do układu współrzędnych jednostkowych.
- Wybrane wartości i zaokrąglenia mogą skutkować artefaktami.



Źródło: https://learnopengl.com/Getting-started/Textures

Mipmapy

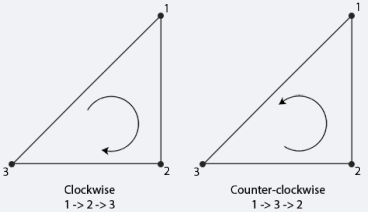
- Nawet przy zastosowaniu filtrowania, mogą się pojawić problemy przy rysowaniu małych, daleko odległych obiektów.
- Problemy zaokrągleń rozwiązuje użycie mniejszych tekstur.
- Mogą być wczytane, lub wygenerowane z oryginalnej tekstury.



Źródło: https://learnopengl.com/Getting-started/Textures

Usuwanie niewidocznych ścian

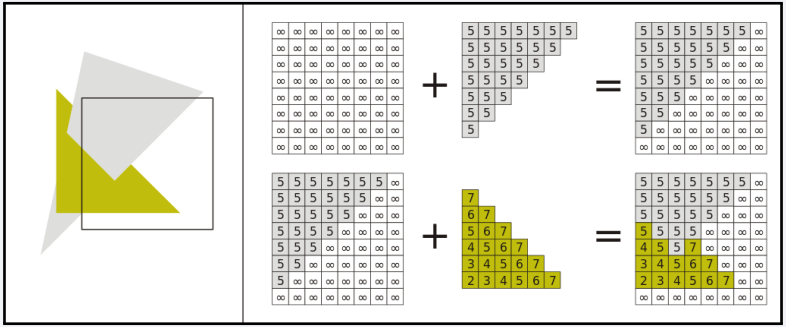
- Zwykle rysowane są wszystkie powierzchnie w bryle widzenia.
- Jedna strona tych powierzchni najczęściej nie jest widoczna.
- Tzw. *Face Culling* pozwala zaoszczędzić część obliczeń GPU.
- Jest to też pomocne przy rysowaniu cienkich obiektów.
- Działanie mechanizmu (to, co widać) jest konfigurowalne.
- Wykorzystuje się tutaj właściwość iloczynu wektorowego.



Źródło: https://learnopengl.com/Advanced-OpenGL/Face-culling

Słowo na temat mechanizmu bufora głębi

- Działa analogicznie, jak bufor koloru – przechowuje dane każdego piksela.
- Pozwala na uzyskanie poprawnego przesłaniania obiektów w przestrzeni.
- Kolejność rasteryzacji obiektów nieprzeźroczystych nie ma znaczenia.
- Wymaga dodatkowej pamięci graficznej do zaalokowania.



Źródło: https://pl.wikipedia.org/wiki/Bufor_Z

Wyświetlanie wielu obiektów

- Obecnie rysowanie następuje z chwilą wywołania funkcji **glDrawArrays()**.
- Aby narysować kilka kopii obiektu, wystarczy wywołać ją kilka razy.
- Żeby obiekty nie znajdowały się w tym samym miejscu należy dokonać zmian w macierzy transformacji – na przykład przesunąć kolejne kopie.
- Obciążamy CPU wywołaniami rysowania i obliczeniami, chociaż w pewnych szczególnych przypadkach możnaby tego uniknąć.
- Przykładowy kod:

```
1| def render(time):
2|     ...
3|
4|     for i in range(10):
5|         M_matrix = glm.translate(M_matrix, glm.vec3(1.0f, 0.0f, 0.0f));
6|         glUniformMatrix4fv(M_location, 1, GL_FALSE, glm.value_ptr(M_matrix))
7|
8|         glDrawArrays(GL_TRIANGLES, 0, 36);
```

Mechanizm rysowania instancjowego

- Pozwala wygenerować wiele kopii tego samego obiektu w bardziej efektywny sposób – bezpośrednio na samej karcie graficznej.
- Aby narysować 10 kopii obiektu, należy wykonać zmianę funkcji rysującej:

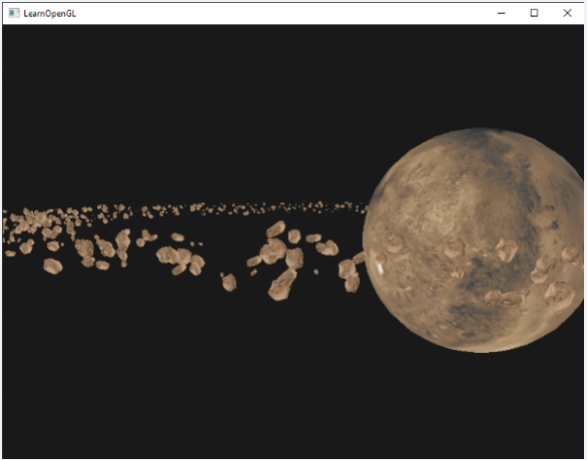
```
glDrawArrays(GL_TRIANGLES, 0, 36)
  ->
glDrawArraysInstanced(GL_TRIANGLES, 0, 36, 10)
            ^^^^^^^^^                   ^^^^
```

- Nowa zmienna **gl_InstanceID**, która numeruje instancje rysowanego obiektu, podobnie jak **gl_VertexID** numeruje instancje shadera.
- Transformację należy uwzględnić bezpośrednio w kodzie shadera,

```
gl_Position = P_matrix * V_matrix * M_matrix * position;
  ->
gl_Position = P_matrix * V_matrix * M_matrix * (
    positon + gl_InstanceID * vec4(1, 0, 0, 0)
);
```

Mechanizm rysowania instancjowego – przykład

- Ostatecznie tworzone obiekty nie muszą być całkowicie identyczne!
- Obecnie dobrą alternatywą jest także wykorzystanie shadera geometrii.



Źródło obrazka: https://learnopengl.com/Advanced-OpenGL/Instancing

Podsumowanie, przyszłość i alternatywy

- Każde wywołanie `glVertex()`, `glColor()`, itd. angażowało procesor.
  - Wąskie gardło w momencie, kiedy obsługujemy złożoną scenę.
- Efektywność w grafice wymaga zrównoleglenia operacji przetwarzania – shadery.
- Jeszcze kolejny krok: podejście **Zero Driver Overhead**.
  - Zmniejszanie narzutu związanego z implementacjami, dostarczanymi w ramach sterowników przez producentów podzespołów graficznych.
- Skutki zmienionej koncepcji potoku graficznego:
  - Większe możliwości i wydajność przetwarzania.
  - Wyższy próg wejścia i złożoność implementacji.
- Alternatywne technologie:
  - **DirectX** – Microsoft.
    - Dostarcza także funkcje multimedialne.
  - **Metal** – Apple.
  - **Vulkan** – Khronos group.

To wszystko na dziś.

Do zobaczenia!