Zachodniopomorski
Uniwersytet
Technologiczny
w Szczecinie

http://bazyluk.net/dydaktyka

Bartosz Bazyluk

Oświetlenie

Zaawansowane techniki oświetlenia, c.d.

Wydział
Informatyki

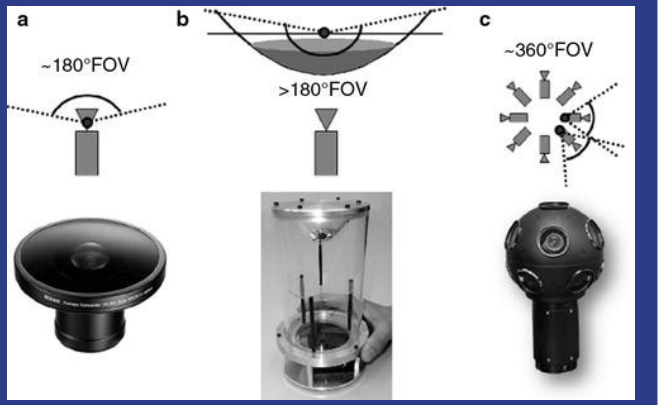
Grafika Komputerowa i Wizualizacja, Informatyka S1, II Rok

Źródło obrazu:
Panocapture.com, Automotive Pack

Źródło obrazu:
Panocapture.com, Automotive Pack

Image Based Lighting (IBL)

- Technika oświetlenia wykorzystująca **mapę środowiska** rzutowaną na powierzchnię oświetlanego obiektu
  - Najczęściej jest to ***omnikierunkowe* zdjęcie**, zwykle *HDR*



Pozyskanie **mapy środowiska** wymaga ultraszerokokątnego obiektywu lub specjalnego rodzaju kamery.

Źródła obrazów:
Panocapture.com, Automotive Pack Springer, Davide Scaramuzza.

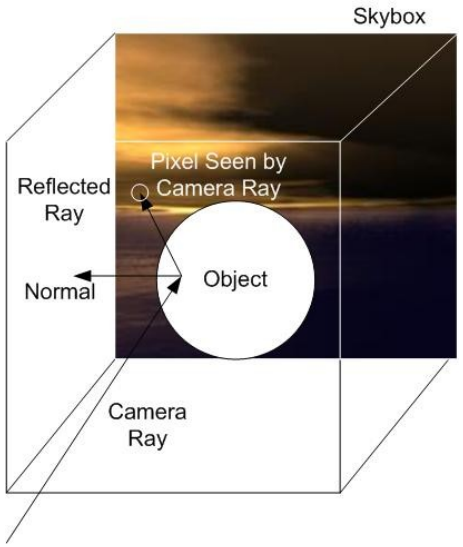
Image Based Lighting (IBL)

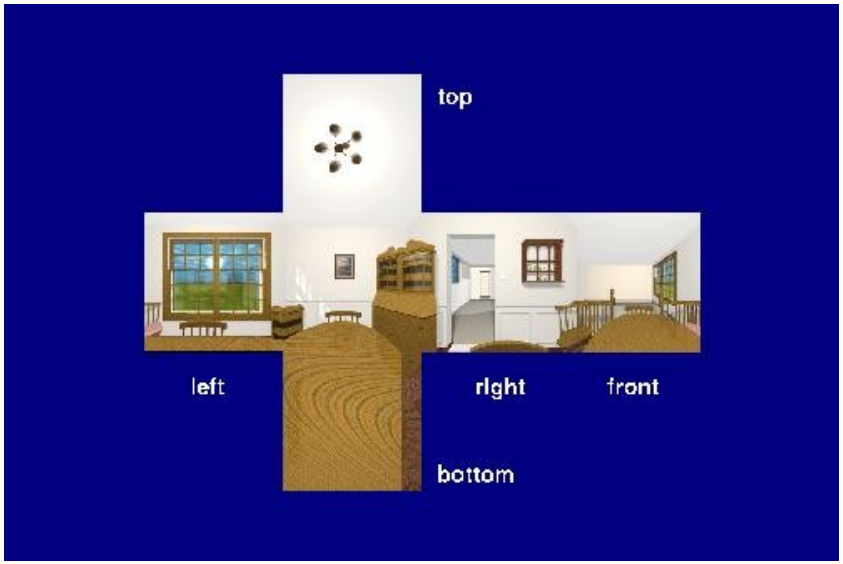
- Technika oświetlenia wykorzystująca **mapę środowiska** rzutowaną na powierzchnię oświetlanego obiektu
  - Najczęściej jest to ***omnikierunkowe* zdjęcie**, zwykle *HDR*

Źródło obrazów:
http://www.hdrlabs.com/sibl

Mapowanie środowiska

- Technika *IBL* wywodzi się z **mapowania środowiska**, które jest techniką rzutowania otoczenia obiektu pod postacią tekstury na jego powierzchnię
  - Mapowanie sferyczne (ang. *sphere mapping*)
  - Mapowanie sześcienne (ang. *cube mapping*)





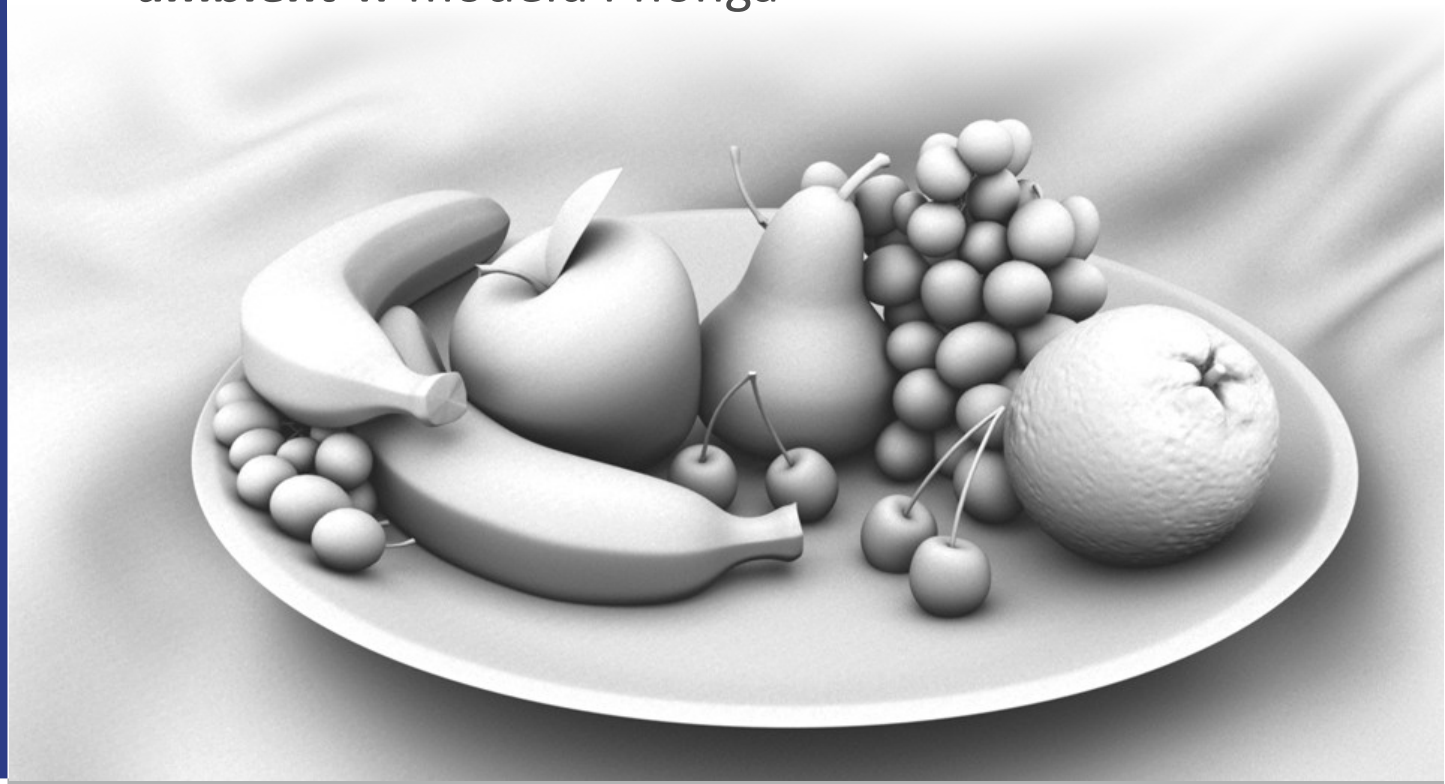
Źródło obrazów:
Wikipedia
http://escience.anu.edu.au/lecture/cg

Ambient Occlusion

- Kolejna, tym razem dokładniejsza niż phongowski komponent *ambient*, **aproksymacja oświetlenia niebezpośredniego**
- Estymacja tego, **jak bardzo dane miejsce *narażone* jest na działanie oświetlenia pochodzącego ze środowiska** (ang. *ambient light*)
- Rezultat może np. **wpłynąć na wartość komponentu *ambient*** w modelu Phonga

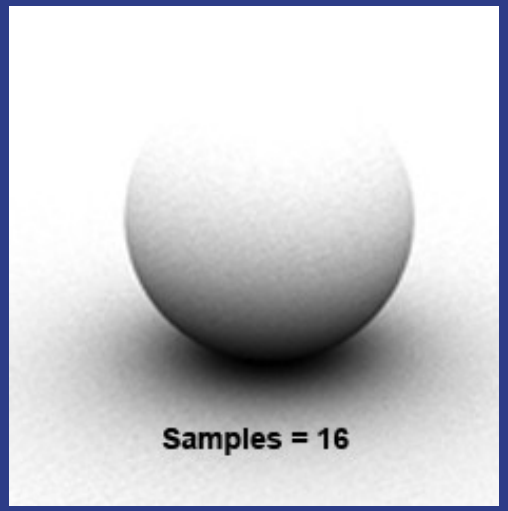
Dla każdego punktu obierana jest **półsfera**. W jej obrębie **wysyłane są promienie**.

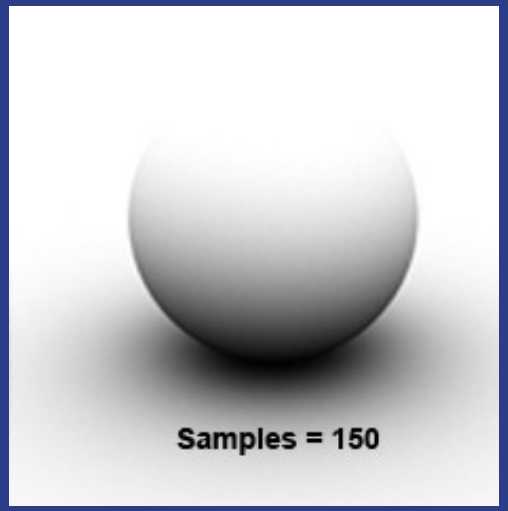
To, **ile z nich** i **w jakiej odległości trafi w inne obiekty**, sugeruje ile światła ze środowiska może dotrzeć do tego punktu.



Źródło obrazu:
http://ucbugg.github.io/learn.ucbugg

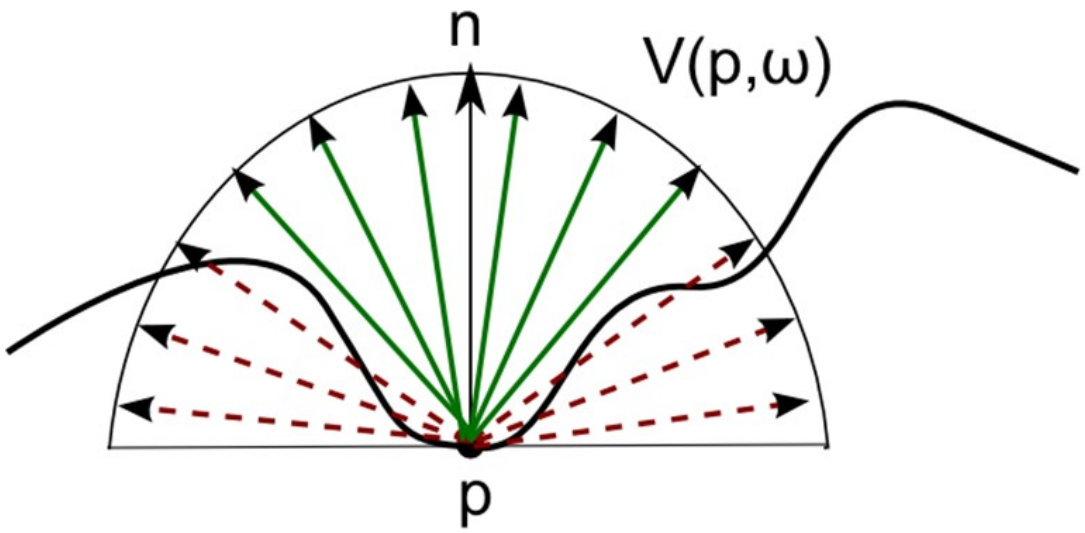
Ambient Occlusion





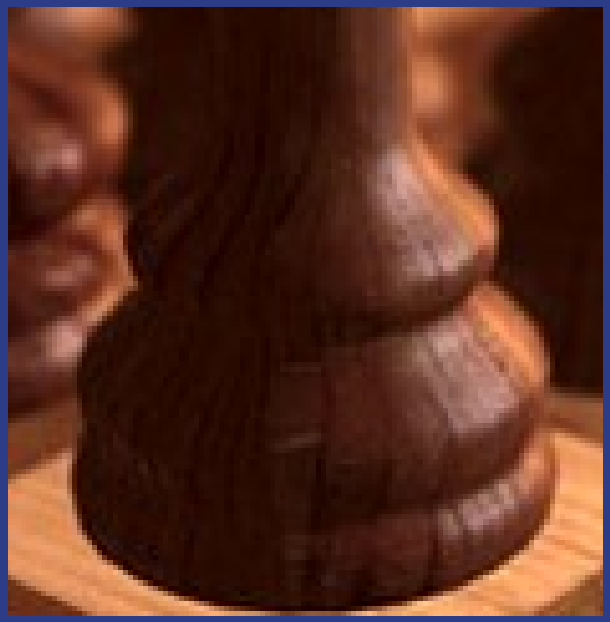
Źródło obrazów:
Nikki Ferchland, http://animationdynamics.com/
https://realtimeillumination.wordpress.com/

- Technika ta **nie bierze pod uwagę źródeł światła**!
- Uwzględnia jedynie **geometrię obiektów** sceny i ich **względne położenie** oraz **odległości**
- Jest to technika nastawiona na **rednerowanie *offline***
  - Bazuje na koncepcji próbkowania / śledzenia promieni, jest więc **bardzo kosztowna**
  - Im mniej próbek, tym większe **zaszumienie**



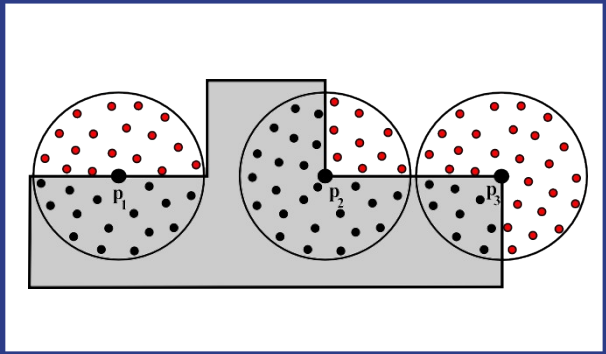
$$AO(\mathbf{p},\mathbf{n}) = \frac{1}{\pi}\int_{\Omega} V(\mathbf{p},\omega)\,\mathbf{n}\cdot\omega\,d\omega,$$

Ambient Occlusion



Źródło obrazów:
Andrew Kramer

Ambient Occlusion



Metody *monte carlo* pozwalają przyspieszyć działanie algorytmu.

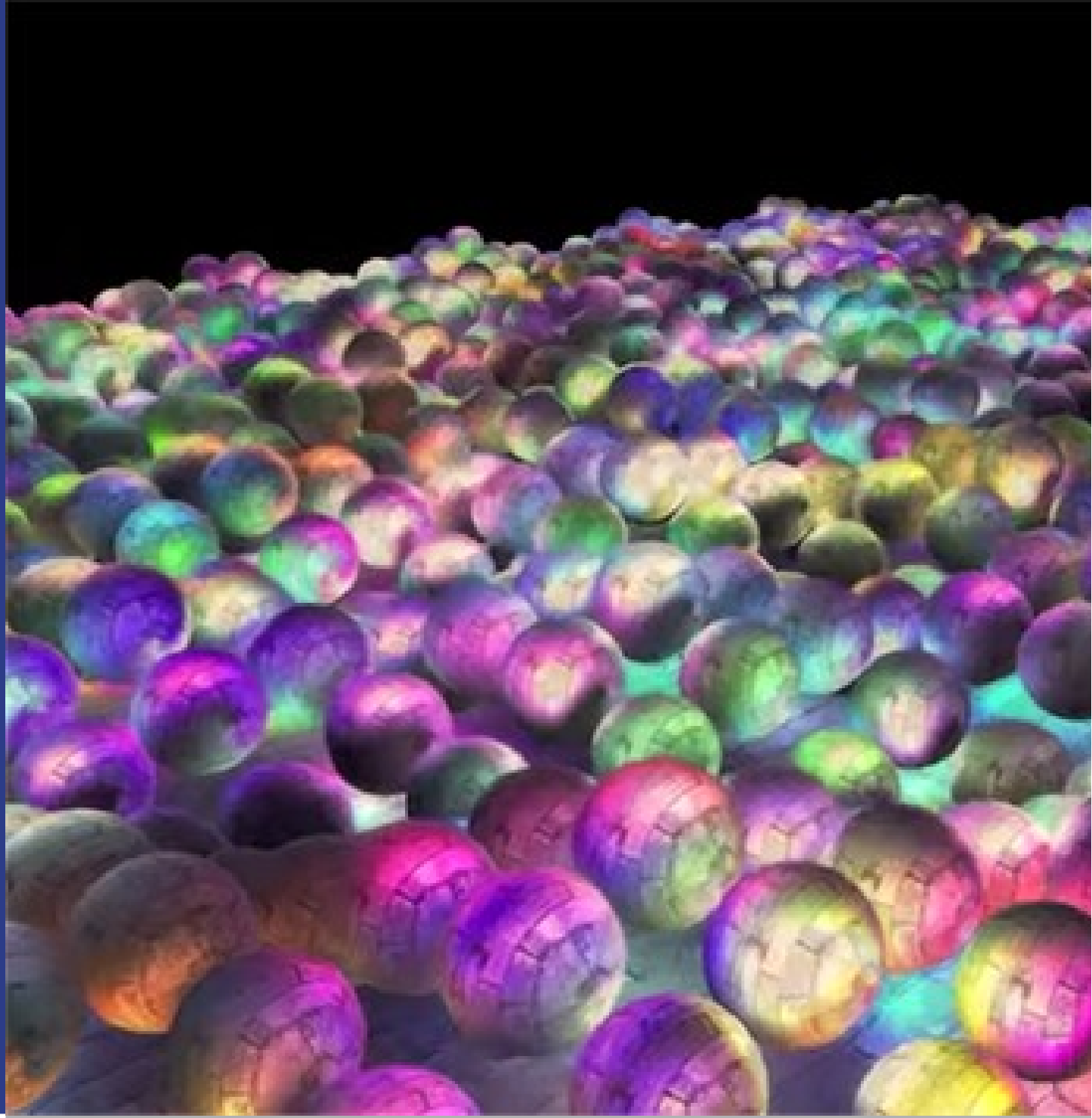
- **Screen Space Ambient Occlusion** (*SSAO*)
  - Operuje nie na geometrii, ale **na zawartości bufora klatki** – dużo szybsze!
  - Porównuje **względne odległości** blisko oddalonych od siebie pikseli – korzysta z *bufora Z*
  - Powoduje artefakty polegające na **zmiennym oświetleniu obiektów zależnie od miejsca i kierunku**, z którego je oglądamy

Źródła obrazów:
https://realtimeillumination.wordpress.com/
http://media.moddb.com/

Deferred Shading

Problem:

W jaki sposób **efektywnie** obliczyć dynamiczne oświetlenie dla **dużej liczby świateł** i **bardzo złożonej geometrii**?



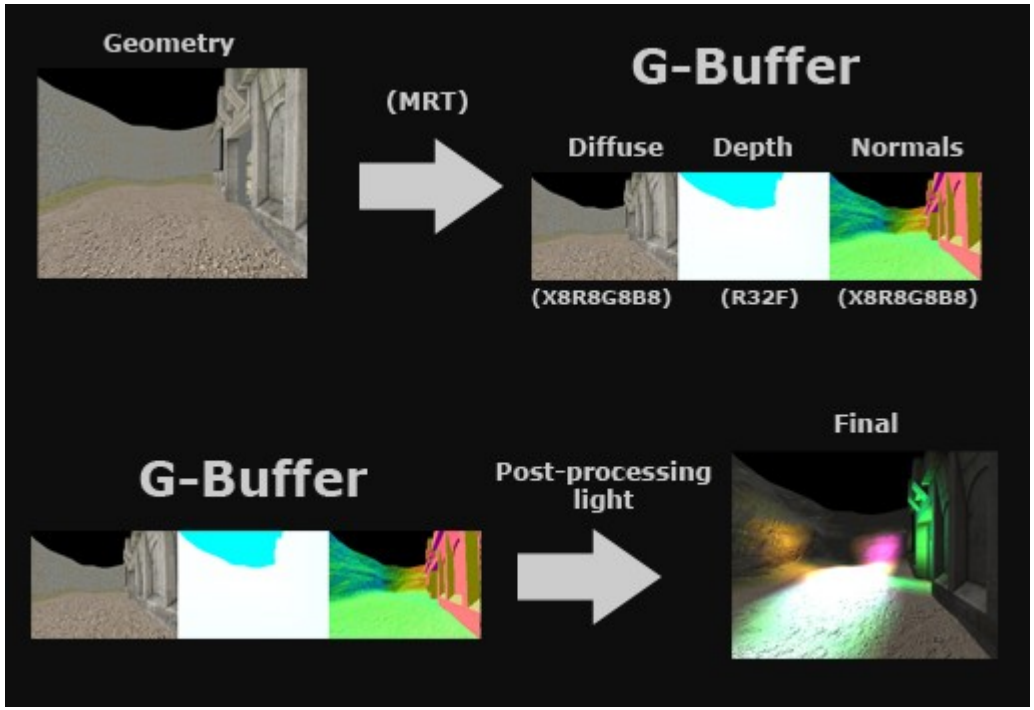
Źródło obrazu:
http://www.learnopengl.com

Deferred Shading

Screen z UE3, prezentuje technikę *Deferred Shading* ze **123 dynamicznymi światłami** (rok 2011)

Źródła obrazów:
Anton Lindeström
Epic Games, UDK

- Metoda **odraczająca** obliczenie oświetlenia do momentu, gdy będzie gotowa zawartość bufora klatki
- Dzieli proces renderowania klatki na **dwa etapy**:
  - Obliczenie **na podstawie geometrii** sceny wartości, które będą niezbędne dla obliczenia oświetlenia
  - Obliczenie właściwego oświetlenia z pomienięciem geometrii, już **w przestrzeni ekranu**

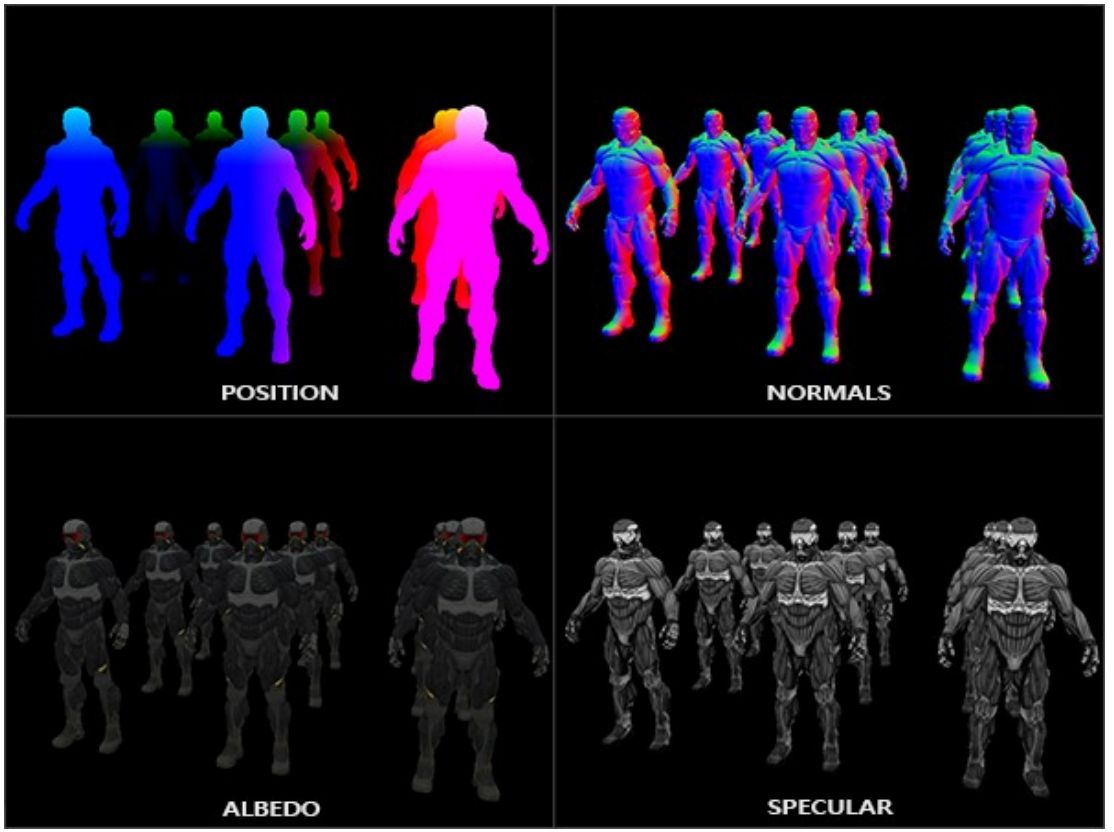


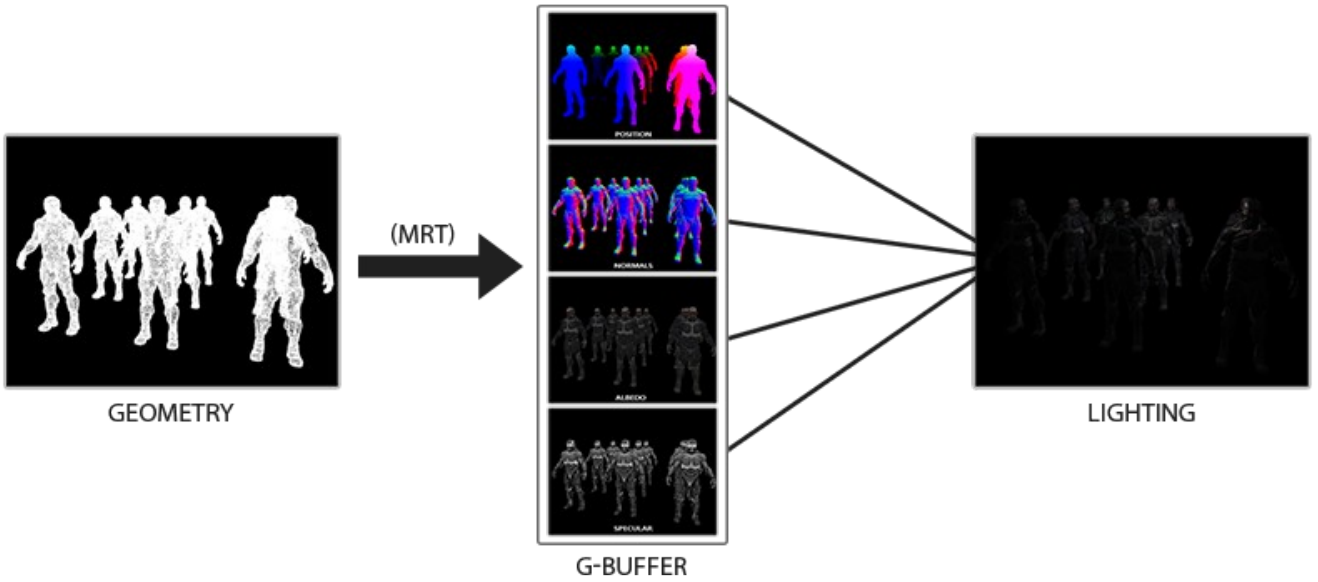
Deferred Shading

Wynik obliczeń z pierwszego etapu przechowywany jest w tzw. ***G-Buffer***.

Dopiero na podstawie jego zawartości obliczane jest faktyczne oświetlenie.

Zawartość G-Buffer:





Źródła obrazów:
http://www.learnopengl.com

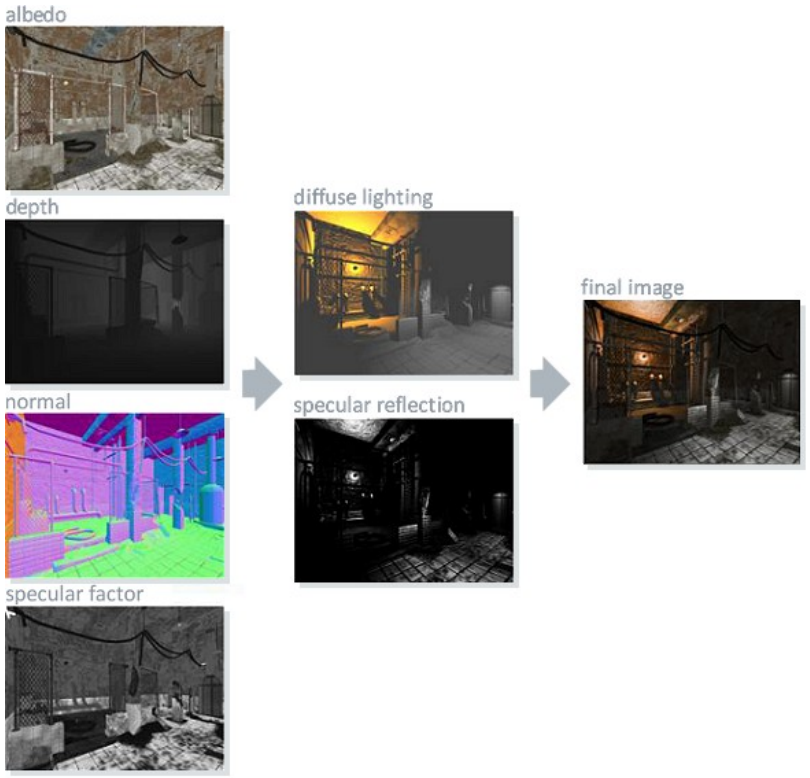
Deferred Shading

- **Zalety** *Deferred Shading*:
  - Złożoność obliczeń związanych z oświetleniem jest **w dużo mniejszym stopniu zależna od złożoności geometrii sceny**
  - Tym samym możliwość wprowadzenia **znacznie większej liczby źródeł światła**
- **Wady** *Deferred Shading*:
  - Problemy z technikami **anti-aliasingu**
    - Właściwy kolor pikseli jest obliczany dopiero na samym końcu, w obrazie o docelowej rozdzielczości
  - Większe **zapotrzebowanie na pamięć**
    - Bufor klatki musi przechować wszystkie wartości wyliczone podczas pierwszego etapu – jako tzw. *G-Buffer*

Źródło obrazu:
http://gamedev.ru

Deferred Lighting

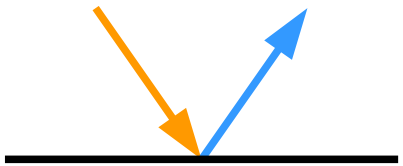
- Modyfikacją techniki *Deferred Shading* jest podejście nazywane ***Deferred Lighting***
  - **Dodatkowy etap pośredni** pracujący na geometrii sceny
    - Obliczenie osobno komponentów *diffuse* i *specular,* *o*statni etap łączy rezultaty
  - **Redukcja ilości pamięci** potrzebnej do przechowania wyjścia z pierwszego etapu



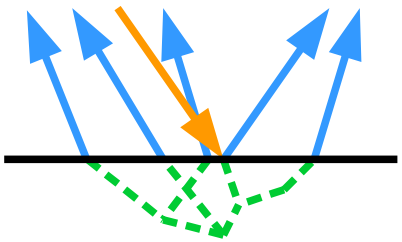
Źródło obrazu:
Azkanan, Leadwerks Software

Subsurface Scattering

- Technika ***subsurface scattering*** (SSS) polega na symulowaniu **podpowierzchniowego rozpraszania światła**
  - Bierze pod uwagę światło, które **przeszło** (nie zostało odbite) **przez częściowo przezroczystą powierzchnię**
  - Światło jest rozpraszane **w głębi obiektu** w wielu kierunkach
  - Promienie światła **wychodzą w innych miejscach powierzchni** obiektu oraz **pod innym kątem** niż gdyby zostały odbite przez tę powierzchnię



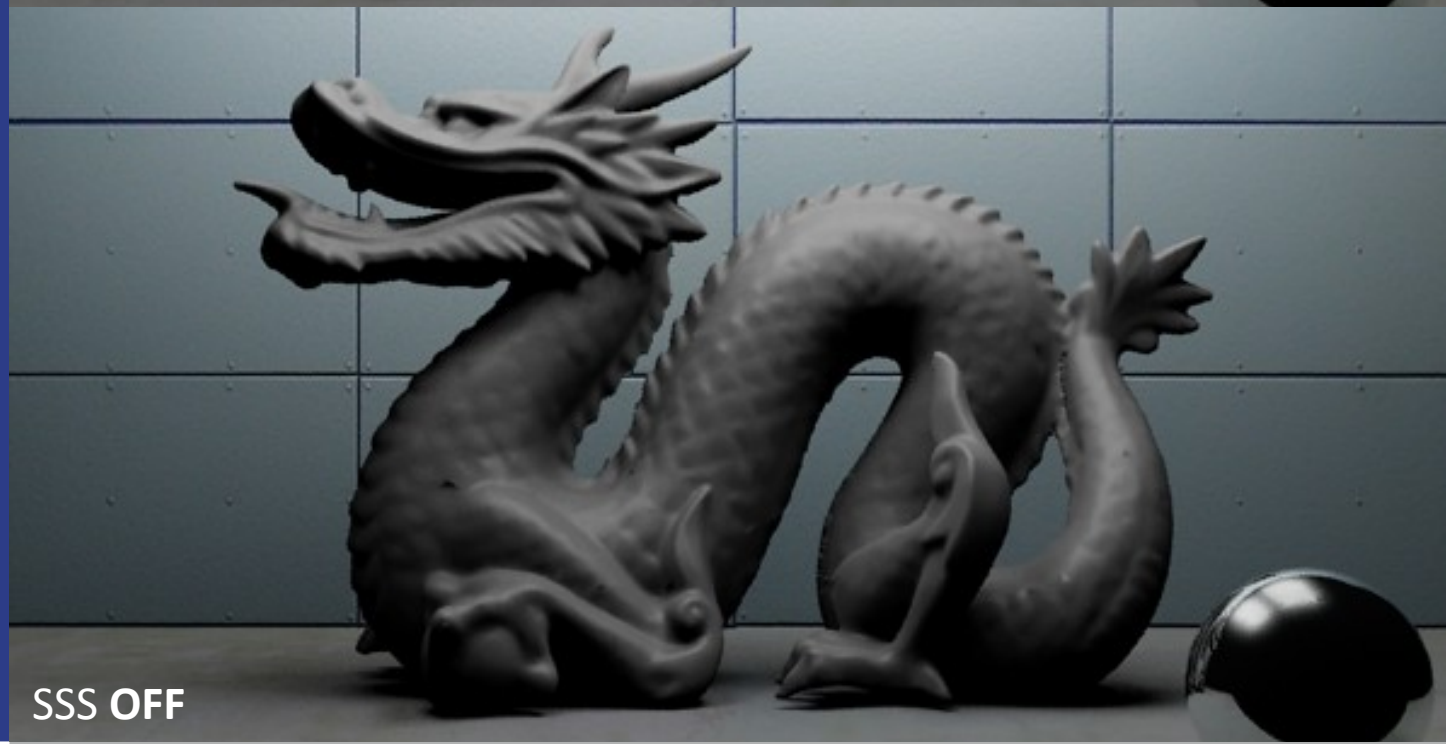
Odbicie powierzchniowe



Rozproszenie podpowierzchniowe

- W rezultacie uwzględniona jest **grubość przedmiotu**, jak i **zmiękczenie ostrych krawędzi cieni**
  - Materiały i obiekty takie jak: wosk, marmur, ludzka skóra, liście

Subsurface Scattering



Źródło obrazów:
MrBluesummers

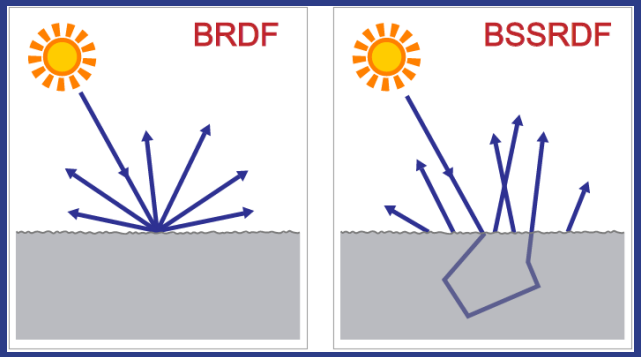
Subsurface Scattering

W rzeczywistym świecie ludzka skóra odbija na powierzchni jedynie 6% światła, pozostałe **94% pochodzi z rozproszenia podpowierzchniowego**!
[Krishnaswamy, Baronoski 2004]

Aby realistycznie renderować skórę, konieczne jest więc uwzględnienie *SSS*.

Źródło obrazu: MrBluesummers

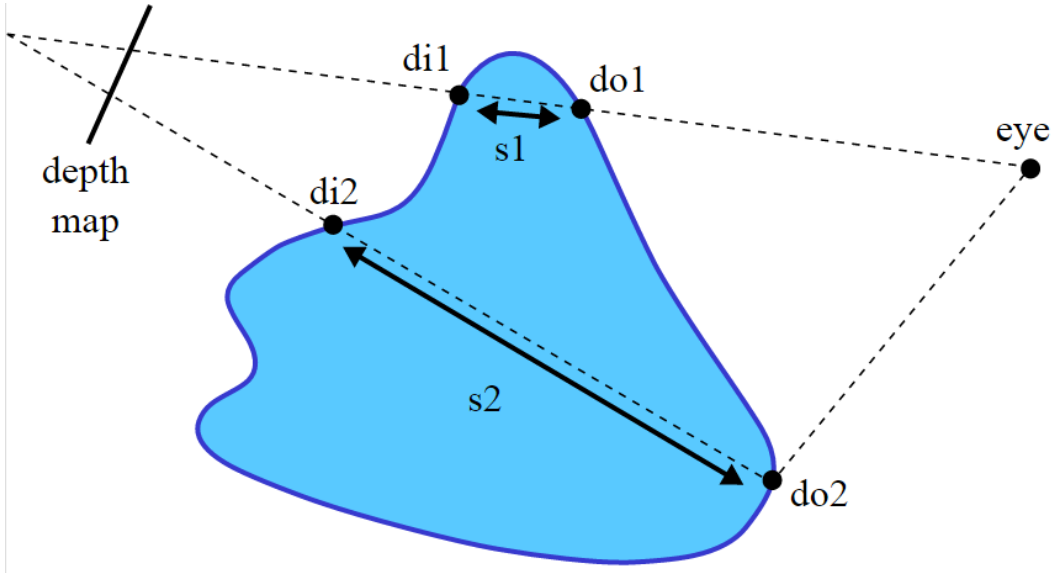
Subsurface Scattering



Odpowiednikiem *BRDF* uwzględniającym pod-powierzchniowe rozproszenie jest ***BSSRDF***

Źródło obrazów: Wikipedia

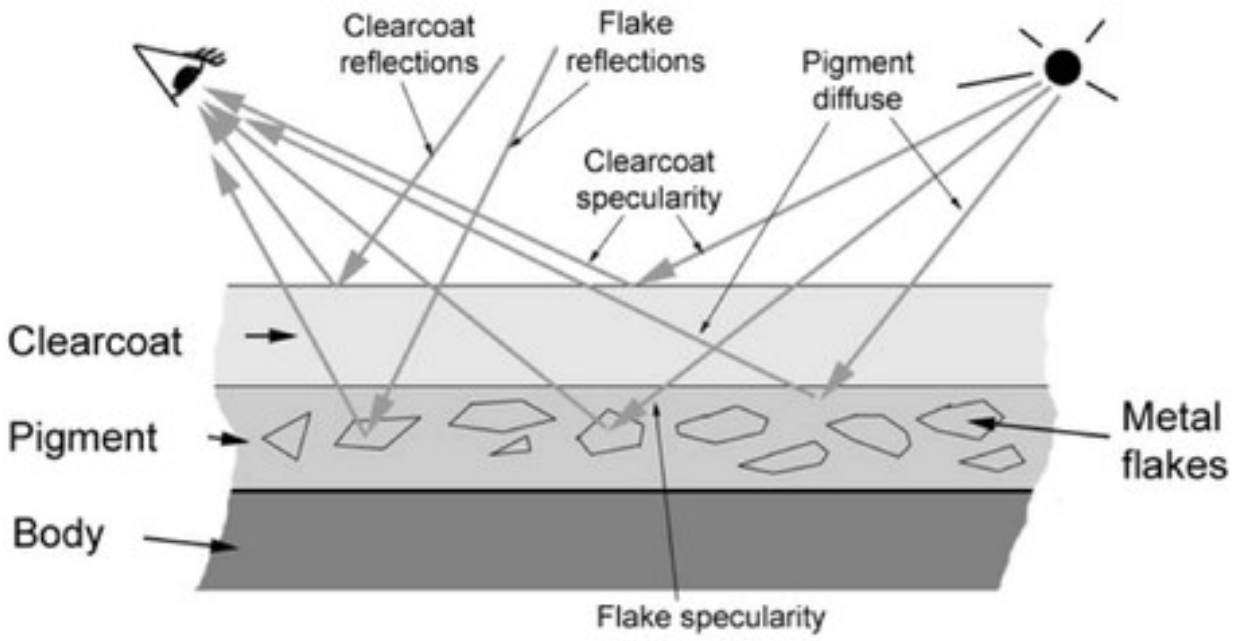
- Symulowanie SSS
  - Podstawowe równanie renderingu obejmuje jedynie powierzchniowe odbicie światła opisane za pomocą BRDF
  - Prosta metoda *czasu rzeczywistego* pozwalająca na przybliżenie wyglądu obiektu zbudowanego z np. **wosku**, polega na wyznaczeniu jego **grubości w danym miejscu** korzystając z **bufora głębokości** uzyskanego z punktu widzenia światła



Materiały wielowarstwowe

- Szczególnym przypadkiem powiązanym z *SSS* jest renderowanie **materiałów wielowarstwowych** (ang. *layered material*)
- Przykładowe zastosowanie: wizualizacja **metalicznego lakieru samochodowego**

Przykład wyrenderowanego, kilkuwarstwowego materiału imitującego **lakier samochodowy**.

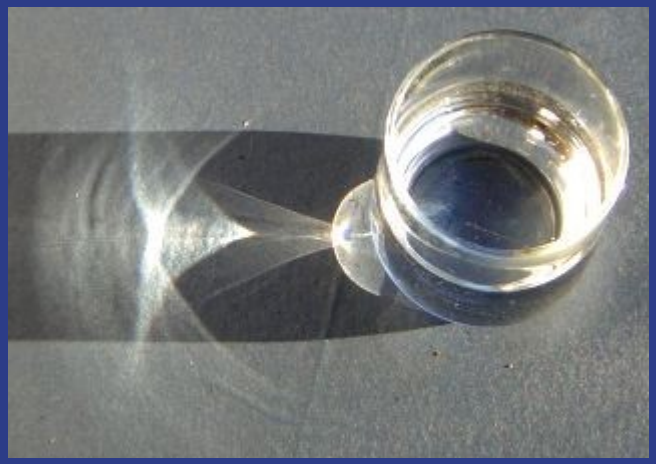


Cross-section of car paint

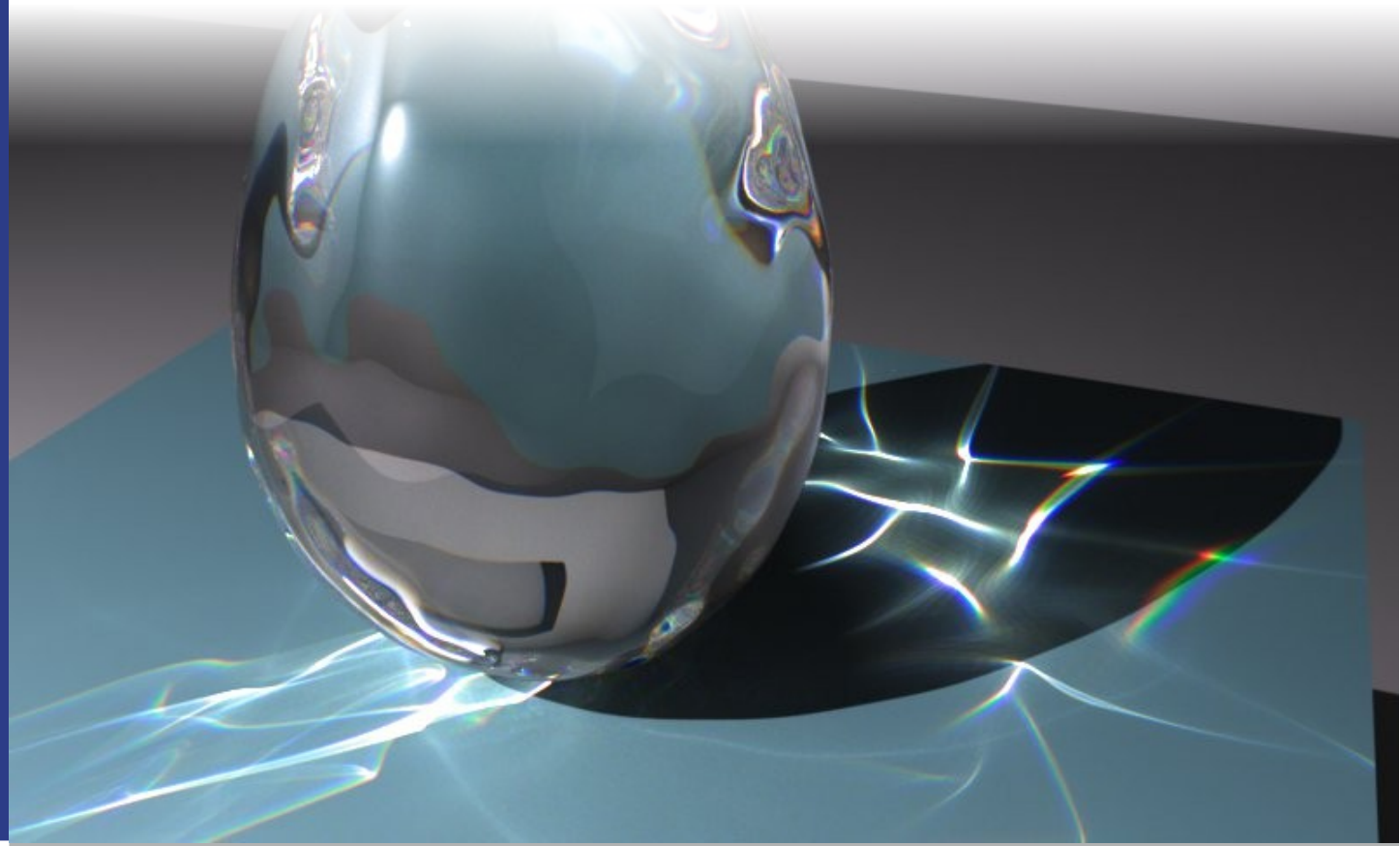
Źródła obrazów:
MrBluesummers
D M Multimedia

Kaustyki

- Zjawisko polegające na trafianiu wielu odbitych lub załamanych fal światła w pobliżu jednego miejsca nazywamy **kaustyką** (ang. *caustic*)
  - Najczęściej mamy do czynienia z takim zjawiskiem np. przy **wklęsłych zwierciadłach i powierzchniach przepuszczających światło** lub przy **falujących powierzchniach płynów**



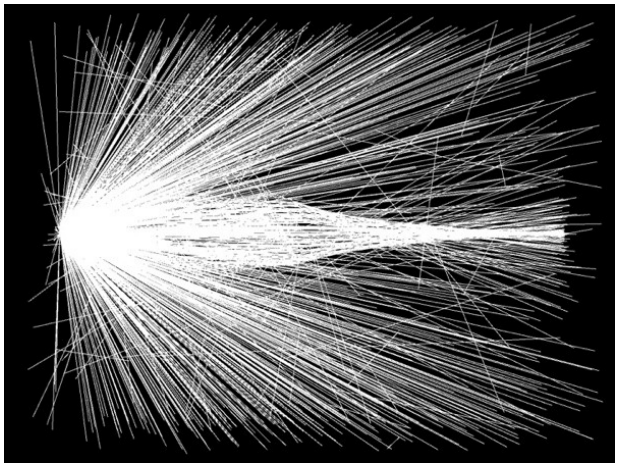
Szklanka wody powoduje **załamanie fal światła** w taki sposób, że powstają kaustyki

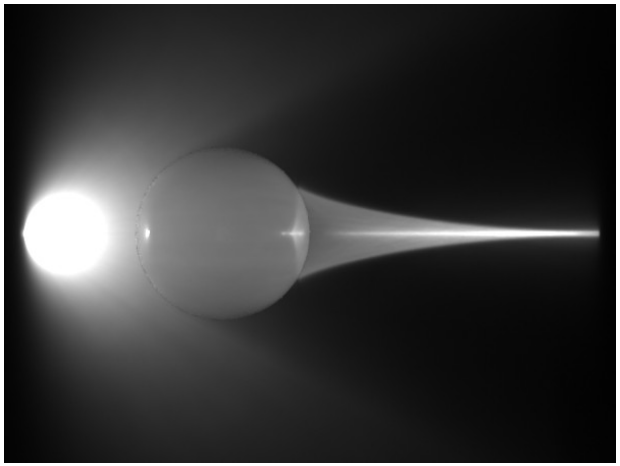


Źródła obrazów:
LuxRender
Wikipedia

Kaustyki

- Realistyczne renderowanie polega na **śledzeniu załamywanych lub odbijanych promieni światła**
  - Szczególnie dobrze widoczne przy technice **mapowania fotonów** (ang. *photon mapping*)





- Metody czasu rzeczywistego polegają m.in. na **próbkowaniu powierzchni**, operacjach *per-fragment* i **nakładaniu tekstur** imitujących obszary skupienia promieni

Źródło obrazów:
FXGuide.com, Mike Seymour. *The State of Rendering - Part 2.*

Zachodniopomorski
Uniwersytet
Technologiczny
w Szczecinie

http://bazyluk.net/dydaktyka

Bartosz Bazyluk

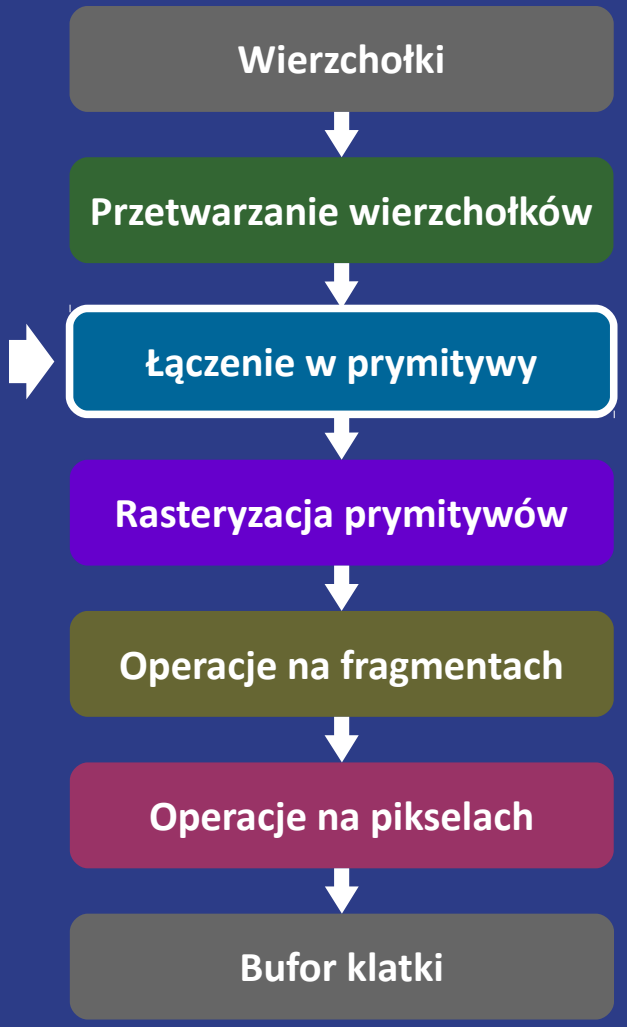
Definicja Geometrii

Wstęp do syntezy trójwymiarowej grafiki komputerowej

Wydział
Informatyki

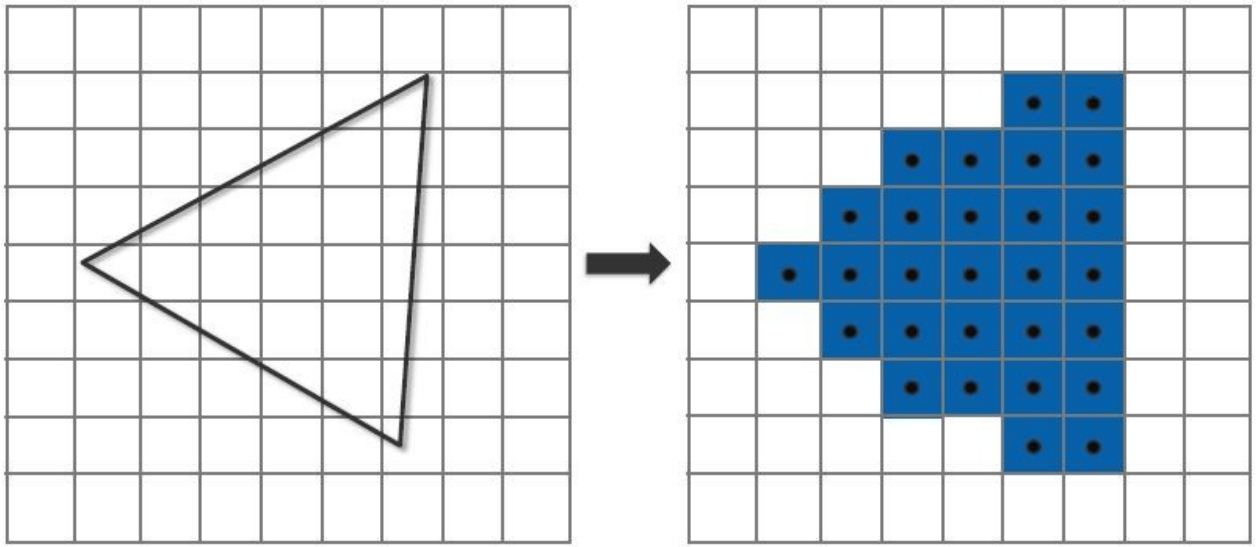
Grafika Komputerowa i Wizualizacja, Informatyka S1, II Rok

Renderowanie geometrii



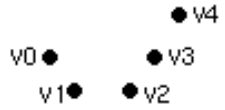
Źródło obrazu:
GPU Design

- Geometria jest opisana przede wszystkim **pozycjami wierzchołków**
- Podczas procesu renderowania powierzchnie obiektów rysowane są za pomocą tzw. **prymitywów**
- Prymitywy to **figury geometryczne** odpowiadające za to, jak informacja o kolejnych pozycjach zostanie wykorzystana podczas rysowania
- Mogą to być np. *punkty*, *linie*, *łamane*, *trójkąty*, *pasy trójkątów*, *czworokąty*, *wielokąty*, itp.

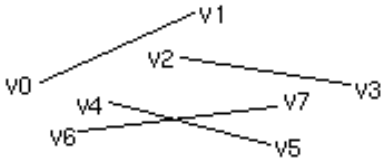


Renderowanie geometrii

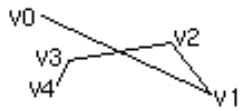
- Prymitywy obsługiwane przez *OpenGL*:



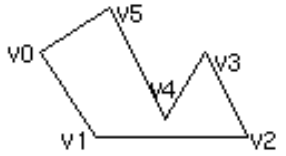
GL_POINTS



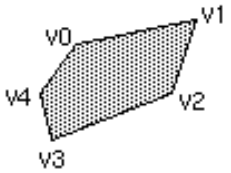
GL_LINES



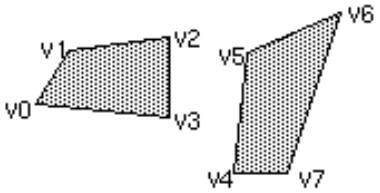
GL_LINE_STRIP



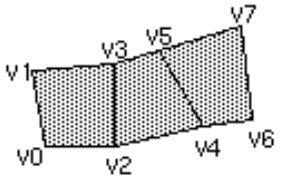
GL_LINE_LOOP



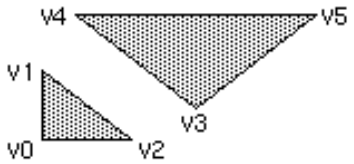
GL_POLYGON



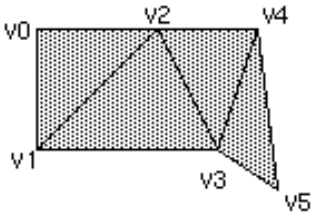
GL_QUADS



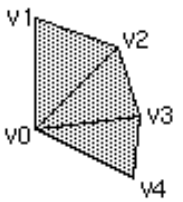
GL_QUAD_STRIP



GL_TRIANGLES



GL_TRIANGLE_STRIP

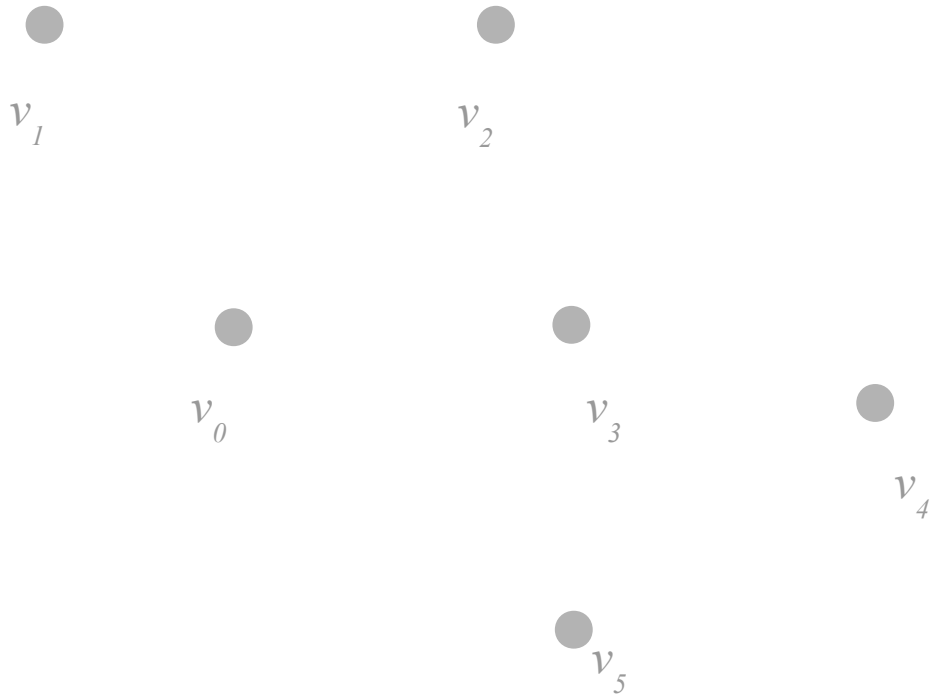


GL_TRIANGLE_FAN

Źródło obrazu:
http://www.opengl.org

Renderowanie geometrii

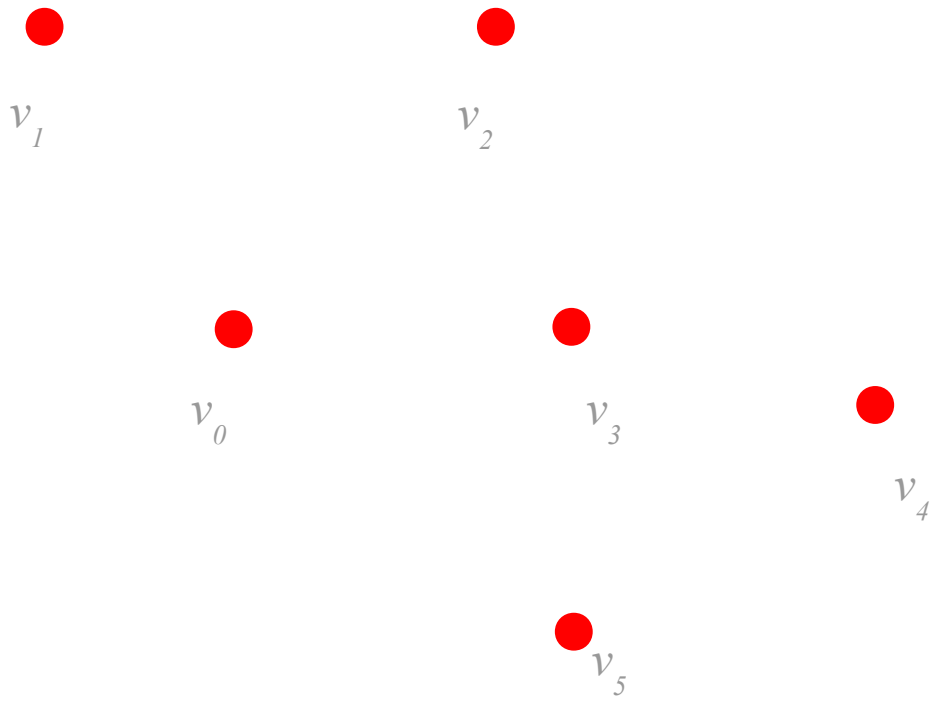
- Przykład:
  - Mamy dane 6 wierzchołków o następujących położeniach:



  - Myślimy o punktach **w przestrzeni 3D**, nie na płaszczyźnie

Renderowanie geometrii

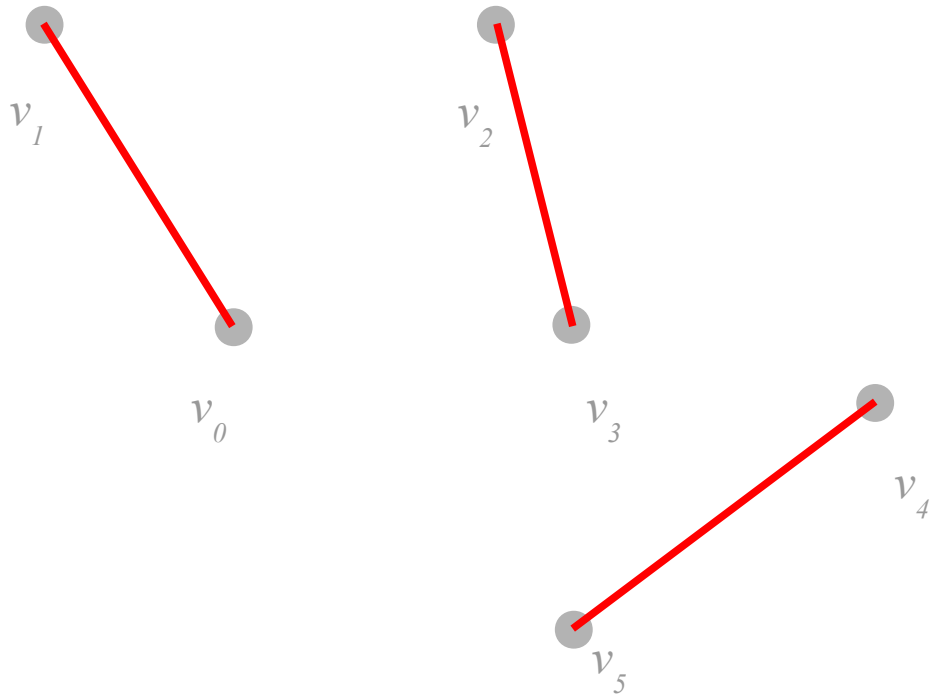
- Przykład:
  - Mamy dane 6 wierzchołków o następujących położeniach:



  - Użycie prymitywu **punkty** (ang. *points*)

Renderowanie geometrii

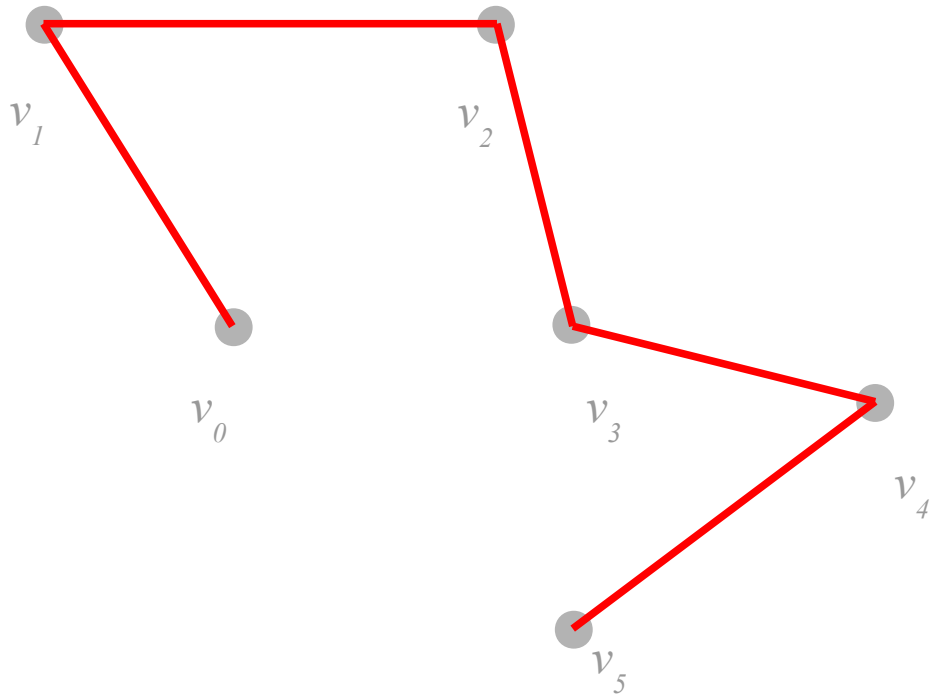
- Przykład:
  - Mamy dane 6 wierzchołków o następujących położeniach:



  - Użycie prymitywu **linie** (ang. *lines*)

Renderowanie geometrii

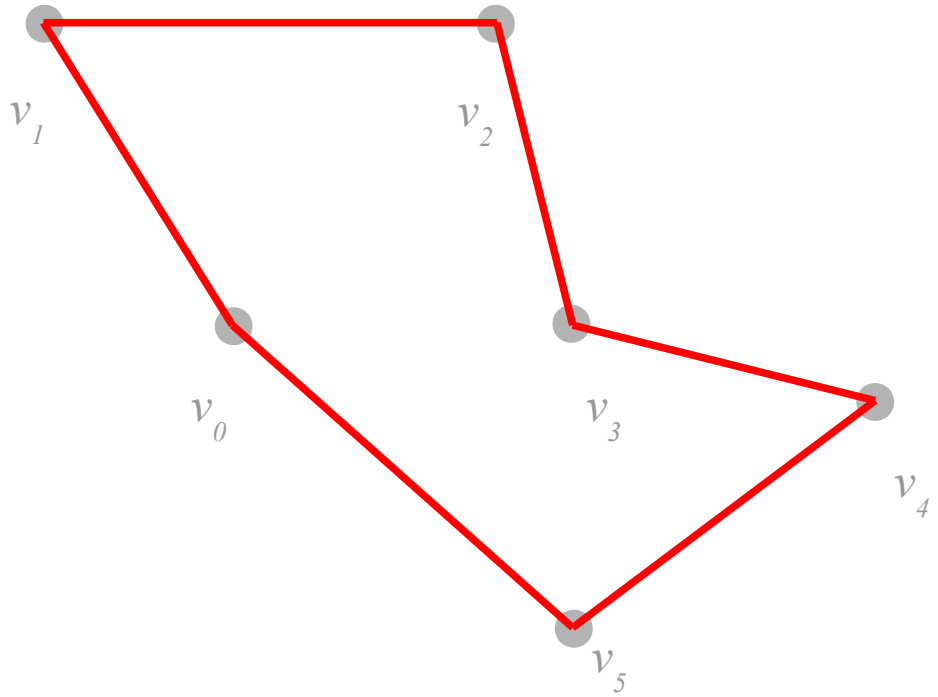
- Przykład:
  - Mamy dane 6 wierzchołków o następujących położeniach:



  - Użycie prymitywu **łamana** (ang. *line strip*)

Renderowanie geometrii

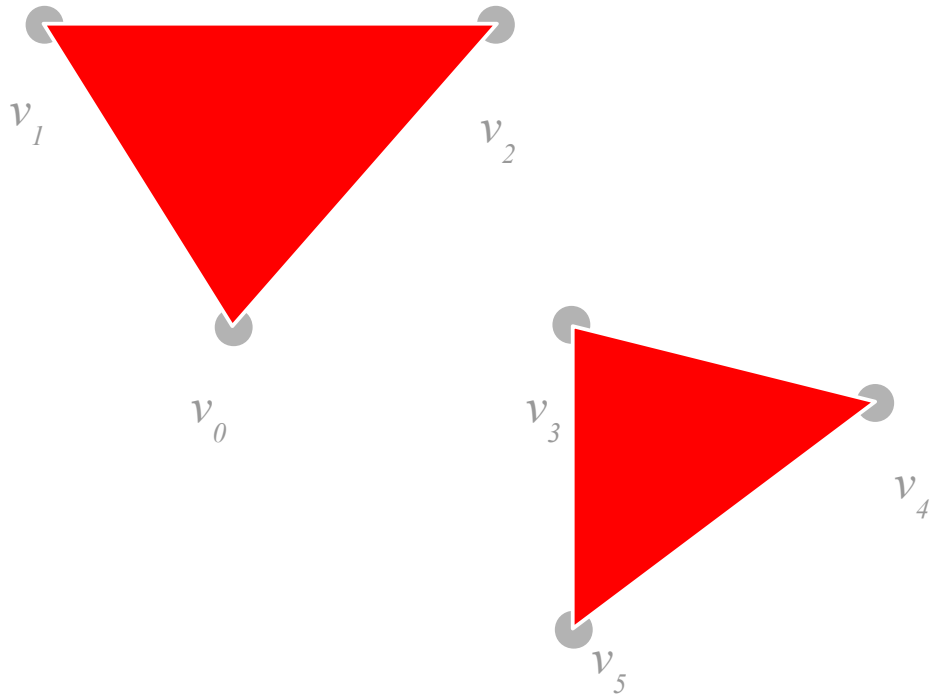
- Przykład:
  - Mamy dane 6 wierzchołków o następujących położeniach:



  - Użycie prymitywu **łamana zamknięta** (ang. *line loop*)

Renderowanie geometrii

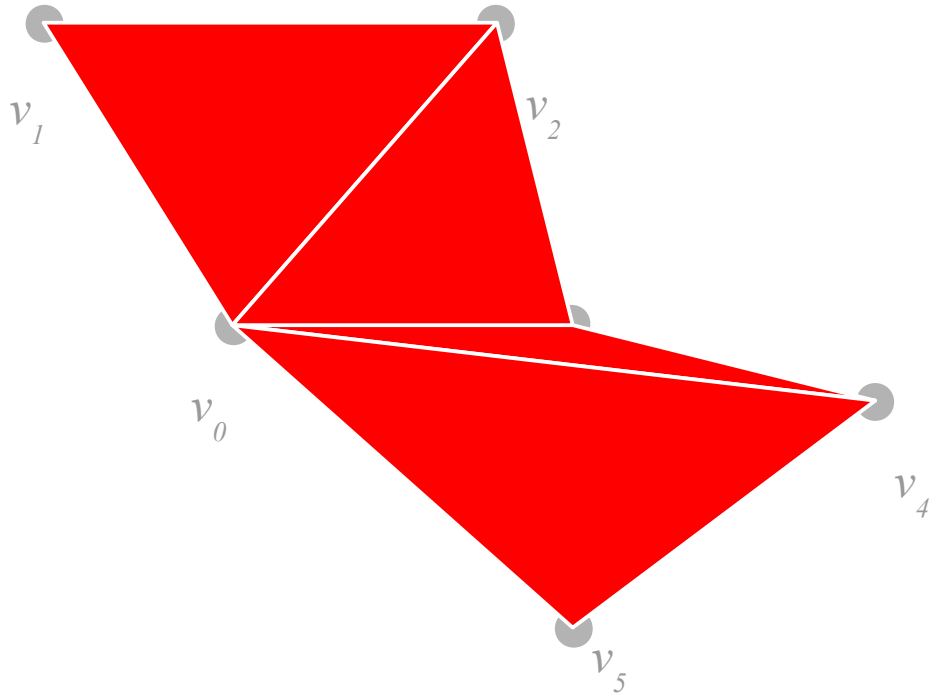
- Przykład:
  - Mamy dane 6 wierzchołków o następujących położeniach:



  - Użycie prymitywu **trójkąty** (ang. *triangles*)

Renderowanie geometrii

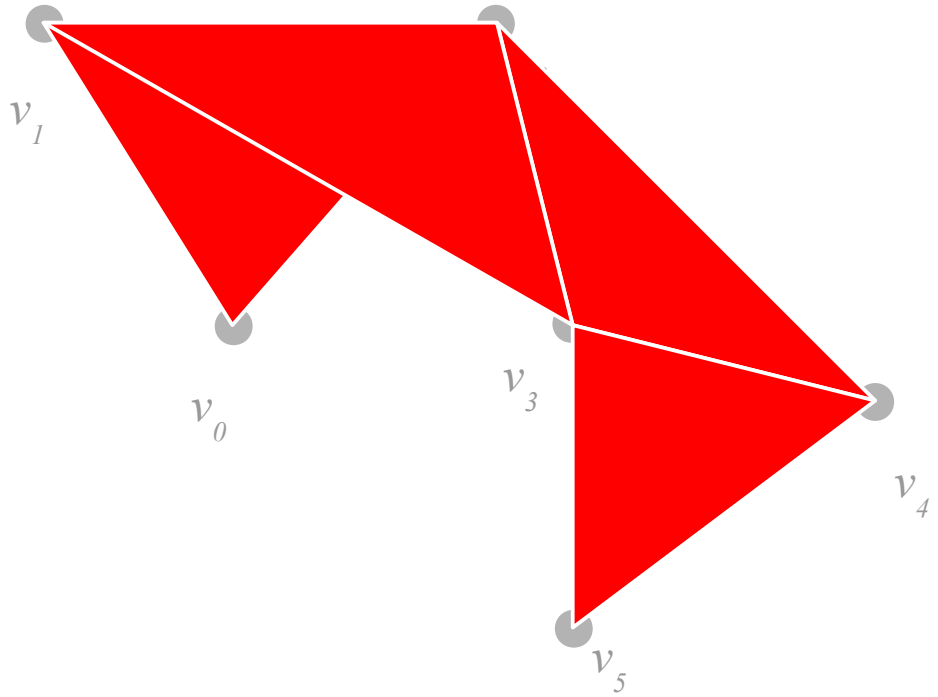
- Przykład:
  - Mamy dane 6 wierzchołków o następujących położeniach:



  - Użycie prymitywu **wachlarz** (ang. *triangle fan*)

Renderowanie geometrii

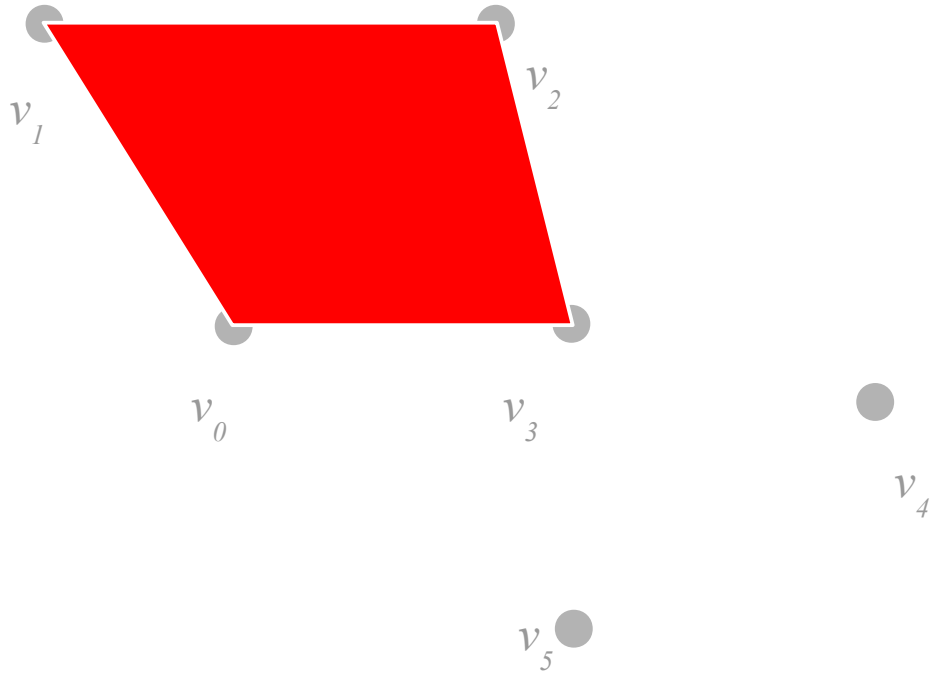
- Przykład:
  - Mamy dane 6 wierzchołków o następujących położeniach:



  - Użycie prymitywu **pas trójkątów** (ang. *triangle strip*)

Renderowanie geometrii

- Przykład:
  - Mamy dane 6 wierzchołków o następujących położeniach:

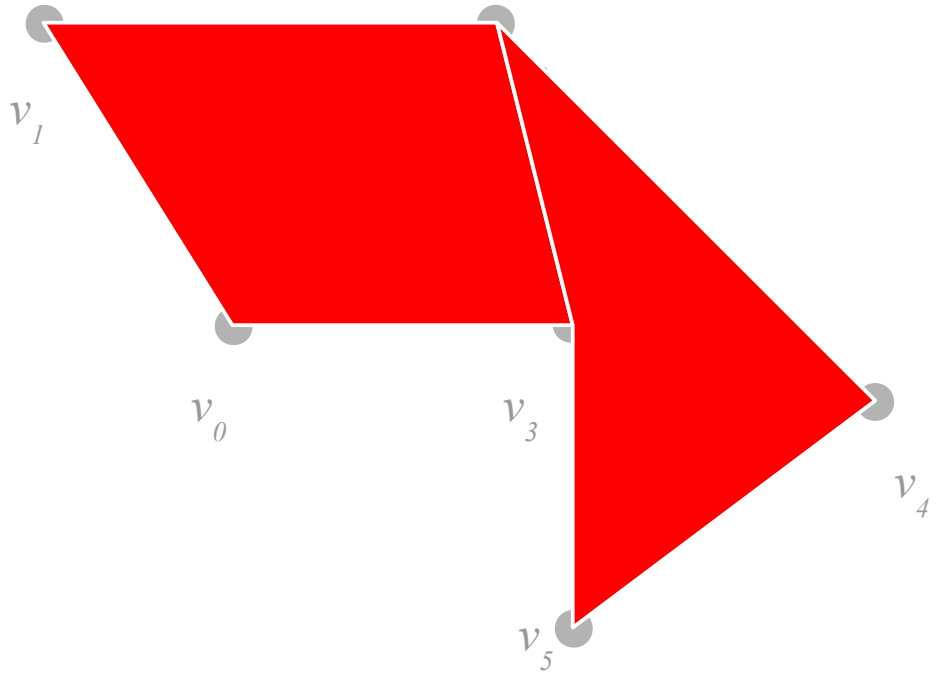


Dwa ostatnie wierzchołki nie są wykorzystane - aby powstał kolejny *quad*, potrzebne byłyby dwa następne!

- Użycie prymitywu **czworokąty** (ang. *quads*)

Renderowanie geometrii

- Przykład:
  - Mamy dane 6 wierzchołków o następujących położeniach:

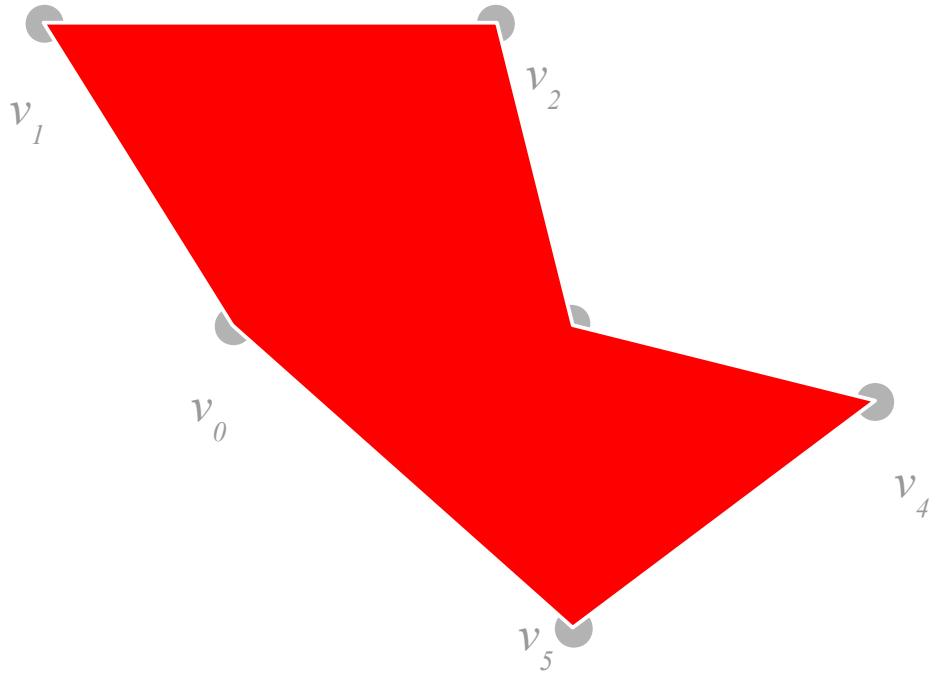


Ten czworokąt jest **niepoprawny** *(„zdegenerowany”)* – nie jest **wypukły**!

- Użycie prymitywu **pas czworokątów** (ang. *quad strip*)

Renderowanie geometrii

- Przykład:
  - Mamy dane 6 wierzchołków o następujących położeniach:

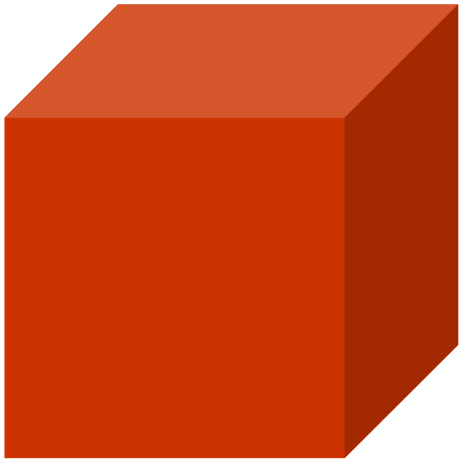


Ten wielokąt jest **niepoprawny** *(„zdegenerowany”)* – nie jest **wypukły**!

- Użycie prymitywu **wielokąt** (ang. *polygon*)

Face culling

- Rozpatrzmy przypadek: scena zawiera **sześcian**

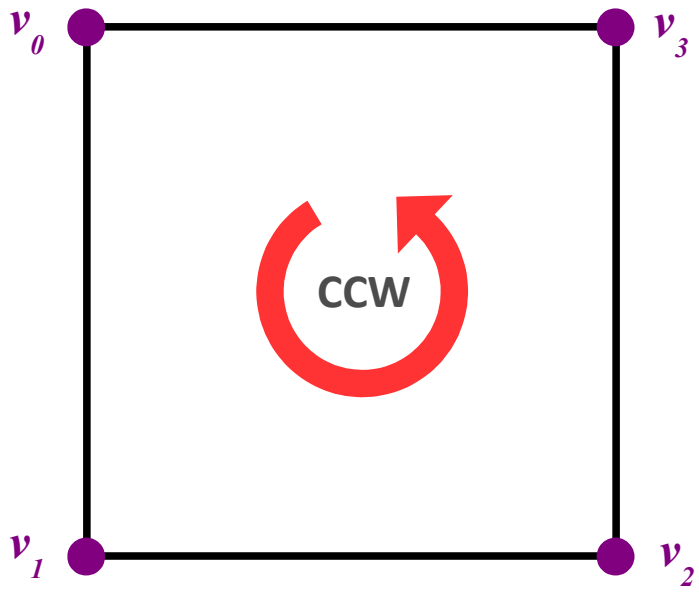


- **Ile jego ścian jest widocznych** na danej klatce animacji?
- **Ile ścian jest renderowanych** podczas rysowania klatki?
- **W którym momencie potoku** pomijane są tylne ściany?

Face culling

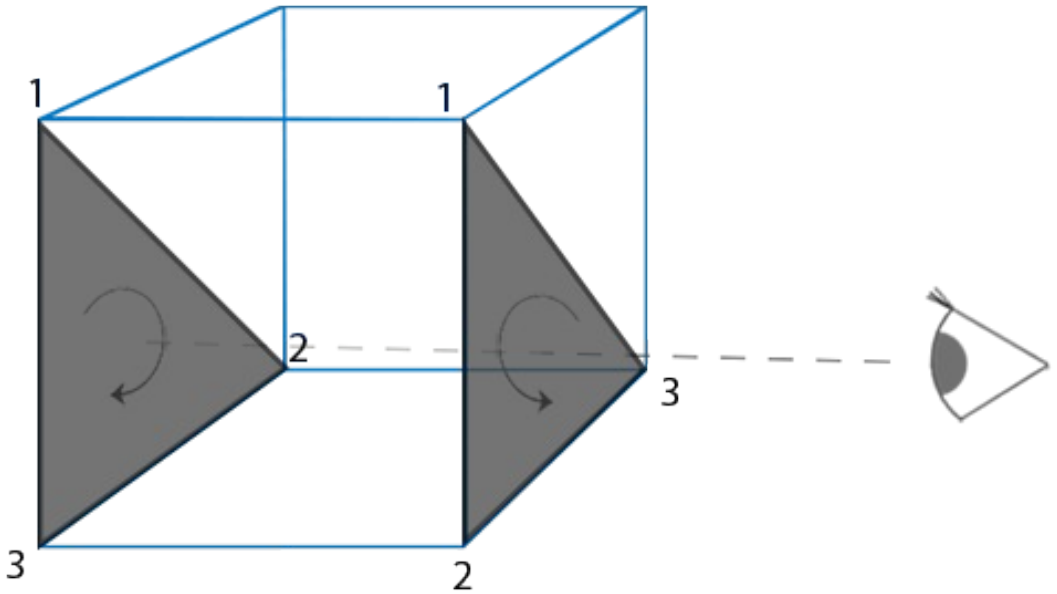
- Technika *face culling* pozwala na **pominięcie ścian widocznych z określonej strony** jeszcze przed rozpoczęciem ich rasteryzacji
- O odrzuceniu decyduje **kolejność wierzchołków po ich projekcji** (ang. *winding order*)
  - **Zgodnie z ruchem wskazówek zegara** (ang. ***c**lock**w**ise*)
  - **Przeciwnie do wskazówek zegara** (ang. ***c**ounter-**c**lock**w**ise*)

Konieczne jest więc **ustalenie w jakiej kolejności definiowane są wierzchołki ścian** gdy patrzymy na ich przednią stronę, a potem **konsekwentne trzymanie się ustalonej kolejności** podczas opisywania geometrii obiektów.



Face culling

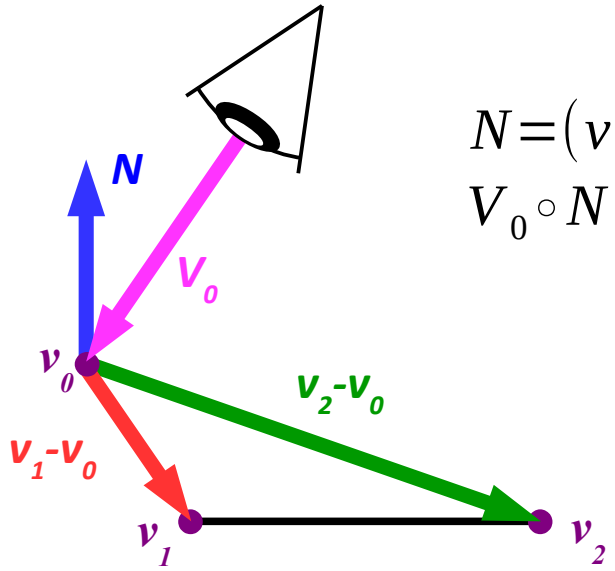
- Jeśli **po przejściu do współrzędnych kamery** kolejność wierzchołków **zgadza się z tą uznaną za przednią**, to znaczy że widzimy przednią stronę ściany i należy ją poddać rasteryzacji
- Jeśli k**olejność jest odwrotna**, to widzimy **tylną stronę ściany** i możemy ją pominąć (jeśli używamy techniki *back face culling*)



Źródło obrazu:
http://www.learningopengl.com

Face culling

- Jak karta graficzna **porównuje kolejność wierzchołków**?
  - Czy **kąt pomędzy wektorem normalnym ściany**, a **wektorem obserwacji** jest **mniejszy lub równy 90 stopni**?
    - **Wektor normalny** obliczamy za pomocą **iloczynu wektorowego** trzech kolejnych wierzchołków w kolejności ich zdefiniowania
    - Równoważne pytanie: Czy **iloczyn skalarny** między tymi wektorami jest **większy lub równy zero**?
  - **Jeśli tak**, to kolejność jest różna od początkowej – **ścianę należy pominąć**!



$$N = (v_1 - v_0) \times (v_2 - v_0)$$
$$V_0 \circ N \geqslant 0$$

Face culling

- Kiedy *back face culling* jest **problematyczny**?
  - Gdy na scenie są **przezroczyste obiekty**
    - Wówczas pomijanie tylnych ścian może być widoczne
  - Gdy występują **obiekty o zerowej grubości**, w których obie strony ścian są widoczne
- Do czego **dodatkowo** może służyć ***back face culling***?
  - Usuwanie "niewidocznych" linii w renderowaniu *wireframe*, pod warunkiem że geometria jest wypukła
- Do czego mógłby służyć ***front face culling***?
  - Ukrywanie przednich ścian może być przydatne podczas renderowania *mapy cieni*, by uniknąć efektu *self-shadowing* gdy sprawdzane jest przysłonięcie sceny z punktu widzenia źródła światła

Face culling

- Jak korzystać z *face cullingu* w **OpenGL**?
  - **Włączanie**:

```
glEnable(GL_CULL_FACE);
```

  - **Wybór kolejności** definicji przednich ścian:

```
glFrontFace(GL_CCW); // przeciwnie do ruchu wskazówek zegara
glFrontFace(GL_CW); // zgodnie z ruchem wskazówek zegara
```

    - dotyczy wierzchołków definiowanych po wywołaniu tej funkcji

  - Wybór ścian, **które mają być pomijane**:

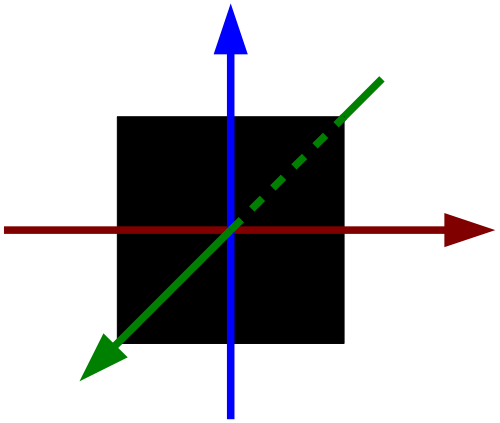
```
glCullFace(GL_BACK);
glCullFace(GL_FRONT);
glCullFace(GL_FRONT_AND_BACK);
```

Definicja geometrii w OpenGL

- Najprostszą metodą definicji i renderowania geometrii jest tzw. *immediate mode*
  - Definicja własności wszystkich wierzchołków **w każdej klatce**
  - Podejście którego **współcześnie** się już **nie stosuje**
    - Jest **nieefektywne**
      - W każdej klatce duża ilość danych przekazywana do karty graficznej
      - Współcześnie używa się buforów w pamięci karty graficznej
    - Jest **najłatwiejsze w zrozumieniu**

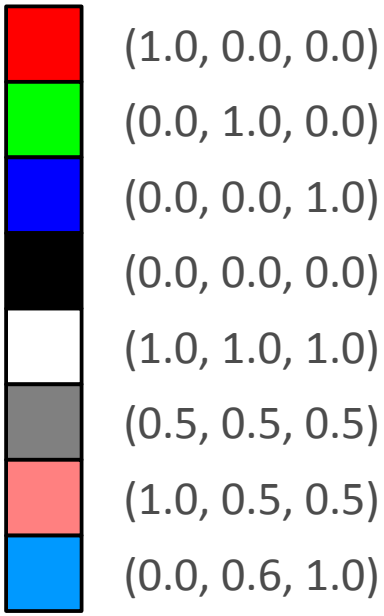
Prymityw

```
glBegin(GL_QUADS);
    glVertex3f(-1.0f, -1.0f, 0.0f);
    glVertex3f(1.0f, -1.0f, 0.0f);
    glVertex3f(1.0f, 1.0f, 0.0f);
    glVertex3f(-1.0f, 1.0f, 0.0f);
glEnd();
```



Kolor w OpenGL

- Częścią stanu OpenGL jest także **kolor <u>wierzchołków</u>**
  - Kolory określa się za pomocą komponentów ***RGB(A)***
    - Wartości są **znormalizowane** (0.0-1.0)
    - Czwarty kanał *A* może określać **przezroczystość**
    - Przykłady:

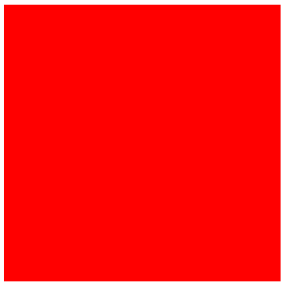


  - Używa się do tego celu funkcji:
    glColor3f(*float r, float g, float b*)
    - lub analogicznych (4f, 3fv, 4fv, ...)

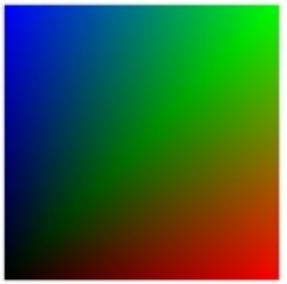
Kolor w OpenGL

- Przykłady:

```
glColor3f(1.0f, 0.0f, 0.0f);
glBegin(GL_QUADS);
        glVertex3f(-1.0f, -1.0f, 0.0f);
        glVertex3f(1.0f, -1.0f, 0.0f);
        glVertex3f(1.0f, 1.0f, 0.0f);
        glVertex3f(-1.0f, 1.0f, 0.0f);
glEnd();
```



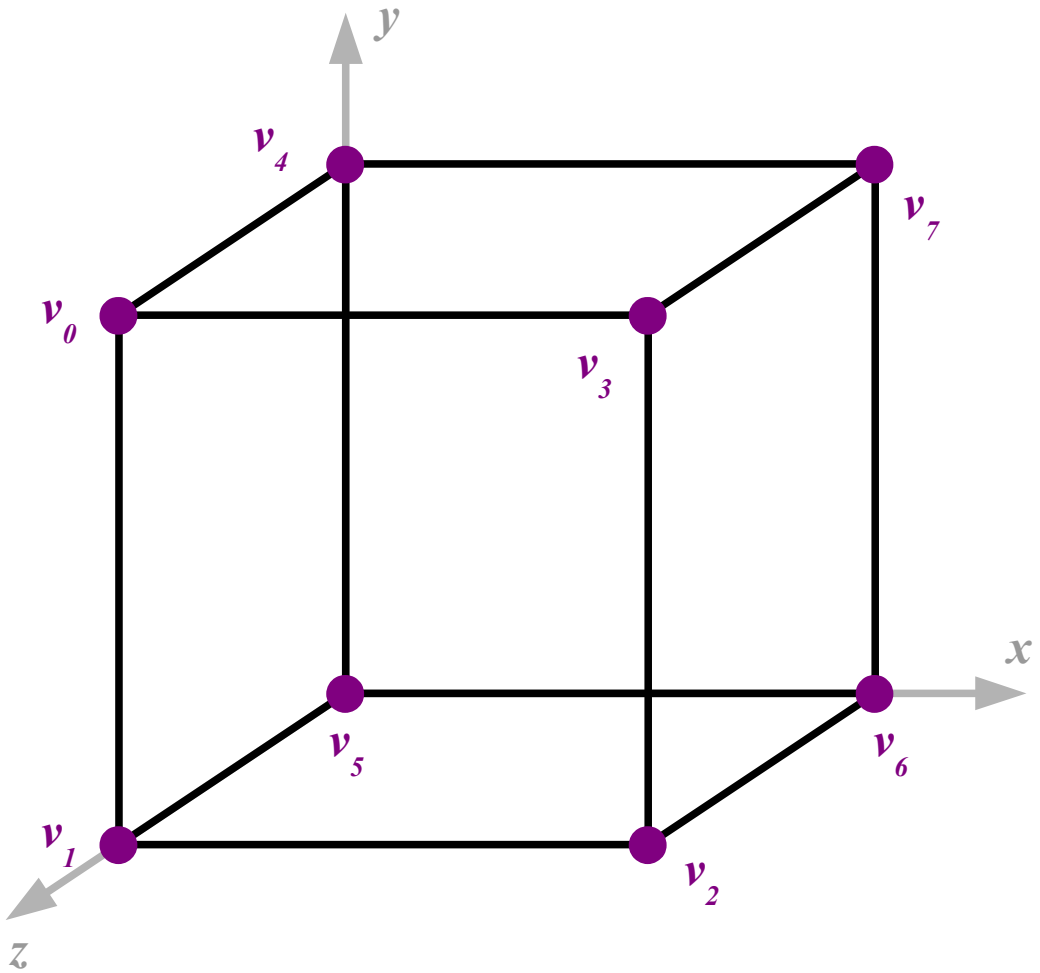
```
glBegin(GL_QUADS);
        glColor3f(0.0f, 0.0f, 0.0f);
        glVertex3f(-1.0f, -1.0f, 0.0f);
        glColor3f(1.0f, 0.0f, 0.0f);
        glVertex3f(1.0f, -1.0f, 0.0f);
        glColor3f(0.0f, 1.0f, 0.0f);
        glVertex3f(1.0f, 1.0f, 0.0f);
        glColor3f(0.0f, 0.0f, 1.0f);
        glVertex3f(-1.0f, 1.0f, 0.0f);
glEnd();
```



DEFINICJA GEOMETRII

Przykładowe dane

- Przykłady będą dotyczyły prostego **sześcianu**:

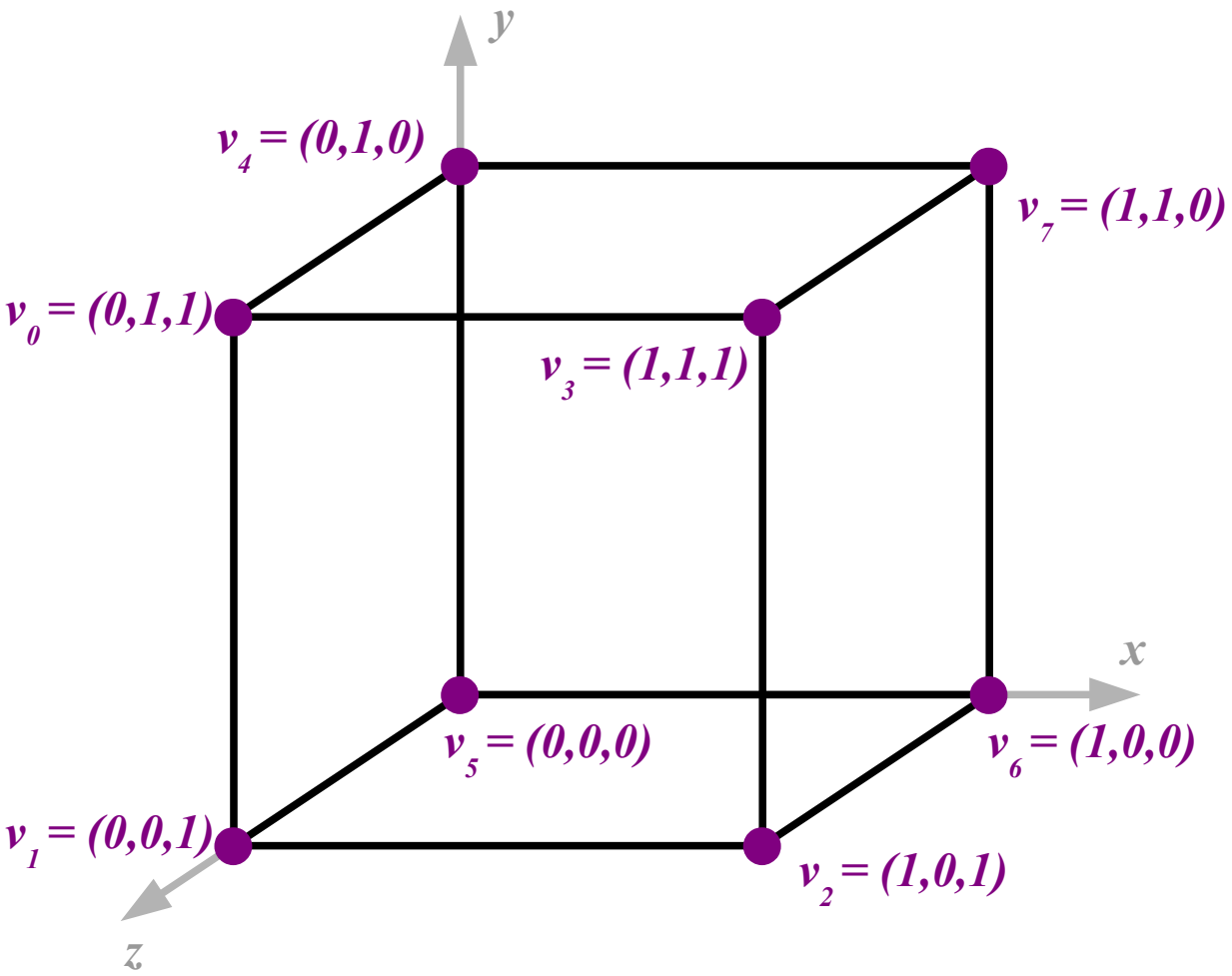


- Dla uproszczenia na razie pomijamy wszystko poza **pozycjami wierzchołków**

DEFINICJA GEOMETRII

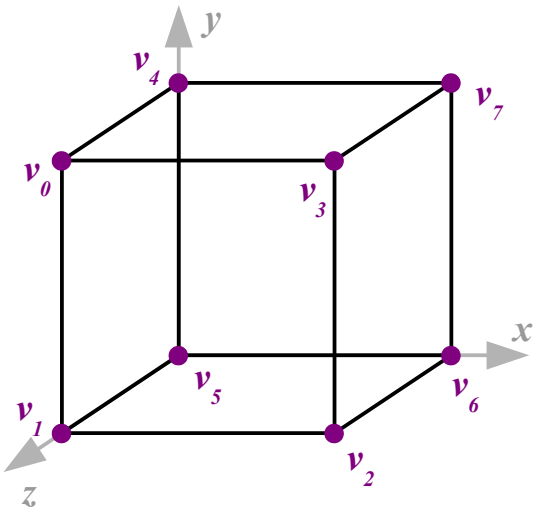
Przykładowe dane

- Pozycje wierzchołków:

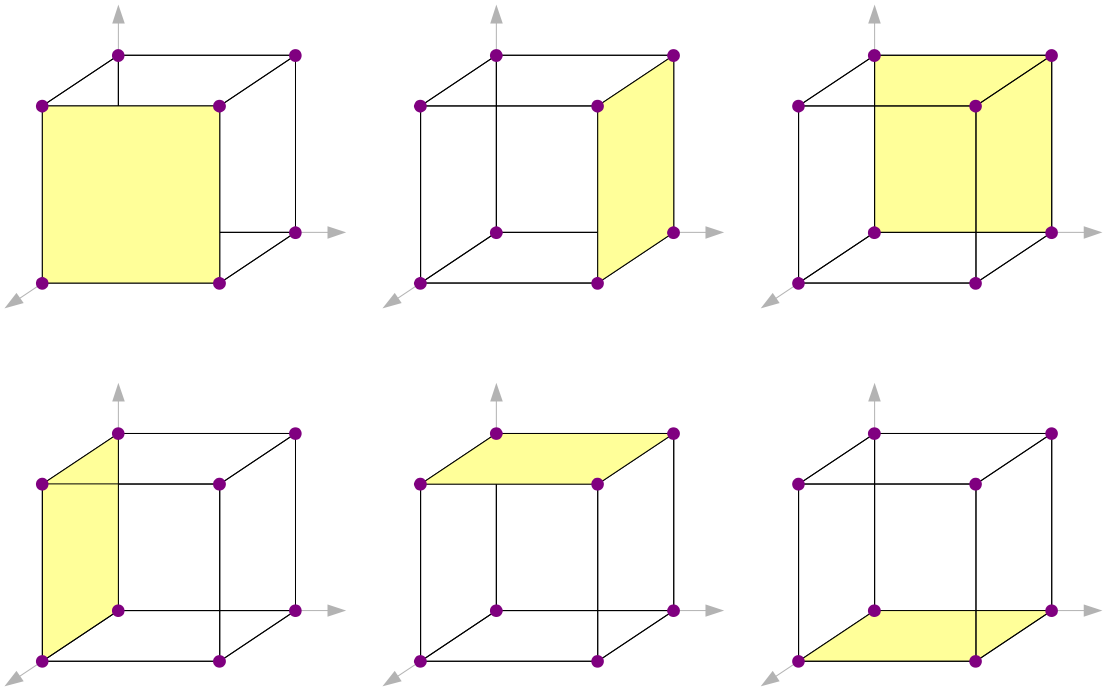


DEFINICJA GEOMETRII

Przykładowe dane



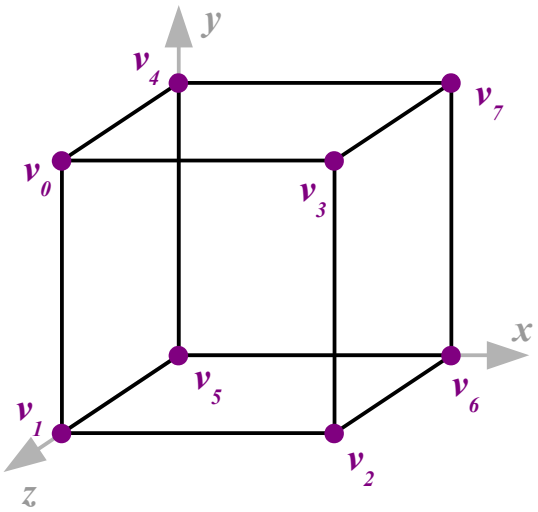
- Kolejność definicji **ścian** naszego sześcianu:



- Zakładamy użycie **prymitywu *GL_QUADS***
  - Oczywiście możliwe będzie zyskanie na wydajności np. używając *GL_QUAD_STRIP*, ale chcemy prosty przypadek
  - ***GL_QUADS* nie jest dostępny we współczesnych wersjach *OpenGL***, ale pozwala łatwiej zobrazować zagadnienie

DEFINICJA GEOMETRII

Przykładowe dane

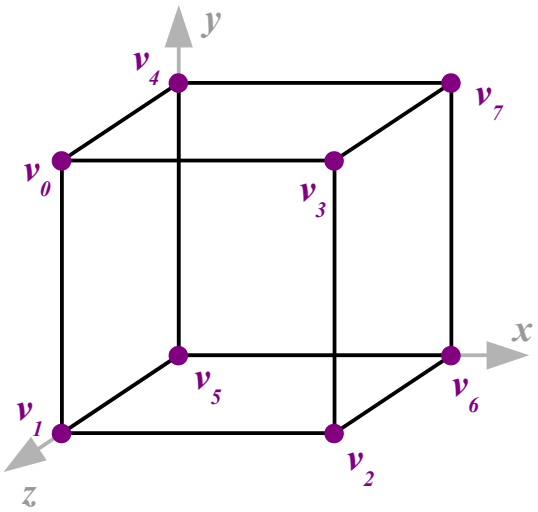


- Kolejność definicji **wierzchołków**:
  - v0 -> v1 -> v2 -> v3
    v3 -> v2 -> v6 -> v7
    v7 -> v6 -> v5 -> v4
    v4 -> v5 -> v1 -> v0
    v0 -> v3 -> v7 -> v4
    v1 -> v5 -> v6 -> v2
  - Czyli: **24 zestawy** po **3 współrzędne** po **4 bajty**
    - 24 × 3 × 4 = **288 bajtów**
  - Pamiętamy o **backface cullingu**
    - zakładamy że *front-face'y* definiowane są **przeciwnie do ruchu wskazówek zegara** (*CCW*)

IMMEDIATE MODE

Ogólna charakterystyka

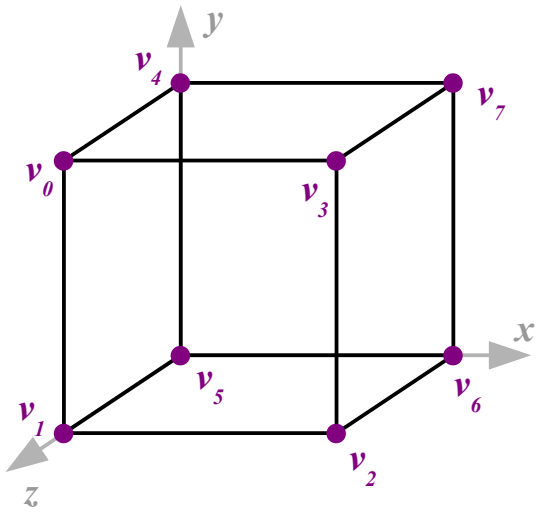
- **Najprostsze** podejście polegające na **bezpośredniej** definicji geometrii
- Wartości (np. współrzędne wierzchołków) dla **każdorazowego** żądania renderowania, **przekazywane są do serwera od nowa**
  - Oznacza to **ogromną ilość danych** przesyłaną co klatkę od klienta do serwera
  - Oznacza to **ogromną liczbę wywołań** funkcji *OpenGL*



IMMEDIATE MODE

Sposób użycia

- **Otwarcie** bloku definicji geometrii
  - Wskazanie pożądanego **prymitywu**
- **Definicja** geometrii
  - **Wierzchołki**, wektory **normalne**, współrzędne **tekstur**...
- **Zamknięcie** bloku
  - **Zlecenie renderowania**, ewentualne buforowanie

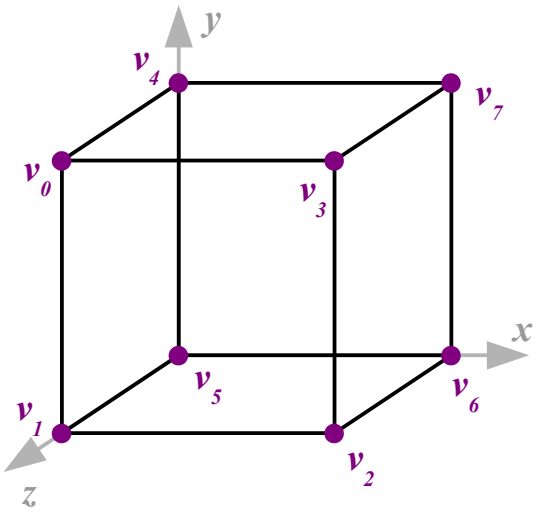


- Przykład dla pierwszego *quada*:

```
glBegin(GL_QUADS);
    glVertex3f(0.0f, 1.0f, 1.0f);
    glVertex3f(0.0f, 0.0f, 1.0f);
    glVertex3f(1.0f, 0.0f, 1.0f);
    glVertex3f(1.0f, 1.0f, 1.0f);
glEnd();
```

IMMEDIATE MODE

Przykład użycia



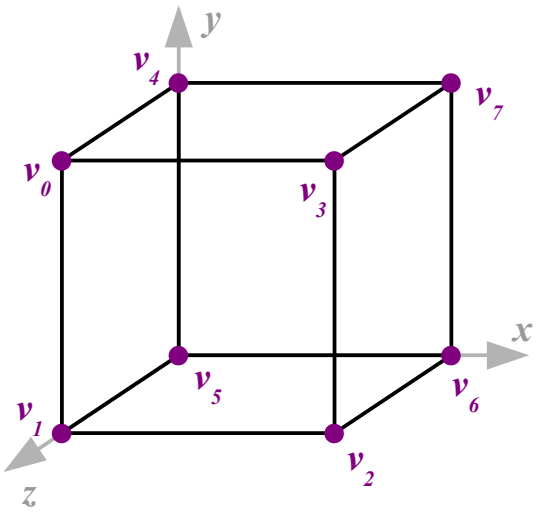
- Dla całego sześcianu:

```
glBegin(GL_QUADS);

        glVertex3f(0.0f, 1.0f, 1.0f);
        glVertex3f(0.0f, 0.0f, 1.0f);
        glVertex3f(1.0f, 0.0f, 1.0f);
        glVertex3f(1.0f, 1.0f, 1.0f);

        glVertex3f(1.0f, 1.0f, 1.0f);
        glVertex3f(1.0f, 0.0f, 1.0f);
        glVertex3f(1.0f, 0.0f, 0.0f);
        glVertex3f(1.0f, 1.0f, 0.0f);

        glVertex3f(1.0f, 1.0f, 0.0f);
        glVertex3f(1.0f, 0.0f, 0.0f);
        glVertex3f(0.0f, 0.0f, 0.0f);
        glVertex3f(0.0f, 1.0f, 0.0f);

        glVertex3f(0.0f, 1.0f, 0.0f);
        glVertex3f(0.0f, 0.0f, 0.0f);
        glVertex3f(0.0f, 0.0f, 1.0f);
        glVertex3f(0.0f, 1.0f, 1.0f);

        glVertex3f(0.0f, 1.0f, 0.0f);
        glVertex3f(0.0f, 1.0f, 1.0f);
        glVertex3f(1.0f, 1.0f, 1.0f);
        glVertex3f(1.0f, 1.0f, 0.0f);

        glVertex3f(0.0f, 1.0f, 1.0f);
        glVertex3f(1.0f, 1.0f, 1.0f);
        glVertex3f(1.0f, 1.0f, 0.0f);
        glVertex3f(0.0f, 1.0f, 0.0f);

        glVertex3f(0.0f, 0.0f, 1.0f);
        glVertex3f(0.0f, 0.0f, 0.0f);
        glVertex3f(1.0f, 0.0f, 0.0f);
        glVertex3f(1.0f, 0.0f, 1.0f);

glEnd();
```

- W **każdej klatce** wysyłamy do serwera pozycje **wszystkich wierzchołków**
  - Nie zmieniają się – **po co?**
- Powtarzające się wierzchołki wysyłane są **kilkukrotnie**
  - Już zostały przesłane – **po co?**

DISPLAY LIST

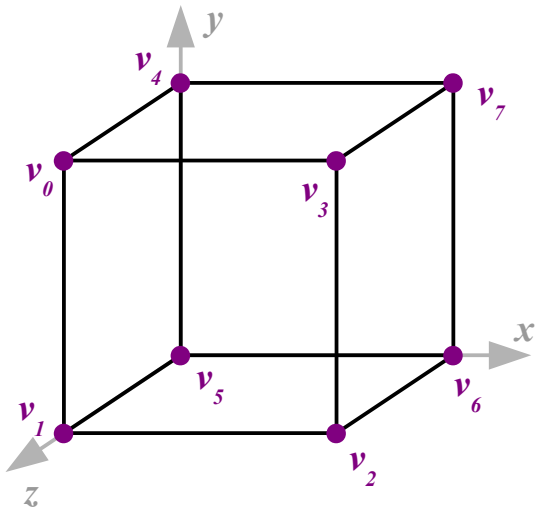
Ogólna charakterystyka



- **Skompilowana** sekwencja instrukcji zmieniających **stan serwera**
- Przechowywana **w pamięci karty graficznej**
- Raz utworzona **nie może być modyfikowana**
  - **Nie nadaje się do animacji bryły**
- Sterownik odpowiada za to, by lista została wykonana **efektywnie**
- Każda lista ma przypisany **identyfikator**, który służy jej późniejszemu wywołaniu (żądaniu renderowania)

DISPLAY LIST

Sposób użycia

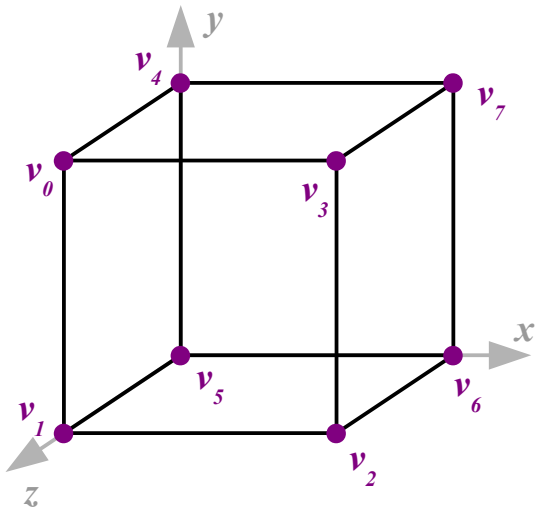


- **Utworzenie** display listy
  - Żądanie **identyfikatora**
- **Kompilacja** display listy
  - **Wierzchołki**, wektory **normalne**, współrzędne **tekstur**...
  - Można dołączyć inne **zmiany stanu serwera**
- **Wywołanie** display listy
  - **Żądanie** renderowania z użyciem **identyfikatora**
- Przykład dla pierwszego *quada*:

```
GLuint id = glGenLists(1);
glNewList(id, GL_COMPILE);
        glBegin(GL_QUADS);
                glVertex3f(0.0f, 1.0f, 1.0f);
                glVertex3f(0.0f, 0.0f, 1.0f);
                glVertex3f(1.0f, 0.0f, 1.0f);
                glVertex3f(1.0f, 1.0f, 1.0f);
        glEnd();
glEndList();
// Renderowanie:
glCallList(id);
```

DISPLAY LIST

Przykład użycia



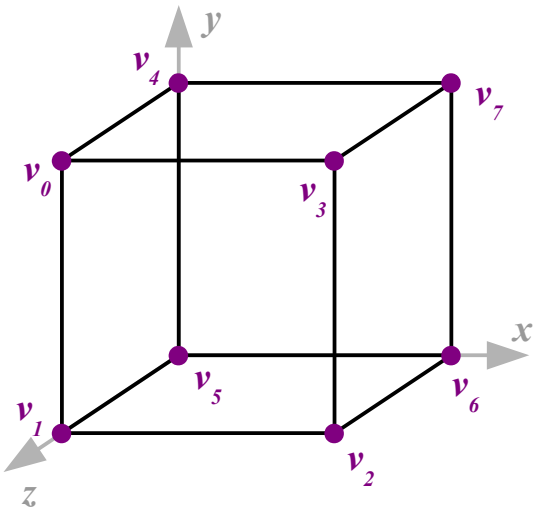
- Dla całego sześcianu:

```
GLuint id = glGenLists(1);
glNewList(id, GL_COMPILE);
        glBegin(GL_QUADS);
                glVertex3f(0.0f, 1.0f, 1.0f);
                glVertex3f(0.0f, 0.0f, 1.0f);
                glVertex3f(1.0f, 0.0f, 1.0f);
                glVertex3f(1.0f, 1.0f, 1.0f);
                glVertex3f(1.0f, 1.0f, 1.0f);
                glVertex3f(1.0f, 0.0f, 1.0f);
                glVertex3f(1.0f, 0.0f, 0.0f);
                glVertex3f(1.0f, 1.0f, 0.0f);
                glVertex3f(1.0f, 1.0f, 0.0f);
                glVertex3f(1.0f, 0.0f, 0.0f);
                glVertex3f(0.0f, 0.0f, 0.0f);
                glVertex3f(0.0f, 1.0f, 0.0f);
                glVertex3f(0.0f, 1.0f, 0.0f);
                glVertex3f(0.0f, 0.0f, 0.0f);
                glVertex3f(0.0f, 0.0f, 1.0f);
                glVertex3f(0.0f, 1.0f, 1.0f);
                glVertex3f(1.0f, 1.0f, 0.0f);
                glVertex3f(1.0f, 1.0f, 1.0f);
                glVertex3f(0.0f, 1.0f, 1.0f);
                glVertex3f(1.0f, 1.0f, 1.0f);
                glVertex3f(1.0f, 1.0f, 0.0f);
                glVertex3f(0.0f, 1.0f, 0.0f);
                glVertex3f(0.0f, 0.0f, 1.0f);
                glVertex3f(0.0f, 0.0f, 0.0f);
                glVertex3f(1.0f, 0.0f, 0.0f);
                glVertex3f(1.0f, 0.0f, 1.0f);
        glEnd();
glEndList();

//...

// Dla każdej klatki:

glCallList(id);
```

- Pozycje **wszystkich wierzchołków** wysyłamy **tylko raz**
- Podczas renderowania kolejnych klatek korzystamy z **danych znajdujących się w pamięci graficznej**
  - Znacząco zyskuje na tym **wydajność**
- Możliwości list wykraczają poza zwykłą definicję geometrii
  - Możliwość użycia innych **instrukcji zmieniających stan**
- Nie ma możliwości **modyfikacji listy** po jej skompilowaniu

VERTEX ARRAY

Ogólna charakterystyka

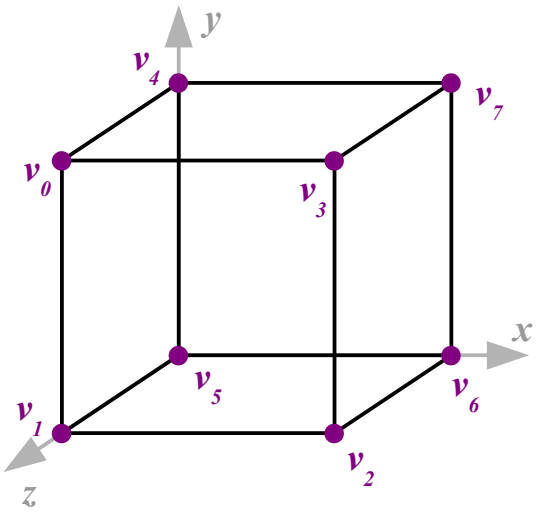


- Ciągła **tablica po stronie klienta** (w pamięci głównej) zawierająca dane wierzchołków
- Możliwość dowolnej **manipulacji** zawartości
- Możliwość **wybiórczego** renderowania
  - Rozwiązuje problem **redefinicji** powtarzających się wierzchołków, co daje **oszczędność pamięci**
- **Interleaved arrays**, czyli łączone tablice dla różnych danych *per-vertex*
  - Np. pozycja, wektor normalny, kolor, współrzędne tekstury...
  - Wszystko w jednej tablicy (co ma i zalety, i wady)

VERTEX ARRAY

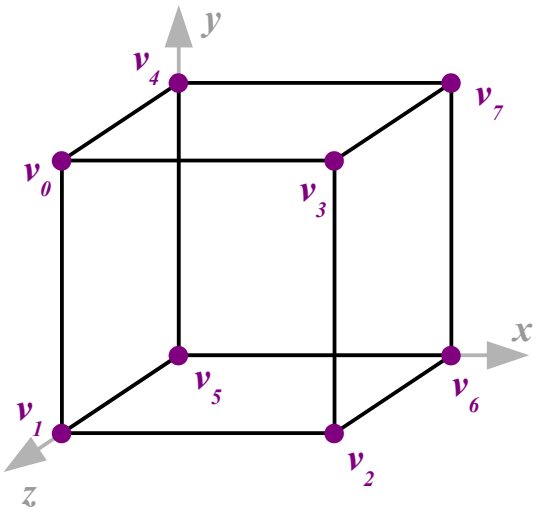
Sposób użycia

- **Utworzenie** tablicy z wartościami
  - Zwyczajna tablica po stronie klienta
- **Wskazanie** serwerowi miejsca w pamięci, gdzie znajdują się dane
  - Przekazanie adresu tablicy
- **Żądanie** renderowania
  - Można określić, **które** z danych i **w jakiej kolejności** nas interesują



VERTEX ARRAY

Sposób użycia (c.d.)



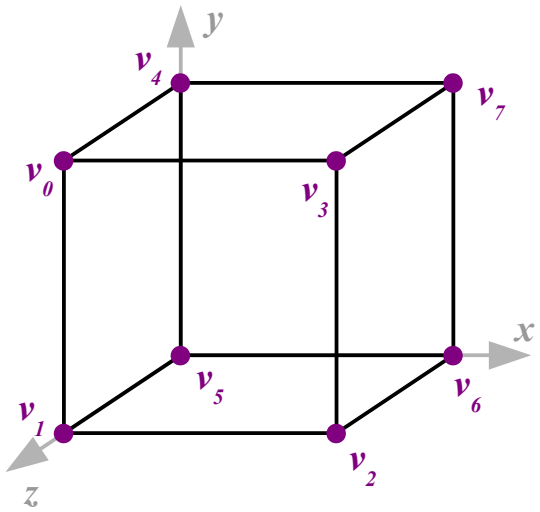
- Przykład dla pierwszego *quada*:

```
GLfloat buffer[] = {
        0.0f, 1.0f, 0.0f,
        0.0f, 1.0f, 1.0f,
        0.0f, 0.0f, 1.0f,
        1.0f, 0.0f, 1.0f
};

// Renderowanie:
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, buffer);

glDrawArrays(GL_QUADS, 0, 4);

glDisableClientState(GL_VERTEX_ARRAY);
```

- Aby wskazać gdzie przechowywane są wartości **innych atrybutów**, można posłużyć się funkcjami:
  - glNormalPointer(), glTexCoordPointer(), glVertexAttribPointer(), ...

VERTEX ARRAY

Wybiórcze renderowanie

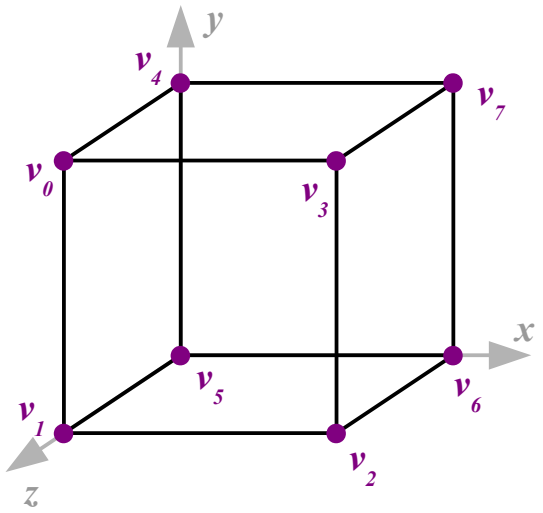


- **Wybiórcze renderowanie** można zrealizować na kilka różnych sposobów:
  - **Podzakres** tablicy *od-do*:
    - **glDrawArrays**(primitive, from, to)
  - Wskazanie **indeksów** wierzchołków:
    - **glDrawElements**(primitive, num_indices, index_type, indices)
    - Dzięki temu **nie musimy redefiniować** wierzchołków!
    - Przykład – **cały** sześcian:

```
GLfloat buffer[] = {
        0.0f, 1.0f, 1.0f,   0.0f, 0.0f, 1.0f,   1.0f, 0.0f, 1.0f,   1.0f, 1.0f, 1.0f,
        0.0f, 1.0f, 0.0f,   0.0f, 0.0f, 0.0f,   1.0f, 0.0f, 0.0f,   1.0f, 1.0f, 0.0f
};
GLubyte indices[] = {
        0, 1, 2, 3,   3, 2, 6, 7,   7, 6, 5, 4,
        4, 5, 1, 0,   0, 3, 7, 4,   1, 5, 6, 2
};

// Renderowanie:
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, buffer);

glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, indices);

glDisableClientState(GL_VERTEX_ARRAY);
```

VERTEX ARRAY

Wybiórcze renderowanie

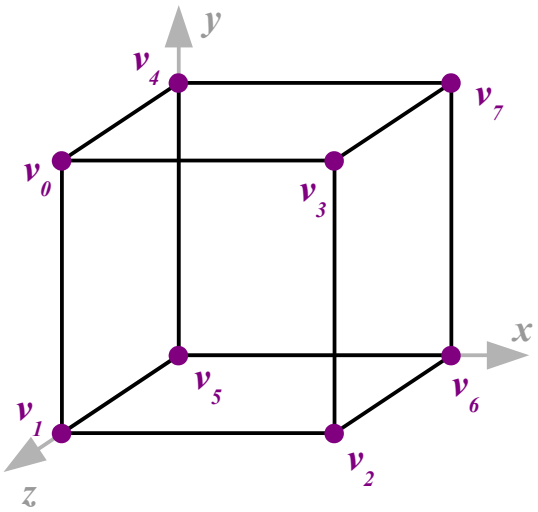


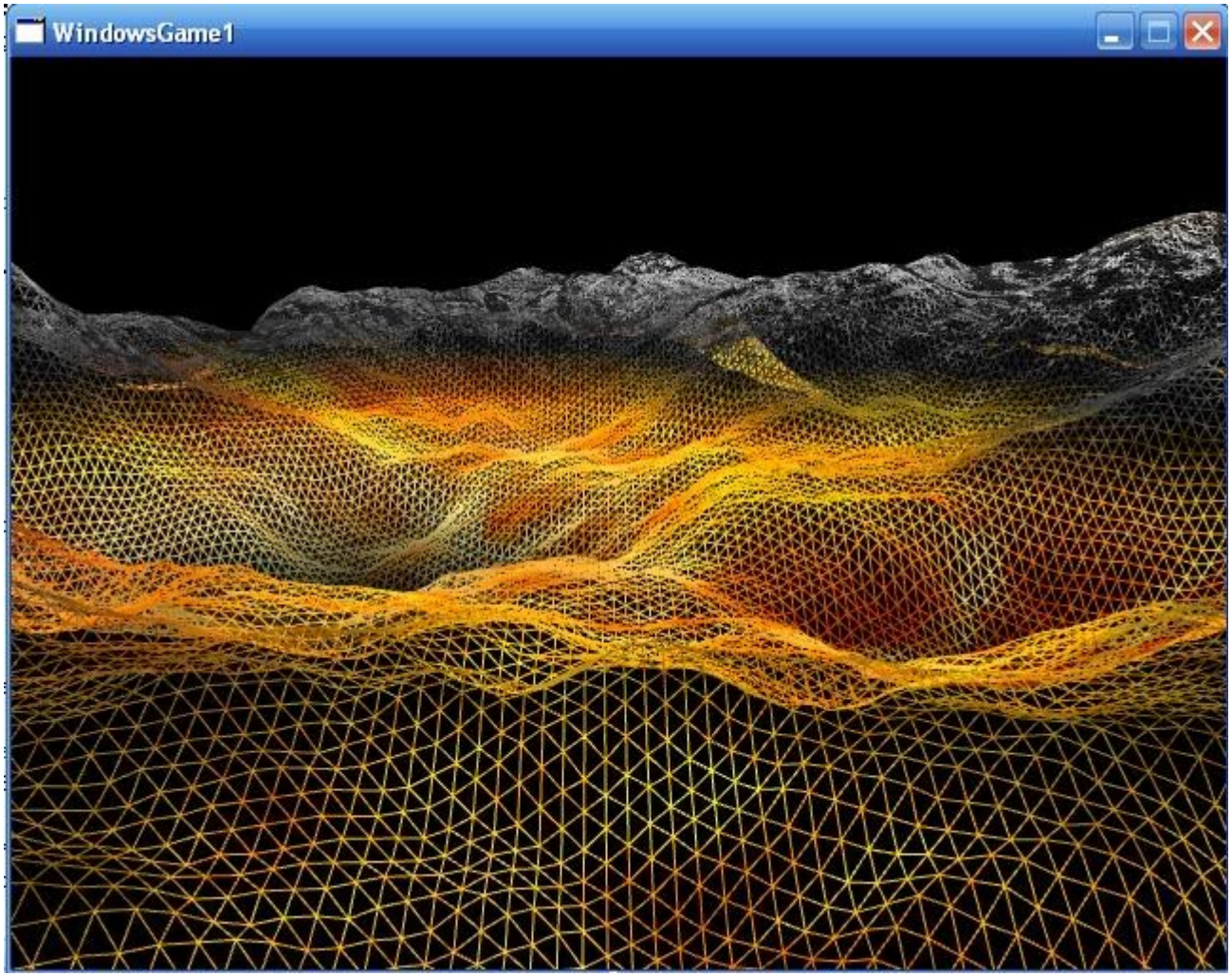
- **Wybiórcze renderowanie** można zrealizować na kilka różnych sposobów (c.d.):
  - Wskazanie **indeksów** wierzchołków oraz ich **zakresu**:
    - **glDrawRangeElements**(primitive, start, end, num_indices, index_type, indices)
    - start i end określają **zakres wartości elementów z tablicy indeksów** (nie zakres tablicy, który zostanie użyty!)
    - Służy to do małej **optymalizacji** – *OpenGL* będzie wiedział, jakiego zakresu indeksów się spodziewać bez konieczności iterowania po tablicy indeksów. W ten sposób może lepiej zaplanować odczyt z *VA*.
- Co cechuje *Vertex Arrays*?
  - Możliwość **dowolnej kontroli** wartości w trakcie renderowania
    - Przydatne przy **animacji bryły**
  - Konieczność **przesłania tablicy** do pamięci karty graficznej
  - Dużo **mniej odwołań** do funkcji *OpenGL*

WYBIÓRCZE RENDEROWANIE

Przykład zastosowania

- Inny przykład praktycznego wykorzystania selektywnego renderowania geometrii z użyciem indeksów wierzchołków:
  - **Level of detail** (*LOD*) dla map wysokości

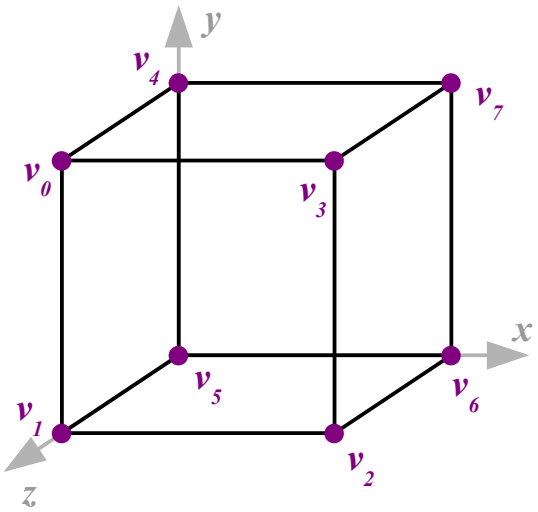




Źródło obrazu:
http://windows-tech.info

VERTEX ARRAY

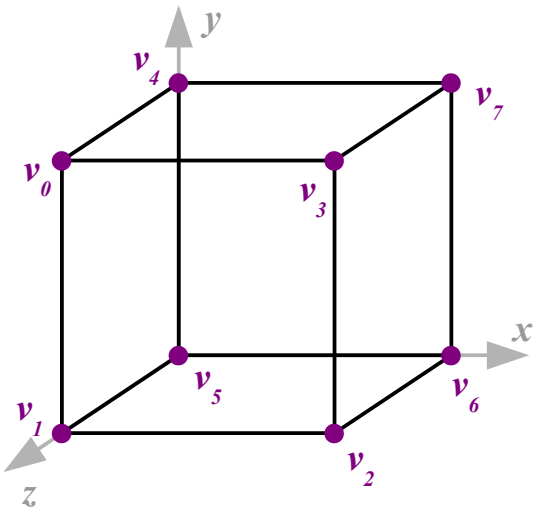
Dodatkowe atrybuty



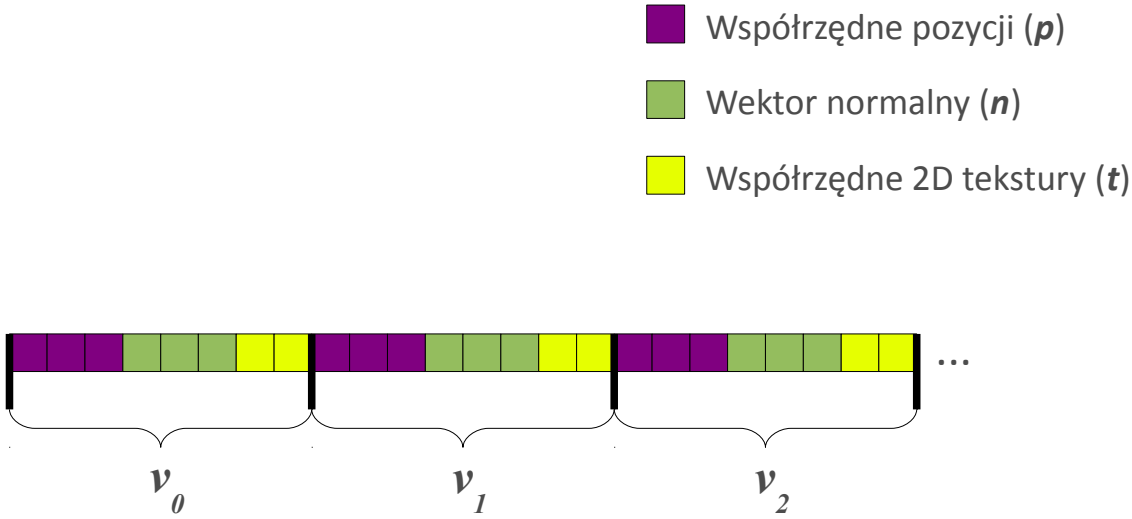
- Przekazywanie wartości **dodatkowych atrybutów** wierzchołków
  - **Nie tylko** pozycja (współrzędne) jest wartością *per-vertex*
    - Wektory normalne, współrzędne tekstur, wektory styczne, kolor, stopień zniszczenia, wiek, stopień deformacji, zabrudzenie, następna pozycja, ...
    - Możemy definiować **własne atrybuty**, które wykorzystamy w naszych shaderach
  - Są **dwa** podejścia rozwiązania tej kwestii:
    - **Oddzielne tablice** dla każdego z atrybutów
    - **Jedna tablica** zawierająca przeplatające się wartości atrybutów (tzw. *interleaved arrays*)

INTERLEAVED ARRAYS

Sposób działania



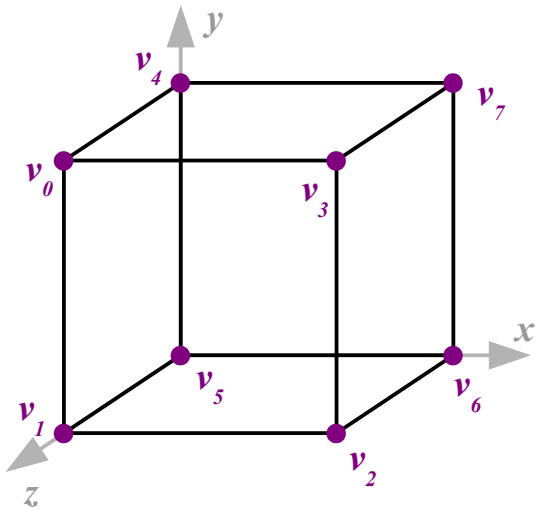
- *Interleaved arrays* – przykład:



- Parametry każdego atrybutu:
  - *offset* – **odległość** w bajtach **od początku** wierzchołka
  - *stride* – **odległość** w bajtach **pomiędzy** wierzchołkami
  - Zakładając, że szerokość każdej wartości to 4B, mamy:
    - *offset*(**p**) = 0, *stride*(**p**) = 32
    - *offset*(**n**) = 12, *stride*(**n**) = 32
    - *offset*(**t**) = 24, *stride*(**t**) = 32

INTERLEAVED ARRAYS

Sposób użycia



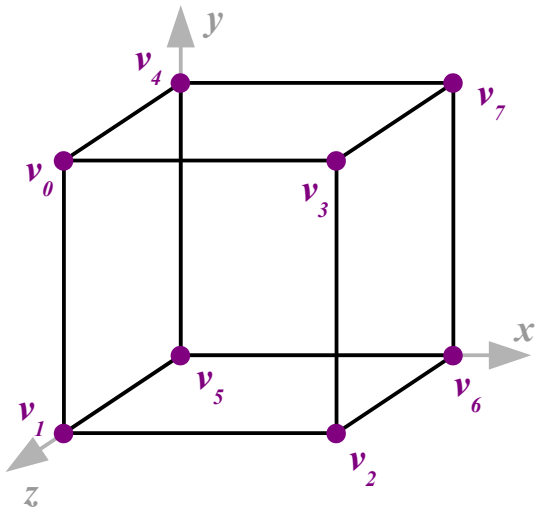
- **Reprezentacja** *interleaved array* **w pamięci głównej** z punktu widzenia programu
  - Wygodnym podejściem jest użycie **struktur**
  - *C++* gwarantuje, że struktura zajmuje **ciągły obszar pamięci**, a składowe mają zachowaną **kolejność**

```
typedef struct {
        GLfloat pos[3];
        GLfloat normal[3];
        GLfloat tex[2];
} SVertex;

SVertex buffer[];

glVertexPointer(3, GL_FLOAT, sizeof(SVertex), buffer);
glNormalPointer(GL_FLOAT, sizeof(SVertex), buffer + 3 * sizeof(GLfloat));
glTexCoordPointer(2, GL_FLOAT, sizeof(SVertex), buffer + 6 * sizeof(GLfloat));
```

- Zamiast arytmetyki wskaźników (mogącej doprowadzić do problemów), warto wykorzystać po prostu:
  - &buffer[0].pos, &buffer[0].normal, &buffer[0].tex

INTERLEAVED ARRAYS

Dodatkowe informacje

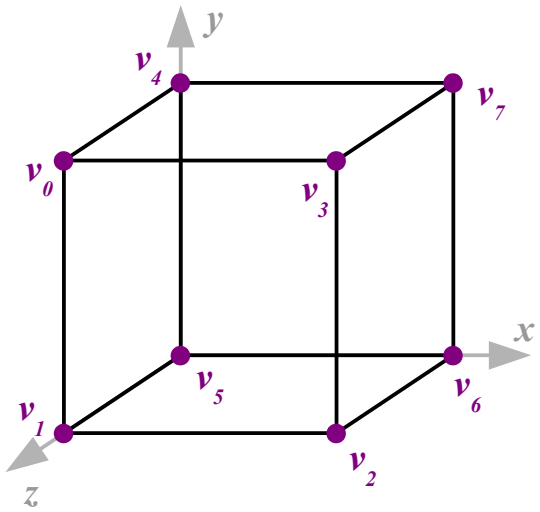


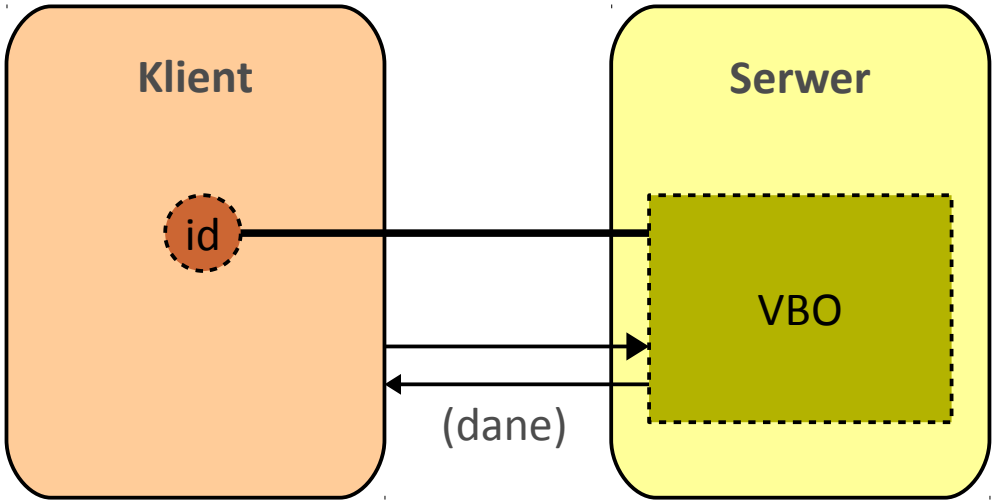
- *Interleaved arrays* – dodatkowe informacje:
  - Jeśli np. planujemy często zmieniać pozycję wierzchołków, pozostawiając pozostałe atrybuty bez zmian, warto użyć oddzielnych tablic
  - Warto dążyć do tego, by *stride* był równy **wielokrotności 32**. Wiele kart graficznych potrafi wtedy lepiej organizować odczyt z pamięci
    - Nawet kosztem zwiększonego zużycia pamięci, wprowadzając **padding**!
  - Możemy określić ***stride* jako 0**, jeśli nasze dane są ciasno upakowane (żadnego paddingu) i tablice zawierają wartości tylko pojedynczych atrybutów
  - Jeśli nasz układ danych jest standardowy, można użyć funkcji **glInterleavedArrays**() w celu wyboru tego układu (oszczędza liczbę wywołań funkcji *OpenGL*)

VBO

Ogólna charakterystyka

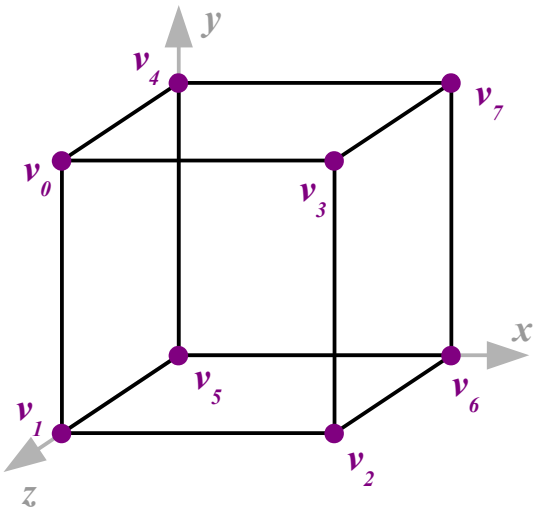
- ***Vertex Buffer Object* (*VBO*)** – technika pozwalająca na łatwe **manipulowanie zawartością pamięci karty graficznej**
- Siostrzana technika do ***Frame Buffer Object* (*FBO*)** i ***Pixel Buffer Object* (*PBO*)**
- Pozwala zarezerwować, wypełnić, odczytać, modyfikować zawartość bufora
- Po stronie klienta przechowywany jest jedynie **identyfikator** bufora





VBO

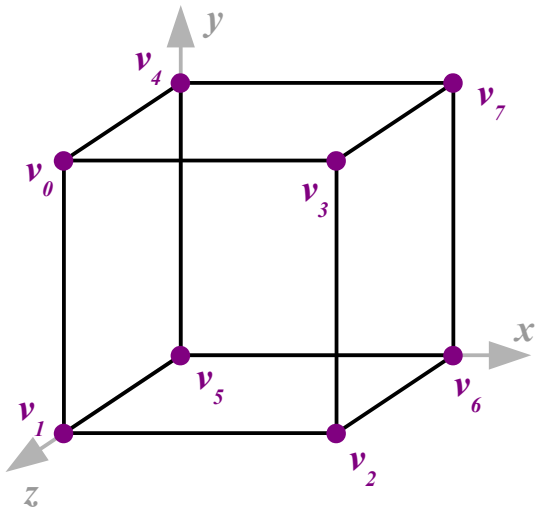
Sposób użycia



- **Wygenerowanie** nowego bufora
  - Otrzymujemy **id** bufora
- **Bindowanie** bufora
  - Wybór bufora, który ma stać się **aktywny**
  - Określenie **rodzaju** bufora (*GL_ARRAY_BUFFER*)
- **Alokacja** bufora
  - **Przekazanie danych** do pamięci karty graficznej
  - Określenie **charakteru danych**: czy będą często modyfikowane, czy też nie...
    - Wpływa na **optymalny przydział pamięci**
- **Wskazanie** miejsc w pamięci
  - Nie przekazujemy wskaźników (bo ich nie mamy), tylko określamy liczbowo *offset* i *stride* dla *VBO*
- **Żądanie renderowania**
  - Można określić, **które** z danych i **w jakiej kolejności** nas interesują

VBO

Przykład użycia

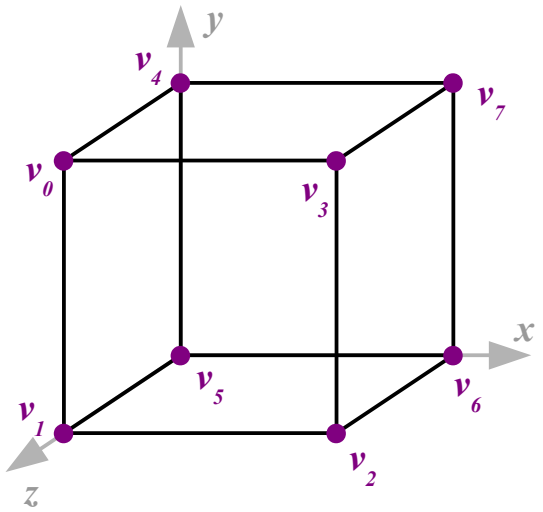


- Przykład dla pierwszego *quada*:

```
GLfloat buffer[] = {0.0f, 1.0f, 0.0f,  0.0f, 1.0f, 1.0f,  0.0f, 0.0f, 1.0f,  1.0f, 0.0f, 1.0f};

GLuint vbo;
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);

glBufferData(GL_ARRAY_BUFFER, sizeof(buffer), buffer, GL_STATIC_DRAW);
delete[] buffer; // Oczywiście jest to błędne użycie (alokacja podczas kompilacji)

glBindBuffer(GL_ARRAY_BUFFER, 0);

// Renderowanie:
glBindBuffer(GL_ARRAY_BUFFER, vbo);

glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, 0);

glDrawArrays(GL_QUADS, 0, 4);

glDisableClientState(GL_VERTEX_ARRAY);
glBindBuffer(GL_ARRAY_BUFFER, 0);

// Sprzątanie:
glDeleteBuffers(1, &vbo);
```

VBO

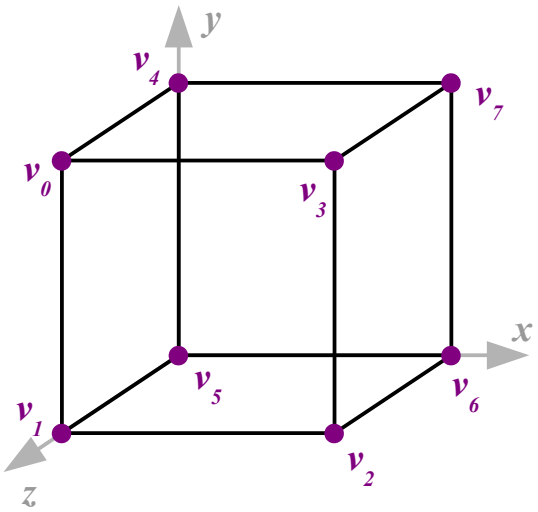
Modyfikacja zawartości



- Istnieje możliwość **modyfikacji** zawartości VBO:
  - Ponowna **alokacja** pamięci i wysłanie danych
    - **glBufferData**(target, size, data, usage)
  - Poprzez **zastąpienie** części danych
    - **glBufferSubData**(target, offset, size, data)
    - Pamięć **nie jest realokowana**!
      Warto użyć nawet, jeśli zamieniamy całą zawartość bufora.
  - Poprzez **mapowanie** pamięci
    - **glMapBuffer**(target, access)
    - **glUnmapBuffer**(target)
    - Otrzymujemy **wskaźnik do pamięci**, możemy dowolnie ją czytać i zmieniać
    - Kosztowna synchronizacja z *GPU*
  - Warto zastanowić się nad **wydajnością i synchronizacją**
    - Wykorzystanie techniki **podwójnego buforowania** (dwa *VBO*, które zamieniamy miejscami – rysujemy z jednego, drugi uaktualniamy) może rozwiązać problem synchronizacji

IBO

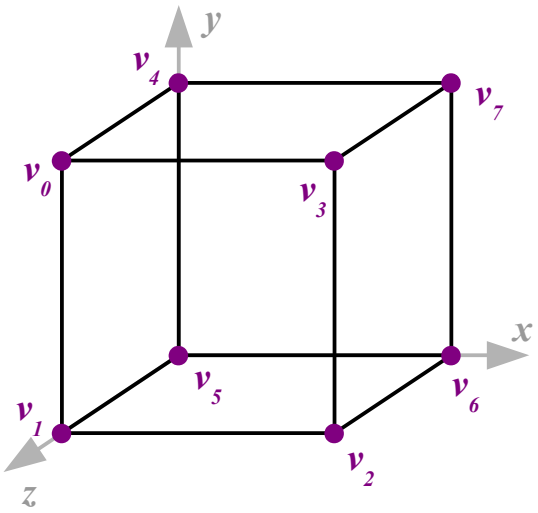
Indeksowanie VBO



- **Wybiórcze renderowanie** z *VBO* jest **jeszcze lepsze**
  - Mamy możliwość stworzenia bufora indeksów: ***Index Buffer Object* (*IBO*)**
  - *IBO* działa **analogicznie** do *VBO*:
    - Musimy go stworzyć, wybrać, zaalokować, wypełnić, zwolnić
    - *GL_ELEMENT_ARRAY_BUFFER*
  - *IBO* jest przechowywany **w pamięci karty graficznej** – odczyt podczas renderowania nie wymaga przesyłania danych z pamięci głównej!

IBO

Przykład użycia



- Przykład – **cały sześcian**:

```
GLfloat buffer[] = {
        0.0f, 1.0f, 1.0f,   0.0f, 0.0f, 1.0f,   1.0f, 0.0f, 1.0f,   1.0f, 1.0f, 1.0f,
        0.0f, 1.0f, 0.0f,   0.0f, 0.0f, 0.0f,   1.0f, 0.0f, 0.0f,   1.0f, 1.0f, 0.0f
};
GLubyte indices[] = {
        0, 1, 2, 3,   3, 2, 6, 7,   7, 6, 5, 4,
        4, 5, 1, 0,   0, 3, 7, 4,   1, 5, 6, 2
};

GLuint vbo, ibo;

glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(buffer), buffer, GL_STATIC_DRAW);
delete[] buffer;

glGenBuffers(1, &ibo);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
delete[] indices;

// Renderowanie:
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, buffer);
glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, 0);
glDisableClientState(GL_VERTEX_ARRAY);

// Sprzątanie:
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
glDeleteBuffers(1, &vbo);
glDeleteBuffers(1, &ibo);
```

Zachodniopomorski
Uniwersytet
Technologiczny
w Szczecinie

http://bazyluk.net/dydaktyka

Bartosz Bazyluk

Definicja Geometrii

Wstęp do syntezy trójwymiarowej grafiki komputerowej

Wydział
Informatyki

Grafika Komputerowa i Wizualizacja, Informatyka S1, II Rok