Zachodniopomorski
Uniwersytet
Technologiczny
w Szczecinie

Bartosz Bazyluk

OpenGL

Współczesne podejście do programowania grafiki

Część I: Definicja geometrii

Programowanie Gier Komputerowych, Informatyka S1, III Rok

Wydział
Informatyki

PLAN WYKŁADU

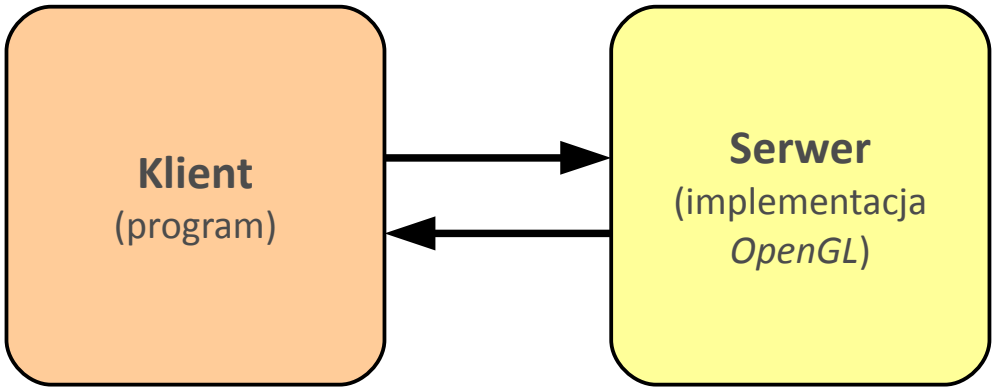
- Sposoby definicji geometrii
  - Immediate mode
  - Display List
  - Vertex Array
    - Interleaved array
  - Vertex Buffer Object
    - Index Buffer Object

OpenGL

Architektura działania

Jest to **warstwa programowa**. Za implementacją *OpenGL* znajduje się **sterownik karty graficznej**, a dopiero dalej właściwy sprzęt.

- Architektura wykorzystania *OpenGL* jako **warstwy pośredniej** pomiędzy aplikacją, a kartą graficzną opiera się o **model klient-serwer**



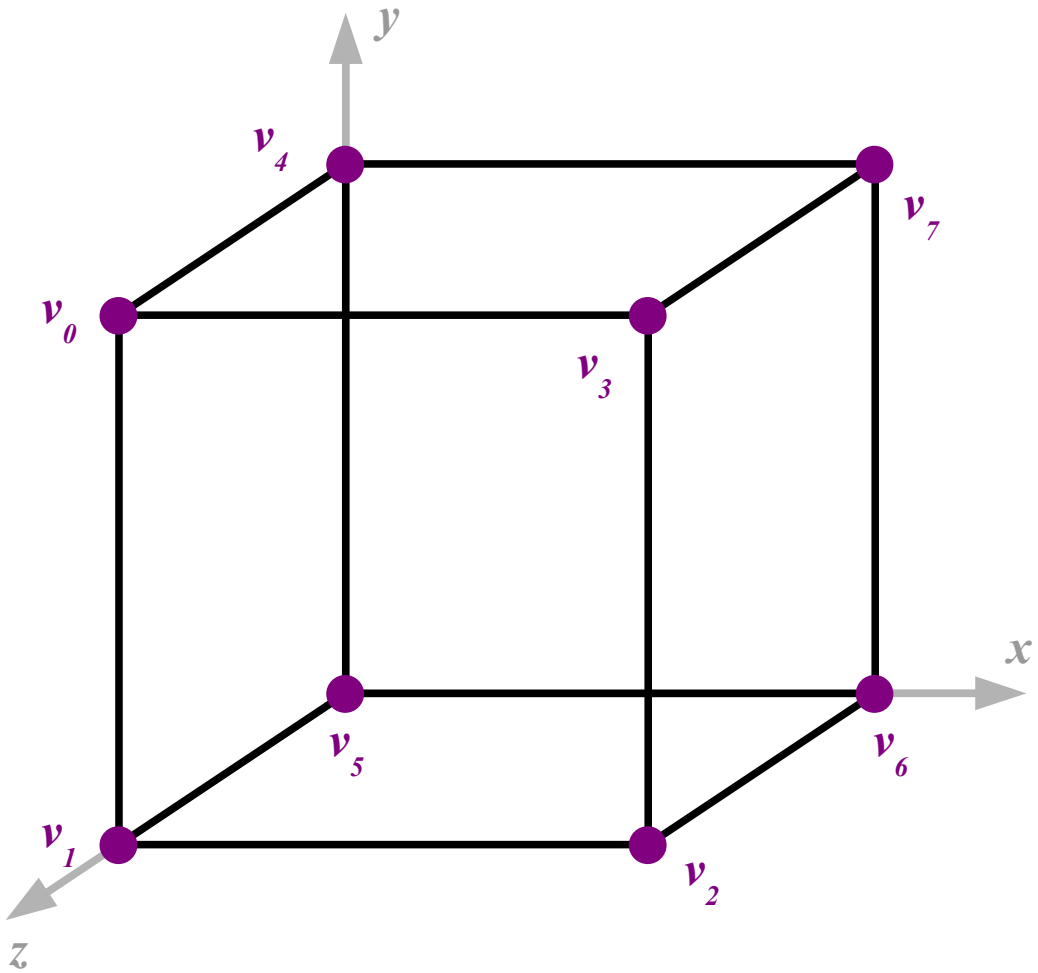
- Zlecenia renderowania
- Opis sceny
- Używa **pamięci głównej**
- Logika programu
- Wyświetlenie rezultatu

- Konteksty
- Renderowanie
- Komunikacja ze sterownikiem
- Dostęp do *GPU* i **pamięci graficznej**

DEFINICJA GEOMETRII

Przykładowe dane

- Przykłady będą dotyczyły prostego **sześcianu**:

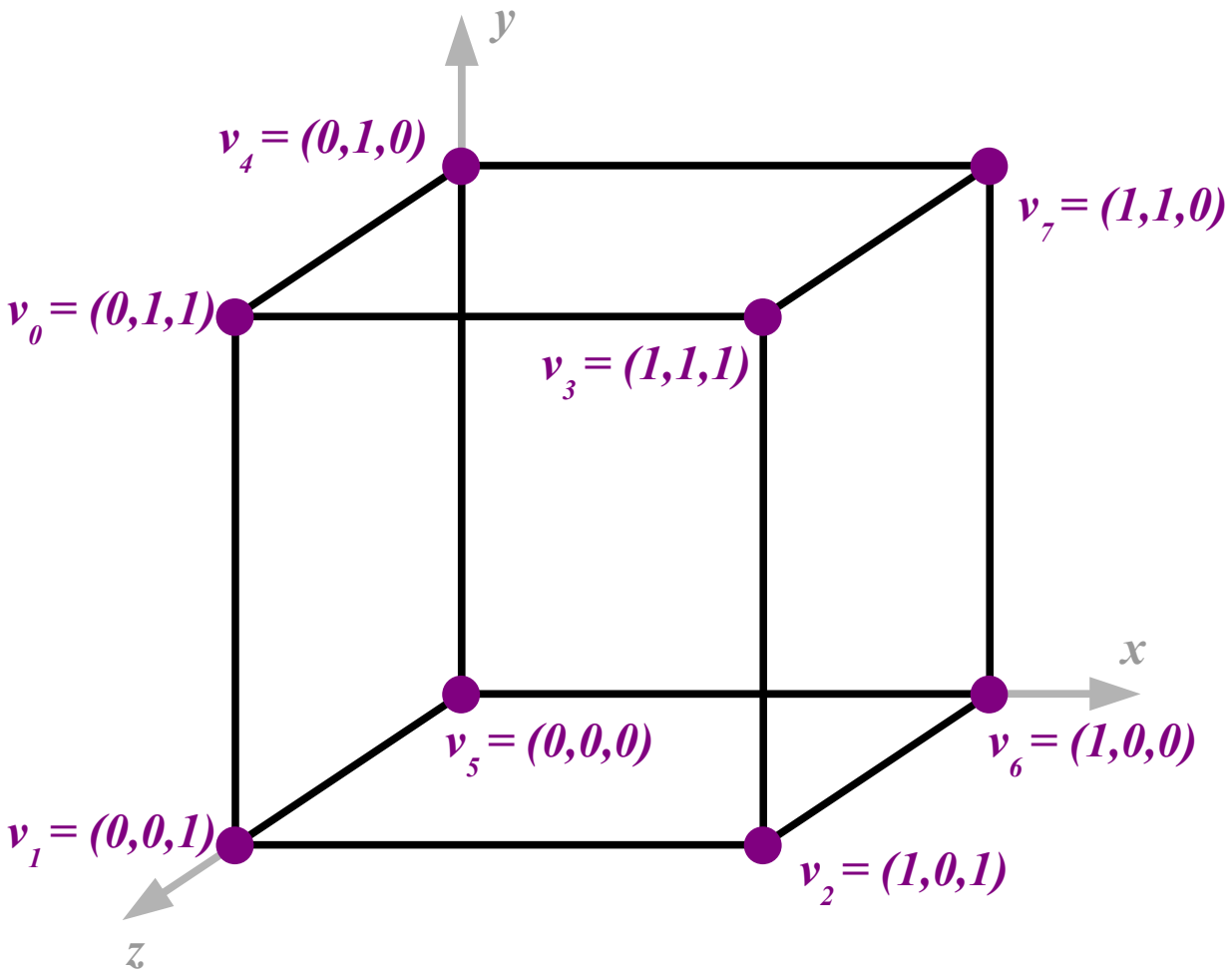


- Dla uproszczenia na razie pomijamy wszystko poza **pozycjami wierzchołków**

DEFINICJA GEOMETRII

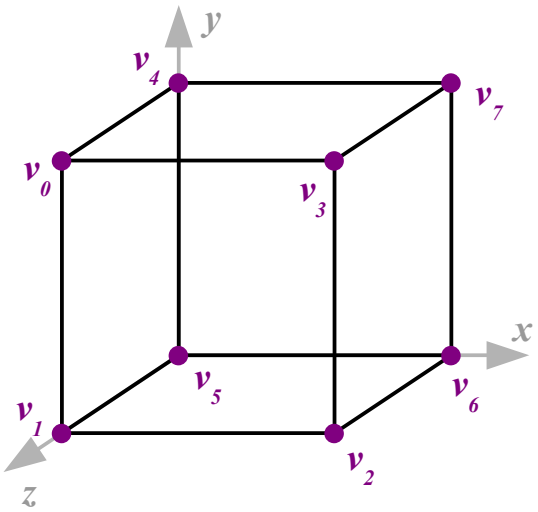
Przykładowe dane

- Pozycje wierzchołków:

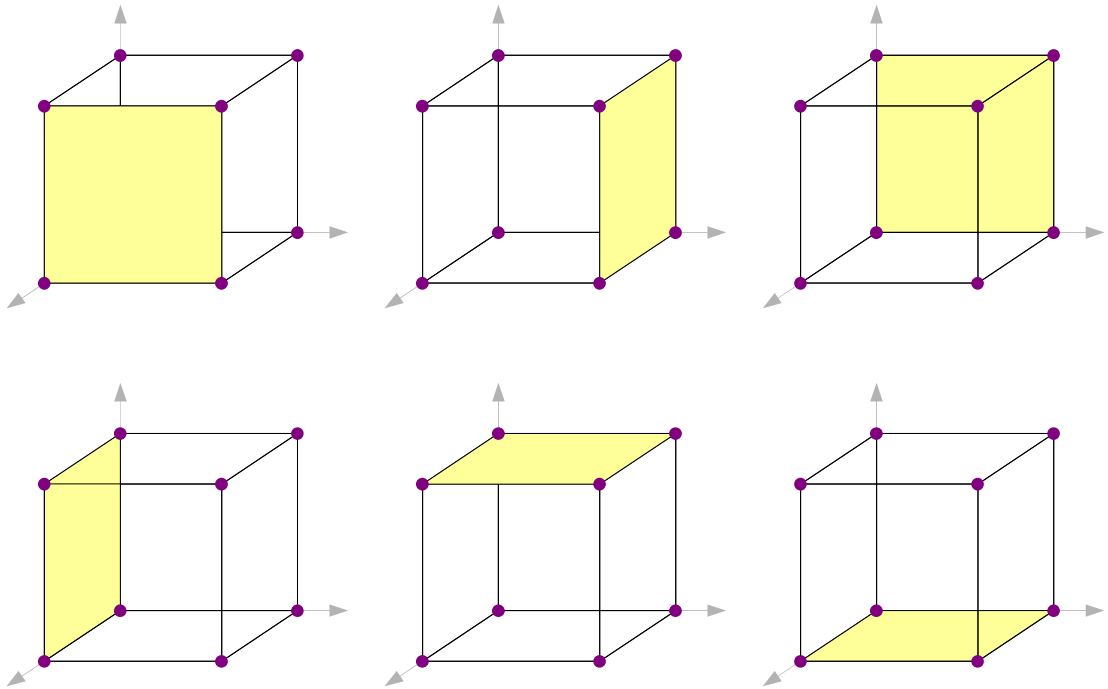


DEFINICJA GEOMETRII

Przykładowe dane



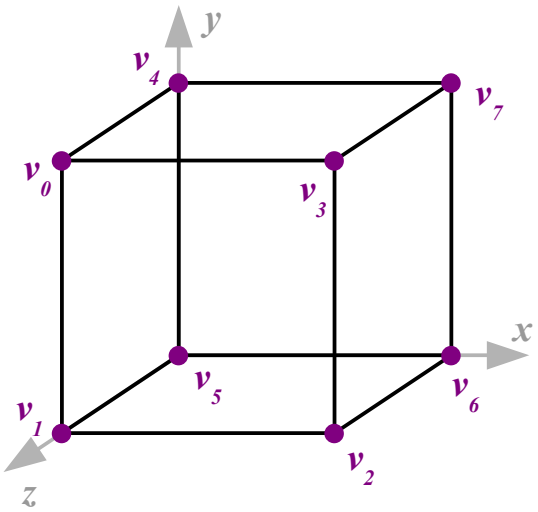
- Kolejność definicji **ścian** naszego sześcianu:



- 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

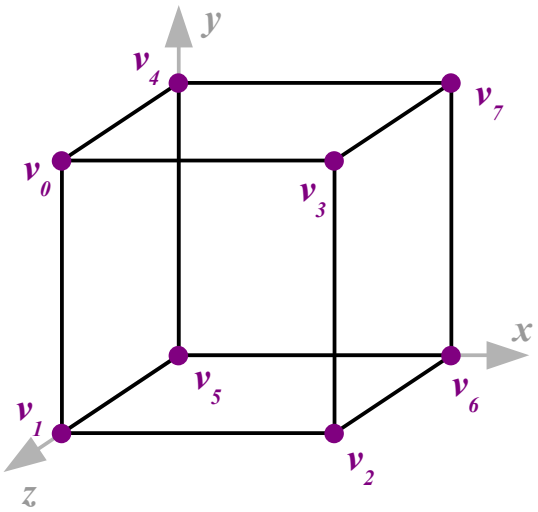


- Kolejność definicji **wierzchołków**:
  - v0 -> v1 -> v2 -> v3
    v3 -> v2 -> v6 -> v7
    v7 -> v6 -> v5 -> v4
    v4 -> v5 -> v1 -> v0
    v0 -> v3 -> v7 -> v4
    v1 -> v5 -> v6 -> v2
  - Czyli: **24 zestawy** po **3 współrzędne** po **4 bajty**
    - 24 × 3 × 4 = **288 bajtów**
  - Pamiętamy o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

IMMEDIATE MODE

Ogólna charakterystyka

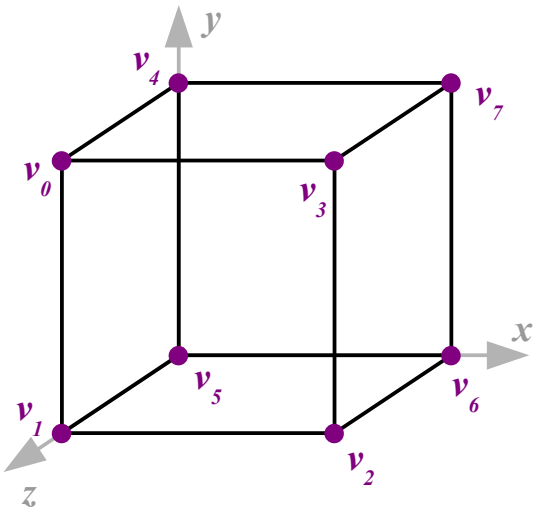
- **Najprostsze** podejście polegające na **bezpośredniej** definicji geometrii
- Wartości (np. współrzędne wierzchołków) dla **każdorazowego** żądania renderowania, **przekazywane są do serwera od nowa**
  - Oznacza to **ogromną ilość danych** przesyłaną co klatkę od klienta do serwera
  - Oznacza to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

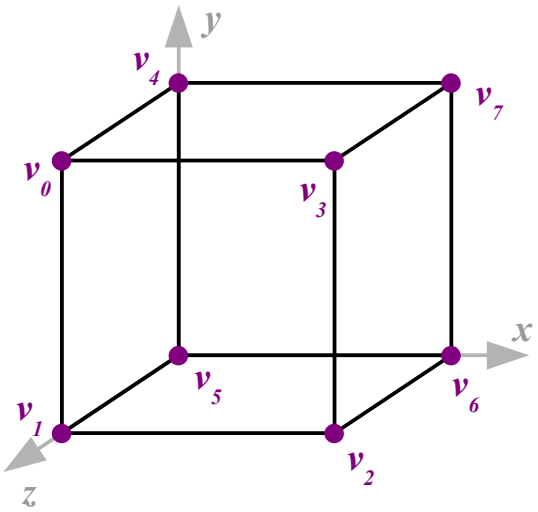


- Przykład dla pierwszego *quada*:

```
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



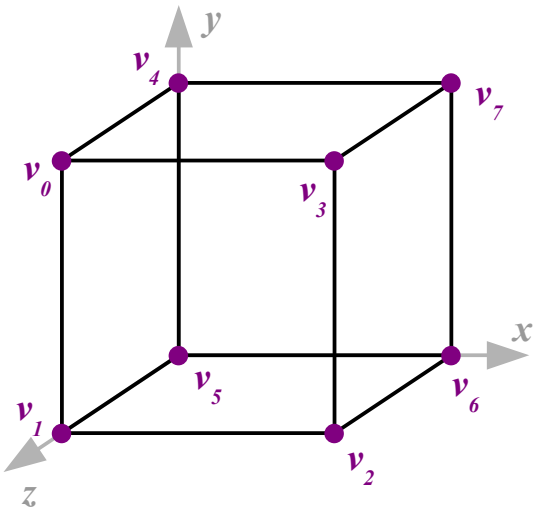
- Dla całego sześcianu:

```
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

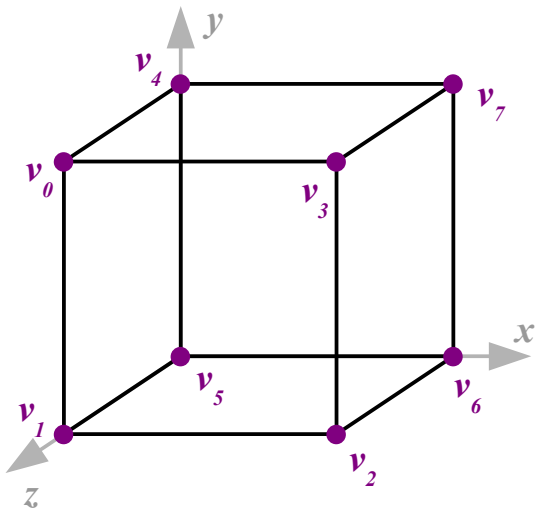
Ogólna charakterystyka



- **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

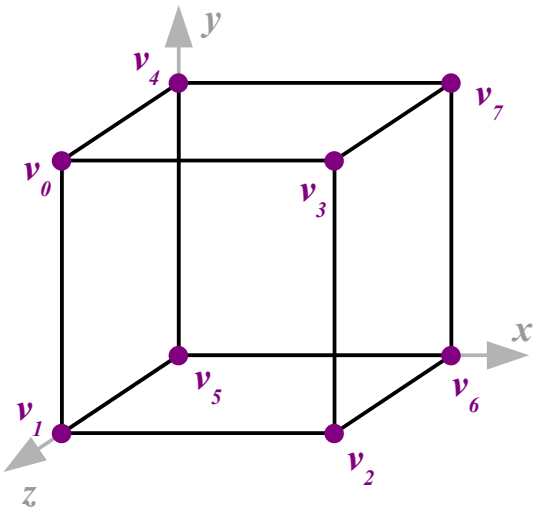


- **Utworzenie** display listy
  - Żądanie **identyfikatora**
- **Kompilacja** display listy
  - **Wierzchołki**, wektory **normalne**, współrzędne **tekstur**...
  - Można dołączyć inne **zmiany stanu serwera**
- **Wywołanie** display listy
  - **Żądanie** renderowania z użyciem **identyfikatora**
-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



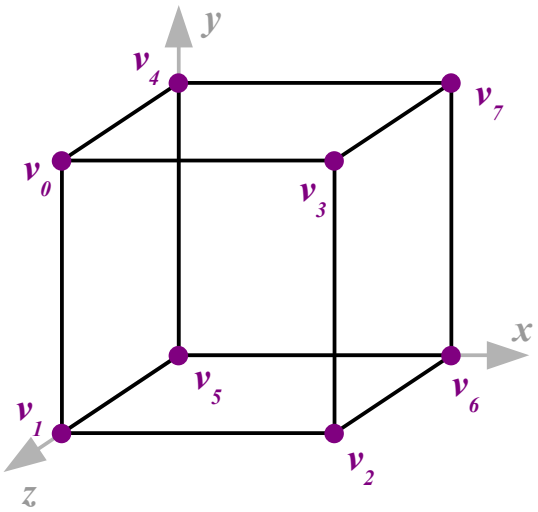
- Dla całego sześcianu:

```
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(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();
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

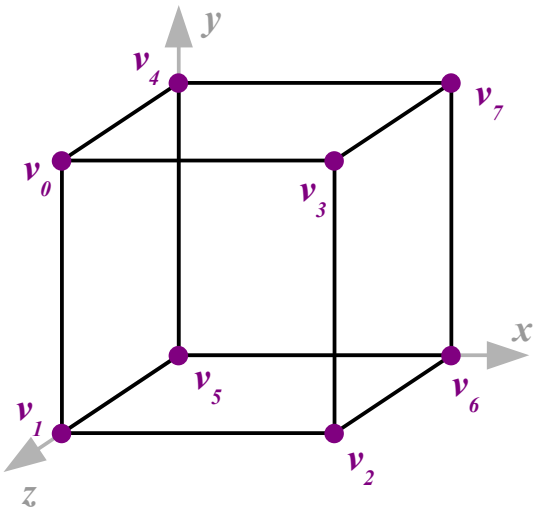


- Ciągła **tablica po stronie klienta** (w pamięci głównej) zawierająca dane wierzchołków
- Możliwość dowolnej **manipulacji** zawartości
- Możliwość **wybiórczego** renderowania
  - Rozwiązuje problem **redefinicji** powtarzających się wierzchołków, co daje **oszczędność pamięci**
- **Interleaved arrays**, 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

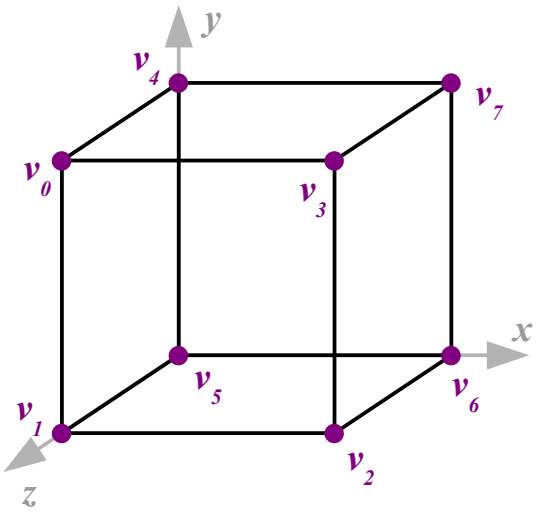
Sposób użycia

- **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.)



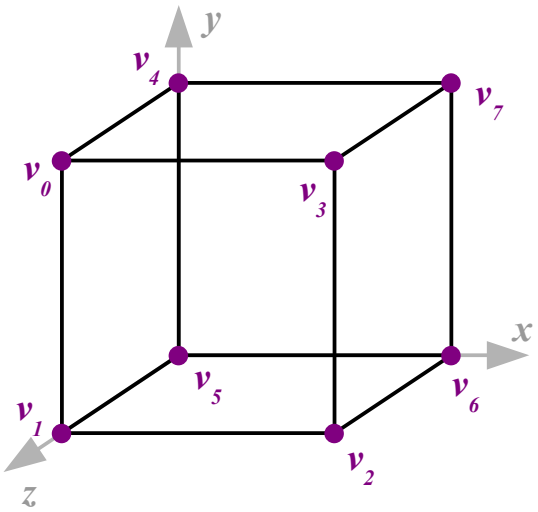
- Przykład dla pierwszego *quada*:

```
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

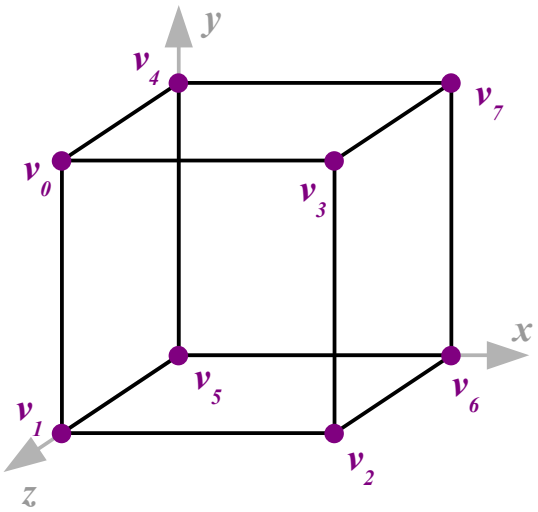


- **Wybiórcze renderowanie** można zrealizować na kilka różnych sposobów:
  - **Podzakres** tablicy *od-do*:
    - **glDrawArrays**(primitive, from, to)
  - Wskazanie **indeksów** wierzchołków:
    - **glDrawElements**(primitive, num_indices, index_type, indices)
    - Dzięki temu **nie musimy redefiniować**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

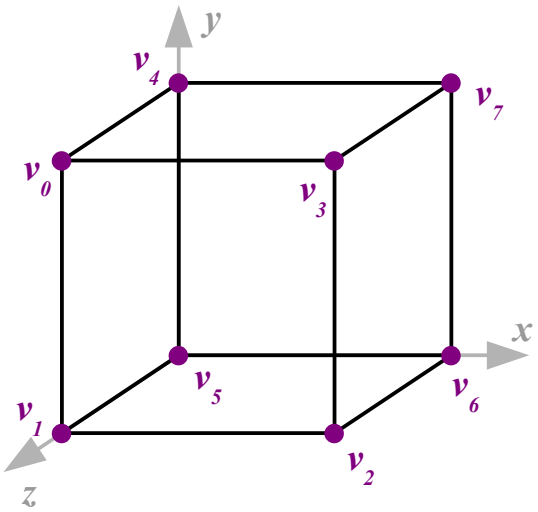


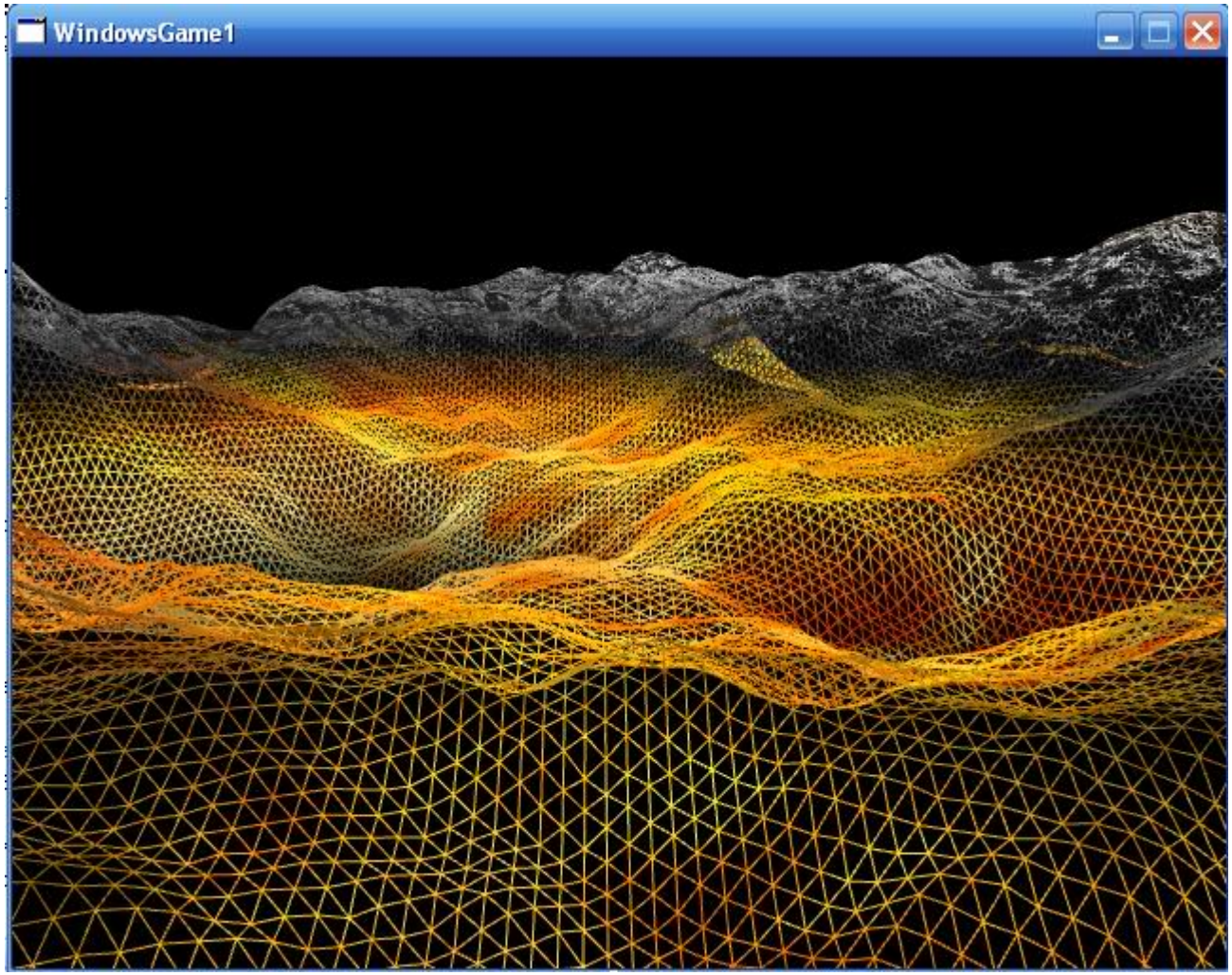
- **Wybiórcze renderowanie** można zrealizować na kilka różnych sposobów (c.d.):
  - Wskazanie **indeksów** wierzchołków oraz ich **zakresu**:
    - **glDrawRangeElements**(primitive, start, end, num_indices, index_type, indices)
    - start i end określają **zakres wartości elementów z tablicy indeksów** (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

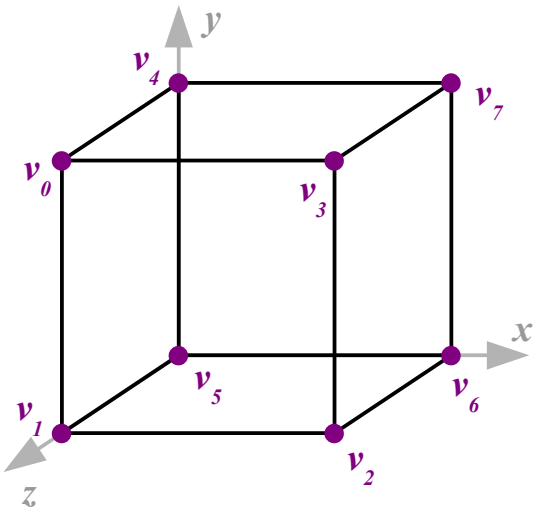




Źródło obrazu:
http://windows-tech.info

VERTEX ARRAY

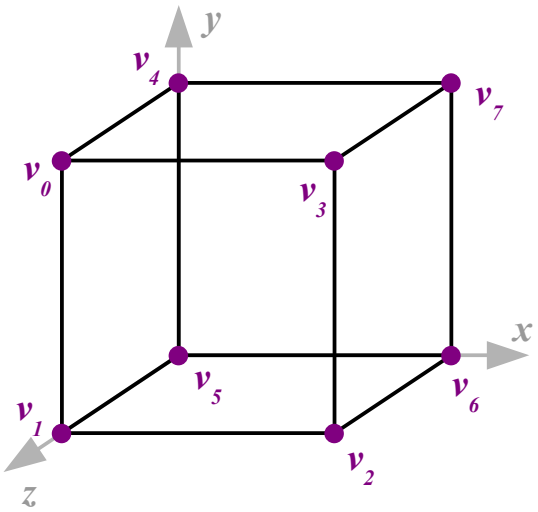
Dodatkowe atrybuty



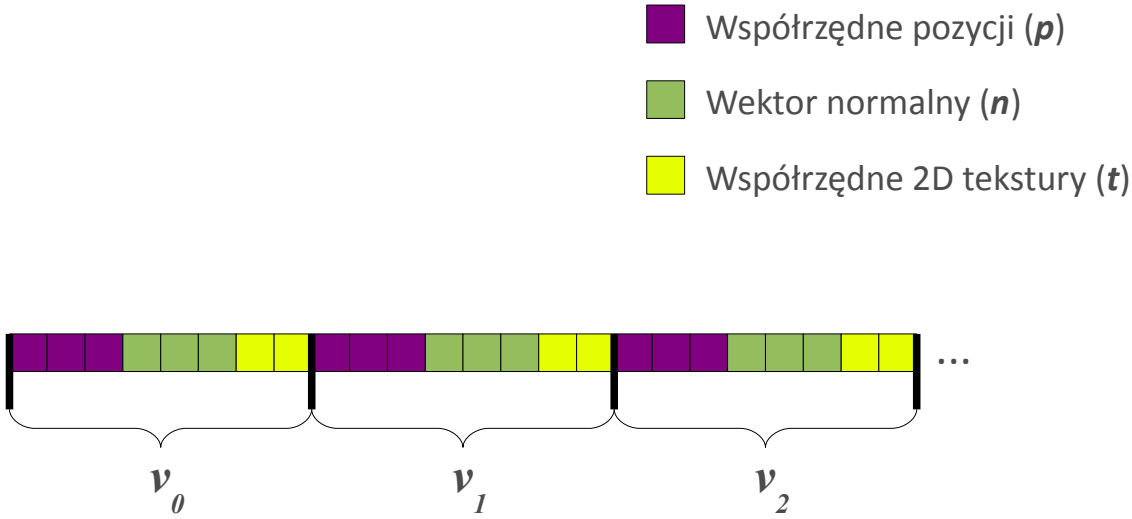
- Przekazywanie wartości **dodatkowych atrybutów** wierzchołków
  - **Nie tylko** pozycja (współrzędne) jest wartością *per-vertex*
    - Wektory normalne, współrzędne tekstur, wektory styczne, kolor, stopień zniszczenia, wiek, stopień deformacji, zabrudzenie, następna pozycja, ...
    - Możemy definiować **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



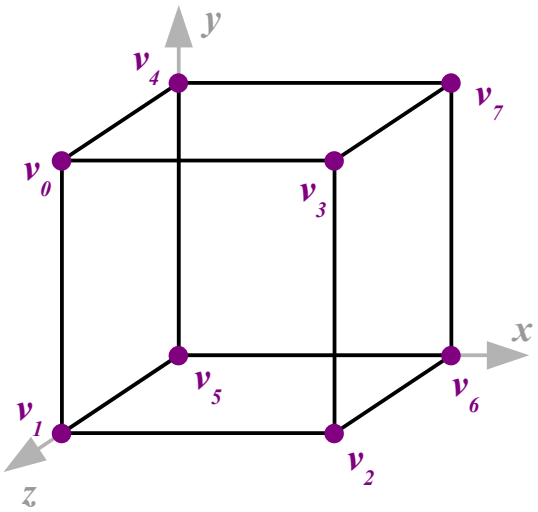
- *Interleaved arrays* – przykład:



- 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



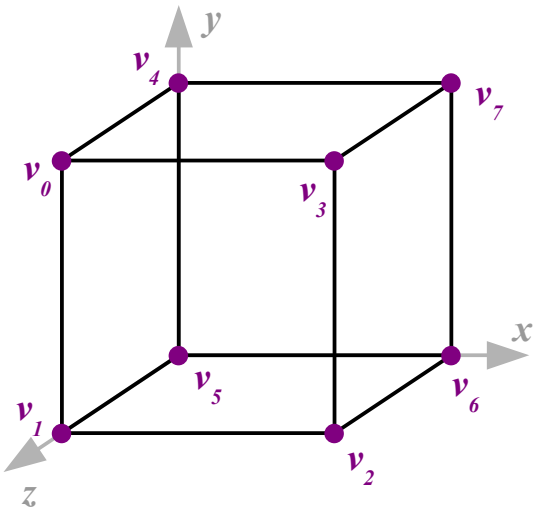
- **Reprezentacja** *interleaved array* **w pamięci głównej** z punktu widzenia programu
  - Wygodnym podejściem jest użycie **struktur**
  - *C++* gwarantuje, że struktura zajmuje **ciągły obszar pamięci**, a składowe mają zachowaną **kolejność**

```
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

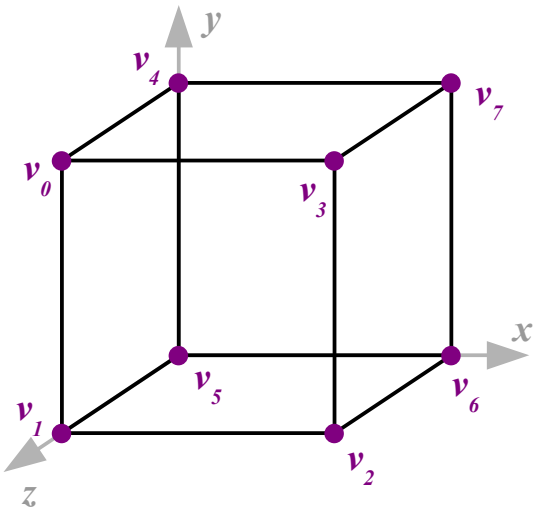


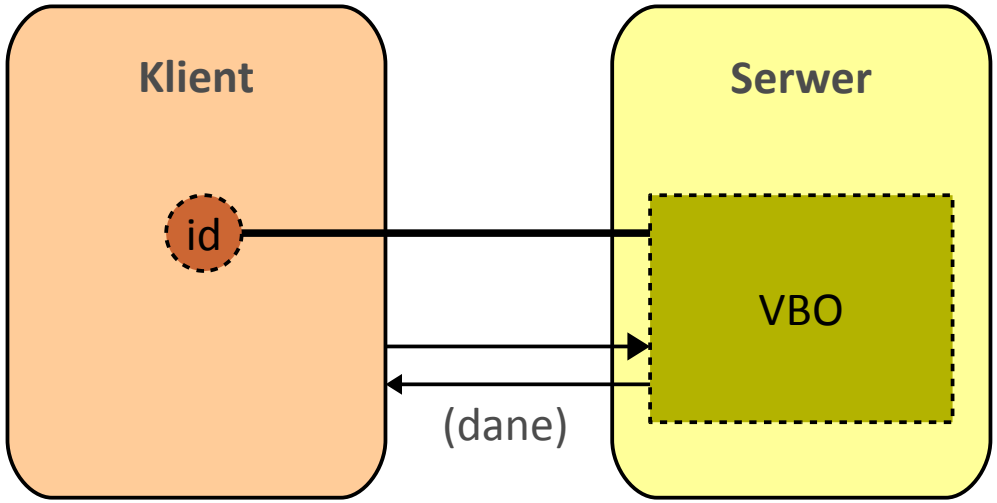
- *Interleaved arrays* – dodatkowe informacje:
  - Jeśli np. planujemy często zmieniać pozycję wierzchołków, pozostawiając pozostałe atrybuty bez zmian, warto użyć oddzielnych tablic
  - Warto dążyć do tego, by *stride* był równy **wielokrotności 32**. Wiele kart graficznych potrafi wtedy lepiej organizować 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

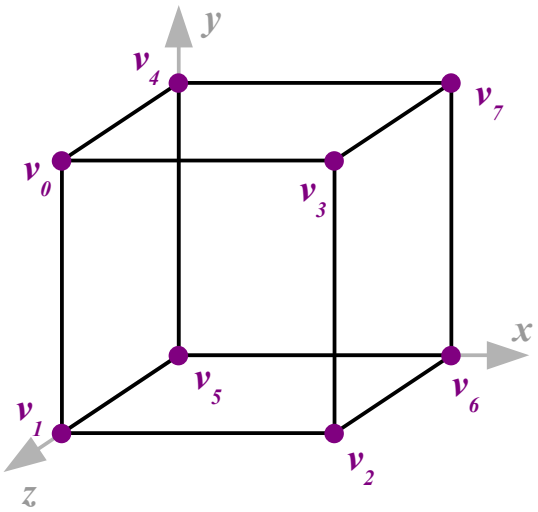
- ***Vertex Buffer Object* (*VBO*)** – technika pozwalająca na łatwe **manipulowanie zawartością pamięci karty graficznej**
- Siostrzana technika do ***Frame Buffer Object* (*FBO*)** i ***Pixel Buffer Object* (*PBO*)**
- Pozwala zarezerwować, wypełnić, odczytać, modyfikować zawartość bufora
- Po stronie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





VBO

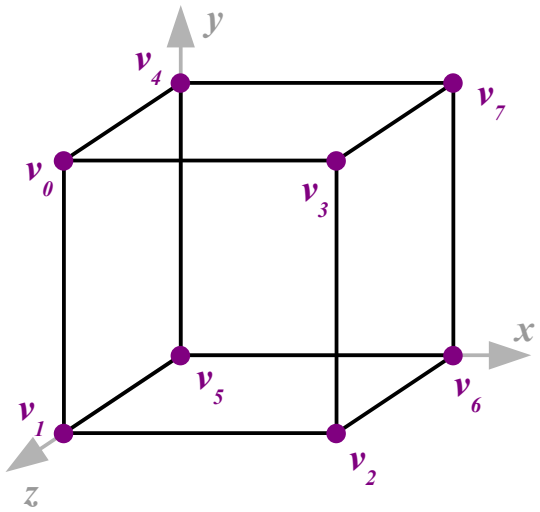
Sposób użycia



- **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

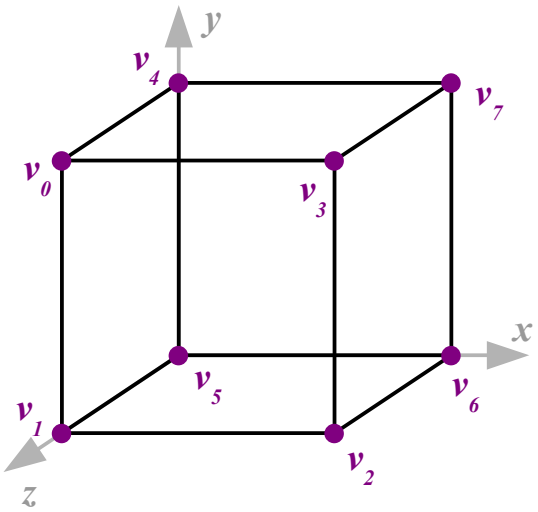


- Przykład dla pierwszego *quada*:

```
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

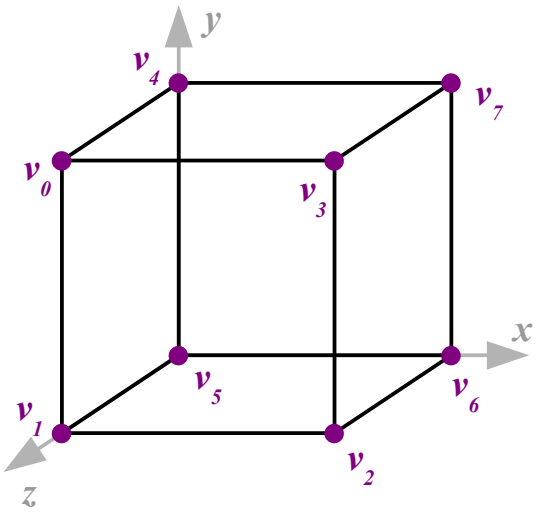


- Istnieje możliwość **modyfikacji** zawartości VBO:
  - Ponowna **alokacja** pamięci i wysłanie danych
    - **glBufferData**(target, size, data, usage)
  - Poprzez **zastąpienie** części danych
    - **glBufferSubData**(target, offset, size, data)
    - Pamięć **nie jest realokowana**!
      Warto 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

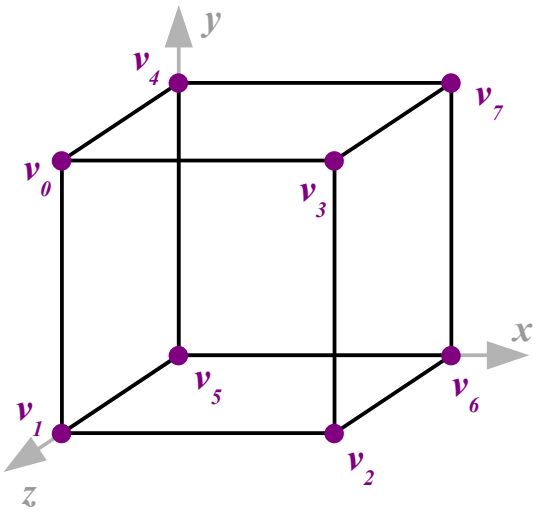
Indeksowanie VBO

- **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

Bartosz Bazyluk

OpenGL

Współczesne podejście do programowania grafiki

Część I: Definicja geometrii

Programowanie Gier Komputerowych, Informatyka S1, III Rok

Wydział
Informatyki