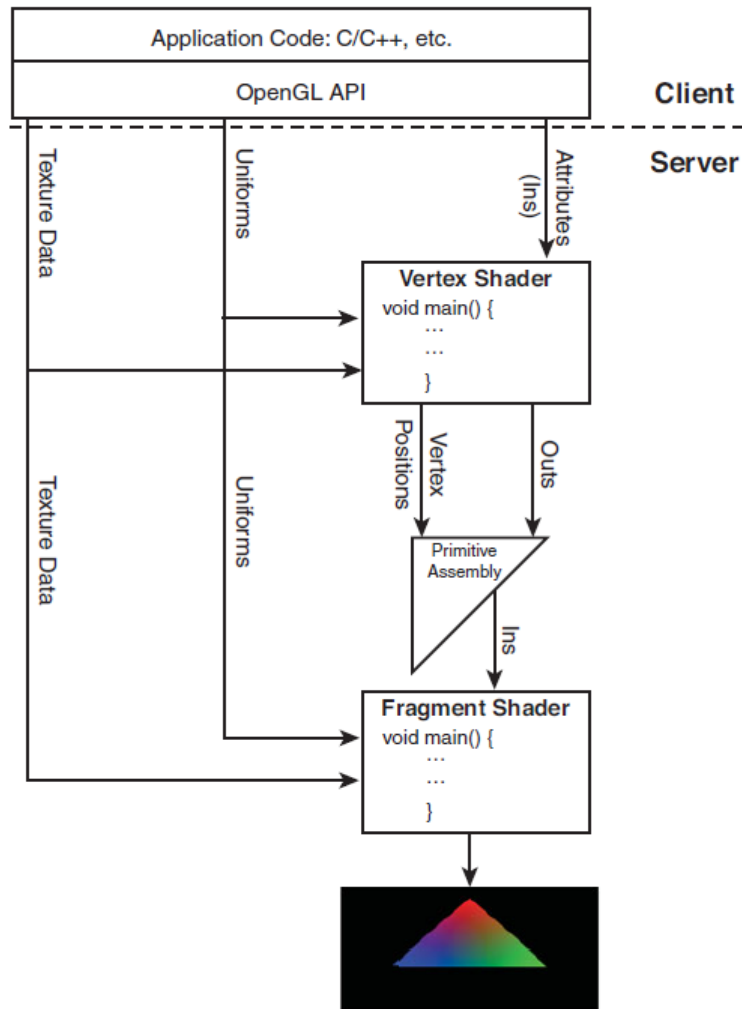


OpenGL Shading Language / WebGL/OpenGL/OpenGL ES 2017

Wprowadzenie do wprowadzenia. Częściowo w oparciu o książkę:
Ed Angel, Interactive Computer Graphics, wyd 7

potok graficzny



Na uproszczonym rysunku
zaznaczone są dwa shadery,
które stanowią minimum:
Vertex Shader i
Fragment Shader

Trzeci, *Geometry Shader*,
na razie pomijamy

Historia GLSL + lekki nieporządek z oznaczeniami

- ▶ OpenGL 2.1 → GLSL 1.20
- ▶ OpenGL 3.0 → GLSL 1.30
- ▶ OpenGL 3.1 → GLSL 1.40 (marzec 2009)
- ▶ OpenGL 3.2 → GLSL 1.50 (sierpień 2009)
- ▶ OpenGL 3.3 → GLSL 3.30 (marzec 2010)
- ▶
- ▶ OpenGL 4.2 → GLSL 4.20 (sierpień 2011)
- ▶ OpenGL 4.3 → GLSL 4.30 (sierpień 2012)
- ▶ OpenGL 4.4 → GLSL 4.40
- ▶ OpenGL 4.5 → GLSL 4.50 (sierpień 2014)



GLSL

- Źródła internetowe :
- <http://www.lighthouse3d.com/tutorials>
- tutorial GLSL, udany, z przykładami, nie tak dawno zaktualizowany;
- <http://www.clockworkcoders.com/ogsl/tutorials.html> -
inny tutorial, też udany choć lekko przestarzały.
- Powyższe przykłady są osadzone w OpenGL, a nie WebGL – jednak GLSL jest w obu przypadkach taki sam



Vertex processor – Vertex shader

- ▶ *Vertex processor* jest programowalną jednostką, na której wykonuje się *Vertex shader*.
- ▶ *Vertex shader* operuje na wierzchołkach.
- ▶ Może wykonywać na nich:
 - ▶ transformacje położeń wierzchołków (z pomocą macierzy widoku i rzutowania)
 - ▶ transformacje normalnych (łącznie z ich normalizacją)
 - ▶ transformacje współrzędnych tekstur
 - ▶ obliczanie oświetlenia w wierzchołkach
 - ▶ obliczanie koloru w wierzchołkach.
- ▶ *Vertex processor* przetwarza niezależnie pojedyncze wierzchołki ('one at a time').



Geometry processor – geometry shader

- ▶ *Geometry processor* jest programowalną jednostką, na której wykonuje się *Geometry shader*.
- ▶ *Geometry shader* przetwarza grupy wierzchołków, które tworzą podstawowe elementy geometryczne (trójkąty, łamane, odcinki, punkty)
- ▶ *Geometry shader* ma dostęp do całej grupy wierzchołków, może zmieniać typ i atrybuty elementu geometrycznego, może dodawać i kasować wierzchołki
- ▶ *Geometry shader* jest opcjonalny. Jeżeli występuje jest umieszczony w potoku graficznym między *Vertex shaderem* i *Fragment shaderem*.



Fragment processor – Fragment shader

Fragment shader działa na fragmentach (~pikselach), czyli na obrazie powstałym na skutek rasteryzacji, umożliwiając znaczną kontrolę nad tym co pojawia się w każdym pikselu.

Na fragmentach można wykonywać:

- obliczanie kolorów i współrzędnych tekstur *per pixel*
- obliczanie mgły
- obliczanie normalnych dla oświetlenia *per pixel*.

Fragment shader nie może zmienić współrzędnych piksela.



Shadery w WebGL

- ▶ Shadery muszą zostać skompilowane i zlinkowane, tak aby utworzyć program wykonywalny
- ▶ WebGL zawiera odpowiedni kompilator i linker
- ▶ Program WebGL musi zawierać vertex i fragment shader

Create
Program

`gl.createProgram()`

Create
Shader

`gl.createShader()`

Load Shader
Source

`gl.shaderSource()`

Compile
Shader

`gl.compileShader()`

Attach Shader
to Program

`gl.attachShader()`

Link Program

`gl.linkProgram()`

Use Program

`gl.useProgram()`

Prostsze podejście, często stosowane

Np. funkcja zaproponowana w kursie Angela i Shreiner

```
initShaders(vFile, fFile );
```

- ▶ `initShaders` ma dwa argumenty
 - ▶ `vFile` - ścieżka do pliku z vertex shaderem
 - ▶ `fFile` - ścieżka do pliku z fragment shaderem
- ▶ Łatwo utworzyć własną podobną funkcję





Elementy GLSL



GLSL

- ▶ Nazwa języka: `OpenGL Shading Language`
- ▶ Język oparty na C z drobnymi elementami C++
- ▶ Wbudowane typy macierzowe i wektorowe (o wymiarze do 4), przydatne w transformacjach i rzutowaniu
- ▶ Zarówno *vertex shadery* jak i *fragment shadery* są napisane w GLSL, niezależnie od tego w jakim języku jest napisany podstawowy kod OpenGL/WebGL
- ▶ Każdy shader zawiera funkcję `main()`



Podstawowe, skalarne typy danych

- ▶ Trzy podstawowe typy skalarne w GLSL:
 - ▶ `float`
 - ▶ `int, uint`
 - ▶ `bool`
- ▶ `float` i `int` zachowują się dokładnie tak jak w C, natomiast typ `bool` tak jak w C++, i przyjmuje wartości `true/false`.
- ▶ W wersji GLSL 3.3 jest tylko pojedyncza precyzja.



Typy wektorowe w GLSL

- ▶ Wektory 2, 3 lub 4 elementowe, są deklarowane jako:
 - ▶ **vec2**, **vec3**, **vec4**: wektory float, 2, 3, 4 elem.
 - ▶ **bvec2**, **bvec3**, **bvec4**: wektory bool
 - ▶ **ivec2**, **ivec3**, **ivec4**: wektory integer
 - ▶ **uvec2**, **uvec3**, **uvec4**: wiadomo...
- ▶ Dla porządku: to nie są klasy, a typy wbudowane w języku GLSL.
- ▶ Wektory można przypisywać, dodawać i mnożyć przez skalar.



Dostęp do składowych wektora

```
vec4  myVector;
```

```
myVector[0] myVector[1] myVector[2] myVector[3]
```

```
myVector.x myVector.y myVector.z myVector.w
```

```
myVector.r myVector.g myVector.b myVector.a
```

```
myVector.s myVector.t myVector.p myVector.q
```

```
myVector.rgb    myVector.b    myVector.xyy
```



Typy macierzowe w GLSL

- ▶ Jest ich kilka, ale same się tłumaczą:
 - ▶ `mat2`, `mat2x2`
 - ▶ `mat3`, `mat3x3`
 - ▶ `mat4`, `mat4x4`
 - ▶ `mat2x3` 2 wiersze x 3 kolumny, itd
 - ▶ `mat2x4`, `mat3x2`, `mat3x4`, `mat4x2`,
`mat4x3`
- ▶ Najważniejszym typem macierzowym jest zapewne `mat4`



Typy związane z próbkowaniem i mapowaniem tekstur

- ▶ Próbkowanie tekstur:

- ▶ `sampler1D`, `sampler2D`, `sampler3D`,
`samplerCube`

- ▶ Zostawiamy to na później



Typ macierzowy w GLSL, cd.

- ▶ Macierz jest w istocie tablicą wektorów.
W GLSL – wektorów kolumnowych.
- ▶ W związku z tym można sprytnie mieszać operacje wektorowe i macierzowe.



Operatory

- ▶ Standardowe operatory arytmetyczne i logiczne C/C++
- ▶ Przeładowane operatory dla operacji macierzowych i wektorowych

```
mat4 m;
```

```
vec4 a, b, c;
```

```
b = a*m;
```

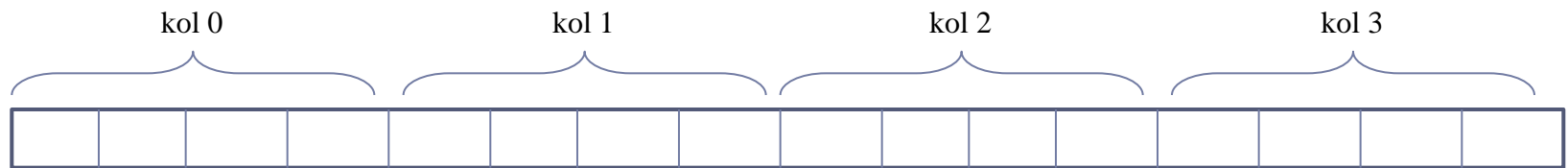
```
c = m*a;
```



Operacje na zmiennych wektorowych i macierzowych

- ▶ Załóżmy, że mamy zdefiniowaną tablicę 4x4 za pomocą typu `mat4` :

```
mat4 mView;
```



```
mView[3] = vec4(0.0f, 0.0f, 0.0f,  
1.0f);
```

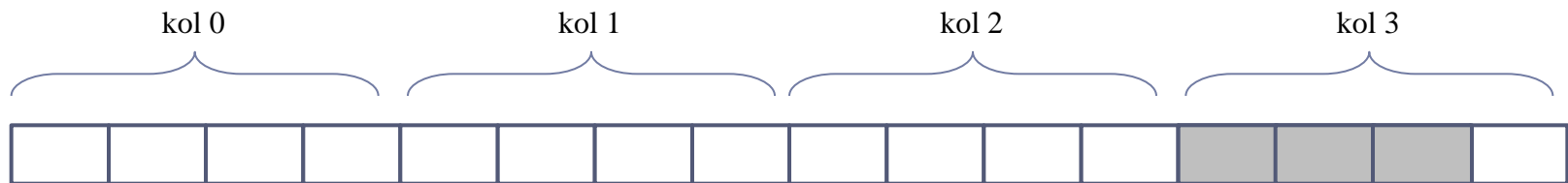
```
vec4 vTranslation = mView[3];
```



Operacje na zmiennych wektorowych i macierzowych

- ▶ Można oczywiście odwoływać się do fragmentów kolumny i wiersza macierzy – tak jak do fragmentów wektorów

```
vec3 vTranslation = mView[3].xyz;
```



Operacje na zmiennych wektorowych i macierzowych

- ▶ Można mnożyć przez siebie macierze i wektory, jeśli są odpowiednio zdefiniowane. I często się to robi:

```
vec4    vVertex;  
mat4   .mvpMatrix;  
...  
...  
vOutPos =.mvpMatrix * vVertex;
```



Nadawanie wartości początkowych macierzy – konstruktor macierzy w stylu C++

- ▶ Można to zrobić ogólnie tak (kolumnami):

```
mat4 vIdent = mat4(1.0f, 0.0f, 0.0f, 0.0f,  
                   0.0f, 1.0f, 0.0f, 0.0f,  
                   0.0f, 0.0f, 1.0f, 0.0f,  
                   0.0f, 0.0f, 0.0f, 1.0f);
```

- ▶ W przypadku macierzy jednostkowej możemy sobie uprościć:

```
mat4 vIdent = mat4(1.0f);
```



Kwalifikatory pamięci (Storage Qualifiers)

- ▶ Deklaracje zmiennych mogą być poprzedzone kwalifikatorami, które wskazują czy:
 - ▶ zmienna jest wejściowa (wejście z OpenGL lub wcześniejszego shadera: z vertex do fragment shadera)
 - ▶ zmienna jest wyjściowa (wyjście do fragment shadera lub buforu)
 - ▶ zmienna wejściowo/wyjściowa – tylko w parametrach shaderów!

Szczegóły w tabelce skopiowanej z książki...



Storage Qualifiers

TABLE 6.3 Variable Storage Qualifiers

Qualifier	Description
<none>	Just a normal local variable, no outside visibility or access.
const	A compile-time constant, or a read-only parameter to a function.
in	A variable passed in from a previous stage.
in centroid	Passed in from a previous state, uses centroid interpolation.
out	Passed out to the next processing stage or assigned a return value in a function.
out centroid	Passed out to the next processing stage, uses centroid interpolation.
inout	A read/write variable. Only valid for local function parameters.
uniform	Value is passed in from client code and does not change across vertices.

Kwalifikator **centroid** – tylko w wypadku *multisamplingu*

Kwalifikator **inout** – pozwala na odsyłanie parametrów na zewnątrz, przy braku w GLSL wskaźników.

Kwalifikator **uniform** używany do przesyłania parametrów sterujących



Komentarz szczegółowy dotyczący Storage Qualifiers i niejasności w przykładach

- ▶ Szczegółową i szybką informację można znaleźć w <https://www.khronos.org/files/opengl42-quick-reference-card.pdf>
 - ▶ **in** – zmienna wejściowa (z programu OpenGL/WebGL lub z wcześniejszego shadera)
 - ▶ **out** – zmienna wyjściowa do następnego etapu
 - ▶ **attribute** – to samo co **in** dla vertex shadera
 - ▶ **varying** – to samo co **out** dla vertex shadera, to samo co **in** dla fragment shadera
 - ▶ **attribute** i **varying** są przestarzałe (deprecated) dla nowszych wersji GLSL
-



Przykład shadera

```
#version 330          //vertex shader
                      //w starszej wersji:
in  vec4  vVertex;    //attribute vec4  vVertex
in  vec4  vColor;     //attribute vec4  vColor

out vec4  vVarColor;  //varying vec4  vVarColor

void main(void)
{
    vVarColor = vColor;
    gl_Position = vVertex;
}
```



Przykład shadera

```
#version 330                                //fragment shader
                                           //w starszej wersji

in vec4 vVarColor;                         //varying vec4...
out vec4 vFragColor;                       //varying vec4...
void main(void)    {
    vFragColor = vVarColor;
    //vFragColor = vec4(1.0f,1.0f,0.0f,0.0f);
}
```



Wbudowane funkcje

- ▶ **Angles & Trigonometry**

- ▶ radians, degrees, sin, cos, tan, asin, acos, atan

- ▶ **Exponentials**

- ▶ pow, exp2, log2, sqrt, inversesqrt

- ▶ **Common**

- ▶ abs, sign, floor, ceil, fract, mod, min, max, clamp



Wbudowane funkcje, cd

► Interpolacje

► **mix**(x, y, a) $x * (1.0 - a) + y * a$

► **step**(edge, x)

$x \leq \text{edge} ? 0.0 : 1.0$

► **smoothstep**(edge0, edge1, x)

$t = (x - \text{edge0}) / (\text{edge1} - \text{edge0}) ;$

$t = \text{clamp}(t, 0.0, 1.0) ;$

$\text{return } t * t * (3.0 - 2.0 * t) ;$



Wbudowane funkcje, cd

- ▶ Geometric

- ▶ `length, distance, cross, dot, normalize, faceForward, reflect`

- ▶ Matrix

- ▶ `matrixCompMult`

- ▶ Vector relational

- ▶ `lessThan, lessThanEqual, greaterThan, greaterThanEqual, equal, notEqual, any, all`



Wbudowane funkcje, cd

▶ Tekstury

- ▶ `texture1D`, `texture2D`, `texture3D`,
`textureCube`
- ▶ `texture1DProj`, `texture2DProj`,
`texture3DProj`, `textureCubeProj`
- ▶ `shadow1D`, `shadow2D`, `shadow1DProj`,
`shadow2Dproj`

▶ Wierzchołki

- ▶ `ftransform` (przestarzałe/deprecated)



Wbudowane zmienne

- ▶ `gl_Position`
 - ▶ (required) output position from vertex shader
- ▶ `gl_FragColor`
 - ▶ (required?) output color from fragment shader
- ▶ `gl_FragCoord`
 - ▶ (optional) input fragment position
- ▶ `gl_FragDepth`
 - ▶ (optional) input depth value in fragment shader



Najprostszy Vertex Shader (w starym stylu)

```
attribute  vec4 vPosition;  
attribute  vec4 vColor;  
  
varying vec4 fColor;  
  
void main()  
{  
    fColor = vColor;  
    gl_Position = vPosition;  
}
```



Najprostszy Fragment Shader (w starym stylu)

```
precision mediump float; //precyzja
                           //obliczania float
                           //lowp,mediump,highp

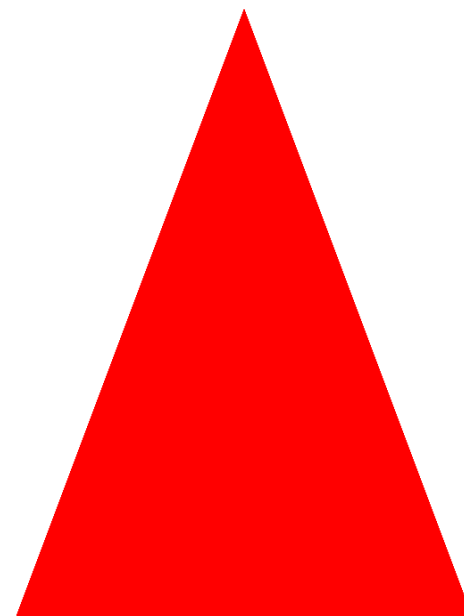
varying vec4 fColor;
void main()
{
    gl_FragColor = fColor;
}
```



Przykłady w czystym WebGL z
shaderami.

Pierwszy najprostszy przykład

- ▶ Tworzymy czerwony trójkąt
- ▶ Zawiera on wszystkie składowe bardziej złożonej aplikacji
 - ▶ vertex shader
 - ▶ fragment shader
 - ▶ HTML canvas
- ▶ Źródło:
www.cs.unm.edu/~angel/WebGL



triangle.html

```
<!DOCTYPE html>
<html>
<head>
<script id="vertex-shader" type="x-shader/x-vertex">
attribute vec4 vPosition;
void main()
{
    gl_Position = vPosition;
}
</script>
<script id="fragment-shader" type="x-shader/x-fragment">
precision mediump float;
void main()
{
    gl_FragColor = vec4( 1.0, 0.0, 0.0, 1.0 );
}
</script>
```



triangle.html

```
<script type="text/javascript" src="../../webgl-utils.js"></script>
<script type="text/javascript" src="../../initShaders.js"></script>
<script type="text/javascript" src="triangle.js"></script>
</head>
<body>
<canvas id="gl-canvas" width="512" height="512">
Oops ... your browser doesn't support the HTML5 canvas element
</canvas>
</body>
</html>
```

webgl-utils.js zawiera zestaw funkcji użytkowych (od Google), które pozwalają na budowanie kontekstu WebGL.

initShaders.js zawiera sekwencję wywołań funkcji WebGL do czytania, kompilowania i linkowania shaderów.



Przykład: triangle.js (1/3)

```
var gl;
var points;

window.onload = function init()
{
    var canvas = document.getElementById( "gl-canvas" );
    gl = WebGLUtils.setupWebGL( canvas );
    if ( !gl ) { alert( "WebGL isn't available" );
    }

    var vertices = new Float32Array([-1, -1, 0, 1, 1, -1]);

    // Konfigurujemy WebGL

    gl.viewport( 0, 0, canvas.width, canvas.height );
    gl.clearColor( 1.0, 1.0, 1.0, 1.0 );
```



triangle.js (2/3)

```
// Load shaders and initialize attribute buffers
```

```
var program = initShaders( gl, "vertex-shader",  
"fragment-shader" );
```

```
gl.useProgram( program );
```

```
// Load the data into the GPU
```

```
var bufferId = gl.createBuffer();  
gl.bindBuffer( gl.ARRAY_BUFFER, bufferId );  
gl.bufferData( gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW );
```



triangle.js (3/3)

```
// Kojarzymy zmienne shadera z danymi bufora
```

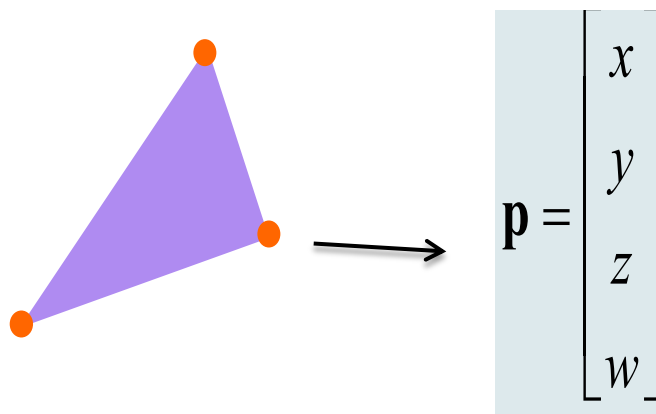
```
var vP = gl.getAttributeLocation( program, "vPosition" );  
gl.vertexAttribPointer( vP, 2, gl.FLOAT, false, 0, 0 );  
gl.enableVertexAttribArray( vP );  
render();  
};
```

```
function render()  
{  
    gl.clear( gl.COLOR_BUFFER_BIT );  
    gl.drawArrays( gl.TRIANGLES, 0, 3 );  
}
```



Reprezentowanie obiektów geometrycznych

- ▶ Obiekty są reprezentowane za pomocą wierzchołków (*vertices*)
- ▶ Wierzchołek jest zbiorem atrybutów (taka jest definicja wierzchołka w OpenGL/WebGL):
 - ▶ Współrzędne położenia
 - ▶ kolory
 - ▶ Współrzędne tekstury
 - ▶ Normalne
 - ▶ Różne inne dane , jakie mogą być przypisane wierzchołkowi
- ▶ Położenia są przechowywane w 4 wymiarowych wektorach
- ▶ Dane o wierzchołkach są przechowywane w buforach VBO (Vertex Buffer Objects)

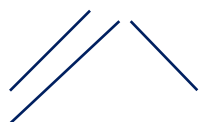


Podstawowe obiekty w OpenGL/WebGL

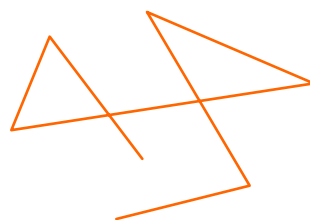
Wszystkie zdefiniowane za pomocą wierzchołków



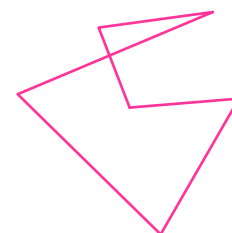
GL_POINTS
gl.POINTS



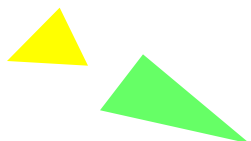
GL_LINES
gl.LINES



GL_LINE_STRIP



GL_LINE_LOOP



GL_TRIANGLES
gl_TRIANGLES



GL_TRIANGLE_STRIP
gl.TRiangle_STRIP



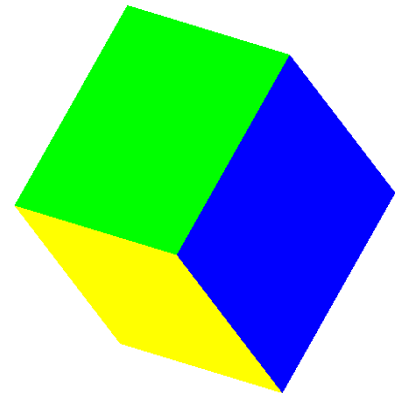
GL_TRIANGLE_FAN



Drugi przykład z sześcianiem w WebGL

Drugi przykład

- ▶ Należy wyrenderować każdą ścianę w sześcianie innym kolorem
- ▶ Przykład pokazuje:
 - ▶ Proste modelowanie obiektów
 - ▶ Budowę obiektu 3D z obiektów podstawowych (primitives)
 - ▶ Budowanie tych 'primitives' z wierzchołków
 - ▶ Zainicjowanie danych (wierzchołków)
 - ▶ Przygotowanie danych do renderowania
 - ▶ interakcję
 - ▶ animację



Rotate X Rotate Y Rotate Z Toggle Rotation



Zainicjowanie danych sześcianu

- ▶ Każdą ścianę sześcianu tworzymy z dwóch trójkątów.
- ▶ Ile potrzebujemy pamięci
 - ▶ $(6 \text{ ścian}) * (2 \text{ trójkąty/ścianę}) * (3 \text{ wierzchołki/trójkąt})$

```
var numVertices = 36;
```

- ▶ W celu uproszczenia komunikacji z GLSL, możemy użyć pakietu **MV.js** który zawiera klasę **vec3**, zbliżoną typu **vec3** w GLSL.



Zanicjowanie danych sześcianu (2)

- ▶ Zanim zainicjujemy VBO, potrzebujemy zdefiniować dane
- ▶ Nasz sześcian ma dwa atrybuty stowarzyszone z każdym wierzchołkiem:
 - ▶ położenie
 - ▶ kolor
- ▶ W tym przykładzie tworzymy dwie tablice z danymi dla dwóch buforów VBO (choć można by się obejść jednym)

```
var points = [];
```

```
var colors = [];
```



Dane sześcianu

- ▶ Wpisujemy wierzchołki jednostkowego sześcianu o środku w początku układu współrzędnych i krawędziach równoległych do osi układu.

```
var vertices = [  
    vec4( -0.5, -0.5,  0.5, 1.0 ),  
    vec4( -0.5,  0.5,  0.5, 1.0 ),  
    vec4(  0.5,  0.5,  0.5, 1.0 ),  
    vec4(  0.5, -0.5,  0.5, 1.0 ),  
    vec4( -0.5, -0.5, -0.5, 1.0 ),  
    vec4( -0.5,  0.5, -0.5, 1.0 ),  
    vec4(  0.5,  0.5, -0.5, 1.0 ),  
    vec4(  0.5, -0.5, -0.5, 1.0 )  
];
```



Dane sześcianu (cd)

- ▶ Następnie wypełniamy tablicę kolorów RGBA
- ▶ Można użyć obiekt `vec3` lub `vec4` (z `MV.js`) lub zwykłą tablicę JavaScript (co zresztą robimy)

```
var vertexColors = [  
    [ 0.0, 0.0, 0.0, 1.0 ], // black  
    [ 1.0, 0.0, 0.0, 1.0 ], // red  
    [ 1.0, 1.0, 0.0, 1.0 ], // yellow  
    [ 0.0, 1.0, 0.0, 1.0 ], // green  
    [ 0.0, 0.0, 1.0, 1.0 ], // blue  
    [ 1.0, 0.0, 1.0, 1.0 ], // magenta  
    [ 0.0, 1.0, 1.0, 1.0 ], // cyan  
    [ 1.0, 1.0, 1.0, 1.0 ]  // white
```

▶];

Dywagacja: tablice w JavaScript

- ▶ Tablica JS jest obiektem wyposażonym m.in. w następujące atrybuty i metody: `length`, `push()` i `pop()`
 - ▶ Tablice JS są inne niż tablice C
 - ▶ Nie mogą być wprost parametrami w funkcjach WebGL
 - ▶ Użyjemy funkcji `flatten()` do wyekstrahowania danych z tablicy JS

```
gl.bufferData( gl.ARRAY_BUFFER,  
               flatten(colors),  
               gl.STATIC_DRAW );
```



Generowanie ściany sześciangu od zera

- ▶ Upraszczamy nasze działania przez użycie własnej funkcji `quad()`
 - ▶ Tworzymy dwa trójkąty tworzące jedną ścianę i przypisujemy kolory wierzchołkom.



Generowanie ściany sześciianu od zera

```
function quad(a, b, c, d) {  
  var indices = [ a, b, c, a, c, d ];  
    for ( var i = 0; i < indices.length; ++i ) {  
      points.push( vertices[indices[i]] );  
  
      // for vertex colors use  
      //colors.push( vertexColors[indices[i]]  
);  
  
      // for solid colored faces use  
      colors.push(vertexColors[a]);  
}
```



Generowanie sześcianu ze ścian

- ▶ Dla sześcianu tworzymy 12 trójkątów
 - ▶ Mamy 36 wierzchołków z 36 kolorami

```
function colorCube() {  
    quad( 1, 0, 3, 2 );  
    quad( 2, 3, 7, 6 );  
    quad( 3, 0, 4, 7 );  
    quad( 6, 5, 1, 2 );  
    quad( 4, 5, 6, 7 );  
    quad( 5, 4, 0, 1 );  
}
```



Przechowywanie atrybutów wierzchołka

- ▶ Dane o wierzchołkach muszą być przechowywane w buforze Vertex Buffer Object (VBO)
- ▶ Do ustawienia VBO musimy:
 - ▶ Utworzyć pusty bufor wywołując

```
vbuffer = gl.createBuffer();
```

- ▶ Uaktywnić określony VBO wywołując:

```
gl.bindBuffer( gl.ARRAY_BUFFER, vBuffer );
```

- ▶ Załadować dane do bufora VBO używając

```
gl.bufferData( gl.ARRAY_BUFFER,  
               flatten(points), gl.STATIC_DRAW );
```



Fragment kodu. Łączymy zmienne shadera z tablicami wierzchołków

```
var cBuffer = gl.createBuffer();  
gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );  
gl.bufferData( gl.ARRAY_BUFFER,  
flatten(colors), gl.STATIC_DRAW );
```


```
var vColor =  
gl.getAttributeLocation( program, "vColor" );  
gl.vertexAttribPointer( vColor, 4,  
gl.FLOAT, false, 0, 0 );  
gl.enableVertexAttribArray( vColor );
```



Fragment kodu. Łączymy zmienne shadera z tablicami wierzchołków

```
var vBuffer = gl.createBuffer();  
gl.bindBuffer( gl.ARRAY_BUFFER, vBuffer );  
gl.bufferData( gl.ARRAY_BUFFER,  
flatten(points), gl.STATIC_DRAW );
```

```
var vPosition = gl.getAttributeLocation(  
program, "vPosition" );  
gl.vertexAttribPointer( vPosition, 3,  
gl.FLOAT, false, 0, 0 );  
gl.enableVertexAttribArray( vPosition );
```



Rysowanie podstawowych elementów geometrycznych (Geometric Primitives)

- ▶ Dla ciągłych grup wierzchołków możemy wykonać proste renderowanie.

```
function render()
{
    gl.clear( gl.COLOR_BUFFER_BIT |
              gl.DEPTH_BUFFER_BIT);
    gl.drawArrays( gl.TRIANGLES, 0, numVertices );
    requestAnimationFrame( render );
}
```

- ▶ `gl.drawArrays` inicjuje vertex shader
- ▶ `requestAnimationFrame` - znane nam z three.js
- ▶ Musimy wyczyścić bufor
- ▶ Bufor głębokości: `gl.enable(gl.GL_DEPTH)` w `init()`



Renderowanie obiektów w WebGL

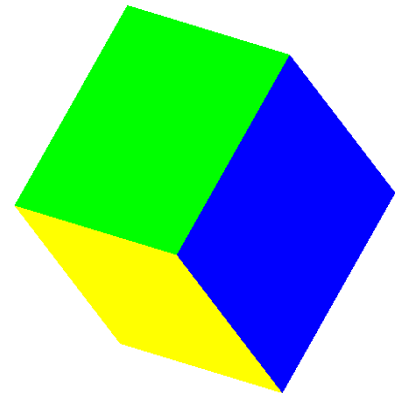
1. Najpierw definiujemy geometrię używając tablic JavaScript.
2. Następnie tworzymy bufory WebGL (VBO, Vertex Buffer Object).
3. W trzecim kroku wskazujemy na VBO utworzone w poprzednim kroku w celu przechowywania współrzędnych wierzchołków.
4. Na koniec wykorzystujemy IBO do wykonania renderowania.



Drugi przykład z sześcianiem
w OpenGL tym razem

Drugi przykład

- ▶ Należy wyrenderować każdą ścianę w sześcianie innym kolorem
- ▶ Przykład pokazuje:
 - ▶ Proste modelowanie obiektów
 - ▶ Budowę obiektu 3D z obiektów podstawowych (primitives)
 - ▶ Budowanie tych 'primitives' z wierzchołków
 - ▶ Zainicjowanie danych (wierzchołków)
 - ▶ Przygotowanie danych do renderowania
 - ▶ interakcję
 - ▶ animację



Rotate X Rotate Y Rotate Z Toggle Rotation



Zainicjowanie danych sześcianu

- ▶ Każdą ścianę sześcianu tworzymy z dwóch trójkątów.
- ▶ Ile potrzebujemy pamięci
 - ▶ $(6 \text{ ścian}) * (2 \text{ trójkąty/ścianę}) * (3 \text{ wierzchołki/trójkąt})$

```
const int NumVertices = 36;
```

- ▶ W celu uproszczenia komunikacji z GLSL, możemy użyć klasy vec4 (implementowanej w C++), zbliżonej do typu vec4 w GLSL.

```
typedef vec4 point4;  
typedef vec4 color4;
```



Zanicjowanie danych sześcianu (2)

- ▶ Zanim zainicjujemy VBO, potrzebujemy zdefiniować dane
- ▶ Nasz sześcián ma dwa atrybuty stowarzyszone z każdym wierzchołkiem:
 - ▶ położenie
 - ▶ kolor
- ▶ W tym przykładzie tworzymy dwie tablice z danymi dla dwóch buforów VBO (choć można by się obejść jednym)

```
point4  vPositions[NumVertices];  
color4  vColors[NumVertices];
```



Dane sześcianu

Wpisujemy wierzchołki jednostkowego sześcianu o środku w początku układu współrzędnych i krawędziach równoległych do osi układu.

```
point4 positions[8] = {  
    point4( -0.5, -0.5,  0.5, 1.0 ),  
    point4( -0.5,  0.5,  0.5, 1.0 ),  
    point4(  0.5,  0.5,  0.5, 1.0 ),  
    point4(  0.5, -0.5,  0.5, 1.0 ),  
    point4( -0.5, -0.5, -0.5, 1.0 ),  
    point4( -0.5,  0.5, -0.5, 1.0 ),  
    point4(  0.5,  0.5, -0.5, 1.0 ),  
    point4(  0.5, -0.5, -0.5, 1.0 )  
};
```



Dane sześcianu (cd)

- ▶ Następnie wypełniamy tablicę kolorów RGBA

```
color4 colors[8] = {  
    color4( 0.0, 0.0, 0.0, 1.0 ),    // black  
    color4( 1.0, 0.0, 0.0, 1.0 ),    // red  
    color4( 1.0, 1.0, 0.0, 1.0 ),    // yellow  
    color4( 0.0, 1.0, 0.0, 1.0 ),    // green  
    color4( 0.0, 0.0, 1.0, 1.0 ),    // blue  
    color4( 1.0, 0.0, 1.0, 1.0 ),    // magenta  
    color4( 1.0, 1.0, 1.0, 1.0 ),    // white  
    color4( 0.0, 1.0, 1.0, 1.0 )    // cyan  
};
```



Generowanie ściany sześciianu od zera

- ▶ Upraszczamy nasze działania przez użycie własnej funkcji `quad()`
 - ▶ Tworzymy dwa trójkąty tworzące jedną ścianę i przypisujemy kolory wierzchołkom.



Generowanie ściany sześciangu od zera

```
int Index = 0; // global variable indexing into VBO arrays
```

```
void quad( int a, int b, int c, int d )  
{  
    vColors[Index] = colors[a]; vPositions[Index] =  
positions[a]; Index++;  
    vColors[Index] = colors[b]; vPositions[Index] =  
positions[b]; Index++;  
    vColors[Index] = colors[c]; vPositions[Index] =  
positions[c]; Index++;  
    vColors[Index] = colors[a]; vPositions[Index] =  
positions[a]; Index++;  
    vColors[Index] = colors[c]; vPositions[Index] =  
positions[c]; Index++;  
    vColors[Index] = colors[d]; vPositions[Index] =  
positions[d]; Index++;  
}
```



Generowanie sześcianu ze ścian

- ▶ Dla sześcianu tworzymy 12 trójkątów
 - ▶ Mamy 36 wierzchołków z 36 kolorami

```
void colorcube()  
{  
    quad( 1, 0, 3, 2 );  
    quad( 2, 3, 7, 6 );  
    quad( 3, 0, 4, 7 );  
    quad( 6, 5, 1, 2 );  
    quad( 4, 5, 6, 7 );  
    quad( 5, 4, 0, 1 );  
}
```



Vertex Array Objects (VAO) - niedostępne OpenGL ES 2.0

- ▶ Tablice VAO przechowują dane o obiekcie
- ▶ Kroki w używaniu VAO
 - ▶ wygeneruj `glGenVertexArrays()`
 - ▶ Dołącz wybrany VAO i zainicjuj przez wywołanie `glBindVertexArray()`
 - ▶ Uaktualnij bufory VBO stowarzyszone z tym VAO
 - ▶ Dołącz (bind) VAO do użycia w renderowaniu
- ▶ To podejście pozwala odwołać się do wszystkich danych obiektu w jednym wywołaniu
 - ▶ Poprzednio mogło być konieczne wiele wywołań



Użycie VAO

► Utwórz VAO

```
GLuint vao;  
glGenVertexArrays( 1, &vao );  
glBindVertexArray( vao );
```



Przechowywanie atrybutów wierzchołka

- ▶ Dane o wierzchołkach muszą być przechowywane w buforze Vertex Buffer Object (VBO)
- ▶ Do ustawienia VBO musimy:
 - ▶ Utworzyć pusty bufor wywołując

```
glGenBuffers()
```

- ▶ Uaktywnić określony VBO wywołując:

```
glBindBuffer( GL_ARRAY_BUFFER, ... )
```

- ▶ Załadować dane do bufora VBO używając

```
glBufferData( GL_ARRAY_BUFFER, ... )
```



Fragment kodu z VBO

► Utwórz i zainicjuj VBO

```
GLuint buffer;  
glGenBuffers( 1, &buffer );  
glBindBuffer( GL_ARRAY_BUFFER, buffer );  
glBufferData( GL_ARRAY_BUFFER,  
              sizeof(vPositions) + sizeof(vColors),  
              NULL, GL_STATIC_DRAW );  
glBufferSubData( GL_ARRAY_BUFFER, 0,  
                sizeof(vPositions), vPositions );  
glBufferSubData( GL_ARRAY_BUFFER, sizeof(vPositions),  
                sizeof(vColors), vColors );
```



Fragment kodu

- ▶ Skojarz zmienne shadera z tablicą wierzchołków
 - ▶ Należy to zrobić po załadowaniu shaderów

```
GLuint vPosition =  
    glGetAttribLocation( program, "vPosition" );  
glEnableVertexAttribArray( vPosition );  
glVertexAttribPointer( vPosition, 4, GL_FLOAT,  
    GL_FALSE, 0, BUFFER_OFFSET(0) );
```

```
GLuint vColor =  
    glGetAttribLocation( program, "vColor" );  
glEnableVertexAttribArray( vColor );  
glVertexAttribPointer( vColor, 4, GL_FLOAT,  
    GL_FALSE, 0,  
    BUFFER_OFFSET(sizeof(vPositions)) );
```



Rysowanie podstawowych elementów geometrycznych (Geometric Primitives)

- ▶ Dla ciągłych grup wierzchołków:

```
glDrawArrays( GL_TRIANGLES, 0, NumVertices );
```

- ▶ Inicjuje vertex shader



Koniec części ogólnej





Co widzimy kamera?

- ▶ Musimy określić ostrosłup widzenia – fragment świata jaki widzimy
- ▶ Robimy to w dwóch krokach
 - ▶ Określamy parametry ostrosłupa (transformacja rzutowania)
 - ▶ Określamy jego położenie w przestrzeni (transformacja modelu – widoku)
- ▶ Wszystko poza ostrosłupem widzenia jest wycinane



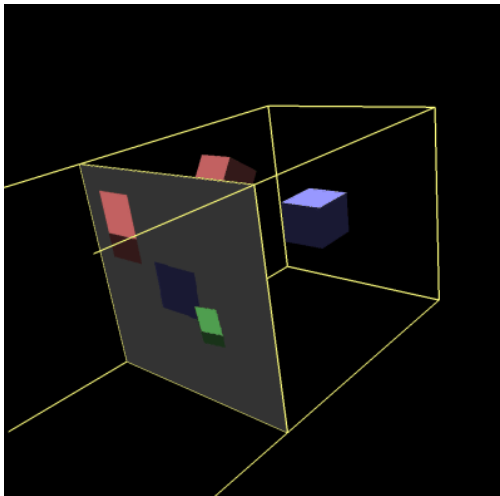
Co widzimy kamerą cd

- ▶ Rzutowanie OpenGL/WebGL używa współrzędnych obserwatora (eye coordinates)
 - ▶ Obserwator (eye) jest umiejscowiony na początku układu współrzędnych
 - ▶ Patrzy w kierunku ujemnych wartości osi z
- ▶ Macierze rzutowania stosują model oparty na sześciu płaszczyznach:
 - ▶ płaszczyzny near i far
 - ▶ określają graniczne odległości od obserwatora (wartości dodatnie)
 - ▶ płaszczyzny ograniczające z boku
 - ▶ top & bottom, left & right

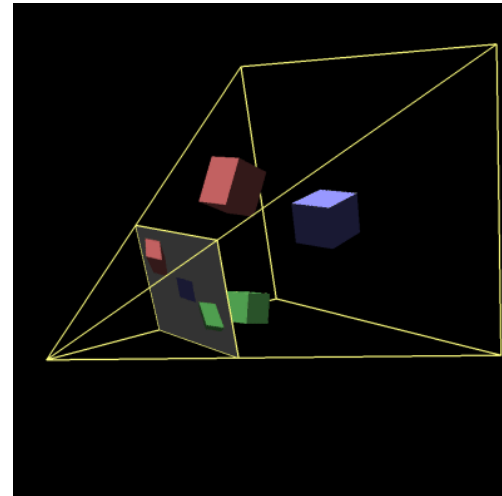


Co widzimy kamerą cd.

Rzut ortogonalny



Rzut perspektywiczny



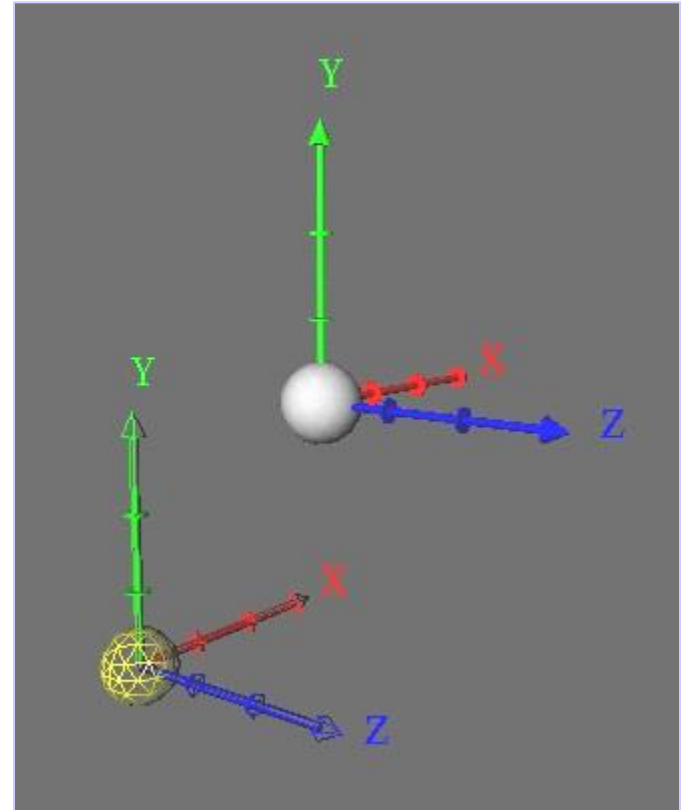
$$O = \begin{bmatrix} \frac{2}{\text{right} - \text{left}} & 0 & 0 & -\frac{\text{right} + \text{left}}{\text{right} - \text{left}} \\ 0 & \frac{2}{\text{top} - \text{bottom}} & 0 & -\frac{\text{top} + \text{bottom}}{\text{top} - \text{bottom}} \\ 0 & 0 & \frac{2}{\text{near} - \text{far}} & \frac{\text{far} + \text{near}}{\text{far} - \text{near}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$P = \begin{bmatrix} \frac{2 \cdot \text{near}}{\text{right} - \text{left}} & 0 & \frac{\text{right} + \text{left}}{\text{right} - \text{left}} & 0 \\ 0 & \frac{2 \cdot \text{near}}{\text{top} - \text{bottom}} & \frac{\text{top} + \text{bottom}}{\text{top} - \text{bottom}} & 0 \\ 0 & 0 & \frac{\text{far} + \text{near}}{\text{far} - \text{near}} & \frac{2 \cdot \text{far} \cdot \text{near}}{\text{far} - \text{near}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Translacja

- ▶ Można interpretować jako przesunięcie początku układu współrzędnych

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Vertex Shader w obrotach sześcianu

```
attribute vec4 vPosition;  
attribute vec4 vColor;  
varying vec4 color;  
uniform vec3 theta;  
  
void main()  
{  
    // Compute the sines and cosines of theta for  
    // each of the three axes in one computation.  
    vec3 angles = radians( theta );  
    vec3 c = cos( angles );  
    vec3 s = sin( angles );
```



Vertex Shader w obrotach sześcianu cd.

// Remember: these matrices are column-major

```
mat4 rx = mat4( 1.0,  0.0,  0.0, 0.0,
                0.0,  c.x,  s.x, 0.0,
                0.0, -s.x,  c.x, 0.0,
                0.0,  0.0,  0.0, 1.0 );
```

```
mat4 ry = mat4( c.y, 0.0, -s.y, 0.0,
                0.0, 1.0,  0.0, 0.0,
                s.y, 0.0,  c.y, 0.0,
                0.0, 0.0,  0.0, 1.0 );
```



Vertex Shader w obrotach sześcianu cd.

```
mat4 rz = mat4( c.z, -s.z, 0.0, 0.0,  
                s.z,  c.z, 0.0, 0.0,  
                0.0,  0.0, 1.0, 0.0,  
                0.0,  0.0, 0.0, 1.0 );
```

```
color = vColor;
```

```
gl_Position = rz * ry * rx * vPosition;
```

```
}
```



Wysyłanie kątów z naszej aplikacji

```
// in init()
```

```
var theta = [ 0, 0, 0 ];  
var axis = 0;  
thetaLoc = gl.getUniformLocation(program,  
"theta");
```

```
// set axis and flag via buttons and event  
listeners
```

```
// in render()
```

```
if(flag) theta[axis] += 2.0;  
gl.uniform3fv(thetaLoc, theta);
```

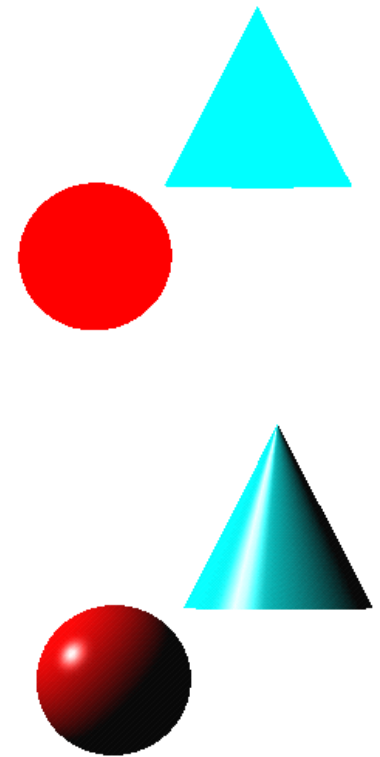




Oświetlenie

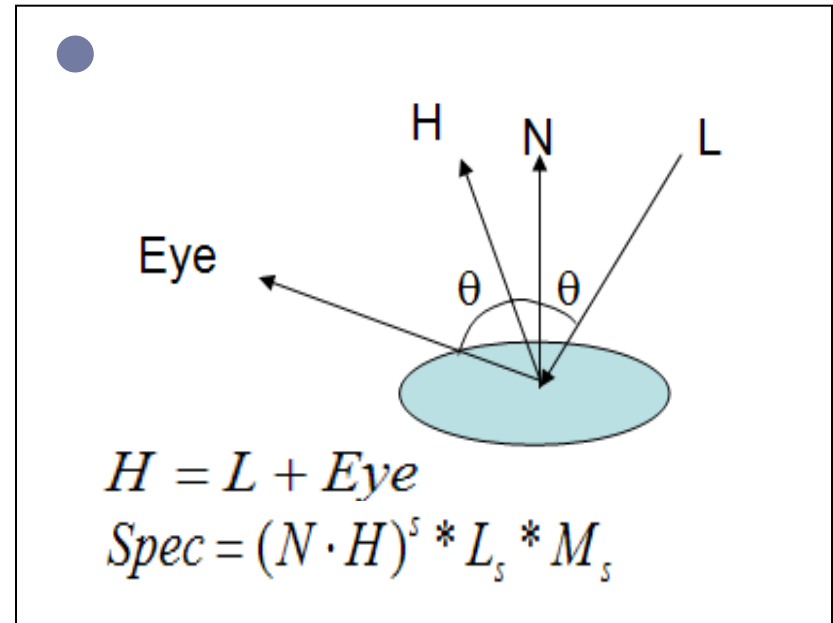
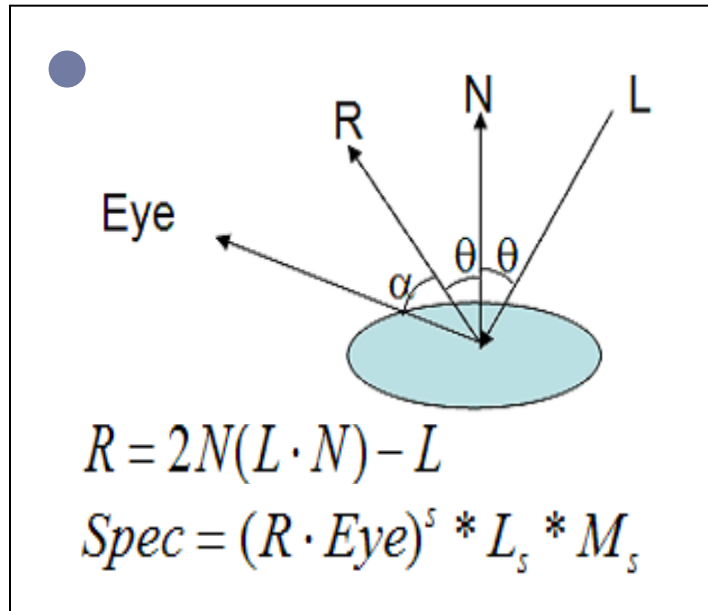
Przypomnienie podstaw

- ▶ **Symulujemy jak obiekty odbijają światło**
 - ▶ materiał
 - ▶ Położenie źródła i barwa światła
 - ▶ Globalne parametry oświetlenia
- ▶ **Implementacja**
 - ▶ vertex shader (szybkość)
 - ▶ fragment shader (lepsza jakość)

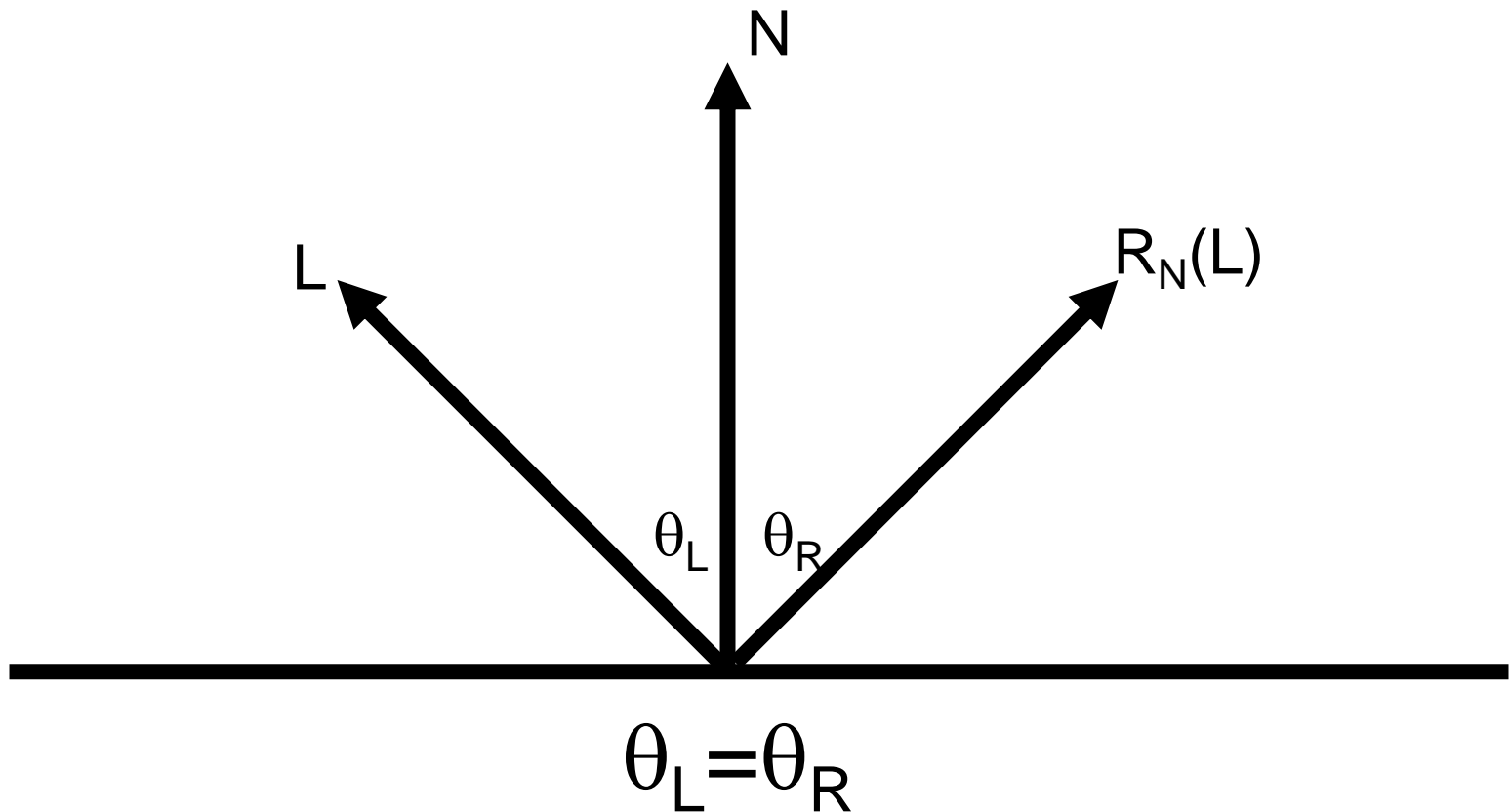


Przypomnienie odbicia połyskliwego.

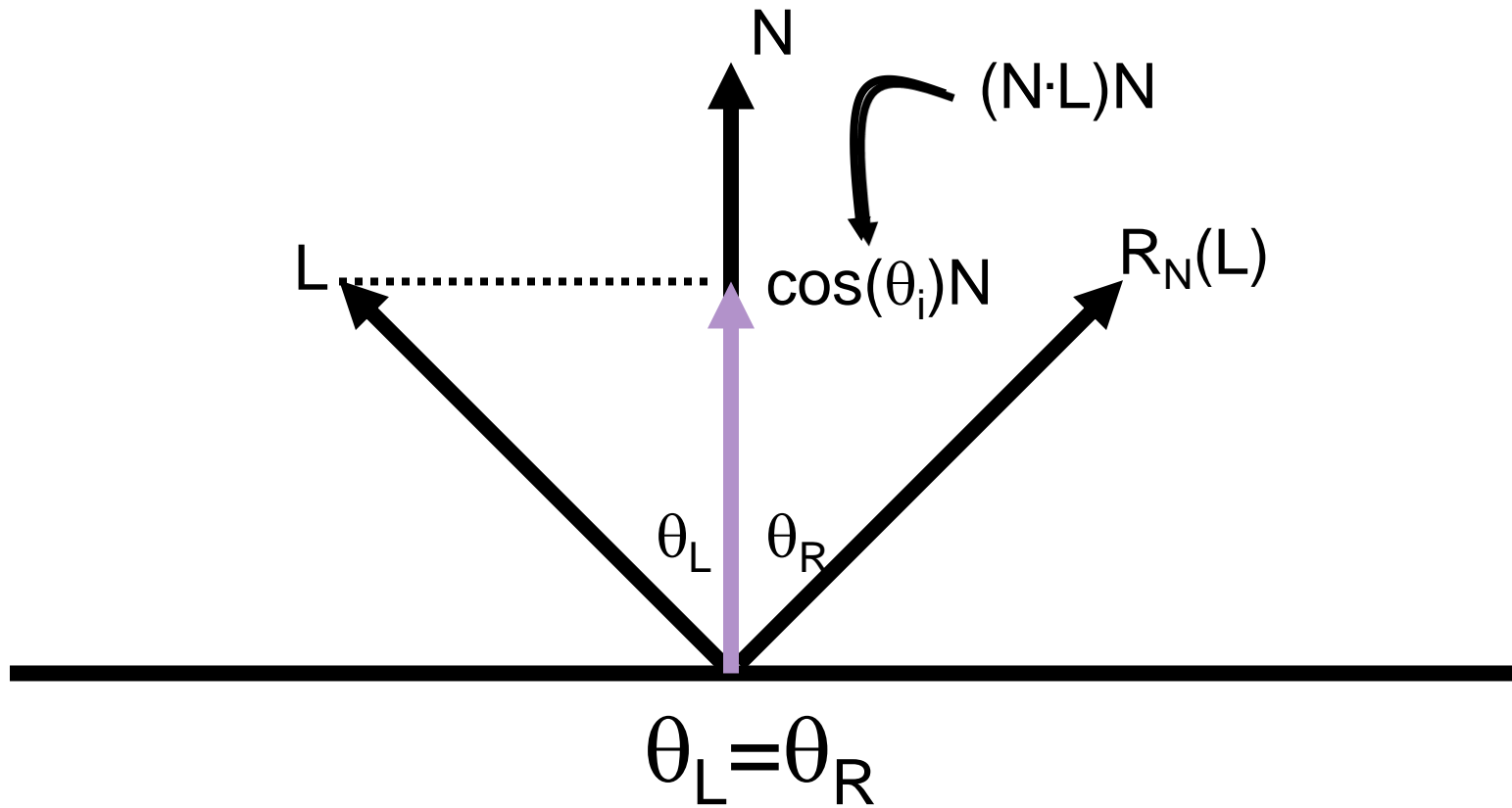
Model Blinna-Phonga vs model Phong



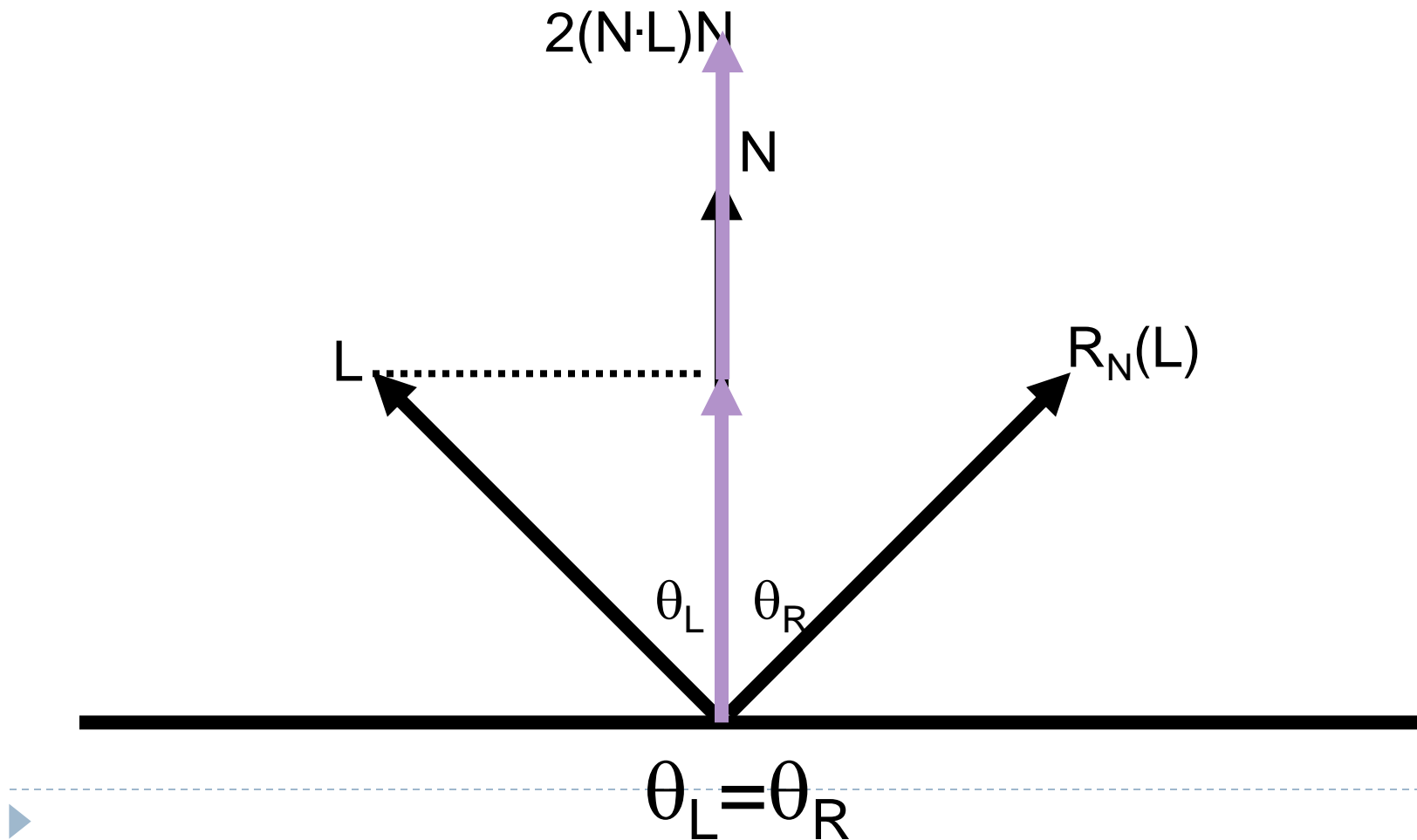
Skąd się bierze wzór w odbiciu Phongu?



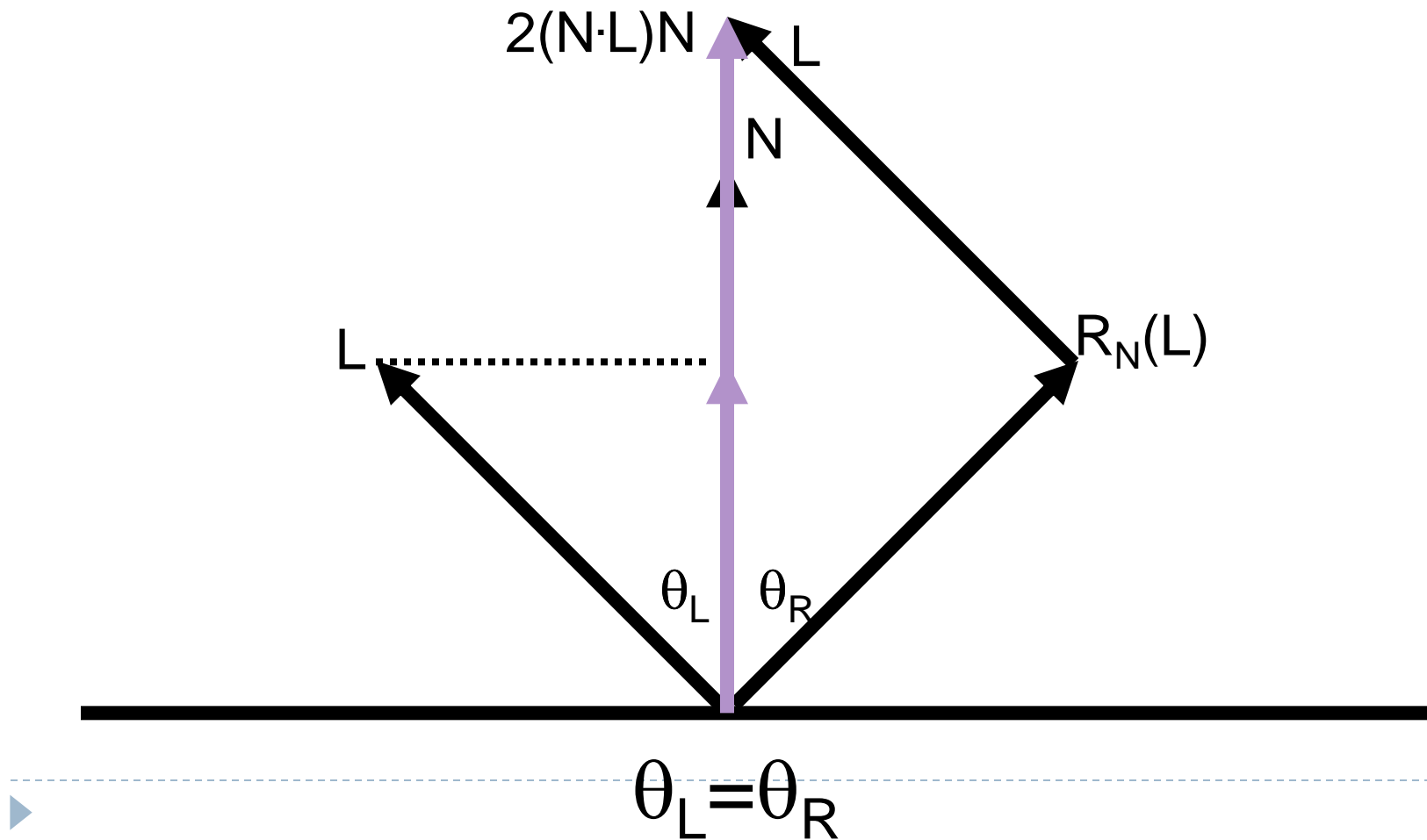
Cd 1.



Cd 2.



Cd 3 ... i gotowe



Dodajemy oświetlenie sześcianu (tutaj całość obliczeń załatwia vertex shader)

```
// vertex shader
```

```
in vec4 vPosition;
```

```
in vec3 vNormal;
```

```
out vec4 color;
```

```
uniform vec4 AmbientProduct, DiffuseProduct,  
             SpecularProduct;
```

```
uniform mat4 ModelView;
```

```
uniform mat4 Projection;
```

```
uniform vec4 LightPosition;
```

```
uniform float Shininess;
```



2

```
void main()
{
    // Transform vertex position into eye
    coordinates
    vec3 pos = vec3(ModelView * vPosition);

    vec3 L = normalize(LightPosition.xyz - pos);
    vec3 E = normalize(-pos);
    vec3 H = normalize(L + E);

    // Transform vertex normal into eye coordinates
    vec3 N = normalize(vec3(ModelView * vNormal));
```



3

```
// Compute terms in the illumination equation
vec4 ambient = AmbientProduct;

float Kd = max( dot(L, N), 0.0 );
vec4  diffuse = Kd*DiffuseProduct;

float Ks = pow( max(dot(N, H), 0.0), Shininess );
vec4  specular = Ks * SpecularProduct;
if( dot(L, N) < 0.0 )
    specular = vec4(0.0, 0.0, 0.0, 1.0)

gl_Position = Projection * ModelView * vPosition;

color = ambient + diffuse + specular;
color.a = 1.0;
}
```

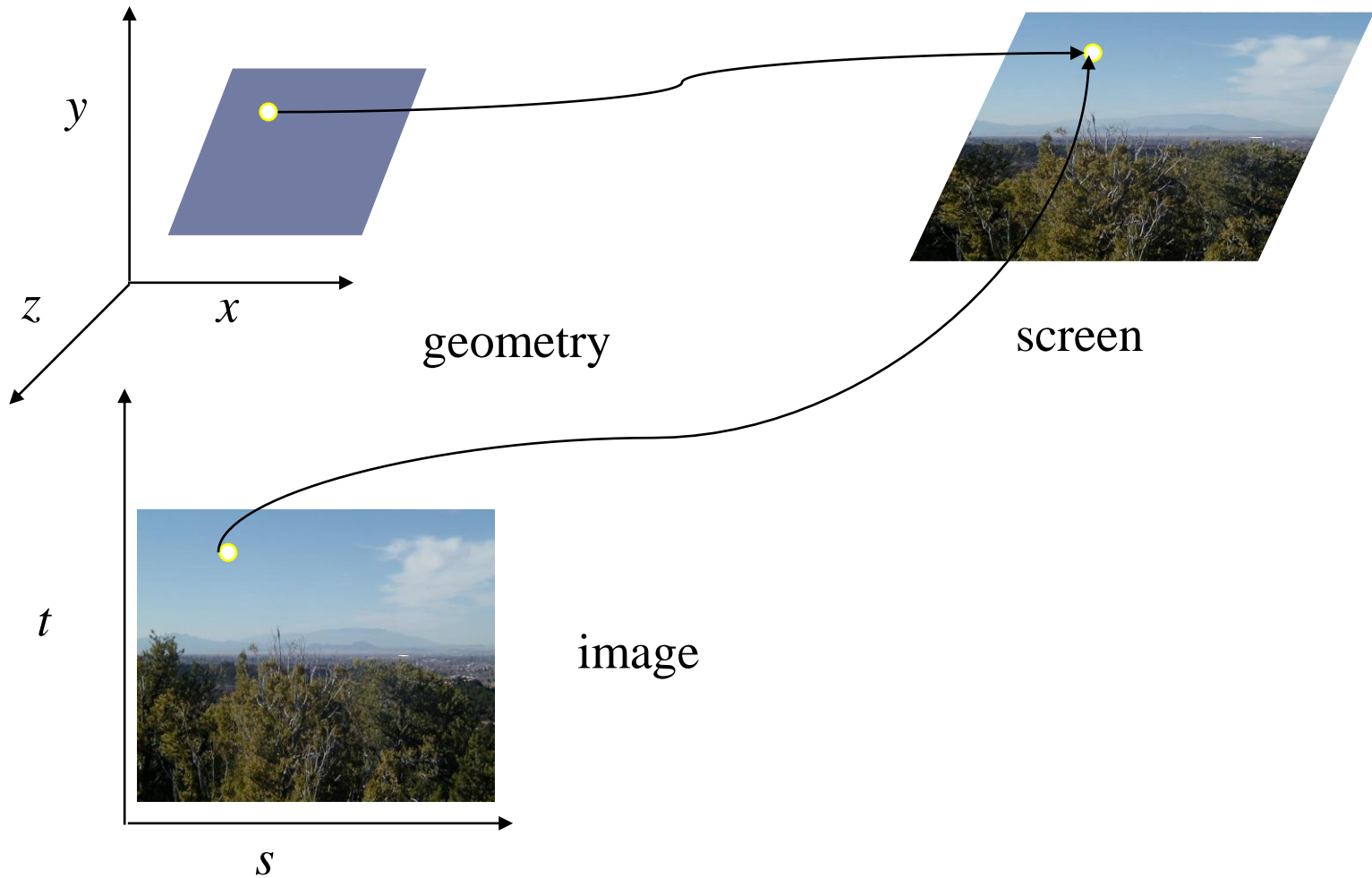


A dark blue vertical bar is positioned on the left side of the slide, spanning the height of the first text box.

Mapowanie tekstur

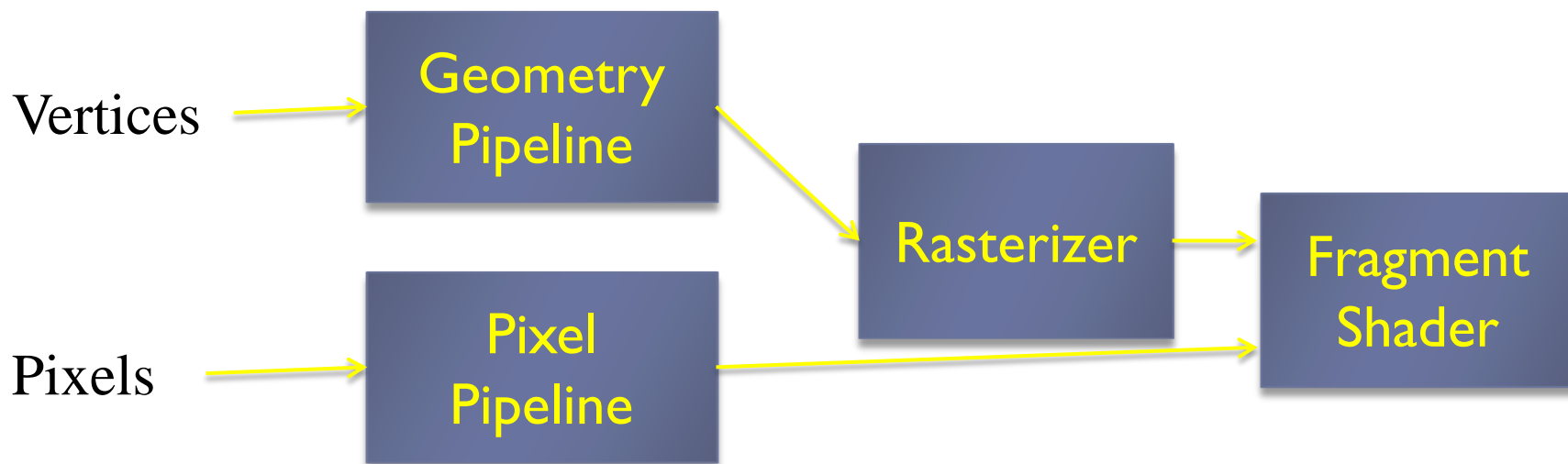


Mapowanie tekstur



Mapowanie tekstur w potoku graficznym OpenGL/WebGL

- ▶ Obrazy (tekstury) i geometria (wierzchołki) są przekazywane różnymi strumieniami danych do GPU i spotykają się na etapie rasteryzacji
 - ▶ Potok graficzny bez tłumaczenia



Użycie tekstur

- ▶ Zwykle sprowadza się do trzech kroków:
 1. Specyfikacji tekstury, czyli:
 - ▶ Wczytania lub wygenerowania obrazu (tablicy tekseli)
 - ▶ Wskazania go jako teksturę (nie każda tablica od razu jest teksturą)
 - ▶ Włączenia opcji teksturowania
 2. Przypisania współrzędnych tekstury do wierzchołków (i na odwrót)
 3. Zdefiniowanie parametrów tekstury
 - ▶ Filtracja, warunki brzegowe, etc.



Tekstury jako obiekty (Texture Objects)

- ▶ WebGL obsługuje obrazy. Zapisuje je w trybie:
 - ▶ one image per one texture object (wiadomo)
- ▶ Najpierw tworzymy pusty obiekt tekstury (texture object):

```
var texture = gl.createTexture();
```

- ▶ Następnie uaktywniamy ten obiekt i określamy jego typ (tekstura 2D – najbardziej typowa, a w WebGL chyba jedyna):

```
gl.bindTexture( gl.TEXTURE_2D, texture );
```



Specyfikowanie tekstury jako obrazu

- ▶ W kolejnym kroku definiujemy teksturę na podstawie tablicy tekselei umieszczonej w głównej pamięci CPU (darujemy sobie w tym miejscu opis parametrów)

```
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, texSize,  
              texSize, 0,  gl.RGBA, gl.UNSIGNED_BYTE, image);
```

- ▶ Możemy też zdefiniować teksturę na podstawie obrazu w standardowej formie wyspecyfikowanego za pomocą tagu <image> w pliku HTML.

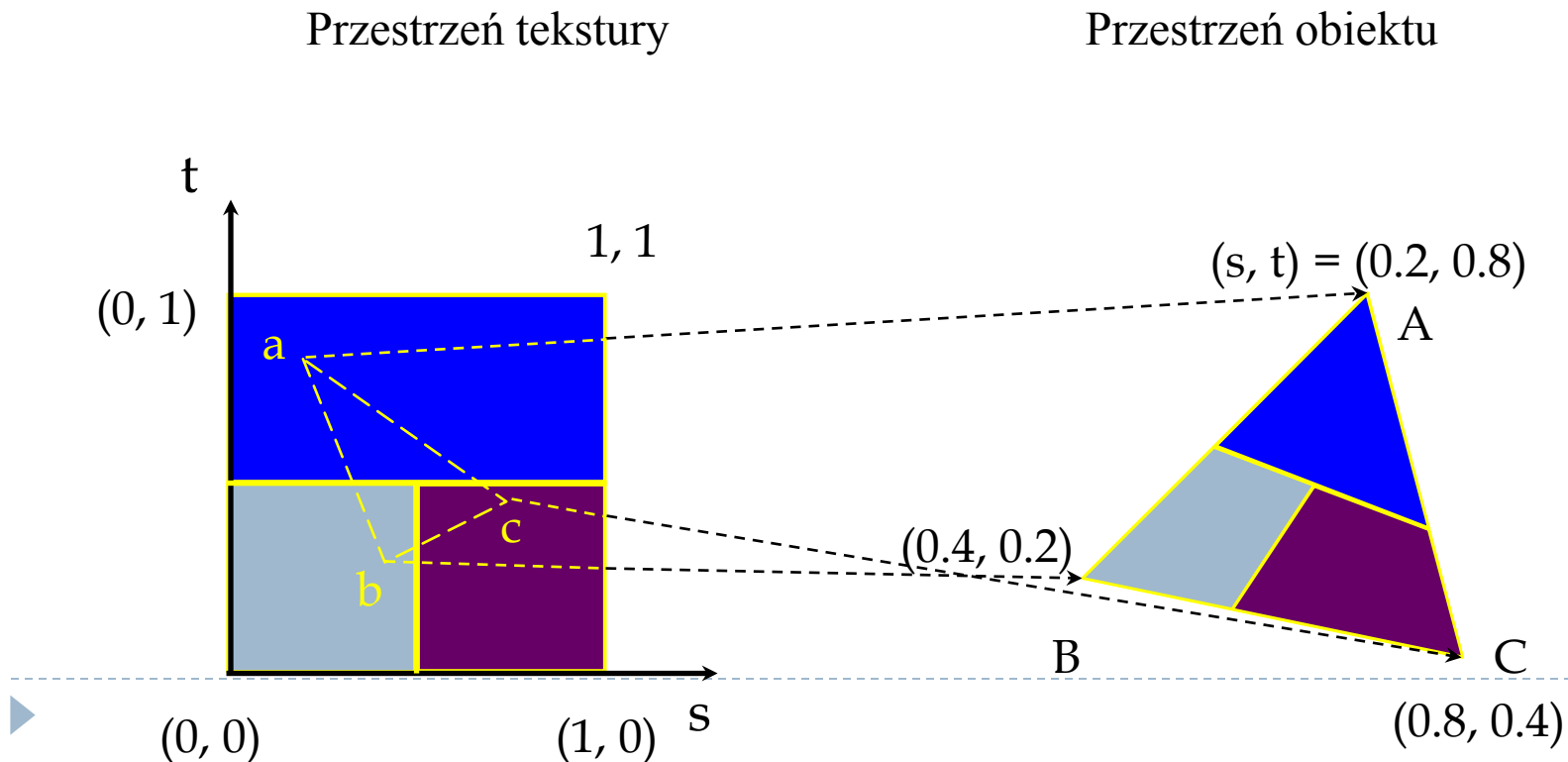
```
var image = document.getElementById("texImage");
```

```
gl.texImage2D( gl.TEXTURE_2D, 0, gl.RGB,  
              gl.RGB, gl.UNSIGNED_BYTE, image );
```



Mapowanie współrzędnych tekstury

- ▶ Opieramy się na współrzędnych tekstury określonych zmiennymi (s, t)
- ▶ Współrzędne muszą być określone w każdym wierzchołku



Użycie tekstury w shaderze

```
precision mediump float;
```

```
varying vec4 fColor;
```

```
varying vec2 fTexCoord;
```

```
uniform sampler2D texture;
```

```
void main()
```

```
{
```

```
gl_FragColor = fColor*texture2D(texture, fTexCoord );
```

```
}
```



Nakładamy teksturę na sześćcian

```
// add texture coordinate attribute to quad  
function
```

```
function quad(a, b, c, d)
```

```
{   pointsArray.push(vertices[a]);  
    colorsArray.push(vertexColors[a]);  
    texCoordsArray.push(texCoord[0]);
```

```
    pointsArray.push(vertices[b]);  
    colorsArray.push(vertexColors[a]);  
    texCoordsArray.push(texCoord[1]);
```

```
    .
```

```
    .
```

```
}  
}
```

Tworzenie obrazu dla tekstury

```
var image1 = new Array()
  for (var i =0; i<texSize; i++)  image1[i] = new Array();
  for (var i =0; i<texSize; i++)
    for ( var j = 0; j < texSize; j++)
      image1[i][j] = new Float32Array(4);
  for (var i =0; i<texSize; i++) for (var j=0; j<texSize; j++) {
    var c = (((i & 0x8) == 0) ^ ((j & 0x8) == 0));
    image1[i][j] = [c, c, c, 1];    }

// Convert floats to ubytes for texture
var image2 = new Uint8Array(4*texSize*texSize);
  for (var i = 0; i < texSize; i++ )
    for (var j = 0; j < texSize; j++ )
      for(var k =0; k<4; k++)
        image2[4*texSize*i+4*j+k] = 255*image1[i][j][k];
```



Tworzenie obiektu tekstury

```
texture = gl.createTexture();

gl.activeTexture( gl.TEXTURE0 );
gl.bindTexture( gl.TEXTURE_2D, texture );
//    gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, texSize,
texSize, 0, gl.RGBA, gl.UNSIGNED_BYTE, image);
gl.generateMipmap( gl.TEXTURE_2D );
gl.texParameteri( gl.TEXTURE_2D,
gl.TEXTURE_MIN_FILTER, gl.NEAREST_MIPMAP_LINEAR );
gl.texParameteri( gl.TEXTURE_2D,
gl.TEXTURE_MAG_FILTER, gl.NEAREST );
```



Vertex Shader

```
attribute vec4 vPosition;  
attribute vec4 vColor;  
attribute vec2 vTexCoord;  
  
varying vec4 color;  
varying vec2 texCoord;  
  
void main()  
{  
    color          = vColor;  
    texCoord       = vTexCoord;  
    gl_Position    = vPosition;  
}
```



Fragment Shader

```
varying vec4 color;  
varying vec2 texCoord;  
  
uniform sampler texture;  
  
void main()  
{  
    gl_FragColor = color * texture( texture, texCoord );  
}
```

