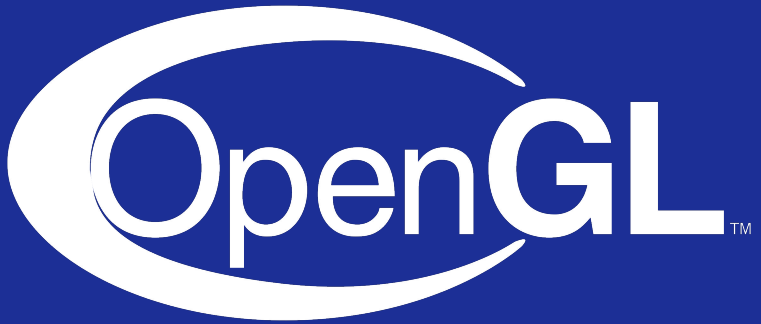
Grafika komputerowa
Wykład 8

OpenGL - shadery



dr inż. Michał Chwesiuk
Michal.Chwesiuk@pw.edu.pl

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

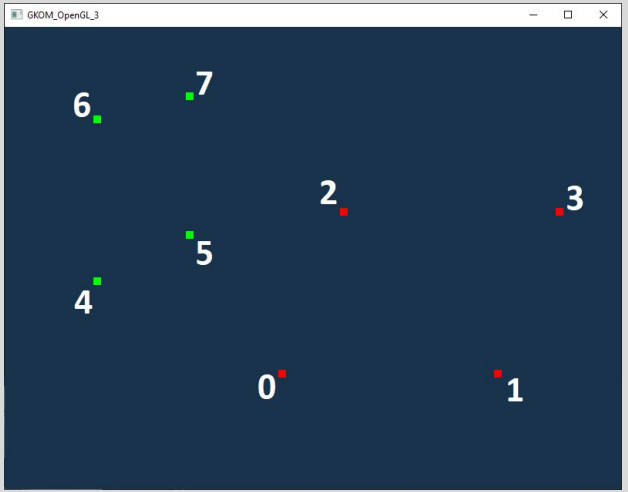
Plan prezentacji

- Indeksowanie VBO
- Back-face culling
- Teksturowanie
- Oświetlenie
- Rendering do tekstury
- Kamera TPP

Indeksowanie VBO

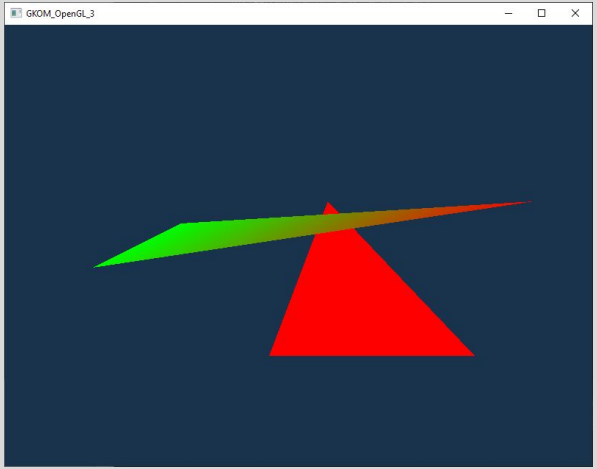
- **Element Array Buffer** pozwala na wskazanie GPU kolejności rysowania wierzchołków
- W celu użycia EAB musimy zamiast funkcji glDrawArrays() wykorzystać **glDrawElements()**
- **Należy pamiętać o przypisaniu VBO lub VAO!**

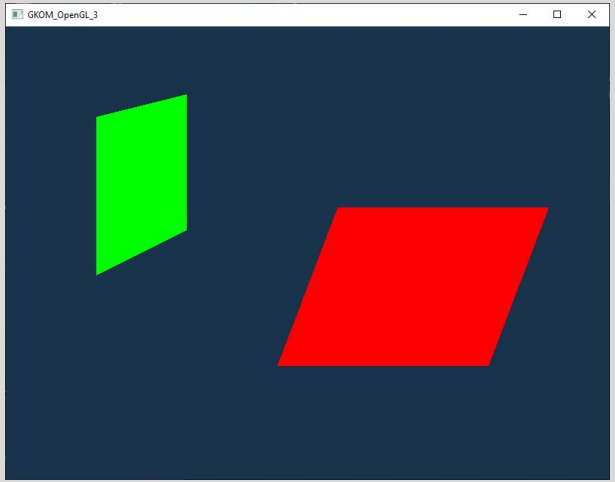
```
std::vector<GLshort> indices;
// First quad
indices.push_back(0);
indices.push_back(1);
indices.push_back(2);
indices.push_back(1);
indices.push_back(2);
indices.push_back(3);
// Second quad
indices.push_back(4);
indices.push_back(5);
indices.push_back(6);
indices.push_back(5);
indices.push_back(6);
indices.push_back(7);

GLuint EAB;
glGenBuffers(1, &EAB);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EAB);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(GLushort),
&indices[0], GL_STATIC_DRAW);
```




```
// DRAW VAO WITH ELEMENT ARRAY BUFFER //
glBindVertexArray(quadsVAO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, quadsEAB);
glDrawElements(
    GL_TRIANGLES,
    quadsElementsCount,
    GL_UNSIGNED_SHORT,
    (void*)0
);

// DRAW VERTEX ARRAY OBJECT //
// glBindVertexArray(quadsVAO);
// glDrawArrays(GL_TRIANGLES, 0, 8);
```

Indeksowanie VBO



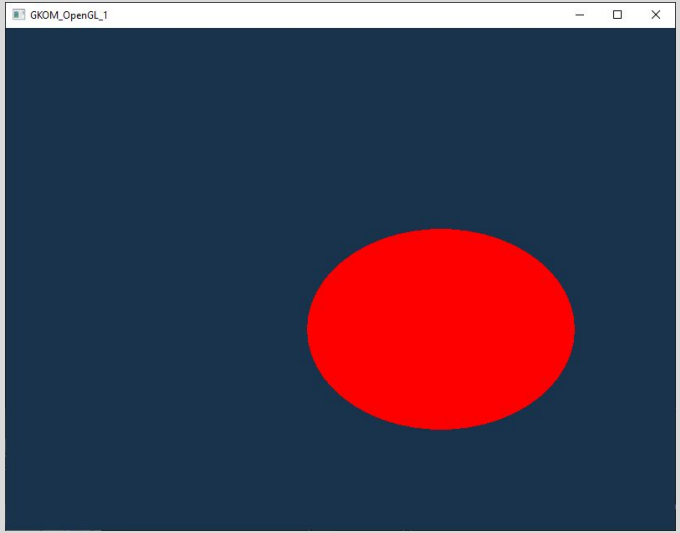



```
// DRAW VERTEX ARRAY OBJECT //
glBindVertexArray(quadsVAO);
glDrawArrays(GL_TRIANGLES, 0, 8);
```

```
// DRAW VAO WITH ELEMENT ARRAY BUFFER //
glBindVertexArray(quadsVAO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, quadsEAB);
glDrawElements(
    GL_TRIANGLES,
    quadsElementsCount,
    GL_UNSIGNED_SHORT,
    (void*)0
);
```

Rysowanie koła


```
glm::vec2 circleMiddle = glm::vec2(0.3f, -0.2f);
float circleRadius = 0.4f;
unsigned int segments = 50;

std::vector<glm::vec2> positions;

positions.push_back(circleMiddle);

for (int i = 0; i <= segments; i++)
{
    float phi = float(i) / segments * 2 * std::numbers::pi;

    float x = circleMiddle.x + circleRadius * std::cos(phi);
    float y = circleMiddle.y + circleRadius * std::sin(phi);

    positions.push_back(glm::vec2(x, y));
}
```

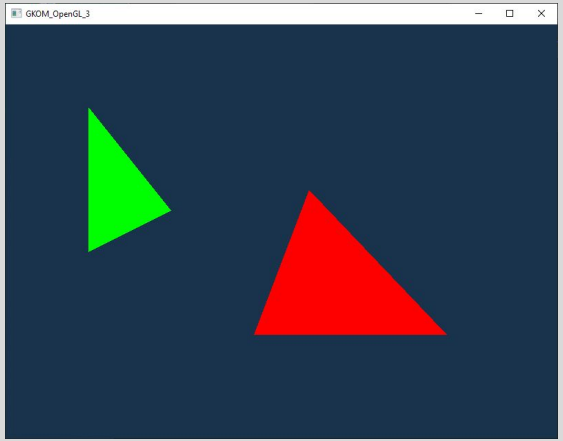



```
while (!glfwWindowShouldClose(_window))
{
    glClearColor(0.1f, 0.2f, 0.3f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    simpleProgram.Activate();

    glBindVertexArray(triangleVAO);
    glDrawArrays(GL_TRIANGLE_FAN, 0, 52);

    processInput();

    glfwSwapBuffers(_window);
    glfwPollEvents();
}
```

Back-face culling

- Odrzucanie tylnych ścian (ang. **Back-face culling**) pozwala na redukcję renderowanej geometrii.
- Należy zdefiniować, która ściana będzie traktowana jako przednia.
  - **GL_CW** - kolejność według ruchu wskazówek zegara (clockwise)
  - **GL_CCW** - kolejność przeciwna do ruchu wskazówek zegara (counter clockwise)
- **Należy pamiętać o zachowaniu kolejności wierzchołków podczas dodawania geometrii!**
- **Back-face culling wraz z parametrami aktywujemy raz!**

```
glFrontFace(GL_CCW); // alternative GL_CW
glCullFace(GL_BACK); // alternative GL_FRONT

glEnable(GL_CULL_FACE);
```




```
std::vector<GLshort> indices;
// First quad
indices.push_back(0);
indices.push_back(1);
indices.push_back(2);
indices.push_back(2); // <-- 1
indices.push_back(1); // <-- 2
indices.push_back(3);
// Second quad
indices.push_back(4);
indices.push_back(5);
indices.push_back(6);
indices.push_back(6); // <-- 5
indices.push_back(5); // <-- 6
indices.push_back(7);
```

Teksturowanie

- **Teksturowanie** jest to technika w grafice komputerowej, której celem jest zwiększenie szczegółowości renderowanych powierzchni za pomocą tekstur.
- Tekstura jest to pewna funkcja (najczęściej w formie bitmapy) zawierająca informację o tym jak przekształcić wartość koloru danego fragmentu powierzchni.
- W procesie teksturowanie zachodzi proces **mapowania tekstury**.

Tworzenie tekstury

- Tekstura jest to funkcja której celem jest modyfikacja powierzchni.
- Teksturę dwuwymiarową tworzymy za pomocą funkcji **glGenTexture()**.

```
GLuint texture;
glGenTextures(1, &texture);
```

- Aby modyfikować parametry tekstury musimy ją "**zbindować**" przy użyciu glBindTexture().

```
glBindTexture(GL_TEXTURE_2D, texture);
```

- Aby wypełnić teksturę danymi o rastrze należy wykorzystać funkcję **glTexImage2D()**.

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, imageWidth, imageHeight,
0, GL_RGB, GL_UNSIGNED_BYTE, image);
```

  - **Należy zwrócić uwagę na zgodność parametrów dot. formatu obrazu** (GL_RBG oraz GL_UNSIGNED_BYTE)

- Oprócz wypełnienia tekstury danymi, należy ustawić parametry **wrappingu** oraz **filtrowania**.
- **W OpenGL Istnieją inne rodzaje tekstur** (np. GL_TEXTURE_1D, GL_TEXTURE_3D, GL_TEXTURE_2D_ARRAY, GL_TEXTURE_CUBE_MAP, GL_TEXTURE_CUBE_MAP_ARRAY ...)

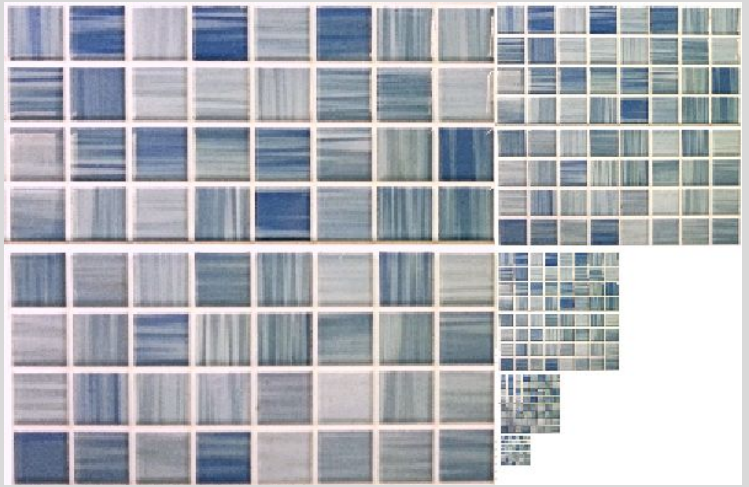
Filtrowanie i wrapping tekstury

- Parametry tekstury ustawia się za pomocą funkcji **glTexParameteri()**
- Filtrowanie tekstury ma na celu zdefiniowanie zachowania się podczas nakładania pomniejszonej, lub powiększonej tekstury:
  - **Zachowanie się dla powiększonej tekstury** - parametr GL_TEXTURE_MIN_FILTER
    - GL_NEAREST - najbliższy piksel
    - GL_LINEAR - interpolacja liniowa
  - **Zachowanie się dla pomniejszonej tekstury** - parametr GL_TEXTURE_MIN_FILTER
    - Takie same jak dla powiększonej tekstury oraz wartości z osobną metodą dla mipmap
- Wybór metody wrappingu tekstury polega na zdefiniowaniu zachowania jej nakłada podczas wyjścia poza zakres <0, 1>:
  - Wrapping stosuje się osobno dla współrzędnej X (parametr GL_TEXTURE_WRAP_S) i współrzędnej Y (GL_TEXTURE_WRAP_T)
  - Możliwe wartości:
    - GL_CLAMP_TO_EDGE - wartość ostatniego piksela
    - GL_CLAMP_TO_BORDER - wartość ramki
    - GL_MIRRORED_REPEAT - powtórzenie tekstury z odbiciem
    - GL_REPEAT - powtórzenie tekstury
    - GL_MIRROR_CLAMP_TO_EDGE - ustawienie ostatniego piksela z drugiej strony

Mipmapping

- **Mipmapy** są to wcześniej wyliczone sekwencje pomniejszonych wersji oryginalnej tekstury.
- Każdy kolejny poziom tekstury to ma wymiary **dwukrotnie mniejsze** dla każdego wymiaru.
- W trakcie procesu teksturowania procesor graficzny wybiera który poziom wykorzystać.
- **Stosuje się w celu przyspieszenia obliczeń kosztem większego zużycia pamięci**, w której przechowywana jest tekstura (⅓ więcej).
- Generowane za pomocą **glGenerateMipmap()**.
- **Wymagane przekazanie do parametru tekstury GL_TEXTURE_MIN_FILTER wartości zawierającej sposób filtrowania mipmap** :
  - GL_NEAREST_MIPMAP_NEAREST
  - GL_LINEAR_MIPMAP_NEAREST
  - GL_NEAREST_MIPMAP_LINEAR
  - GL_LINEAR_MIPMAP_LINEAR

```
glGenerateMipmap(GL_TEXTURE_2D);
```



Teksturowanie w programie cieniującym

- Celem przekazania tekstury do programu cieniującego należy stworzyć w shaderze zmienną uniform typu **sampler2D**.
- **Zmiennej jednorodnej przypisujemy wartość numer jednostki teksturującej** z przedziału od 0 do ilości jednostek teksturujących (minus jeden) w procesorze graficznym, np. 5.
- **Aktywujemy wybraną przez nas jednostkę teksturującą i bindujemy teksturę**.
- Do pobrania koloru tekstury w danej współrzędnej UV w GLSL służy funkcja **texture()**.
- **Współrzędne tekstury wyliczamy poprzez interpolację atrybutu zawierającego UV**.

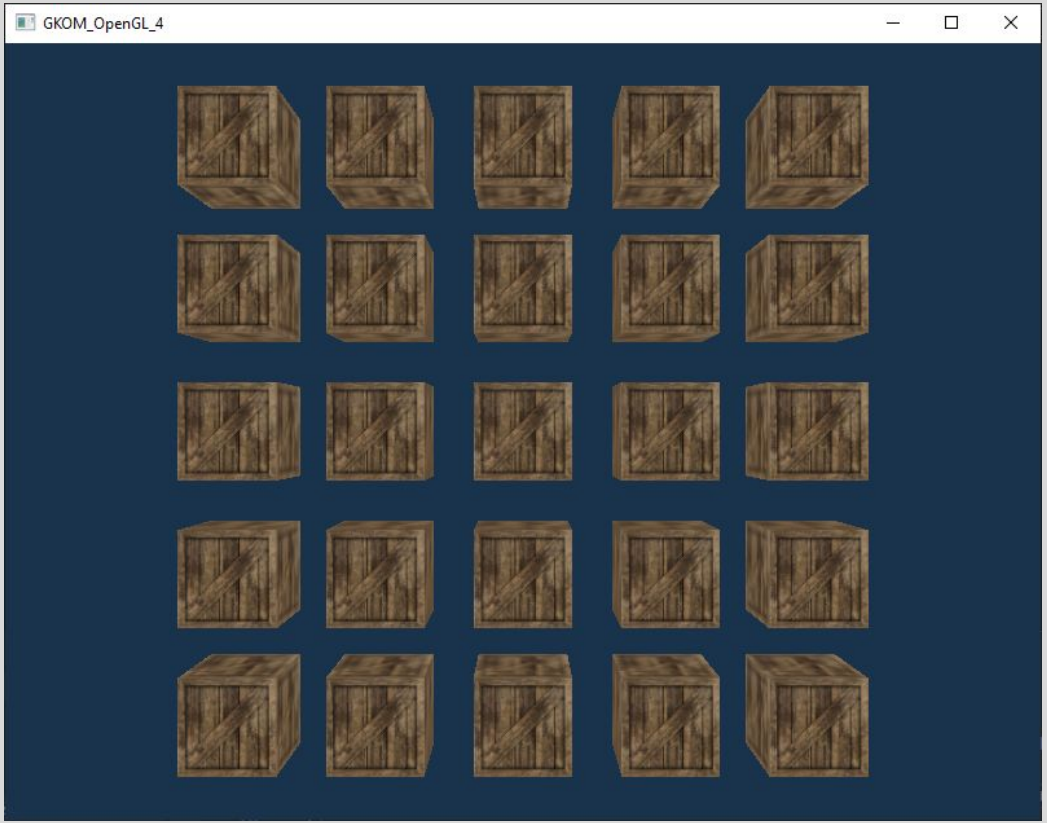
```
glUniform1i(textureProgram.GetUniformID("uTexture"), 5);
glActiveTexture(GL_TEXTURE5);
glBindTexture(GL_TEXTURE_2D, boxTexture);
```

Kod w pętli renderującej

```
#version 330 core

in vec2 vTexCoord;

out vec4 FragColor;

uniform sampler2D uTexture;

void main()
{
    vec4 texColor = texture(uTexture, vTexCoord);

    FragColor = vec4(texColor.xyz, 1.0f);
}
```

Fragment shader

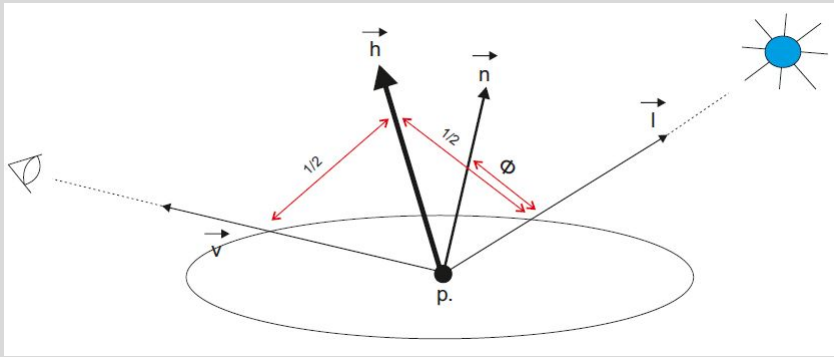
Teksturowanie - wynik



Oświetlenie

- Wzór na **model oświetlenia Blinna Phonga**

$$I = \sum_{i=1}^{N} I_a + I_d + I_s$$



$$I_a = M_a \cdot L_a$$

$$I_d = M_d \cdot L_d \cdot (|\vec{n}| \cdot |\vec{l}|)$$

$$I_s = M_s \cdot L_s \cdot (|\vec{n}| \cdot |\vec{h}|)^{M_{shi}}$$

Oświetlenie

- Przykład fragment shadera realizującego model oświetlenia.
- **Program nie jest kompletny**. Należy zaimplementować vertex shader, zdefiniować atrybuty, uniformy itd.
- **Do shadera należy przekazać parametry światła**, takie jak:
  - pozycja
  - ambient
  - diffuse
  - specular
  - shininess
- Parametry światła można **trzymać w GLSL jako strukturę**.

```
void main() {

    highp vec3 tex = texture2D(texture, fragUV).xyz;
    highp vec3 colorFull = vec3(0, 0, 0);

    highp vec3 N = normalize(fragNormal);
    highp vec3 V = normalize(vertexWorldSpace - cameraPosition);

    highp vec3 L = normalize(light.position - vertexWorldSpace);
    highp vec3 H = normalize(L + V);

    highp float cosNL = dot(N, L);
    cosNL = clamp(cosNL, 0.0, 1.0);

    highp float cosVH = dot(V, H);
    cosVH = clamp(cosVH, 0.0, 1.0);

    highp vec3 colorAmb = material.ambient  * light.ambient;
    highp vec3 colorDif = material.diffuse  * light.diffuse * cosNL;
    highp vec3 colorSpe = material.specular * light.specular * pow(cosVH, material.shininess);

    if(hasTexture == 1)
    {
        colorAmb = colorAmb * tex;
        colorDif = colorDif * tex;
    }

    colorFull = clamp(colorAmb + colorDif + colorSpe, 0.0, 1.0);

    gl_FragColor = vec4(colorFull, 1.0);
}
```

WYDZIAŁ ELEKTRONIKI I TECHNIK INFORMACYJNYCH

Framebuffer

- Bufor ramki (ang. **Framebuffer**) jest to wycinek pamięci w procesorze graficzny, w której przetrzymywana jest bitmapa wypełniana pikselami renderowanej sceny.
- Domyślnie w OpenGL() zadeklarowane są dwa framebuffer'y, do których są przekazywane piksele renderowanej sceny oraz są naprzemiennie rysowane są na wyświetlaczu.
- Możliwe jest stworzenie dodatkowych framebuffer'ów, pełniących zadeklarowane przez nas czynności.
- Utworzone framebuffer'y zawierają dołączone tekstury, do których wstawiamy wygenerowaną przez program zawartość.
- **Framebuffer może posiadać więcej niż jedną teksturę wyjściową!**
- Do poprawnego renderingu sceny do tekstury **konieczne jest ręczne wstawienie bufora odpowiedzialnego za przechowywanie głębokości**.
- **Rozdzielczość tekstur i bufora głębokości w framebuffer'ach powinna być równa rozdzielczości okna**.
- Dodatkowe bufory ramek stosuje się do bardziej zaawansowanych technik renderingu, takich jak mapowanie cieni, postprocessing, forward rendering itd.

Framebuffer

- Framebuffer w OpenGL() tworzymy za pomocą funkcji **glGenFramebuffers()**.

```
GLuint framebuffer;
glGenFramebuffers(1, &framebuffer);
```

- Framebuffer aktywujemy za pomocą **glBindFramebuffer()**
    - Przed renderingiem należy pamiętać o ustawieniu poprawnej wartości transformacji odcinania kamery.

```
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);

glViewport(0, 0, resolution.x, resolution.y);
```

- Aby przywrócić domyślny framebuffer, do funkcji glBindFramebuffer() przekazujemy wartość 0.

```
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

- **Render call wywołujemy w ten sam sposób jak dla domyślnego bufora ramki**.

Dowiązywanie tekstur

- Przez renderowaniem do utworzonego bufora ramki, **należy dowiązać teksturę, w której będzie zapisywany wynik renderingu**.
  - Dowiązywanie tekstury wykonujemy za pomocą funkcji **glFramebufferTexture()**

```
glFramebufferTexture(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, texture, 0);
```

  - GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1, GL_COLOR_ATTACHMENT2
    - identyfikator powiązanej tekstury do bufora ramki.
- Filtrowanie tekstur w buforze ramki powinno być ustawione na GL_NEAREST.
- Za pomocą funkcji **glDrawBuffers()** może wskazać, do których tekstur będziemy przekazywać wynik renderingu.
  - Pozwala to na zmianę tekstury docelowej do renderingu innych obiektów w potoku.

```
GLenum DrawBuffers[2] = { GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1 };

glDrawBuffers(2, DrawBuffers);
```

Bufor głębi

- Aby poprawnie wyrenderować teksturę, **do utworzonych framebuffer'ów musimy dołączyć bufor głębokości!**
  - Tworzymy bufor renderingu za pomocą funkcji **glGenRenderbuffers()**.
  - Bindujemy OpenGL() do utworzonego bufora.
  - Alokujemy pamięć za pococą **glRenderbufferStorage()**.
  - Za pomocą **glFramebufferRenderbuffer()** przekazujemy bufor do framebuffer'a jako bufor głębokości.

```
glGenRenderbuffers(1, &depthBuffer);
glBindRenderbuffer(GL_RENDERBUFFER, depthBuffer);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, resolution.x, resolution.y);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, depthBuffer);
```

- **Alternatywnie możemy wykorzystać teksturę do przechowywania informacji o głębi.**
  - W takim przypadku należy wykorzystać wcześniej użytą funkcję **glFramebufferTexture()** i wskazać typ dołączenia jako głębie (GL_DEPTH_ATTACHMENT).
  - **Rozwiązanie to jest znacznie wolniejsze**, lecz pozwala na wykorzystanie danych o głębi w kolejnych etapach renderingu.

Rendering bufora ramki

```
projectionMatrix = glm::perspective(glm::radians(fieldOfView), windowResolution.x / windowResolution.y, 0.1f, 100.0f);
cameraMatrix = glm::lookAt(cameraPosition, cameraPosition + cameraDirection, cameraUp);
modelMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f, 0.0f));

textureProgram.Activate();
glUniformMatrix4fv(textureProgram.GetUniformID("uCameraMatrix"), 1, GL_FALSE, glm::value_ptr(cameraMatrix));
glUniformMatrix4fv(textureProgram.GetUniformID("uProjectionMatrix"), 1, GL_FALSE, glm::value_ptr(projectionMatrix));
glUniformMatrix4fv(textureProgram.GetUniformID("uModelMatrix"), 1, GL_FALSE, glm::value_ptr(modelMatrix));

framebuffer.Clear();    // glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
framebuffer.Activate(); // glBindFramebuffer(GL_FRAMEBUFFER, _framebufferID);
                        // glViewport(0, 0, FBresolution.x, FBresolution.y);

glUniform1i(textureProgram.GetUniformID("uTexture"), 0);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, boxTexture);

glBindVertexArray(boxVAO);
glDrawArrays(boxVAOPrimitive, 0, boxVAOVertexCount);

glBindFramebuffer(GL_FRAMEBUFFER, 0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glViewport(0, 0, windowResolution.x, windowResolution.y);

/*
 * Do wyświetlenia Framebuffer'a na ekranie, wykorzystujemy tzw. Fullscreen Quad.
 * Czworokąt, który obejmuje cały ekran.
 */

fullscreenQuadProgram.Activate();

glUniform1i(textureProgram.GetUniformID("uTexture"), 0);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, framebuffer.GetTexture(0));

glBindVertexArray(fullscreenQuadVAO);
glDrawArrays(fullscreenQuadPrimitive, 0, fullscreenQuadVAOVertexCount);
```

Kamera TPP

- Obrót kamery według jednej osii można osiągnąć za pomocą **współrzędnych biegunowych**.
- Najczęściej podczas opisywania punktu w przestrzeni dwuwymiarowej wykorzystujemy współrzędne kartezjańskie, które opisują punkt za pomocą dwóch zmiennych : X i Y.
- Współrzędne biegunowe polegają na opisaniu tych współrzędnych za pomocą dwóch innych atrybutów :
  - promień wodzący r - odległość punktu P od środka układu współrzędnych O
  - amplituda punktu φ - kąt pomiędzy osią OX, a wektorem OP

Transformacje punktów do układów

- Z układu kartezjańskiego do układu biegunowego

$$r = \sqrt{X^2 + Y^2}$$

$$\phi = arctan(\frac{Y}{X})$$

- Z układu biegunowego do układu kartezjańskiego

$$X = r * cos(\phi)$$

$$Y = r * sin(\phi)$$

Kamera TPP

- Obrót kamery według dwóch osii można osiągnąć za pomocą **współrzędnych sferycznych**.
- Do przedstawienia tych współrzędnych bardzo pomocna jest wikipedia.
    - Przekształcenie do układu sferycznego

```
r = sqrt(X^2, Y^2, Z^2) // w naszym przypadku zawsze równe 1, nie trzeba liczyć
phi = atan2(Z, X)
theta = acos(Y / r)
```

    - Przekształcenie z układu sferycznego

```
X = r * sin(theta) * cos(phi)
Y = r * cos(theta)
Z = r * sin(theta) * sin(phi)
```

- Aby obracać kamerą za pomocą myszy musimy uzyskać dx oraz dy wskazujące zmianę pozycji myszy.

```
phi = phi + dx * 0.01;
theta = theta + dy * 0.01;
```