

Zaawansowane systemy programowania grafiki. Modelowanie głębi ostrości

Aleksander Denisiuk
Uniwersytet Warmińsko-Mazurski
Olsztyn, ul. Słoneczna 54
denisjuk@matman.uwm.edu.pl

1 czerwca 2021

Modelowanie głębi ostrości

Wstęp

Renderowanie do
tektury

Compute Shader

Box Filter

Implementacja

Najnowsza wersja tego dokumentu dostępna jest pod adresem

<http://wmii.uwm.edu.pl/~denisjuk/uwm>

Wstęp

- ❖ Zagadnienie
- ❖ Modelowanie
- ❖ Technologie

Renderowanie do
tektury

Compute Shader

Box Filter

Implementacja

Wstęp

Zagadnienie

Wstęp

❖ **Zagadnienie**

❖ Modelowanie

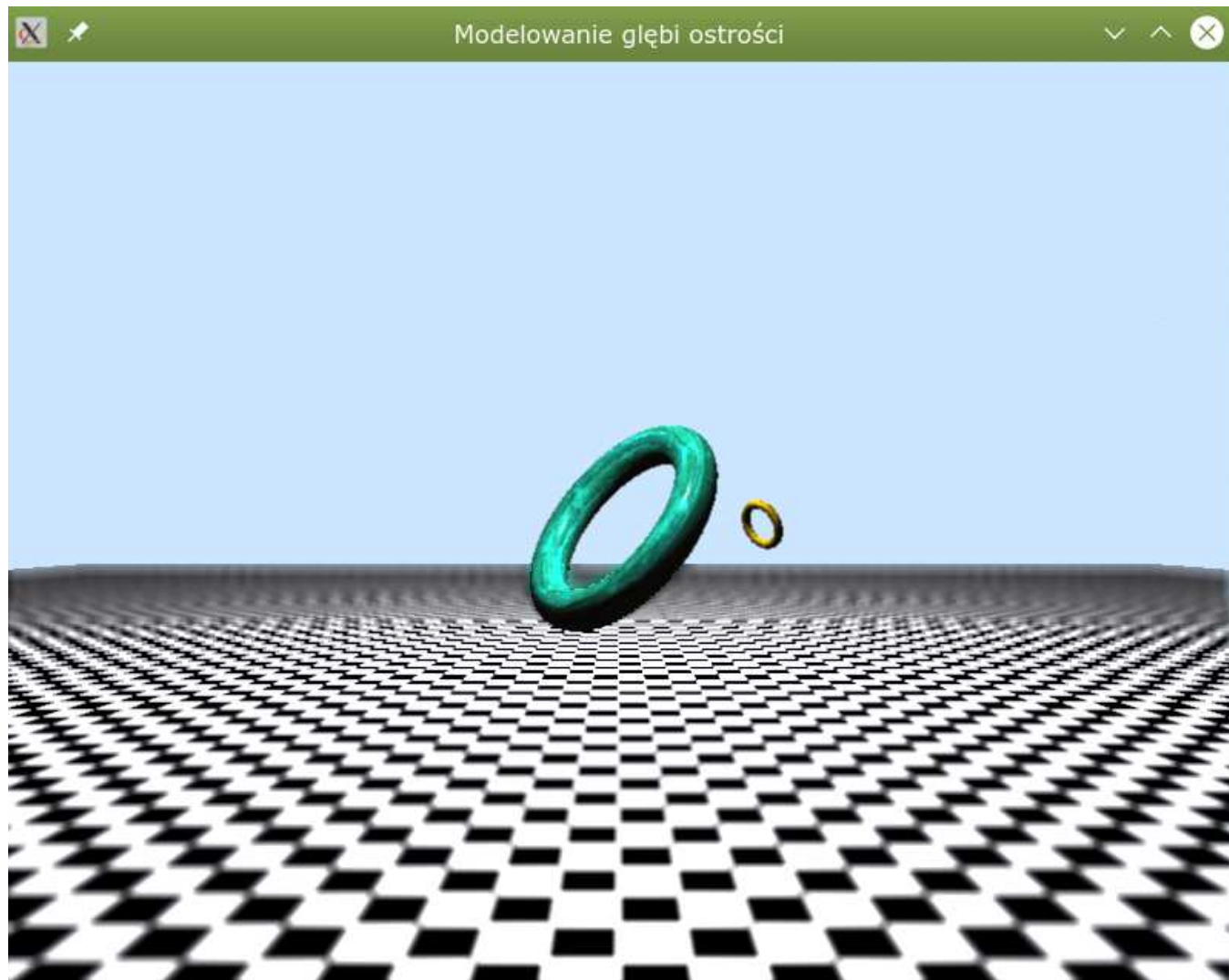
❖ Technologie

Renderowanie do
tekstury

Compute Shader

Box Filter

Implementacja



Parametry

Wstęp

❖ Zagadnienie

❖ Modelowanie

❖ Technologie

Renderowanie do
tektury

Compute Shader

Box Filter

Implementacja

- `focus_distance` — odległość ogniskowa
- `focus_depth` — głębina ostrości

Modelowanie

Wstęp

❖ Zagadnienie

❖ Modelowanie

❖ Technologie

Renderowanie do
tekstury

Compute Shader

Box Filter

Implementacja

- Wyrenderować obraz
- Dla każdego piksela zapisać odległość od kamery (bufor głębokości)
- Jeżeli odległość różni się od ogniskowej więcej, niż głębokości ostrości, zastosować rozmycie
 - ◆ średnia z sąsiednich pikseli — *box* filtr
 - ◆ im więcej odległość różni się od ogniskowej, tym większy jest promień filtru

Technologie

Wstęp

❖ Zagadnienie

❖ Modelowanie

❖ Technologie

Renderowanie do
tekstury

Compute Shader

Box Filter

Implementacja

- Renderowanie do tekstury — obiekt bufora ramki (framebuffer)
 - ✦ odległość zapisać jako czwartą współrzędną koloru (kanał α)
- Otrzymaną teksturę filtrować i wyświetlić jako prostokąt
 - ✦ bardzo dużo obliczeń
 - można obliczyć równoległe
 - ◆ Compute Shader

Wstęp

**Renderowanie do
tektury**

- ❖ Framebuffer
- ❖ Tekstury
- ❖ Ustawienia
- ❖ Renderowanie

Compute Shader

Box Filter

Implementacja

Renderowanie do tektury

Utworzenie obiektu buforu ramki

Wstęp

Renderowanie do
tekstury

❖ Framebuffer

❖ Tekstury

❖ Ustawienia

❖ Renderowanie

Compute Shader

Box Filter

Implementacja

```
GLuint fbo = 0;  
glGenFramebuffers(1, &fbo);  
glBindFramebuffer(GL_FRAMEBUFFER, fbo);
```

- domyślnie obiekt buforu ramki z numerem 0 — ekran

Utworzenie tekstury dla obrazu

Wstęp

Renderowanie do
tekstury

❖ Framebuffer

❖ Tekstury

❖ Ustawienia

❖ Renderowanie

Compute Shader

Box Filter

Implementacja

```
GLuint color_texture;  
glGenTextures(1, &color_texture);  
glBindTexture(GL_TEXTURE_2D, color_texture);
```

```
// Pusty obrazek (ostatnie zero)  
glTexStorage2D(GL_TEXTURE_2D, 1,  
               GL_RGBA32F, width, height);
```

```
// Parametry interpolacji (koniecznie)  
glTexParameteri(GL_TEXTURE_2D,  
                GL_TEXTURE_MAG_FILTER, GL_NEAREST);  
glTexParameteri(GL_TEXTURE_2D,  
                GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

- Parametry tekstury mogą być inne

Utworzenie tekstury dla buforu głębokości

Wstęp

Renderowanie do
tekstury

❖ Framebuffer

❖ Tekstury

❖ Ustawienia

❖ Renderowanie

Compute Shader

Box Filter

Implementacja

```
GLuint depth_texture;  
glGenTextures(1, &depth_texture);  
glBindTexture(GL_TEXTURE_2D, depth_texture);  
glTexStorage2D(GL_TEXTURE_2D, 11,  
               GL_DEPTH_COMPONENT32F,  
               width, height);  
  
// parametry interpolacji tekstury  
glTexParameteri(GL_TEXTURE_2D,  
                GL_TEXTURE_MIN_FILTER, GL_LINEAR);  
glTexParameteri(GL_TEXTURE_2D,  
                GL_TEXTURE_MAG_FILTER, GL_LINEAR);  
  
// parametry ekstrapolacji tekstury  
glTexParameteri(GL_TEXTURE_2D,  
                GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);  
glTexParameteri(GL_TEXTURE_2D,  
                GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

- Można utworzyć tylko jedną z możliwych tekstur

Konfiguracja buforu ramki

Wstęp

Renderowanie do
tekstury

❖ Framebuffer

❖ Tekstury

❖ Ustawienia

❖ Renderowanie

Compute Shader

Box Filter

Implementacja

```
glFramebufferTexture(GL_FRAMEBUFFER,  
                     GL_DEPTH_ATTACHMENT, depth_texture, 0);  
glFramebufferTexture(GL_FRAMEBUFFER,  
                     GL_COLOR_ATTACHMENT0, color_texture, 0);
```

```
// Set the list of draw buffers.
```

```
GLenum draw_buffers[1] = {GL_COLOR_ATTACHMENT0};  
glDrawBuffers(1, draw_buffers);
```

- 1 — to ilość buforów wejściowych
 - ◆ może ich być więcej
 - ◆ w shaderze fragmentów:

```
layout(location = 0) out vec3 color;
```
 - ◆ $\text{location} = 0 \iff \text{GL_COLOR_ATTACHMENT0}$

Sprawdzenie buforu ramki

Wstęp

Renderowanie do
tekstury

❖ Framebuffer

❖ Tekstury

❖ **Ustawienia**

❖ Renderowanie

Compute Shader

Box Filter

Implementacja

```
if (glCheckFramebufferStatus (GL_FRAMEBUFFER)  
    != GL_FRAMEBUFFER_COMPLETE)  
    return false;
```

Renderowanie

Wstęp

Renderowanie do
tekstury

❖ Framebuffer

❖ Tekstury

❖ Ustawienia

❖ **Renderowanie**

Compute Shader

Box Filter

Implementacja

```
glBindFramebuffer(GL_FRAMEBUFFER, fbo);  
glViewport(0, 0, 1024, 768);  
...  
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Wstęp

Renderowanie do
tekstury

Compute Shader

- ❖ Compute Shader
- ❖ Kompilowanie
- ❖ Wykonanie
- ❖ Wejście/Wyjście
- ❖ Pamięć
współdzielona
- ❖ Synchronizacja

Box Filter

Implementacja

Compute Shader

Compute Shader

Wstęp

Renderowanie do
tekstury

Compute Shader

❖ Compute Shader

❖ Kompilowanie

❖ Wykonanie

❖ Wejście/Wyjście

❖ Pamięć
współdzielona

❖ Synchronizacja

Box Filter

Implementacja

- W OpenGL od 4.3 (6 sierpnia 2012 roku)
 - ✦ w 4.2 jako rozszerzenie
- Wykorzystywany do obliczeń numerycznych, zwłaszcza równoległych
- Nie jest na stałe wpisany do potoku renderinga
- Przetwarza dane (z jednej tekstury) na dane (z innej tekstury)
- Jest oddzielny od shaderów graficznych i nie może być z nimi połączony
- Jest pisany w języku GLSL
- Typowe zastosowania:
 - ✦ *preprocessing*
 - ✦ *postprocessing*

Kompilowanie

Wstęp

Renderowanie do
tekstury

Compute Shader

❖ Compute Shader

❖ **Kompilowanie**

❖ Wykonanie

❖ Wejście/Wyjście

❖ Pamięć
współdzielona

❖ Synchronizacja

Box Filter

Implementacja

- Kompiluje się jak każdy inny shader, z parametrem `GL_COMPUTE_SHADER`
- Jest linkowany w program w sposób zwykły
- Jest usuwany tradycyjnie
- `glUseProgram()` działa jak zwykle
- Zmienne uniform też są dostępne

Przykład

Wstęp

Renderowanie do
tekstury

Compute Shader

❖ Compute Shader

❖ **Kompilowanie**

❖ Wykonanie

❖ Wejście/Wyjście

❖ Pamięć
współdzielona

❖ Synchronizacja

Box Filter

Implementacja

```
GLuint compute_shader;  
GLuint compute_program;
```

```
static const GLchar * source[]={  
    "#version 430 core\n"  
    "\n"  
    "layout(local_size_x=32,local_size_y=32) in;\n"  
    "void main(void){\n"  
    "\n"  
    "}\n"  
};
```

Obiekt shadera. Działania

Wstęp

Renderowanie do
tekstury

Compute Shader

❖ Compute Shader

❖ **Kompilowanie**

❖ Wykonanie

❖ Wejście/Wyjście

❖ Pamięć
współdzielona

❖ Synchronizacja

Box Filter

Implementacja

```
compute_shader  
    = glCreateShader(GL_COMPUTE_SHADER);  
glShaderSource(compute_shader, 1, source, NULL);  
glCompileShader(compute_shader);
```

```
compute_program = glCreateProgram();  
glAttachShader(compute_program, compute_shader);  
glLinkProgram();
```

.....

```
glDeleteShader(compute_shader);
```

Wykonanie

Wstęp

Renderowanie do
tekstury

Compute Shader

❖ Compute Shader

❖ Kompilowanie

❖ Wykonanie

❖ Wejście/Wyjście

❖ Pamięć
współdzielona

❖ Synchronizacja

Box Filter

Implementacja

```
void glDispatchCompute (  
    GLuint  num_groups_x,  
    GLuint  num_groups_y,  
    GLuint  num_groups_z) ;
```

Grupy robocze (work groups)

Wstęp

Renderowanie do
tekstury

Compute Shader

❖ Compute Shader

❖ Kompilowanie

❖ Wykonanie

❖ Wejście/Wyjście

❖ Pamięć
współdzielona

❖ Synchronizacja

Box Filter

Implementacja

- Przy odpalaniu programu zostaje utworzona *globalna grupa robocza*
- Globalna grupa robocza zostaje podzielona na lokalne grupy
- Każda lokalna grupa robocza jest sześcianem $x \times y \times z$ fragmentów (pikseli), na których zostanie wykonany kod shadera
 - ◆ każdy z wymiarów może być jeden
 - ◆ kolejność wykonania nie jest określona
 - ◆ możliwe są obliczenia równoległe
- Rozmiar lokalnych grup określony jest w shaderze, jako dane wejściowe, przykładowo:

```
layout (local_size_x = 4, local_size_y = 7,  
        local_size_z=10) in
```

- ◆ domyślnym wymiarem jest 1

Wejście/Wyjście

Wstęp

Renderowanie do
tekstury

Compute Shader

❖ Compute Shader

❖ Kompilowanie

❖ Wykonanie

❖ Wejście/Wyjście

❖ Pamięć
współdzielona

❖ Synchronizacja

Box Filter

Implementacja

● Dostęp do wbudowanych zmiennych

```
const uvec3 gl_WorkGroupSize;  
in uvec3 gl_NumWorkGroups;  
in uvec3 gl_WorkGroupID;  
in uvec3 gl_LocalInvocationID;  
in uvec3 gl_GlobalInvocationID;  
in uint gl_LocalInvocationIndex;
```

◆ $gl_GlobalInvocationID = gl_WorkGroupID * gl_WorkGroupSize + gl_LocalInvocationID$

◆ $gl_LocalInvocationIndex = gl_LocalInvocationID.z * gl_WorkGroupSize.x * gl_WorkGroupSize.y + gl_LocalInvocationID.y * gl_WorkGroupSize.x + gl_LocalInvocationID.x$

Wejście/Wyjście

Wstęp

Renderowanie do
tekstury

Compute Shader

❖ Compute Shader

❖ Kompilowanie

❖ Wykonanie

❖ Wejście/Wyjście

❖ Pamięć
współdzielona

❖ Synchronizacja

Box Filter

Implementacja

- Brak niestandardowych zmiennych wejściowych/wyjściowych
- Dane się pobiera/zapisuje do pamięci w sposób jawny

```
#version 430 core

layout(local_size_x=32, local_size_y=32);
layout(binding=0, rgba32f) uniform image2D img_in;

layout(binding=1, rgba32f) uniform image2D img_out;

void main(void) {
    vec4 texel;
    ivec2 p = ivec2(gl_GlobalInvocationID.xy);

    texel = imageLoad(img_in, p);
    texel = vec4(1.0) - texel;
    imageStore(img_out, p, texel);
}
```

Pamięć współdzielona

Wstęp

Renderowanie do
tekstury

Compute Shader

❖ Compute Shader

❖ Kompilowanie

❖ Wykonanie

❖ Wejście/Wyjście

❖ Pamięć
współdzielona

❖ Synchronizacja

Box Filter

Implementacja

- Deklaracja zmiennej z modyfikatorem `shared`
- Dostępna dla wszystkich instancji shadera w tej samej grupie roboczej
- Dostęp jest znacznie szybszy, niż do ogólnej pamięci
- Mała ilość (co najmniej 32KB)

Zagadnienie

Wstęp

Renderowanie do
tekstury

Compute Shader

❖ Compute Shader

❖ Kompilowanie

❖ Wykonanie

❖ Wejście/Wyjście

❖ Pamięć
współdzielona

❖ Synchronizacja

Box Filter

Implementacja

- Ponieważ instancje shadera działają równolegle, nigdy nie wiadomo, w jakim stanie są obliczenia
- Shader, który wykorzystuje wyniki obliczenia innych instancji powinien mieć pewność, że te obliczenia już są wykonane
- Funkcja `barrier()` zapewnia, że wszystkie instancje doszły w obliczeniach do tego punktu
- Funkcja `memoryBarrierShared()` zapewnia, że wszystkie instancje już zapisały dane do pamięci współdzielonej

Przykład

Wstęp

Renderowanie do
tekstury

Compute Shader

❖ Compute Shader

❖ Kompilowanie

❖ Wykonanie

❖ Wejście/Wyjście

❖ Pamięć
współdzielona

❖ Synchronizacja

Box Filter

Implementacja

● Przesunięcie obrazu o jeden w prawo

```
#version 430
```

```
layout(local_size_x = 1024) in;
```

```
layout(binding=0, r32ui) uniform uimageBuffer  
img_in;
```

```
layout(binding=1) uniform uimageBuffer img_out;
```

```
shared uint temp_storage[1024];
```

```
void main(void) {
```

```
    uint n = imageLoad(img_in, gl_invocationID.x).x;
```

```
    temp_storage[gl_invocationID.x] = n;
```

```
    barrier();
```

```
    memoryBarrierShared();
```

```
    n = temp_storage[(gl_invocationID.x-1)&1023];
```

```
    imageStore(img_out, gl_invocationID.x, n);
```

```
}
```

Wstęp

Renderowanie do
tektury

Compute Shader

Box Filter

- ❖ Sumy prefiksowe
- ❖ Shader dla sum
- ❖ Sumy 2D
- ❖ 2D shader
- ❖ Shader dla box
filtra

Implementacja

Box Filter

Sumy prefiksowe

- Wstęp
- Renderowanie do tekstury
- Compute Shader
- Box Filter
- ❖ Sumy prefiksowe
- ❖ Shader dla sum
- ❖ Sumy 2D
- ❖ 2D shader
- ❖ Shader dla box filtra
- Implementacja

- Dany jest ciąg x_0, x_1, \dots, x_n
- Obliczyć ciąg *sum prefiksowych* y_0, y_1, \dots, y_n , gdzie

$$y_0 = x_0,$$

$$y_1 = x_0 + x_1,$$

$$y_2 = x_0 + x_1 + x_2,$$

.....

$$y_n = x_0 + x_1 + x_2 + \dots + x_n.$$

- $1, 2, 3, 4 \mapsto 1, 3, 6, 10$
- Zastosowanie: $\sum_{i=i_1}^{i_2} x_i = y_{i_2} - y_{i_1-1}$

Równoległe

Wstęp

Renderowanie do
tekstury

Compute Shader

Box Filter

❖ Sumy prefiksowe

❖ Shader dla sum

❖ Sumy 2D

❖ 2D shader

❖ Shader dla box
filtra

Implementacja

● Jak obliczyć równoległe?

$$y_0 = x_0,$$

$$y_1 = x_0 + x_1,$$

$$y_2 = x_0 + x_1 + x_2,$$

$$y_3 = x_0 + x_1 + x_2 + x_3.$$

Równoległe

Wstęp

Renderowanie do
tekstury

Compute Shader

Box Filter

❖ Sumy prefiksowe

❖ Shader dla sum

❖ Sumy 2D

❖ 2D shader

❖ Shader dla box
filtra

Implementacja

● Jak obliczyć równoległe?

$$y_0 = x_0,$$

$$y_1 = x_0 + x_1,$$

$$y_2 = x_0 + x_1 + x_2,$$

$$y_3 = x_0 + x_1 + x_2 + x_3.$$

● A tak:

$$y_0 = x_0,$$

$$y_1 = x_0 + x_1, \quad y_2 = y_2 + y_1,$$

$$y_2 = x_2, \quad y_3 = y_3 + y_1.$$

$$y_3 = x_2 + x_3,$$

Równoległe obliczenie sum prefiksowych

Wstęp

Renderowanie do
tekstury

Compute Shader

Box Filter

❖ Sumy prefiksowe

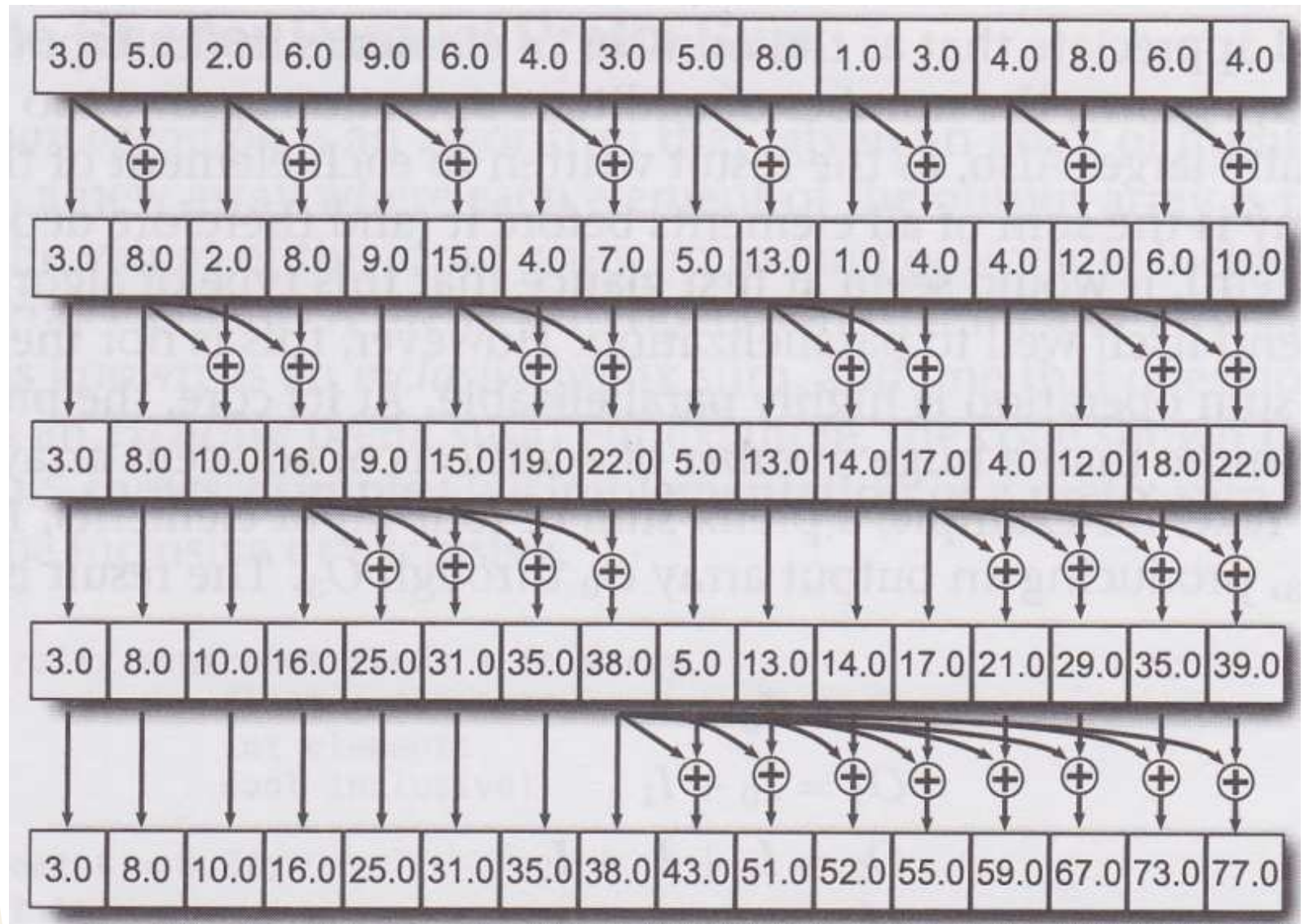
❖ Shader dla sum

❖ Sumy 2D

❖ 2D shader

❖ Shader dla box
filtra

Implementacja



Shader

Wstęp

Renderowanie do
tekstury

Compute Shader

Box Filter

❖ Sumy prefiksowe

❖ Shader dla sum

❖ Sumy 2D

❖ 2D shader

❖ Shader dla box
filtra

Implementacja

```
#version 430 core
```

```
layout (local_size_x = 1024) in;
```

```
layout (binding = 0) coherent buffer block1{  
    float input_data[gl_WorkGroupSize.x];  
}
```

```
layout (binding = 1) coherent buffer block2{  
    float output_data[gl_WorkGroupSize.x];  
}
```

```
shared float shared_data[gl_WorkGroupSize.x*2];
```

```
.....
```


Kopiowanie do pamięci współdzielonej

Wstęp

Renderowanie do
tekstury

Compute Shader

Box Filter

❖ Sumy prefiksowe

❖ Shader dla sum

❖ Sumy 2D

❖ 2D shader

❖ Shader dla box
filtra

Implementacja

```
.....  
void main(void) {  
    uint id = gl_LocalInvocationID.x;  
    uint rd_id;  
    uint wr_id;  
    uint mask;  
  
    const uint steps  
            = uint(log2(gl_WorkGroupSize.x))+1  
    uint step=0;  
  
    shared_data[id*2] = input_data[id*2];  
    shared_data[id*2+1] = input_data[id*2+1];  
    barrier();  
    memoryBarrierShared();  
  
    .....
```

Pętla

Wstęp

Renderowanie do
tekstury

Compute Shader

Box Filter

❖ Sumy prefiksowe

❖ Shader dla sum

❖ Sumy 2D

❖ 2D shader

❖ Shader dla box
filtra

Implementacja

```
.....  
    for (step = 0; step < steps; step++) {  
        mask = (1 << step) - 1;  
        rd_id = ((id >> step) << (step+1)) + mask;  
        wr_id = rd_id + (id & mask);  
  
        shared_data[wr_id] += shared_data[rd_id];  
  
        barrier();  
        memoryBarrierShared();  
    }  
  
    output_data[id*2] = shared_data[id*2];  
    output_data[id*2+1] = shared_data[id*2+1];  
}
```

Dwuwymiarowe sumy prefiksowe a box filtr

Wstęp

Renderowanie do
tekstury

Compute Shader

Box Filter

❖ Sumy prefiksowe

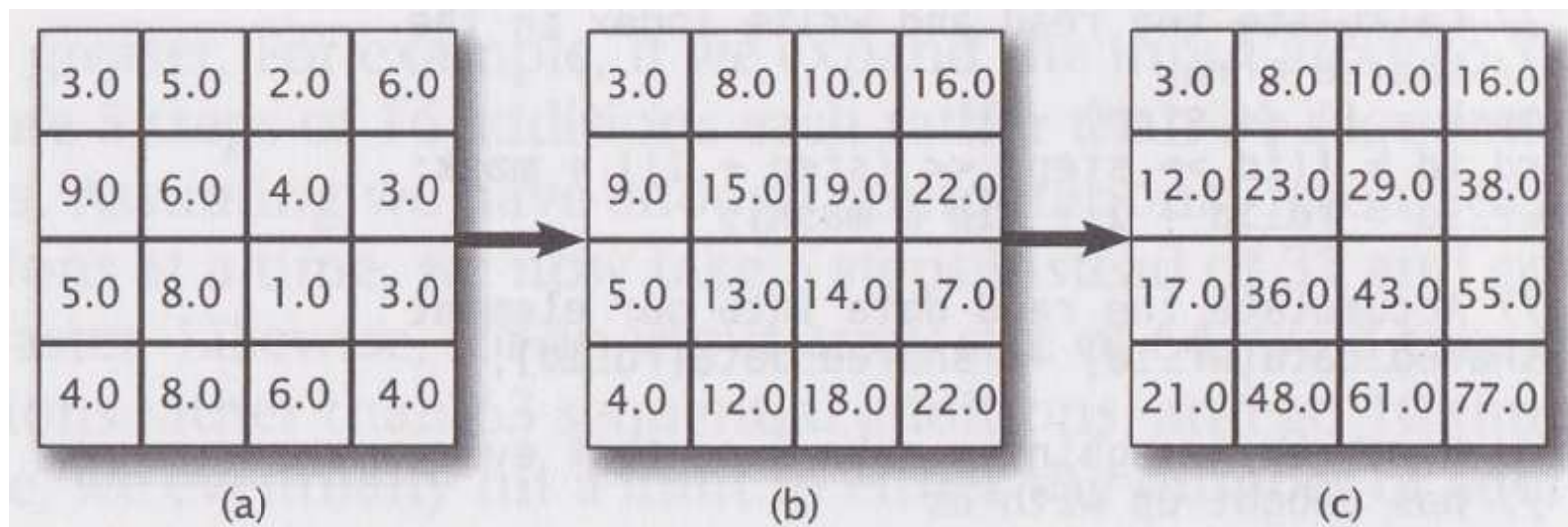
❖ Shader dla sum

❖ Sumy 2D

❖ 2D shader

❖ Shader dla box
filtra

Implementacja



- $$\sum_{i=i_1+1, j=j_1+1}^{i_2, j_2} a_{ij} = S(i_1, j_1) + S(i_2, j_2) - S(i_1, j_2) - S(i_2, j_1)$$
- Algorytm jednowymiarowy wzdłuż wierszy, potem — wzdłuż kolumn
 - ◆ Optymalizacja: algorytm jednowymiarowy wzdłuż wierszy, zapisać wynik do kolumn

Shader

Wstęp

Renderowanie do
tekstury

Compute Shader

Box Filter

❖ Sumy prefiksowe

❖ Shader dla sum

❖ Sumy 2D

❖ 2D shader

❖ Shader dla box
filtra

Implementacja

- Kanał α zawiera odległość od kamery

```
#version 440 core
```

```
layout (local_size_x = 1024) in;
```

```
shared vec3 shared_data[gl_WorkGroupSize.x * 2];
```

```
layout (binding = 0, rgba32f)
```

```
    readonly uniform image2D input_image;
```

```
layout (binding = 1, rgba32f)
```

```
    writeonly uniform image2D output_image;
```

Przygotowanie

Wstęp

Renderowanie do
tekstury

Compute Shader

Box Filter

❖ Sumy prefiksowe

❖ Shader dla sum

❖ Sumy 2D

❖ 2D shader

❖ Shader dla box
filtra

Implementacja

```
void main(void) {  
    uint id = gl_LocalInvocationID.x;  
    uint rd_id;  
    uint wr_id;  
    uint mask;  
    ivec2 P0 = ivec2(id*2, gl_WorkGroupID.x);  
    ivec2 P1 = ivec2(id*2 + 1, gl_WorkGroupID.x);  
  
    const uint steps  
        =uint(log2(gl_WorkGroupSize.x)) + 1;  
    uint step = 0;
```

Kopiowanie do pamięci współdzielonej

Wstęp

Renderowanie do
tekstury

Compute Shader

Box Filter

❖ Sumy prefiksowe

❖ Shader dla sum

❖ Sumy 2D

❖ 2D shader

❖ Shader dla box
filtra

Implementacja

```
vec4 i0 = imageLoad(input_image, P0);
```

```
vec4 i1 = imageLoad(input_image, P1);
```

```
sharedData[P0.x] = i0.rgb;
```

```
sharedData[P1.x] = i1.rgb;
```

```
barrier();
```

Pętla

Wstęp

Renderowanie do
tekstury

Compute Shader

Box Filter

❖ Sumy prefiksowe

❖ Shader dla sum

❖ Sumy 2D

❖ 2D shader

❖ Shader dla box
filtra

Implementacja

```
for (step = 0; step < steps; step++) {  
    mask = (1 << step) - 1;  
    rd_id = ((id >> step) << (step + 1)) + mask;  
    wr_id = rd_id + 1 + (id & mask);  
  
    shared_data[wr_id] += shared_data[rd_id];  
  
    barrier();  
}
```

Zapisywanie z transponowaniem

Wstęp

Renderowanie do
tekstury

Compute Shader

Box Filter

❖ Sumy prefiksowe

❖ Shader dla sum

❖ Sumy 2D

❖ 2D shader

❖ Shader dla box
filtra

Implementacja

● Kanał α bez zmian

```
imageStore(output_image, P0.yx,  
           vec4(shared_data[P0.x], i0.a));  
imageStore(output_image, P1.yx,  
           vec4(shared_data[P1.x], i1.a));  
}
```


Shader dla box filtra

- Wstęp
- Renderowanie do tekstury
- Compute Shader
- Box Filter
 - ❖ Sumy prefiksowe
 - ❖ Shader dla sum
 - ❖ Sumy 2D
 - ❖ 2D shader
 - ❖ Shader dla box filtra
- Implementacja

- Zakładamy, że już mamy teksturę z obliczonymi sumami prefiksowymi
- Wyświetlamy kwadrat z użyciem tej tekstury
- Kanał α zawiera odległość od kamery
- Zwykły potok renderingu:
Vertex Shader \rightsquigarrow Rasteryzator \rightsquigarrow Fragment Shader

Vertex Shader

Wstęp

Renderowanie do
tekstury

Compute Shader

Box Filter

❖ Sumy prefiksowe

❖ Shader dla sum

❖ Sumy 2D

❖ 2D shader

❖ Shader dla box
filtra

Implementacja

```
#version 430 core
```

```
void main(void) {  
    const vec4 vertex[] = vec4[]  
        ( vec4(-1.0, -1.0, 0.5, 1.0),  
          vec4( 1.0, -1.0, 0.5, 1.0),  
          vec4(-1.0,  1.0, 0.5, 1.0),  
          vec4( 1.0,  1.0, 0.5, 1.0)  
        );  
  
    gl_Position = vertex[gl_VertexID];  
}
```

Fragment Shader

Wstęp

Renderowanie do
tekstury

Compute Shader

Box Filter

❖ Sumy prefiksowe

❖ Shader dla sum

❖ Sumy 2D

❖ 2D shader

❖ Shader dla box
filtra

Implementacja

```
#version 430 core
```

```
layout (binding = 0) uniform sampler2D  
    input_image;
```

```
layout (location = 0) out vec4 color;
```

```
uniform float focal_distance = 50.0;
```

```
uniform float focal_depth = 30.0;
```

```
void main(void) {
```

```
    // s will be used to scale our texture
```

```
    vec2 s = 1.0 / textureSize(input_image, 0);
```

```
    // C is the center of the filter
```

```
    vec2 C = gl_FragCoord.xy;
```

```
    vec4 v = texelFetch(input_image,
```

```
        ivec2(gl_FragCoord.xy), 0).rgba;
```

Promień rozmycia

Wstęp

Renderowanie do
tekstury

Compute Shader

Box Filter

❖ Sumy prefiksowe

❖ Shader dla sum

❖ Sumy 2D

❖ 2D shader

❖ Shader dla box
filtra

Implementacja

- N potrzeby aplikacji wyczyszczamy ekran z kanałem α równym zero

```
glClearColor(0.8f, 0.85f, 1.0f, 0.0f);
```

- Jeżeli zostało zero, to znaczy, że ten piksel nie był renderowany, nie ma po co filtrować

```
float m;  
if (v.w == 0.0) {  
    m = 0.5;  
}  
else {  
    m = abs(v.w - FocalDistance);  
    m = 0.5 + smoothstep(0.0, FocalDepth, m) * 7.5;  
}
```

Obliczenie filtra

- Wstęp
- Renderowanie do tekstury
- Compute Shader
- Box Filter
 - ❖ Sumy prefiksowe
 - ❖ Shader dla sum
 - ❖ Sumy 2D
 - ❖ 2D shader
 - ❖ Shader dla box filtra
- Implementacja

```
vec2 P0 = C + vec2(-m, -m);  
vec2 P1 = C + vec2(-m, m);  
vec2 P2 = C + vec2(m, -m);  
vec2 P3 = C + vec2(m, m);
```

```
P0 *= S;  
P1 *= S;  
P2 *= S;  
P3 *= S;
```

```
vec3 a = textureLod(input_image, P0, 0).rgb;  
vec3 b = textureLod(input_image, P1, 0).rgb;  
vec3 c = textureLod(input_image, P2, 0).rgb;  
vec3 d = textureLod(input_image, P3, 0).rgb;  
  
vec3 f = a - b - c + d;
```

Zakończenie

Wstęp

Renderowanie do
tekstury

Compute Shader

Box Filter

❖ Sumy prefiksowe

❖ Shader dla sum

❖ Sumy 2D

❖ 2D shader

❖ Shader dla box
filtra

Implementacja

```
// Scale radius -> diameter  
m *= 2;
```

```
// Divide through by area  
f /= float(m * m);
```

```
// Outut final color  
color = vec4(f, 1.0);  
}
```

Wstęp

Renderowanie do
tektury

Compute Shader

Box Filter

Implementacja

- ❖ Framgent Shader
- ❖ Tekstury
- ❖ Programy
- ❖ Modele
- ❖ Framebuffer
- ❖ Window

Implementacja

Poprawki w Shaderze

- Wstęp
- Renderowanie do tekstury
- Compute Shader
- Box Filter
- Implementacja
 - ❖ Fragment Shader
 - ❖ Tekstury
 - ❖ Programy
 - ❖ Modele
 - ❖ Framebuffer
 - ❖ Window

- Jako odległość od kamery używamy współrzędną z wektora `viewDir`

```
layout (location = 0) out vec4 out_color;
```

```
.....
```

```
vec4 color = material.emission;
```

```
.....
```

```
out_color = vec4(color.rgb,  
                 frag_vertex.view_dir.z);
```


Nowe obiekty tekstury

- Wstęp
- Renderowanie do tekstury
- Compute Shader
- Box Filter
- Implementacja
 - ❖ Fragment Shader
 - ❖ **Tekstury**
 - ❖ Programy
 - ❖ Modele
 - ❖ Framebuffer
 - ❖ Window

```
class ColorFBOTexture : public Texture{  
public:  
    void Initialize(GLint width, GLint height);  
};
```

```
class DepthTexture : public Texture{  
public:  
    void Initialize(int width, int height);  
};
```

Inicjalizacja ColorFBOTexture

Wstęp

Renderowanie do
tekstury

Compute Shader

Box Filter

Implementacja

❖ Fragment Shader

❖ Tekstury

❖ Programy

❖ Modele

❖ Framebuffer

❖ Window

```
void ColorFBOTexture::Initialize(int width,  
                                   int height) {  
    // nowy indeks tekstury  
    glGenTextures(1, &texture_);  
    // aktywacja  
    glBindTexture(GL_TEXTURE_2D, texture_);  
    // parametry ekstrapolacji tekstury  
    glTexParameteri(GL_TEXTURE_2D,  
                    GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);  
    glTexParameteri(GL_TEXTURE_2D,  
                    GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);  
    // Utwórzmy pustą teksturę  
    glTexStorage2D(GL_TEXTURE_2D, 11,  
                  GL_RGBA32F, width, height);  
}
```

Inicjalizacja DepthTexture

- Wstęp
- Renderowanie do tekstury
- Compute Shader
- Box Filter
- Implementacja
 - ❖ Fragment Shader
 - ❖ Tekstury
 - ❖ Programy
 - ❖ Modele
 - ❖ Framebuffer
 - ❖ Window

```
void DepthTexture::Initialize(int width,  
                                int height){  
    // nowy indeks tekstury  
    glGenTextures(1, &texture_);  
    // aktywacja  
    glBindTexture(GL_TEXTURE_2D, texture_);  
    // Utwórzmy pustą teksturę  
    glTexStorage2D(GL_TEXTURE_2D,  
                   11, GL_DEPTH_COMPONENT32F, width, height);  
  
    .....  
}
```

Interpolacja i ekstrapolacja

- Wstęp
- Renderowanie do tekstury
- Compute Shader
- Box Filter
- Implementacja
 - ❖ Fragment Shader
 - ❖ **Tekstury**
 - ❖ Programy
 - ❖ Modele
 - ❖ Framebuffer
 - ❖ Window

```
.....  
    // parametry interpolacji tekstury  
    glTexParameteri(GL_TEXTURE_2D,  
                    GL_TEXTURE_MIN_FILTER, GL_LINEAR);  
    glTexParameteri(GL_TEXTURE_2D,  
                    GL_TEXTURE_MAG_FILTER, GL_LINEAR);  
    // parametry ekstrapolacji tekstury  
    glTexParameteri(GL_TEXTURE_2D,  
                    GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);  
    glTexParameteri(GL_TEXTURE_2D,  
                    GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);  
}
```

ComputeProgram

Wstęp

Renderowanie do
tekstury

Compute Shader

Box Filter

Implementacja

❖ Fragment Shader

❖ Tekstury

❖ Programy

❖ Modele

❖ Framebuffer

❖ Window

```
class ComputeProgram : public BaseProgram{
public:
    ~ComputeProgram();
    void Initialize
        (const char *compute_shader_file);
    operator GLuint() const {return program_;}
protected:
    GLuint program_;
    GLuint compute_shader_;
    GLuint LinkProgramOrDie(GLuint compute_shader);
};
```

ComputeProgram, inicjalizacja

Wstęp

Renderowanie do
tekstury

Compute Shader

Box Filter

Implementacja

❖ Fragment Shader

❖ Tekstury

❖ Programy

❖ Modele

❖ Framebuffer

❖ Window

```
void ComputeProgram::Initialize
(const char *compute_shader_file) {

    compute_shader_
        = LoadAndCompileShaderOrDie (
            compute_shader_file,
            GL_COMPUTE_SHADER);
    program_ = LinkProgramOrDie (compute_shader_);

    glUseProgram (program_);
}
```

DisplayProgram

Wstęp

Renderowanie do
tekstury

Compute Shader

Box Filter

Implementacja

❖ Fragment Shader

❖ Tekstury

❖ Programy

❖ Modele

❖ Framebuffer

❖ Window

```
class DisplayProgram : public BaseProgram{
public:
    void Initialize(
        const char *vertex_shader_file,
        const char *fragment_shader_file);
    void SetFocalDepth(GLfloat depth);
    void SetFocalDistance(GLfloat distance);
private:
    GLint focal_depth_location_;
    GLint focal_distance_location_;
};
```

Uzupełnienie funkcji

LoadAndCompileShaderOrDie

Wstęp

Renderowanie do
tekstury

Compute Shader

Box Filter

Implementacja

❖ Fragment Shader

❖ Tekstury

❖ Programy

❖ Modele

❖ Framebuffer

❖ Window

```
if (!compiled) {  
    switch (type) {  
        case GL_VERTEX_SHADER:  
            cerr << "vertex ";  
            break;  
        case GL_FRAGMENT_SHADER:  
            cerr << "fragment ";  
            break;  
        case GL_COMPUTE_SHADER:  
            cerr << "compute ";  
            break;  
    }  
    cerr << ...  
}
```


Pusty obiekt

- Wstęp
- Renderowanie do tekstury
- Compute Shader
- Box Filter
- Implementacja
 - ❖ Framgent Shader
 - ❖ Tekstury
 - ❖ Programy
 - ❖ Modele
 - ❖ Framebuffer
 - ❖ Window

```
class EmptyVAOObject {  
public:  
    ~EmptyVAOObject();  
    void Initialize();  
    operator GLuint() { return vao_; }  
protected:  
    GLuint vao_;  
};  
void EmptyVAOObject::Initialize() {  
    glGenVertexArrays(1, &vao_);  
}  
EmptyVAOObject::~~EmptyVAOObject() {  
    glBindVertexArray(0);  
    glDeleteVertexArrays(1, &vao_);  
}
```

Kwadrat

- Wstęp
- Renderowanie do tekstury
- Compute Shader
- Box Filter
- Implementacja
 - ❖ Fragment Shader
 - ❖ Tekstury
 - ❖ Programy
 - ❖ Modele
 - ❖ Framebuffer
 - ❖ Window

```
class Quad : public EmptyVAOObject,
             public TextureModel{
public:
    void Draw(const DisplayProgram & pr) const;
};

void Quad::Draw(const DisplayProgram & pr) const{
    glUseProgram(pr);
    glActiveTexture(texture_unit_);
    glBindTexture(GL_TEXTURE_2D, texture_);

    glBindVertexArray(vao_);
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
    glBindVertexArray(0);
}
```

Framebuffer

- Wstęp
- Renderowanie do tekstury
- Compute Shader
- Box Filter
- Implementacja
 - ❖ Framgent Shader
 - ❖ Tekstury
 - ❖ Programy
 - ❖ Modele
 - ❖ Framebuffer
 - ❖ Window

```
class FrameBuffer{  
public:  
    ~FrameBuffer();  
    void Initialize  
        (GLuint depth_texture,  
         GLuint color_texture);  
    operator GLuint() {return fbo_;}  
private:  
    GLuint fbo_;  
};
```

Framebuffer, Inicjalizacja

Wstęp

Renderowanie do
tekstury

Compute Shader

Box Filter

Implementacja

❖ Fragment Shader

❖ Tekstury

❖ Programy

❖ Modele

❖ **Framebuffer**

❖ Window

```
void FrameBuffer::Initialize(
    GLuint depth_texture,
    GLuint color_texture) {
    // Obiekt buforu ramki
    glGenFramebuffers(1, &fbo_);
    glBindFramebuffer(GL_FRAMEBUFFER, fbo_);

    glFramebufferTexture(GL_FRAMEBUFFER,
        GL_DEPTH_ATTACHMENT, depth_texture, 0);
    glFramebufferTexture(GL_FRAMEBUFFER,
        GL_COLOR_ATTACHMENT0, color_texture, 0);
    GLenum draw_buffers[1]
        = {GL_COLOR_ATTACHMENT0};
    glDrawBuffers(1, draw_buffers);
}
```

Sprawdzenie

Wstęp

Renderowanie do
tekstury

Compute Shader

Box Filter

Implementacja

❖ Fragment Shader

❖ Tekstury

❖ Programy

❖ Modele

❖ Framebuffer

❖ Window

```
GLuint fbo_status
    = glCheckFramebufferStatus (GL_FRAMEBUFFER) ;
if (fbo_status
    != GL_FRAMEBUFFER_COMPLETE) {
    std::cerr
        << "glCheckFramebufferStatus error "
        << fbo_status<<std::endl;
    glfwTerminate();
    exit(EXIT_FAILURE);
}
glBindFramebuffer (GL_FRAMEBUFFER, 0);
}
```

Nowe dane

Wstęp

Renderowanie do
tekstury

Compute Shader

Box Filter

Implementacja

❖ Framgent Shader

❖ Tekstury

❖ Programy

❖ Modele

❖ Framebuffer

❖ Window

```
Quad quad_;
```

```
ComputeProgram compute_program_;
```

```
DisplayProgram display_program_;
```

```
DepthTexture depth_texture_;
```

```
ColorFBOTexture fbo_texture_;
```

```
ColorFBOTexture tmp_fbo_texture_;
```

```
Framebuffer fbo_;
```

- Odpowiednie zmiany w konstruktorze, inicjalizacji i destruktorze

- ◆ pamiętać o zerze w bajcie przezroczystości:

```
glClearColor(0.8f, 0.85f, 1.0f, 0.0f);
```

Renderowanie

Wstęp

Renderowanie do
tekstury

Compute Shader

Box Filter

Implementacja

❖ Fragment Shader

❖ Tekstury

❖ Programy

❖ Modele

❖ Framebuffer

❖ Window

```
.....  
// Render to a texture
```

```
glBindFramebuffer(GL_FRAMEBUFFER, fbo_);  
glViewport(0, 0, width_, height_);  
glClear(GL_COLOR_BUFFER_BIT  
        | GL_DEPTH_BUFFER_BIT);  
glEnable(GL_DEPTH_TEST);  
  
tori_.Draw(point_program_);  
plane_.Draw(point_program_);  
  
glBindFramebuffer(GL_FRAMEBUFFER, 0);  
  
.....
```

Filtrowanie

Wstęp

Renderowanie do
tekstury

Compute Shader

Box Filter

Implementacja

❖ Fragment Shader

❖ Tekstury

❖ Programy

❖ Modele

❖ Framebuffer

❖ Window

```
.....  
glUseProgram(compute_program_);  
glBindImageTexture(0, fbo_texture_,  
    0, GL_FALSE, 0, GL_READ_ONLY, GL_RGBA32F);  
glBindImageTexture(1, tmp_fbo_texture_,  
    0, GL_FALSE, 0, GL_WRITE_ONLY, GL_RGBA32F);  
glDispatchCompute(kFBOSize, 1, 1);  
glMemoryBarrier(  
    GL_SHADER_IMAGE_ACCESS_BARRIER_BIT);  
glBindImageTexture(0, tmp_fbo_texture_,  
    0, GL_FALSE, 0, GL_READ_ONLY, GL_RGBA32F);  
glBindImageTexture(1, fbo_texture_,  
    0, GL_FALSE, 0, GL_WRITE_ONLY, GL_RGBA32F);  
glDispatchCompute(kFBOSize, 1, 1);  
glMemoryBarrier(  
    GL_SHADER_IMAGE_ACCESS_BARRIER_BIT);  
.....
```


Wyświetlanie

Wstęp

Renderowanie do
tekstury

Compute Shader

Box Filter

Implementacja

❖ Fragment Shader

❖ Tekstury

❖ Programy

❖ Modele

❖ Framebuffer

❖ Window

```
.....  
glDisable(GL_DEPTH_TEST);  
  
glUseProgram(display_program);  
glBindTexture(GL_TEXTURE_2D, fbo_texture);  
  
quad_>bind();  
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);  
quad_>release();  
  
.....
```