Wyjaśnienie kodu projektu 'Symulator Płynów' z podstaw informatyki

Użyte biblioteki

```
#include <SFML/Graphics.hpp>
#include <cmath>
#include <iostream>
#include <iomanip>
#include <time.h>
#include <functional>
```

Użyto SFML do wyświetlania symulacji - obiektów geometrycznych i tekstu.

Biblioteka cmath dostarcza operacje matematyczne potęgowania i pierwiastkowania.

Biblioteka iostream wykorzystana by wyświetlić błąd w przypadku problemu z załadowaniem czcionki.

Biblioteka iomanip użyta do strumieniowania tekstu z ilością klatek.

Biblioteka time.h wykorzystana do odczytania aktualnej daty i godziny by zseedować generator liczb losowych.

Biblioteka functional użyta by umożliwić przekazywanie funkcji jako wartości atrybuty klasy `Button`. Funkcja ta zostanie uruchomiona po kliknięciu przycisku.

Zmienne globalne

```
bool show_menu = true;
bool show_coloring = true;
```

Zmienna `show_menu` jest odpowiednio modyfikowana po kliknięciu przycisków START/RESET i by na jej podstawie warunkowo wyświetlać obiekty w głównej pętli.

Zmienna `show_coloring` jest modyfikowana po kliknięciu przycisku SHOW PRESSURE by na jej podstawie wizualizować lub nie ciśnienie cząsteczek z poziomu metody `FluidSimulator.draw`.

Klasa `Button`

```
class Button {
private:
    sf::RectangleShape shape;
    sf::Text text;
    sf::Font font;
    bool enabled = true;
    std::function<void()> callback;
public:
    Button(float x, float y, float width, float height, const std::string&
label, const sf::Font& font) {
        shape.setPosition(x, y);
        shape.setSize(sf::Vector2f(width, height));
        shape.setFillColor(sf::Color::Blue);
        shape.setOutlineColor(sf::Color::Black);
        shape.setOutlineThickness(2);

        this->font = font;
        text.setFont(this->font);
        text.setString(label);
        text.setCharacterSize(20);
        text.setFillColor(sf::Color::White);

        // Center text
        sf::FloatRect textBounds = text.getLocalBounds();
        text.setOrigin(textBounds.left + textBounds.width / 2.0f,
textBounds.top + textBounds.height / 2.0f);
        text.setPosition(x + width / 2.0f, y + height / 2.0f);
    }

    void setCallback(const std::function<void()>& func) {
        callback = func;
    }

    void setEnabled(bool value) {
        enabled = value;
    }

    void draw(sf::RenderWindow& window) {
        window.draw(shape);
        window.draw(text);
    }

    void handleEvent(const sf::Event& event, const sf::RenderWindow& window)
{
        if (enabled && event.type == sf::Event::MouseButtonPressed &&
event.mouseButton.button == sf::Mouse::Left) {
            sf::Vector2i mousePos = sf::Mouse::getPosition(window);
            if (shape.getGlobalBounds().contains(static_cast<sf::Vector2f>
(mousePos))) {
```

```
                if (callback) callback(); // Run the given function
            }
        }
    }
};
```

SFML nie udostępnia wysokopoziomowych abstrakcji do tworzenia UI, więc klasy `Button` i `Slider` są nieodłączną częścią projektu.

Klasa `Button` ma pięć prywatnych atrybutów określających m. in. kształt, tekst i czcionkę. Atrybut `enabled` ma dedykowany setter, a jego wartość `true` oznacza, że przycisk jest klikalny co odzwierciedla pierwszy warunek uruchomienia metody `Button.callback`.

Po zdefiniowaniu przycisku należy określić jego funkcję metodą `Button.setCallback`.

Metoda `Button.callback` wywołuję wcześniej przekazaną funkcję. Przykładowo, umożliwia to zmianę obiektów wyświetlanych w głównej pętli.

Konstruktor wykorzystuję argumenty przekazane podczas definicji przycisku do sprecyzowania wyglądu i pozycji atrybutów `shape` i `text` używając ich SFML'owych metod. Tekst jest wyrównywany do środka przycisku.

Klasa `Slider`

```
class Slider {
private:
    sf::RectangleShape track;
    sf::CircleShape knob;
    sf::Font font;
    sf::Text valueText;
    sf::Text NameText;

    float trackStartX, trackEndX;
    int minValue, maxValue;
    int currentValue;
    bool isDragging = false;
public:
    Slider(float x, float y, float width, int minValue, int maxValue,
std::string name)
        : minValue(minValue), maxValue(maxValue), currentValue(minValue) {
        // Set up the track
        track.setSize(sf::Vector2f(width, 5));
        track.setFillColor(sf::Color::White);
        track.setPosition(x, y);

        // Set up the knob
        knob.setRadius(10);
        knob.setFillColor(sf::Color::Red);
```

```
        knob.setOrigin(knob.getRadius(), knob.getRadius());
        knob.setPosition(x, y + track.getSize().y / 2);

        // Calculate the bounds
        trackStartX = x;
        trackEndX = x + width;

        // Load the font for text
        if (!font.loadFromFile("./resources/tuffy.ttf")) { // Ensure you
have a font file in your project directory
            std::cerr << "Failed to load font\n";
        }

        // Value display
        valueText.setFont(font);
        valueText.setCharacterSize(16);
        valueText.setFillColor(sf::Color::White);
        valueText.setString(std::to_string(currentValue));
        valueText.setPosition(x + width + 20, y - 5);

        // Name
        NameText.setFont(font);
        NameText.setCharacterSize(21);
        NameText.setFillColor(sf::Color::White);
        NameText.setString(name);
        NameText.setPosition(x + width/3, y - 36);
    }

    void handleEvent(const sf::Event& event, const sf::RenderWindow& window)
{
        if (event.type == sf::Event::MouseButtonPressed &&
event.mouseButton.button == sf::Mouse::Left) {
            sf::Vector2f mousePos(event.mouseButton.x, event.mouseButton.y);
            if (knob.getGlobalBounds().contains(mousePos)) {
                isDragging = true;
            }
        } else if (event.type == sf::Event::MouseButtonReleased &&
event.mouseButton.button == sf::Mouse::Left) {
            isDragging = false;
        } else if (event.type == sf::Event::MouseMoved && isDragging) {
            float mouseX = static_cast<float>(event.mouseMove.x);
            if (mouseX < trackStartX) mouseX = trackStartX;
            if (mouseX > trackEndX) mouseX = trackEndX;
            knob.setPosition(mouseX, knob.getPosition().y);

            // Map mouseX to the range of minValue to maxValue
            float percentage = (mouseX - trackStartX) / (trackEndX -
trackStartX);
            currentValue = static_cast<int>(minValue + percentage *
(maxValue - minValue));
```

```
            valueText.setString(std::to_string(currentValue));
        }
    }

    void draw(sf::RenderWindow& window) {
        window.draw(track);
        window.draw(knob);
        window.draw(valueText);
        window.draw(NameText);
    }

    int getValue() const {
        return currentValue;
    }
};
```

Klasa `Slider` (pl. suwak) to ważna część interfejsu użytkownika, pozwalająca użytkownikowi wybranie wartości z przedziału. Klasa zawiera 11 prywatnych atrybutów i razem z konstruktorem 4 publiczne metody, którę wyjaśnie niżej.

Atrybuty:

- `track` to prostokątny kształt reprezentujący ścieżkę po której porusza się suwak
- `knob` to kształt koła reprezentujące uchwyt suwaka
- `font` to czcionka ładowana w konstruktorze
- `valueText` to tekst który będzie wyświetlany po prawej od suwaka oznaczający aktualną wartość. Wraz z tym jak użytkownik przesuwa suwak, `valueText` będzie się zmieniać
- `NameText` to teksty który będzie wyświetlny nad suwakiem oznaczający parametr który będziemy modyfikować
- `trackStartX, trackEndX` to współrzędne X początku i końca ścieżki
- `minValue, maxValue` to przedział wartości suwaka
- `currentValue` to wartość liczbowa, na podstawie której będziemy aktualizować `valueText`, a także do której możemy się dostać z innych części kodu getterem `Slider.getValue`
- `isDragging` z wartością `true` będzie oznaczał, że metoda `Slider.handleEvent` będzie śledziła pozycję myszki i na jej podstawie aktualizowała suwak

Metody:

- Konstruktor ustawia szczegóły atrybutów `track`, `knob`, `valueText`, `NameText` używając odpowiednich SFML'owych metod, a także nadaję `trackStartX, trackEndX` wartości na podstawie argumentów `x` (koordynat X) i `width` (długość suwaka)
- `handleEvent` zostanie przekazana do obsługi zdarzeń w głównej pętli. Po przytrzymaniu lewego przycisku myszy `isDragging` zostanie ustawione na true, a po

puszczeniu wróci do false. Tak długo jak `isDragging` utrzyma się na true, koordynaty myszki będą śledzone i odpowiednie atrybuty suwaka będą aktualizowane

- `draw` rysuje suwak
- `getValue` zwraca wartość liczbową reprezentującą aktualną wartość tego suwaka

Klasa `FPSCounter`

```
class FPSCounter {
private:
    float fps;
    sf::Clock clock;
    sf::Time previousTime;

    static const int SAMPLE_SIZE = 10;
    std::array<float, SAMPLE_SIZE> fpsHistory;
    int currentSample = 0;

public:
    FPSCounter() : fps(0.0f), currentSample(0) {
        previousTime = clock.getElapsedTime();
        fpsHistory.fill(0.0f);
    }

    void update() {
        sf::Time currentTime = clock.getElapsedTime();
        sf::Time deltaTime = currentTime - previousTime;
        previousTime = currentTime;

        // Calculate instantaneous fps
        float currentFps = 1.0f / deltaTime.asSeconds();

        // Store in circular buffer
        fpsHistory[currentSample] = currentFps;
        currentSample = (currentSample + 1) % SAMPLE_SIZE;

        // Calculate average fps
        float sum = 0.0f;
        for (float sample : fpsHistory) {
            sum += sample;
        }
        fps = sum / SAMPLE_SIZE;
    }

    std::string getFPSString() const {
        std::stringstream ss;
        ss << std::fixed << std::setprecision(1) << fps << " FPS";
        return ss.str();
    }
```

```
    void draw(sf::RenderWindow& window, const sf::Font& font,
              unsigned int characterSize = 20,
              sf::Vector2f position = sf::Vector2f(27, 25)) {
        sf::Text text;
        text.setFont(font);
        text.setString(getFPSString());
        text.setCharacterSize(characterSize);
        text.setFillColor(sf::Color::White);
        text.setPosition(position);
        window.draw(text);
    }
};
```

Klasa ta służy do wyświetlania średnich klatek na sekundę białym tekstem w lewym górny rogu, na podstawie ostatnich 10 klatek.

W metodzie update zmienna currentFps jest liczona dzieląc 1 przez różnicę czasu przed i po poprzedniej klatce.

Wyświetlanie samego currentFps nie jest polecane, bo zmienia wartości zbyt chaotycznie. Dlatego, poprzednie 10 zapisów currentFps jest przechowywane w okrężnym buforze (liście, gdzie nowe wartości zastępują stare) i finalnie pokazywane FPSy biorą się z sumy poprzednich 10 podzielonych przez 10.

Funkcja `dot`

```
float dot(const sf::Vector2f& a, const sf::Vector2f& b) {
    return a.x * b.x + a.y * b.y;
}
```

Funkcja `dot` kalkuluję iloczyn skalarny dwóch wektorów dwuwymiarowych. Wykorzystywana w metodach klasy `FluidSimulator`.

Struktura `Particle`

```
struct Particle {
    sf::CircleShape shape;
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f force;
    float density;
    float pressure;

    Particle(float radius) : shape(radius), density(0.f), pressure(0.f) {
        shape.setFillColor(sf::Color::Cyan);
```

```
    }
};
```

Particle to pojedyńcza cząsteczka wody reprezentowana poprzez kształt koła. Wykorzystana w metodzie `FluidSimulator.addParticle`.

Klasa `FluidSimulator`

```
class FluidSimulator {
private:
    sf::Vector2f gravity;
    sf::FloatRect bounds;
    std::vector<Particle> particles;

    const float VISCOSITY = 7000.f;
    const float REST_DENSITY = 1000.f;
    const float GAS_CONSTANT = 100.f;
    const float SMOOTHING_LENGTH = 15.f;
    const float SMOOTHING_LENGTH_SQ = SMOOTHING_LENGTH * SMOOTHING_LENGTH;
    const float POLY6_SCALE = 315.f / (64.f * 3.14 *
std::pow(SMOOTHING_LENGTH, 4));
    const float SPIKY_GRAD_SCALE = -45.f / (3.14 *
std::pow(SMOOTHING_LENGTH, 6));
    const float VISC_LAP_SCALE = 45.f / (3.14 * std::pow(SMOOTHING_LENGTH,
6));
public:
    float PARTICLE_RADIUS = 5.f;
    float DAMPING = 0.4f;
    float MAX_VELOCITY = 300.f;
    float PARTICLE_MASS = 5.0f;

    FluidSimulator(const sf::FloatRect& boundsRect, const sf::Vector2f&
gravityVec = sf::Vector2f(0.f, 981.f))
        : gravity(gravityVec), bounds(boundsRect) {}

    void addParticle(const sf::Vector2f& pos) {
        Particle p(PARTICLE_RADIUS);
        p.position = pos;
        p.velocity = sf::Vector2f(0.f, 0.f);
        p.force = sf::Vector2f(0.f, 0.f);
        p.shape.setPosition(pos);
        particles.push_back(p);
    }

    void removeAllParticles() {

        particles.clear();
    }
```

```
    void update(float dt) {
        computeDensityPressure();
        computeForces();
        integrate(dt);
    }

    void shake() {
        for (auto& p : particles) {
                switch(rand() % 4) {
                case 0:
                    p.velocity += sf::Vector2f(0.f, static_cast<float>
(rand() % 10000));
                    break;
                case 1:
                    p.velocity += sf::Vector2f(static_cast<float>(rand() %
10000), 0.f);
                    break;
                case 2:
                    p.velocity += sf::Vector2f(0.f, -static_cast<float>
(rand() % 10000));
                    break;
                case 3:
                    p.velocity += sf::Vector2f(-static_cast<float>(rand() %
10000), 0.f);
                    break;
            }
        }
    }

    void wind(int direction, float force) {
        // 0123 - up right down left
        switch(direction) {
            case 0:
                for (auto& p : particles) {
                    p.velocity += sf::Vector2f(0.f, -force);
                }
                break;
            case 1:
                for (auto& p : particles) {
                    p.velocity += sf::Vector2f(force, 0.f);
                }
                break;
            case 2:
                for (auto& p : particles) {
                    p.velocity += sf::Vector2f(0.f, force);
                }
                break;
            case 3:
                for (auto& p : particles) {
```

```
                    p.velocity += sf::Vector2f(-force, 0.f);
                }
                break;
        }
    }

    void draw(sf::RenderWindow& window) {
        // Find max pressure in current frame for dynamic scaling
        float max_pressure = 0.0f;
        for (const auto& p : particles) {
            max_pressure = std::max(max_pressure, p.pressure);

        }

        // Avoid division by zero
        max_pressure = std::max(max_pressure, 0.0001f);

        for (auto& p : particles) {
            // Normalize pressure between 0 and 1
            float pressure_scale = p.pressure / max_pressure;

            // Create a color gradient from blue (low pressure) to red (high
pressure)
            sf::Color color(
                static_cast<sf::Uint8>(200 * pressure_scale),
// Red
                static_cast<sf::Uint8>(100 * (1.0f - pressure_scale)),
// Green
                static_cast<sf::Uint8>(255 * (1.0f - pressure_scale))
// Blue
            );
            if (show_coloring)
                p.shape.setFillColor(color);
            else
                p.shape.setFillColor(sf::Color::Cyan);
            p.shape.setPosition(p.position);
            window.draw(p.shape);
        }
    }
private:
    void computeDensityPressure() {
        for (auto& pi : particles) {
            pi.density = 0.f;
            for (auto& pj : particles) {
                sf::Vector2f diff = pi.position - pj.position;
                float r2 = diff.x * diff.x + diff.y * diff.y;

                if (r2 < SMOOTHING_LENGTH_SQ) {
                    pi.density += PARTICLE_MASS * POLY6_SCALE *
std::pow(SMOOTHING_LENGTH_SQ - r2, 3.f);
```

```
                }
            }


            pi.pressure = GAS_CONSTANT * (pi.density - REST_DENSITY);
        }
    }

    void computeForces() {
        for (auto& pi : particles) {
            sf::Vector2f pressure_force(0.f, 0.f);
            sf::Vector2f viscosity_force(0.f, 0.f);

            for (auto& pj : particles) {
                if (&pi == &pj) continue;

                sf::Vector2f diff = pi.position - pj.position;
                float r = std::sqrt(diff.x * diff.x + diff.y * diff.y);

                if (r < SMOOTHING_LENGTH && r > 0.0001f) {
                    // Pressure force
                    float pressure_scale = (pi.pressure + pj.pressure) /
(2.f * pi.density * pj.density);
                    sf::Vector2f normalized_diff = diff / r;
                    pressure_force += normalized_diff * (PARTICLE_MASS *
pressure_scale *
                        SPIKY_GRAD_SCALE * std::pow(SMOOTHING_LENGTH - r,
2.f));

                    // Viscosity force
                    viscosity_force += (pj.velocity - pi.velocity) *
                        (PARTICLE_MASS * VISCOSITY / pj.density *
VISC_LAP_SCALE * (SMOOTHING_LENGTH - r));
                }


                // Check for overlap (distance between particles < 2 *
radius)
                if (r < 2 * PARTICLE_RADIUS) {
                    sf::Vector2f normalized_diff = diff / r; // Collision
normal

                    // Calculate relative velocity
                    sf::Vector2f relative_velocity = pi.velocity -
pj.velocity;

                    // Normal velocity component (along collision normal)
                    float normal_velocity = dot(relative_velocity,
normalized_diff);
```

```
                    // Only resolve if particles are moving toward each
other
                    if (normal_velocity < 0) {
                        // Coefficient of restitution (1.0 = perfectly
elastic)
                        const float RESTITUTION = 0.8f;

                        // Calculate impulse
                        float impulse = -(1.0f + RESTITUTION) *
normal_velocity;
                        impulse /= 2.0f; // Assuming equal mass for both
particles

                        // Apply impulse
                        pi.velocity += normalized_diff * impulse;
                        pj.velocity -= normalized_diff * impulse;

                        // Separate particles to prevent overlap
                        float overlap = 2 * PARTICLE_RADIUS - r;
                        sf::Vector2f separation = normalized_diff * (overlap
* 0.5f);
                        pi.position += separation;
                        pj.position -= separation;

                        // Clear forces since we're handling collision
response through velocity
                        pi.force = sf::Vector2f(0.0f, 0.0f);
                        pj.force = sf::Vector2f(0.0f, 0.0f);
                    }
                }
            }

            // Combine all forces: pressure, viscosity, and gravity
            pi.force = pressure_force + viscosity_force + gravity *
pi.density;

            // Limit force magnitude
            float force_magnitude = std::sqrt(pi.force.x * pi.force.x +
pi.force.y * pi.force.y);
            if (force_magnitude > MAX_VELOCITY * pi.density) {
                pi.force *= (MAX_VELOCITY * pi.density / force_magnitude);
            }
        }
    }

    void integrate(float dt) {

        for (auto& p : particles) {
            // Update velocity with force
            p.velocity += dt * p.force / p.density;
```

```
            // Clamp velocity magnitude
            float speed = std::sqrt(p.velocity.x * p.velocity.x +
p.velocity.y * p.velocity.y);
            if (speed > MAX_VELOCITY) {
                p.velocity *= MAX_VELOCITY / speed;
            }

            // Update position
            p.position += dt * p.velocity;

            // Border collision with particle radius
            if (p.position.x + PARTICLE_RADIUS < bounds.left) {
                p.position.x = bounds.left - PARTICLE_RADIUS;
                p.velocity.x *= -DAMPING;
            }
            if (p.position.x + PARTICLE_RADIUS > bounds.left + bounds.width)
{
                p.position.x = bounds.left + bounds.width - PARTICLE_RADIUS;
                p.velocity.x *= -DAMPING;
            }
            if (p.position.y + PARTICLE_RADIUS < bounds.top) {
                p.position.y = bounds.top - PARTICLE_RADIUS;
                p.velocity.y *= -DAMPING;
            }
            if (p.position.y + PARTICLE_RADIUS > bounds.top + bounds.height)
{
                p.position.y = bounds.top + bounds.height - PARTICLE_RADIUS;
                p.velocity.y *= -DAMPING;
            }
        }
    }
};
```

Prywatne zmienne tej klasy pozwalają na dostrojenie symulacji:

- gravity to grawitacja
- bounds bierze się z ograniczenia poprzez białą ramkę wokół
- particles to lista przechowywująca wszystkie cząstki
- VISCOSITY to lepkość, oznaczająca jak bardzo cząstki przyklejają się do siebie
- REST_DENSITY to teoretyczna gęstość globalna do której będzie porównywana gęstość lokalna
- GAS_CONSTANT to stała modyfikująca jak bardzo bliskość (gęstość) cząstek wpływa na zmianę ciśnienia
- SMOOTHING_LENGTH to minimalna długość od której dwie cząstki zwiększają swoją gęstość
- SMOOTHING_LENGTH_SQ to po prostu kwadrat powyższego, dla szybszych obliczeń

- POLY6_SCALE to tzw. kernel gęstości oznaczający jak bardzo odległość wpływa na gęstość. Im bliżej siebie są dwie cząstki tym bardziej zwiększy się ich gęstość
- SPIKY_GRAD_SCALE to kernel ciśnienia
- VISC_LAP_SCALE to kernel lepkości
- PARTICLE RADIUS to promień cząstki
- DAMPING to tłumienie zderzeń. Im większy, tym szybciej cząstki stracą swoją prędkość po zderzeniu
- MAX_VELOCITY to maksymalna dozwolona prędkość z jaką mogą poruszać się cząstki
- PARTICLE_MASS to masa pojedyńczej cząstki

Metody:

- Konstruktor ustawia grawitację i ograniczenia wokół krawędzi ekranu
- Metoda `FluidSimulator.addParticle` dodaje jedną cząstkę wody do listy `particles`. Metoda `FluidSimulator.removeAllParticles` czyści listę `particles` (używana po kliknięciu przycisku RESET)
- `draw` rysuje cząstki z lub bez koloryzowania względem ciśnienia na podstawie globalnej zmiennej `show_coloring`
- `wind` nadaję wszystkim cząstkom prędkość w kierunku określonym przez argument tej funkcji
- `shake` nadaję każdej cząstcę wysoką prędkość w losowym kierunku
- `computeDensityPressure` kalkuluję gęstość wykorzystując kernel gęstości gdy odległość pomiędzy dowolnymi dwoma cząstkami jest mniejsza niż `SMOOTHING_LENGTH` i zmienia wartość `pressure` cząstki (dla każdej cząstki)
- `computePressure` Kalkuluję ciśnienie i lepkość. Rozdziela cząstki zbyt blisko siebie. Ustawia `force` każdej cząstki na sumę lepkości, ciśnienia i grawitacji pomnożonej przez gęstość. Na koniec ogranicza prędkość cząstek na podstawie MAX_VELOCITY
- `integrate` zamienia siłę `force` na prędkość `velocity` i odpowiednio modyfikuję pozycję cząstek. zanim nowa pozycja zostanie wyświetlona, funkcja sprawdza jeszcze czy nie wykracza ona poza granice

Setup przed rozpoczęciem głównej pętli

```
// Seed rand
srand(time(NULL));

// SFML Window Setup
sf::RenderWindow window(
        sf::VideoMode(800, 600),
        "Fluid Sim",
        sf::Style::Titlebar | sf::Style::Close
);
window.setFramerateLimit(60);
```

```
const float DELTA_TIME = 1.f / 60.f;

// FPS Counter Setup
sf::Font font;
font.loadFromFile("./resources/tuffy.ttf");
FPSCounter fps_counter;

// Border Setup
const float BORDER_PADDING = 20.f;
const float BORDER_THICKNESS = 4.f;
sf::RectangleShape border;
border.setPosition(BORDER_PADDING, BORDER_PADDING);
border.setSize(sf::Vector2f(
        window.getSize().x - 2 * BORDER_PADDING,
        window.getSize().y - 2 * BORDER_PADDING
));
border.setFillColor(sf::Color::Transparent);
border.setOutlineColor(sf::Color::White);
border.setOutlineThickness(BORDER_THICKNESS);

sf::FloatRect bounds(
        BORDER_PADDING + BORDER_THICKNESS,
        BORDER_PADDING + BORDER_THICKNESS,
        window.getSize().x - 2 * (BORDER_PADDING + BORDER_THICKNESS),
        window.getSize().y - 2 * (BORDER_PADDING + BORDER_THICKNESS)
);

// Define FluidSimulator
FluidSimulator simulator(bounds);

// Button & Slider setup
Button button_start(300, 200, 200, 50, "Start", font);
Button button_reset(550, 90, 220, 50, "Reset", font);
Button button_coloring(550, 30, 220, 50, "Show Pressure ON/OFF", font);
Slider slider_gridsize(300, 300, 200, 1, 35, "Grid Size");
Slider slider_radius(300, 360, 200, 3, 10, "Particle Radius"); // default 5
Slider slider_damping(300, 420, 200, 0, 100, "Damping%"); // default 7000
Slider slider_max_velocity(300, 480, 200, 300, 1000, "Max Velocity"); //
default 300
Slider slider_mass(300, 540, 200, 4, 10, "Particle Mass"); // default 5

button_start.setCallback([&show_menu, &bounds, &simulator, &button_reset,
&button_start, &slider_gridsize, &slider_radius, &slider_damping
,&slider_max_velocity, &slider_mass]() {
        show_menu = false;
        button_reset.setEnabled(true);
        button_start.setEnabled(false);

        // Modify Fluid Simulator attributes
        simulator.PARTICLE_RADIUS = static_cast<float>
```

```
(slider_radius.getValue()); // od 2 do 10
        simulator.MAX_VELOCITY = static_cast<float>
(slider_max_velocity.getValue()); // od 300 do 1000
        simulator.PARTICLE_MASS = static_cast<float>
(slider_mass.getValue()); //
        simulator.DAMPING = static_cast<float>(1.f -
slider_damping.getValue()/100.f); // do 7000

        const int GRID_SIZE = slider_gridsize.getValue();
        const float SPACING = 12.f;
        const float startX = bounds.left + bounds.width * 0.25f;
        const float startY = bounds.top + bounds.height * 0.25f;

        for (int row = 0; row < GRID_SIZE; row++) {
                for (int col = 0; col < GRID_SIZE; col++) {
                        simulator.addParticle(sf::Vector2f(
                                startX + col * SPACING - 1 + (rand() % 3),
                                startY + row * SPACING - 1 + (rand() % 3)
                        ));
                }
        }
});

button_reset.setCallback([&show_menu, &simulator, &button_reset,
&button_start]() {
        show_menu = true;
        button_reset.setEnabled(false);
        button_start.setEnabled(true);

        simulator.removeAllParticles();
});

button_coloring.setCallback([]() {
        if (show_coloring)
                show_coloring = false;
        else
                show_coloring = true;
});
```

Inicjalizuję okno SFML jako `window` 800x600 pikseli z ograniczeniem do 60 FPS.

Ładuję czcionkę i inicjalizuję licznik FPS.

Tworzę białą granicę ograniczającą ruch cząstek w okolicach krawędzi `window`.

Definiuję FluidSimulator jako simulator.

Tworzę interfejs użytkownika używając klas `Button` i `Slider`. Definiuję funkcję które będą uruchomione po naciśnięciu przycisków. Po naciśnięciu przycisku START pobieram dane ze

wszystkich sliderów i rozpoczynam symulację. Przycisk RESET usuwa wszystkie cząstki.

Callback dla przycisku od zmiany koloru modyfikuję zmienną globalną `show_coloring`.

Główna pętla - Event handling

```
while (window.isOpen()) {
        // Event handling
        sf::Event event;
        while (window.pollEvent(event)) {
                slider_gridsize.handleEvent(event, window);
                slider_radius.handleEvent(event, window);
                slider_damping.handleEvent(event, window);
                slider_max_velocity.handleEvent(event, window);
                slider_mass.handleEvent(event, window);
                switch (event.type) {
                        case sf::Event::Closed:
                                window.close();
                                break;
                        case sf::Event::KeyPressed:
                                switch(event.key.code) {
                                case sf::Keyboard::Q:
                                        window.close();
                                        break;
                                case sf::Keyboard::Space:
                                        simulator.shake();
                                        break;
                                case sf::Keyboard::Up:
                                        simulator.wind(0, 10.f);
                                        break;
                                case sf::Keyboard::Right:
                                        simulator.wind(1, 10.f);
                                        break;
                                case sf::Keyboard::Down:
                                        simulator.wind(2, 10.f);
                                        break;
                                case sf::Keyboard::Left:
                                        simulator.wind(3, 10.f);
                                        break;
                                default:
                                        break;
                                }

                                break;
                        case sf::Event::MouseButtonPressed:
                                button_start.handleEvent(event, window);
                                button_reset.handleEvent(event, window);
                                button_coloring.handleEvent(event, window);
                                break;
```

```
                        default:
                                break;
                }
        }

        // Update, clear, draw, display
        [...]
}
```

To, co użytkownik robi na klawiaturze i myszce wywołuję funkcje i metody pożądane dla konkretnych klawiszy, jak zamknięcie programu pod klawiszem Q, uruchomienie metody `handleEvent` dla przycisków, uruchomienie metody `simulator.wind` na strzałki góra/lewo/prawo/dół, uruchomienie metody `simulator.shake` na spację.

Główna pętla - Aktualizowanie i rysowanie obiektów

```
while (window.isOpen()) {
        // Event handling
        [...]

        // Update, clear, draw, display
        fps_counter.update();

        window.clear();

        button_coloring.draw(window);
        if (show_menu) {
                button_start.draw(window);
                slider_gridsize.draw(window);
                slider_radius.draw(window);
                slider_damping.draw(window);
                slider_max_velocity.draw(window);
                slider_mass.draw(window);
        } else {
                simulator.update(DELTA_TIME);
                simulator.draw(window);
                button_reset.draw(window);
        }

        window.draw(border);
        fps_counter.draw(window, font);
        window.display();
}
```

Elementy są chowane lub pokazywane na podstawie zmiennej globalnej `show_menu`.