Dokumentacja projektu

# Symulacja mechaniki płynów metodą SPH

*Iwo Plaza, Tomasz Koszarek, Paweł Świder, Maciej Czyjt*
*Kraków, 2021/2022*

# Spis treści

# Opis projektu

Projekt powstał na zajęciach z przedmiotu Design Patterns na kierunku Informatyka na Akademii Górniczo-Hutniczej. Zakłada stworzenie oprogramowania symulującego i wizualizującego mechanikę płynu w celu empirycznego dowodu Prawa Bernoulliego.

# Technologia

Oprogramowanie napisane zostało w języku Python. Korzysta z biblioteki CUDA umożliwiającej masowo równoległe obliczenia z wykorzystaniem karty graficznej.

# Model fizyczny

Główną ideą fizyczną stojącą za symulacją płynów są równania Naviera-Stokesa, opisujące ruch płynu na podstawie jego ciśnienia, gęstości oraz lepkości:

$$\rho \frac{Dv}{Dt} = \rho g - \nabla p + \mu \nabla^2 v$$

Pozwala ono obliczyć przyspieszenie płynu dla dowolnego punktu jego objętości.

# Technika SPH

SPH (Smoothed Particle Hydrodynamics) symuluje płyn przy użyciu cząsteczek będących nośnikami informacji o płynie (takich jak gęstość, ciśnienie, itp.) zdolnych do poruszania się zgodnie z dynamiką Newtona. W miejscach, w których znajdują się cząsteczki, cechy płynu są dokładnie znane, natomiast w pozostałych punktach wartości te można obliczyć, przy pomocy wartości z otoczenia.

Wartości te są interpolowane przy pomocy funkcji bazowej A:

$$A(r) = \int A(r')W(r - r', h)\, dr' \approx \sum_b A(r_b)W(r - r_b, h)$$

$r - pozycja\ rozpatrywanego\ punktu$
$A(r) - wartość\ szukanej\ wielkości/cechy\ dla\ tego\ punktu$
$r' - pozycja\ dowolnego\ innego\ punktu$
$h - promień\ wpływu$
$W(r - r', h) - wartość\ funkcji\ wagowej$
$b - indeks\ cząsteczki$

Funkcja ta zwraca wartość cechy *A* w punkcie *r*, sumując (całkując) wartości tej cechy z otoczenia oraz mnożąc je przez funkcję wagową *W*, która to zwraca wartości malejące z odległością. W szczególności istnieje promień maksymalnego wpływu, poza którym zwracana wartość będzie zawsze zerowa.

Technika ta pozwala zdyskretyzować równania *N-S* i obliczać cechy cząsteczki przy użyciu innych cząsteczek z jej otoczenia. Zdyskretyzowane równanie *N-S*:

$$\frac{dv_i}{dt} = g - \frac{1}{\rho_i}\nabla p + \frac{\mu}{\rho_i}\nabla^2 v \quad (1)$$

Powyższe równanie pozwala obliczyć przyspiesznie i-tej cząsteczki na podstawie jej sił zewnętrznych (np. grawitacja), gęstości, ciśnienia i lepkości. Podane są przybliżenia na wyrazy prawej strony równania korzystające w podanej wyżej idei interpolacji:

$$\rho_i \approx \sum_j m_j W(r - r_j, h) \quad (2)$$

$$\frac{\nabla p_i}{\rho_i} \approx \sum_j m_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2}\right) \nabla W(r - r_j, h) \qquad (3)$$

$$\frac{\mu}{\rho_i} \nabla^2 v_i \approx \frac{\mu}{\rho_i} \sum_j m_j \left(\frac{v_j - v_i}{\rho_j}\right) \nabla^2 W(r - r_j, h) \qquad (4)$$

Dzięki tym równaniom możemy w każdym kroku symulacji, dla każdej cząsteczki obliczyć potrzebne wartości na podstawie ich sąsiadów, następnie obliczyć przyspiesznie i zcałkować numerycznie z krokiem czasowym *dt*.

Funkcja wagowa *W* oraz jej gradient i laplasjan, potrzebne do obliczeń:

$$W(r - r_b, h) = \frac{315}{64\pi h^9}\left(h^2 - \left|\left|r - r_b\right|\right|^2\right)^3$$

$$\nabla W(r - r_b, h) = \frac{-45}{\pi h^6}\left(h - \left|\left|r - r_b\right|\right|^2\right)\frac{r - r_b}{\left|\left|r - r_b\right|\right|}$$

$$\nabla^2 W(r - r_b, h) = \frac{45}{\pi h^6}\left(h - \left|\left|r - r_b\right|\right|^2\right)$$

W związku z faktem, iż funkcja wagowa *W* zwraca wartość 0 dla wszystkich punktów znajdujących się poza promieniem wpływu, pojawia się miejsce na optymalizajcę. Zamiast w sposób naiwny, dla każdej cząsteczki obliczać wartość w oparciu o wszystkie inne, można robić to w oparciu tylko o jej sąsiadów.

# Zrównoleglenie obliczeń przy użyciu karty graficznej

Obliczenia symulacji polegać będą na przetworzeniu każdej z cząsteczek w sposób niezależny. Z tego powodu, w celu zwiększenia wydajności oprogramowania, w projekcie wykorzystujemy kartę graficzną przy pomocy biblioteki CUDA.

CUDA umożliwia definiowanie funkcji uruchamianych na karcie graficznej (nazywanych dalej kernelami), które wykonają się na setkach procesorów karty graficznej. Każda cząsteczka zatem może być przetwarzana na jednym wątku CUDA'y.

# Implementacja oprogramowania

Projekt zakłada wizualizację offline, w związku z czym powstały dwa programy, *symulator* odpowiedzialny za przeprowadzenie symulacji i zapisanie jej przebiegu w pliku oraz *wizualizator* odpowiedzialny za odczytanie symulacji i zwizualizowanie jej (z potencjalną możliwością małej interakcji). Stworzyliśmy również niewielki moduł *serializer* dostarczający obu programom możliwość zapisywania do / odczytywania z plików.

## Krótki opis architektury i działania programów

a) Symulator na podstawie parametrów początkowych oblicza kolejne stany symulacji. Stan symulacji jest zapisem pozycji, prędkości i gęstości każdej cząsteczki w danym punkcie czasu. Obliczenie jednego stanu polega na przetworzeniu każdej cząsteczki. Sprowadza się to do znalezienia jej sąsiadów „odbudowując" pomocniczą strukturę dzielącą przestrzeń na sektory a następnie obliczenie dla niej wartości z równań (2-4). Obliczana jest siła wypadkowa cząsteczki i wykonywane jest numeryczne całkowanie i w rezultacie znajdowana jest nowa pozycja cząsteczki. Płyn przepływa przez rurkę, która może składać się z kilku segmentów. Segment w ogólności ma dowolną długość i jest ściętym stożkiem, tzn. promień na jednym z jego końców może być różny od promienia na drugim końcu.
b) Wizualizator pobiera wygenerowane wcześniej stany symulacji oraz parametry środowiska, przygotowuje środowisko wizualizacyjne i pozwala użytkownikowi na przegląd przebiegu

symulacji. Każda cząsteczka odpowiada pół-przezroczystej, niebieskiej kropce w scenie trójwymiarowej, a rurka ograniczająca ciecz ujawnia się jako uproszczony obrys faktycznej formy (gdzie fizycznie potrzeba by nieskończonej ilości wierzchołków żeby odwzorować kształt ściętego stożka).

## Szczegółowy opis architektury i działania programów

### Moduł config

Jest modułem, do którego mają dostęp wszystkie inne moduły. Zawiera zmienne konfiguracyjne takie jak, liczba cząsteczek symulacji, czas trwania, kształt przestrzeni, itp.
Zawiera również funkcje pozwalające skonstruować potrzebne symulacji paramtery oraz stan początkowy.
Umożliwia skonfigurowanie aplikacjji w trybie BOX lub PIPE
Tryb BOX polega na umieszczeniu płynu w prostopadłościennym „akwarium".
Tryb PIPE umieszcza płyn w zdefiniowanej wcześniej rurce.

### Moduł common

Opis klas

***SimulationParameters***:
Jest dataklasą odpowiedzialną za przechowywanie stałych parametrów symulacji, do których należą:
- liczba cząsteczek
- siłą zawnętrzna (np. grawitacja)
- czas trwania symulacji w sekundach
- liczba klatek na sekundę
- definicja rurki
- rozmiar przestrzeni
- rozmiar voxela

***SimulationState***:
Jest dataklasą odpowiedzialną za przechowywanie informacji o stanie symulacji, w skład których wchodzą 3 tablice:
- position (przechowuje pozycje wszystkich cząsteczek)
- velocity (prędkości cząsteczek)
- density (gęstości cząsteczek)

***Segment:***
Jest dataklasą odpowiedzialną za przechowywanie informacji o segmencie rurki, przez którą przepływa płyn. Segment definiowany jest przez punkt o współrzędnych trójwymiarowych, w którym się zaczyna, długość segmentu, promień początkowy oraz końcowy.
Udostępnia metodę *radius_at(x)* przyjmującą jednowymiarowy punkt i zwracającą długość promienia walca w tym punkcie.

***Pipe:***
Enkapsuluje listę segmentów. Posiada metody:
- *get_length()* zwracającą całkowitą długosc rurki, czyli sumę długości segmentów
- *find_segment(x)* przyjmującą jednowymiarowy punkt i zwracjącą numer segmentu, do którego ten punkt należy
- *radius_at(x)* analogicznie do *Segment.radius_at(x)* zwraca wartość promienia dla danego punktu.

***PipeBuilder:***
Builder ułatwiający budowę instancji klasy *Pipe,* dba o to aby segmenty na siebie nie nachodziły oraz aby nie było pomiędzy nimi luk. Użycie budowniczego sprowadza się do dwóch kroków: ustawienia pierwszego segmentu oraz dodawania kolejnych segmentów które pojawiają się na końcu rurki.

- *with_starting_position(self, position: Tuple[float, float, float])* – ustawia punkt początkowy środka rurki
- *with_starting_radius(self, radius: float)* – ustawia długość promienia na początku pierwszego segmentu
- *with_ending_radius(self, radius: float)* – ustawia długość promianie na końcu pierwszego segmentu
- *with_starting_length(self, length: float)* – ustawia długość pierwszego segmentu
- *add_roller_segment(self, length)* – dodaje segment o określonej dłogości i jednakowym promieniu początkowym i końcowym równym promieniowi końcowemu ostatniego segmentu
- *add_lessening_segment(self, length, change)* – dodaje segment o określonej długości o różnych promieniach gdzie promień początkowy jest równy promieniowi końcowemu poprzedniego segmentu a promień końcowy nowego segmentu jest mniejszy od początkowego o wartość change
- *add_increasing_segment(self, length, change)* - dodaje segment o określonej długości o różnych promieniach gdzie promień początkowy jest równy promieniowi końcowemu poprzedniego segmentu a promień końcowy nowego segmentu jest większy od początkowego o wartość change
- *transform(self, space_size_x: float, space_size_yz: float, max_radius: float = None)* – tranformuje rurkę tak aby: łączna dłogość segmentów wynosiła *space_size_x*, współrzędne y oraz z dla osi symetrii rurki wynosiły *space_size_yz* / 2.0 oraz aby maksymalny promień wynosił wartość *max_radius* lub *space_size_yz* gdy nie jest wskazany. Funkcja zachowuje wszystkie proporcje pomiędzy długościami oraz promieniami.
- *get_result(self)* – zwraca gotową instancję klasy *Pipe.*

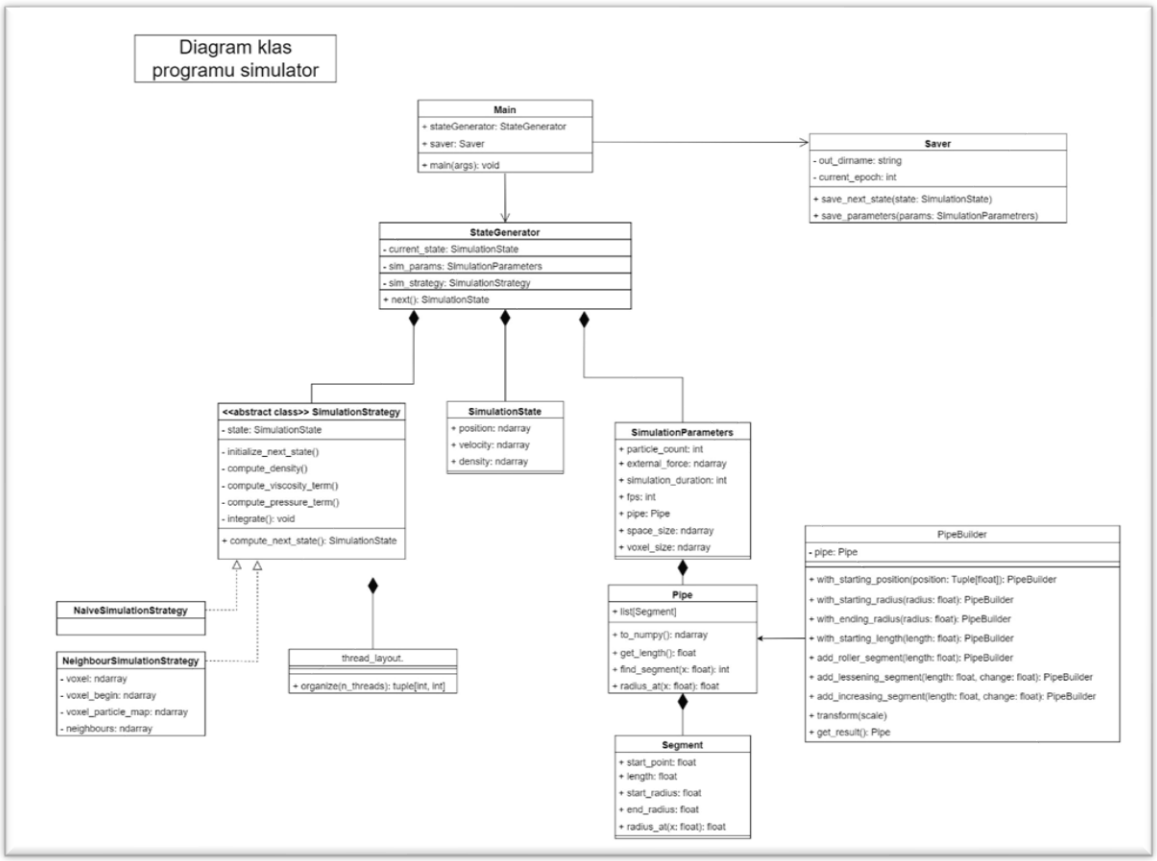
***Saver:***
Klasa odpowiedzialna za zapisywanie *SimulationParameters* oraz *SimulationState* dla każdej epoki. Tworzy folder gdzie zapisywane są dane. Dane reprezentowane w tablicach numpy zapisywane są do pliku .npy, a parametry symulacji do pliku w formacie JSON

***Loader:***
Wczytuje dane zapisane przez klasę *Saver*, udostępnia metody:

- *load_simulation_parameters(self)* – odpowiedzialną za wczytanie parametrów symulacji
- *load_simulation_state(self, epoch: int)* – odpowiedzialną za wczytanie stanu symulacji we wskazanej epoce.

## Program symulator

Diagram klas




Opis klas

***StateGenerator***:
Klasa wyeksponowana do użytku z zewnątrz udostępniająca API logiki biznesowej. Generuje ona kolejne stany symulacji (Python generator). Przy inicjalizacji potrzebuje obiektu *SimulationParameters* (na jego podstawie zna m. in. długość symulacji i liczbą klatek na sekundę) oraz *SimulationState* (pierwszy stan symulacji). Klasa monitoruje ilość i kolejność obliczanych stanów, których tworzenie deleguje w całości do *AbstractSPHStrategy*.

***AbstractSPHStrategy***:
Klasa odpowiedzialna za algorytm SPH. Jest używana przez *StateGenerator* do obliczania kolejnych stanów symulacji przy użyciu metody *compute_next_state()*, wewnątrz której wykonywane są kroki algorytmu. Klasa jest abstrakcyjna i realizuje wzorzec projektowy Strategy, umożliwiając uwtorzenie podklas zawierających wariacje algorytmu, które będą operować na obiektach klasy *SimulationState* na swoich zasadach. Metoda *compute_next_state()* definiuje ogólne kroki algorytmu. Są to:
- initialize_computations()
- compute_density()
- compute_pressure()
- compute_viscosity()
- integrate()
- collide()
- finalize_computations()

Inicjalizacja, całkowanie i kolizje są wspólne dla wszystkich podstrategii i są zaimplementowane wewątrz klasy abstrakcyjnej, natomiast obliczanie gęstości, ciśnienia i lepkości zależy od konkretnych podklas.

Inicjalizacja polega na przesłaniu danych aktualnego stanu do karty graficznej.

Obliczanie gęstości, ciśnienia i lepkości polega na obliczeniu wartości z wzorów (2-4) dla każdej cząsteczki. Wartości te posłużą do obliczenia siły wypadkowej cząsteczki.

Całkowanie to proces odpowiadający właściwemu przemieszaniu cząsteczek, na podstawie obliczonych wcześniej sił. Odbywa się ono w następujący sposób:

1. oblicz przyśpieszenie cząsteczki, dzieląc działającą na nią siłę wypadkową przez jej masę
2. oblicz przyrost prędkości cząsteczki, mnożąc przyśpieszenie przez krok czasowy *dt* i dodaj go do przechowywanej w stanie prędkości, aktualizując ją
3. oblicz zmianę pozycji cząsteczki, mnożąc prędkość przez krok czasowy *dt* i dodaj ją do przechowywanej w stanie pozycji cząsteczki, dokonując przemieszczenia.

Wykrywanie i rozwiązywanie kolizji polega na:
- znalezieniu segmentu rurki, w którym znajduje się dana cząsteczka
- sprawdzeniu, czy znajduje się ona poza rurką
- jeśli tak, należy statycznie przemieścić czasteczkę z powrotem do wnętrza rurki, a następnie obliczyć wektor prędkości, który uległ zmianie w wyniku odbicia

***NaiveSPHStrategy:***
Klasa dziedzicząca po *AbstractSPHStrategy,* wykonująca algorytm w sposób naiwny, tj. przy przetwarzaniu cząsteczki bierze pod uwagę wszystkie inne. Korzysta ona bezpośrednio z API CUDA, jednak abstrachuje je w taki sposób, że z zewnątrz nie jest ono widoczne. Realizuje wzorzec projektowy Facade.
Podczas inicjalizowania obliczeń dokonywany jest transfer danych do karty graficznej.
Obliczanie gęstości, ciśnienia i lepkości realizowane są przez wywołanie odpowiednich kerneli (w wersji naiwnej) wykonywanych wielowątkowo na GPU. Pozostałe kroki, czyli całkowanie oraz kolizje (zdefiniowane w klasie bazowej) nie są przeciążone, gdyż są wspólne dla wszystkich strategii. Przy finalizacji obliczeń dane pobierane są z karty graficznej i zapisywane do nowego obiektu *SimulationState* i zwracane.

***VoxelSPHStrategy:***
Klasa dziedzicząca po *AbstractSPHStrategy*, realizująca algorytm przetwarzając cząsteczki tylko na podstawie jej sąsiadów, co przyspiesza obliczenia. Podobnie jak strategia naiwna, jest fasadą dla CUDA API.

<u>Opis algorytmu i struktury danych służących do znajdowania sąsiadów cząsteczki:</u>

Input: pozycja rozpatrywanej cząsteczki
Output: lista jej sąsiadów

Algorytm dzieli się na dwa zasadnicze kroki:
a) zbudowanie odpowiednich struktur danych
b) przeszukanie ich w odpowiedni sposób, wydobywając szukanych sąsiadów.

krok a)
1. oblicz trójwymiarowy indeks voxela, w którym znajduje się cząsteczka na podstawie jej pozycji
2. stwórz listę 27 voxeli sąsiednich (oddalonych o 1 w każdym kierunku)
3. oblicz jednowymiarowy numer (indeks) każdego z tych voxeli
4. stwórz listę par (voxel_id, cząsteczka_id)
5. posortuj wyżej stworzoną liste według voxel_id
6. stwórz listy początków voxeli (początek voxela to indeks z listy par z kroku wyżej)

Powyższy proces odbywa się przy inicjalizacji obliczeń, a stworzone struktury danych wysyłane są na GPU wraz z danymi stanu.
W związku z tym poszczególne kernele przetwarzające cząsteczki optymalizują obliczenia korzystając z listy sąsiadów:

krok b)
1. dla każdego z sąsiednich voxeli odczytaj jego początek *begin* z tablicy początków
2. oblicz koniec *end* z tej samej tablicy (początek następnego niepustego voxela)

3. odwołując się do tablicy par, iterując po indeksach [*begin, end)* dodawaj do listy wynikowej czasteczki- sąsiadów

Opis mechanizmu rozdzielania zadań na wątki CUDA

Mechanizm ten zdefiniowany jest w module *thread_layout.py*.
Udostępniona jest funkcja *organize(total_thread_count)*. Przyjmuje ona całkowitą liczbę wątków, które chcemy uzyskać i zwraca *grid_size* (liczbę bloków w gridzie) oraz *block_size* (liczbę wątków w bloku). Wątki zatem ułożone są w jednym wymiarze i podzielone na bloki.
Funkcja ta pobiera dzięki CUDA API informację o obecnej karcie graficznej i jej *compute capability*. Wartość ta jest następnie odwzorowywana na najbardziej optymalny rozmiar bloku (zwykle około 128 wątkó na blok). Odwzorowanie to jest z góry zdefiniowane i powstało na podstawie dokumentacji CUDA.

## Program wizualizator

Ideą przy tworzeniu oprogramowania wizualiującego było zaprojektowanie architektury umożliwiającej łatwą podmianę zarówno technologii realizującej proces renderingu jak i bibliotek tworzących i zarządzających oknem programu.

Tworzenie okna i renderowanie

Poniżej znajduje się opis klas abstrakcyjnych, które operują między sobą i tworzą strukturę niezależną od implementacji:

Window

Symbolizuje okno programu. Organizuje wizualizację w listę warstw, gdzie kolejne warstwy przykrywają poprzednie.

| **Parametry inicjalizujące** | | |
|---|---|---|
| title: str | - | Tytuł okna. |
| width: int | default = 800 | Szerokość okna |
| height: int | default = 600 | Wysokość okna |

Layer

| **Interfejs** | |
|---|---|
| add(component: Component) | Dodaje komponent do tej warstwy. |
| register(identifier: str, component: Component) | Rejestruje komponent pod daną nazwą, aby była możliwa komunikacja między-warstwami. M.in. komendy mają dostęp do wszystkich rejestrowanych komponentów. |
| draw(delta_time: float) | Wywołuje krok `_update`, oraz rysuje komponenty w kolejności ich dodania wykorzystując krok czasowy delta_time (w sekundach). |
| **Metody protected** | |
| _update(delta_time: float) | Domyślnie bierna. Klasa podrzędna może zaimplementować kroki aktualizujące stan obiektów zawartych w warstwie na podstawie kroku czasowego delta_time (w sekundach). |

Każda warstwa przekazuje dalej powiadomienia które do niej nie należą według wzorca Chain of Responsibility. Takimi powiadomieniami są:

- **on_mouse_btn_pressed** – Naciśnięcie przycisku myszy nie powinno wpływać na warstwy pod aktywnym elementem
- **on_key_pressed** – Naciśnięcie klawisza nie powinno wywoływać dwóch akcji. Wyższe warstwy mają pierwszeństwo przechwycenia.

Są też powiadomienia które są przekazywane do każdej warstwy:

- **on_mouse_move** – Ruch myszą zwykle przydatny jest po uwczesnym kliknięciu przycisku, więc nie chcemy aby np. obrót kamerą był przerywany po najechaniu na przycisk.
- **on_mouse_btn_released** – Unikamy wymagania, żeby zaprzestanie obrotu kamerą musiało się odbyć z brakiem kursora na elemencie interfejsu.
- **on_key_released** – Upuszczenie klawisza musi być poprzedzone jego wciśnięciem, tak więc kontrolę nad przechwyceniem pozostawiamy samemu wciśnięciu

SceneComponentFactory

Używany przez warstwy, które tworzą część sceny trójwymiarowej.

| **Interfejs** | |
|---|---|
| create_point_field(origin: Vec3f, scale: Vec3f) → PointField | Metoda dostarczająca implementację komponentu typu PointField. |
| create_cube(origin: Vec3f, scale: Vec3f) → Cube | Metoda dostarczająca implementację komponentu typu Cube. |
| create_camera(origin: Vec3f, yaw: float, pitch: float) → Camera | Metoda dostarczająca implementację komponentu typu Camera. |
| create_wire_cylinder(start: Vec3f, end: Vec3f, start_radius: float, end_radius: float) → WireCylinder | Metoda dostarczająca implementację komponentu typu WireCylinder. |
| def create_wire_pipe(self, pipe: Pipe): | Metoda dostarczająca implementację komponentu typu WirePipe. |
| create_wire_cube(self, position: Vec3f, scale: Vec3f) → WireCube | Metoda dostarczająca implementację komponentu typu WireCube. |

UIComponentFactory

Używany przez warstwy, które tworzą część interfejsu graficznego.

| **Interfejs** | |
|---|---|
| create_stack_layout(self, pos: Tuple[int, int], spacing: int = 0) → StackLayout | Metoda dostarczająca implementację komponentu typu StackLayout. Służy do układania komponentów interfejsu graficznego w jednym rzędzie. |
| create_button(font: Font, pos: Tuple[int, int], label: str, click_command) → Button | Metoda dostarczająca implementację komponentu typu Button. |

LayerContext

Używany przez warstwy każdego rodzaju. Służy jako interfejs do komunikacji warstwy ze światem zewnętrznym.

| **Interfejs** | |
|---|---|
| invoke_command(command: Command) | Przekazuje komendę do wywołania w odpowiednim kontekście i do ewentualnego przetworzenia (np. logging) |
| create_font(path: str, font_size: int = 20) → Font | Metoda dostarczająca implementację obiektu typu Font. |

Implementacja z użyciem GLUT i OpenGL

Abstrakcyjną strukturę opisaną powyżej postanowiliśmy zaimplementować z użyciem OpenGL jako głównego API do komunikacji z kartą graficzną oraz wykorzystując GLUT do zarządzania oknem programu.