POLITECHNIKA WARSZAWSKA

DYSCYPLINA NAUKOWA INFORMATYKA
TECHNICZNA I TELEKOMUNIKACJA /
DZIEDZINA NAUK INŻYNIERYJNO-TECHNICZNYCH

# Rozprawa doktorska

mgr inż. Piotr Nowakowski

**Zaawansowane algorytmy proceduralnej generacji geometrii na karcie graficznej na potrzeby wizualizacji pól wektorowych**

Promotor
prof. dr hab. inż. Przemysław Rokita

WARSZAWA 2023

# Zaawansowane algorytmy proceduralnej generacji geometrii na karcie graficznej na potrzeby wizualizacji pól wektorowych

**Streszczenie.**

Wizualizacja pól wektorowych jest szeroko wykorzystywana zarówno w obszarze badań naukowych jak i w zastosowaniach przemysłowych. Jednym z głównych problemów tej tematyki jest duża ilość danych wejściowych, a co za tym idzie, duży koszt obliczeniowy. Z tego powodu obecnym trendem w dziedzinie jest opracowywanie algorytmów wykorzystujących równoległe przetwarzanie danych. Taką możliwość przetwarzania danych oferują najnowsze karty graficzne.

Przeprowadzone badania przedstawione w niniejszej pracy miały na celu opracowanie sposobów przechowywania informacji w pamięci karty graficznej o polu wektorowym oraz przeniesienie na kartę wszystkich istotnych obliczeń potrzebnych do wizualizacji pola. Opracowane techniki wizualizacji zostały oparte o proceduralną generację geometrii. Przedstawione metody generacji proceduralnej wykorzystują technologię shaderów siatki, która została wprowadzona na rynek w najnowszej generacji kart graficznych. Technologia ta nie została jeszcze w dostateczny sposób przebadana ani opisana w dostępnej literaturze naukowej.

**Słowa kluczowe:** grafika komputerowa, pola wektorowe, OpenGL, GLSL

# Advanced procedural geometry generation algorithms on a graphics card for vector field visualisation

**Abstract.**

Vector field visualisation has found broad use in both science and industry applications. One of the main challenges faced by researchers of this field is a large amount of input data required to be processed and the related computational cost. For this reason a certain trend in the field can be seen in which modern visualisation algorithms are created specifically for graphics card usage to utilize the excellent parallel processing capabilities offered by such devices.

The aim of the research done in this thesis was to propose ways of storage of field data in the video memory of the card as well as to create procedural geometry generation algorithms that utilize the available processing power. A recently released technology — mesh shaders — was used for this purpose. Due to their novelty mesh shaders are not yet sufficiently covered in the available body of scientific work.

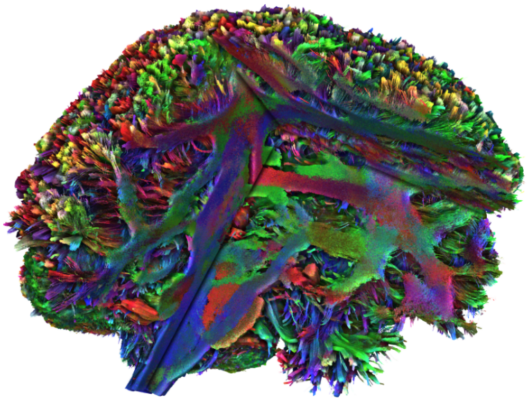
**Keywords:** computer graphics, vector fields, OpenGL, GLSL

# Spis treści

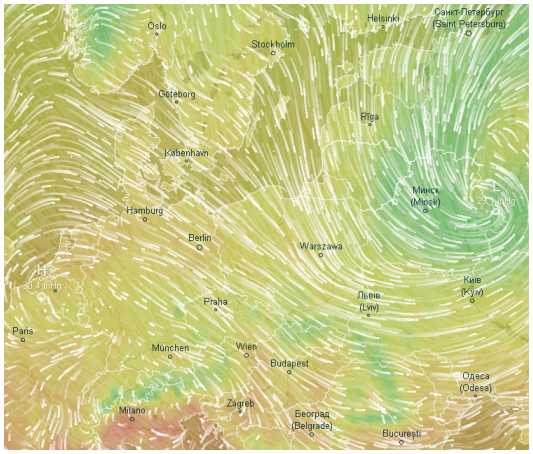
# 1. Wprowadzenie

## 1.1. Motywacja

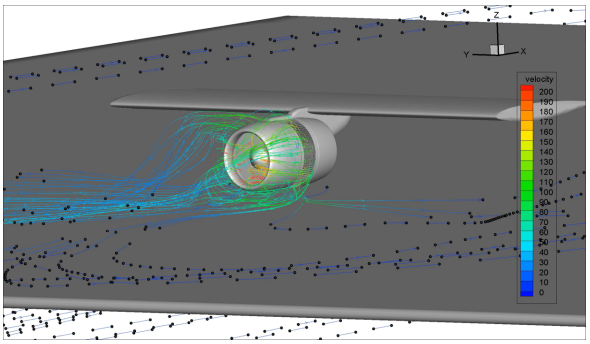
Pola wektorowe to fundamentalny dział matematyki, który znalazł bardzo szerokie zastosowanie przy tworzeniu modeli zjawisk fizycznych takich jak pola elektromagnetyczne lub przy symulacji przepływu cieczy i gazów. Pola, które można znaleźć w przyrodzie, zawierają skomplikowane wewnętrzne struktury oraz są zazwyczaj niewidzialne. Utrudnia to ich analizę bez dodatkowej pomocy w postaci graficznej reprezentacji. Wizualizacja pól jest często wykorzystywana w praktyce (np. w medycynie [1], inżynierii mechanicznej [2], czy nauce o pogodzie [3] — Rysunek 1.1), dlatego nie powinien dziwić fakt, że powstało wiele różnych metod wizualizacji dopasowanych do potrzeb danej dziedziny.



**(a)** Wizualizacja traktografii mózgu [1].




**(b)** Wizualizacja stanu atmosfery [3].




**(c)** Wizualizacja przepływu powietrza przez silnik lotniczy [2].

**Rys. 1.1.** Przykłady praktycznego zastosowania wizualizacji pól wektorowych.

Jednym z głównych problemów natury technicznej stojących przed zagadnieniem wizualizacji trójwymiarowych pól wektorowych jest ilość danych wejściowych. Takie dane muszą zawierać listę trójelementowych wektorów odpowiadających wszystkim punktom w analizowanej przestrzeni, co daje sześcienną zależność ich liczby od gęstości próbkowania obszaru zainteresowania. Należy przy tym zwrócić uwagę, że taki pojedynczy zbiór reprezentuje jedynie stan pola w pewnym punkcie na osi czasu — jeżeli wizualizacja wymaga pokazania zmian w charakterystyce pola w dłuższym okresie, rozmiar danych do przetworzenia może być jeszcze kilkukrotnie zwiększony [4].

Z dużą ilością danych do przetworzenia wiąże się duży koszt obliczeniowy algorytmów. Z tego powodu obowiązującym obecnie trendem w dziedzinie jest implementacja rozwiązań uruchamianych na kartach graficznych po to, aby wykorzystać ich szerokie możliwości równoległego przetwarzania danych. W literaturze można znaleźć wiele przykładów metod, które (choć pierwotnie stworzone do działania na głównym procesorze komputera) zostały ulepszone do pracy w wyżej wymieniony sposób [5].

Podejście to wiąże się z oddzielnym zestawem problemów do rozwiązania. Wąskim gardłem ograniczającym wydajność algorytmów działających na karcie graficznej jest stosunkowo niewielka przepustowość magistrali obsługującej główną pamięć wideo w porównaniu z możliwością przetwarzania danych oferowaną przez rdzenie (których liczba w najnowszych kartach sięga kilku tysięcy) [6]. Z tego powodu wydajność można w wielu przypadkach poprawić wykorzystując teselację lub algorytmy proceduralnej generacji (po to, aby zmniejszyć rozmiar potrzebnych danych wejściowych) — nawet jeżeli wiąże się to ze zwiększeniem ilości wymaganych obliczeń.

Architektura kart graficznych jest nieustannie rozwijana. W 2018 r. firma NVIDIA wprowadziła na rynek nową mikroarchitekturę kart graficznych Turing [7]. Oprócz dedykowanych rdzeni do generowania grafiki za pomocą techniki *ray tracing*, ta rodzina kart umożliwia również proceduralne generowanie geometrii przy użyciu nowego rodzaju shadera — shadera siatki [8]. Przeprowadzenie badań nad wykorzystaniem nowej technologii do opracowania wydajniejszych algorytmów wizualizacji pól wektorowych było jednym z celów tej rozprawy doktorskiej.

Praca ma również wymiar praktyczny. Będąc wieloletnim członkiem Kolabora-

cji ALICE w CERN oraz współautorem wykorzystywanego w niej oprogramowania $O^2$ [9] (zbioru bibliotek oraz gotowych programów używanych do przetwarzania danych otrzymywanych w trakcie eksperymentów fizycznych, szczegółowo opisanego w Rozdziale 2), dostrzegłem duże zapotrzebowanie społeczności naukowej na wizualizację danych pochodzących bezpośrednio z pomiarów, jak również wizualizacji różnych aspektów technicznych detektora zderzeń cząstek. W szczególności dotyczy to wizualizacji pola magnetycznego, które generowane jest przez wbudowane w detektor elektromagnesy. Jego właściwości zostały dokładnie określone przez inżynierów Kolaboracji, a powstały w ten sposób zbiór danych o wektorach pola został udostępniony publicznie [10]. Wizualizacja tego pola z jednej strony stanowiła interesujące wyzwanie, a z drugiej strony dostarczyła idealnego zbioru danych testowych, na których sprawdzać mogłem wydajność oraz efekt działania opracowanych przeze mnie algorytmów dla kart graficznych opisanych w dalszych rozdziałach niniejszej rozprawy. Należy przy tym zwrócić uwagę na to, że stworzone algorytmy wizualizacji mają charakter ogólny i z powodzeniem mogą zostać zastosowane również do innych pól wektorowych, co zostało zaprezentowane w Rozdziale 7.

### 1.2. Cel pracy

W ramach mojej rozprawy doktorskiej postanowiłem zrealizować następujące cele badawcze:

- *Opracowanie oraz implementacja skutecznej metody przechowywania oraz ewaluacji danych o polu wektorowym na karcie graficznej* [11] (na przykładzie modelu pola magnetycznego detektora ALICE), jako kroku niezbędnego do wykorzystania tego modelu do wizualizacji w ramach dalszych rozważań,
- *Opracowanie oraz implementacja metody propagacji trajektorii cząstek na karcie graficznej* [12], jako formy wizualizacji efektu oddziaływania pola wektorowego (tj. przedstawienia trajektorii lotu cząstek wykrytych przez detektor oraz jej zakrzywienia w efekcie oddziaływania pola magnetycznego) na przykładzie detektora ALICE,
- *Opracowanie oraz implementacja algorytmów wizualizacji pól wektorowych z wy-*

*korzystaniem potoku shaderów siatki* [13], gdzie model pola ALICE został wykorzystany jako jeden z przykładów demonstrujących efekt działania stworzonych metod,

- *Integracja wizualizacji pola magnetycznego z oprogramowaniem ALICE $O^2$*, jako mój wkład w kontynuację badań prowadzonych przez CERN.

## 1.3. Plan pracy

Dalsza część pracy podzielona jest na następujące części:

- Rozdział 2 zawiera opis ośrodka naukowo-badawczego CERN, Wielkiego Zderzacza Hadronów, detektora ALICE oraz oprogramowania $O^2$. Jego celem jest wyjaśnienie kontekstu niezbędnego w interpretacji moich oryginalnych osiągnięć opisanych w dalszych fragmentach pracy. Rozdział ten zawiera również szczegółowy opis wybranych algorytmów istniejących już w $O^2$, które posłużyły mi za punkt wyjścia w opisanych badaniach.
- Rozdział 3 zawiera opis architektury kart graficznych oraz potoku przetwarzania grafiki skupiający się na metodach generacji geometrii (punktów, linii, trójkątów) bezpośrednio na karcie. Porównuję w nim możliwości, jakie dostępne są przy zastosowaniu standardowego potoku graficznego z możliwościami oferowanymi przez nowy potok graficzny — potok shaderów siatki, który został zaprezentowany w 2018 r. przez firmę NVIDIA. Potok shaderów siatki wykorzystałem w nowatorskim rozwiązaniu wizualizacji pól wektorowych opisanym w Rozdziale 7.
- Rozdział 4 zawiera przegląd istniejących rozwiązań wizualizacji pól wektorowych, które posłużyły mi za inspirację do przeprowadzenia przedstawionych w pracy badań.
- Rozdział 5 zawiera opis moich badań dotyczących implementacji modelu pola magnetycznego detektora ALICE na karcie graficznej. Zapewnienie dostępu do próbek pola z poziomu kodu wykonywanego na karcie było pierwszym niezbędnym krokiem po to, aby jego wizualizacja była możliwa za pomocą rozwijanych w Rozdziale 7 technik. Zaproponowałem tu cztery autorskie sposoby przechowywania modelu w pamięci karty, które następnie przetestowałem pod kątem wydaj-

ności oraz wierności w stosunku do modelu referencyjnego zaimplementowanego w ramach oprogramowania $O^2$. Na podstawie tych badań powstał artykuł, który został opublikowany w czasopiśmie *Computer Physics Communications* [11].

- Rozdział 6 zawiera opis moich badań (przedstawionych we wstępnej formie na konferencji *Quark Matter* [14], a następnie przygotowanych do publikacji w czasopiśmie *Computer Physics Communications* — preprint [12]) dotyczących zastosowania modelu pola magnetycznego w obliczaniu oraz wizualizacji trajektorii lotu cząstek bezpośrednio na karcie graficznej.
- Rozdział 7 zawiera opis moich badań nad sposobami wykorzystania potoku shaderów siatki do wizualizacji pól wektorowych w czasie rzeczywistym. Na ich podstawie powstało oprogramowanie *FieldView* umożliwiające wyświetlanie charakterystyki dowolnego pola wektorowego za pomocą linii, wstążek, tub oraz powierzchni. Osiągnięcie zostało przygotowane do publikacji w czasopiśmie *SoftwareX* — preprint [13]. Wyniki badań w postaci wizualizacji pola magnetycznego detektora ALICE zostały również wdrożone jako część oprogramowania $O^2$.
- Rozdział 8 prezentuje podsumowanie pracy, komentarz do uzyskanych wyników oraz pomysły na dalsze ulepszenia opracowanych algorytmów.
- Załącznik A zawiera opis obsługi programu *FieldView*, który powstał na podstawie opracowanych w ramach Rozdziału 7 algorytmów wizualizacji pól wektorowych.
- Załącznik B przedstawia w całości mój dorobek naukowy podzielony na publikacje wykorzystane w niniejszej pracy, pozostałe publikacje oraz uczestnictwo w projektach badawczych.
- Załącznik C zawiera podziękowania nadesłanie przez zarząd Kolaboracji ALICE za mój dotychczasowy wkład pracy badawczej oraz inżynieryjnej wniesiony w okresie studiów doktoranckich w rozwój oprogramowania detektora.

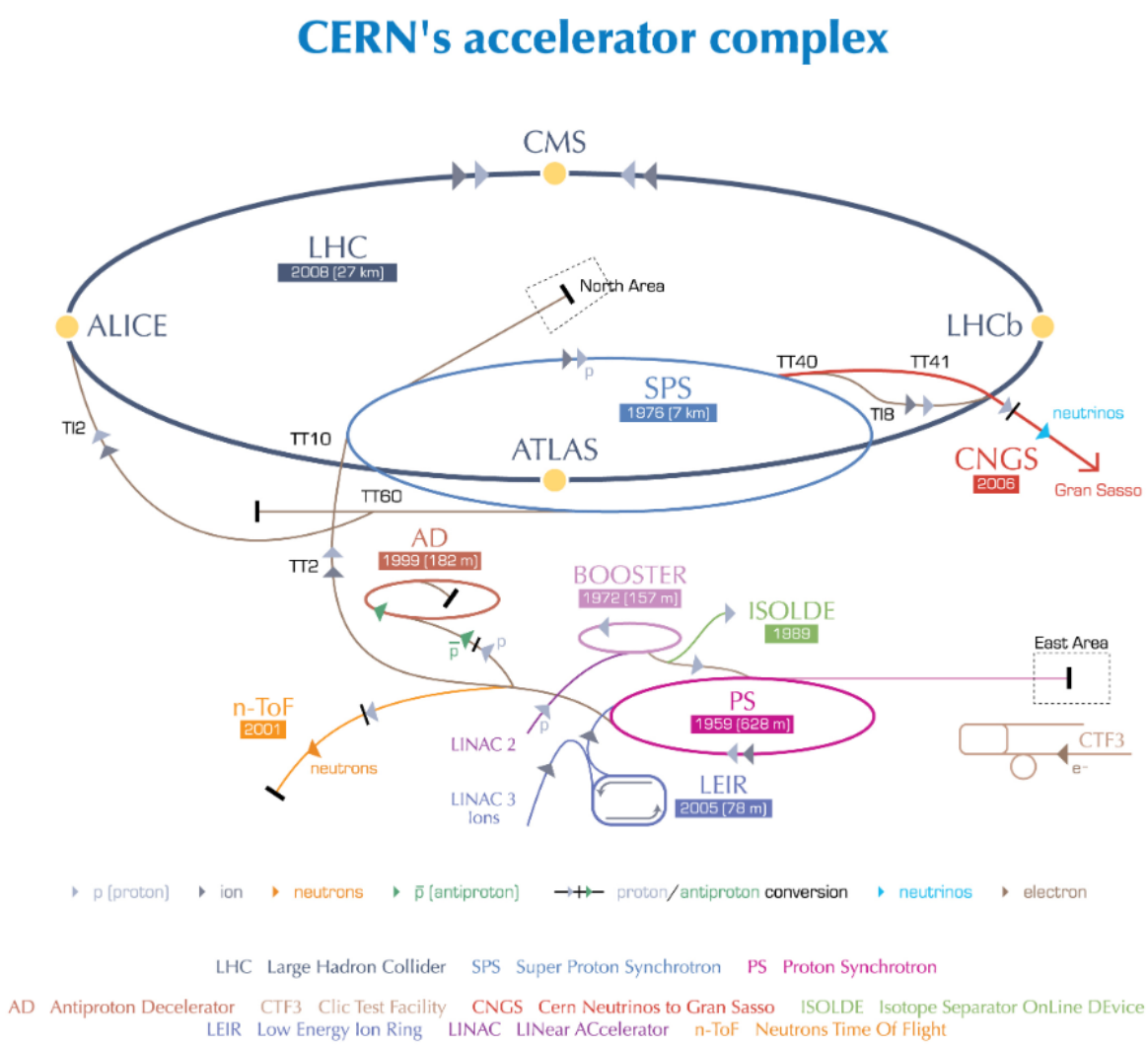
# 2. Eksperyment ALICE

Rozdział ma na celu przedstawienie najważniejszych informacji na temat placówki CERN oraz Eksperymentu ALICE (w tym sposób działania detektora oraz jego konstrukcję). Opis ten jest konieczny do interpretacji warunków w jakich wykorzystywany jest istniejący algorytm dostępu do modelu pola magnetycznego oraz algorytm rekonstrukcji trajektorii lotu cząstek przez niego wykrytych. Pierwszy z nich posłużył jako punkt wyjścia do opracowania sposobów dostępu do danych pola z poziomu pamięci karty graficznej (co jest sprawą kluczową umożliwiającą jakiekolwiek dalsze badania nad jego wizualizacją) opisanych w Rozdziale 5. Drugi posłużył jako punkt wyjścia w rozważaniach dotyczących wizualizacji efektu oddziaływania pola na trajektorię cząstek opisanych w Rozdziale 6.

## 2.1. Wielki Zderzacz Hadronów

Wielki Zderzacz Hadronów (*Large Hadron Collider*, LHC) [16] to akcelerator cząstek zbudowany przez Europejską Organizacja Badań Jądrowych (*Organisation Européenne pour la Recherche Nucléaire*, wcześniej *Conseil Européen pour la Recherche Nucléaire*, CERN). Ma on kształt okręgu o średnicy ok. 27 km, w którym dwie wiązki cząstek (protonów albo jąder atomów ołowiu) przyspieszane są w przeciwnych kierunkach, a następnie zderzane ze sobą. Przyspieszanie następuje w kilku etapach: zaczynając od *Linear Accelarator 3* (LINAC 3), poprzez *Proton Synchrotron* (PS) i *Super Proton Synchrotron* (SPS) i kończąc na LHC, gdzie cząstki osiągają docelową prędkość — 99.9999991% prędkości światła w próżni [17].

Zderzenia wiązek następują tam, gdzie zainstalowana jest aparatura służąca do rejestracji efektów kolizji. Są to cztery miejsca na LHC zaznaczone żółtymi kropkami na Rysunku 2.1, czyli detektory *A Toroidal LHC ApparatuS* (ATLAS), *Compact Muon Solenoid* (CMS), *Large Hadron Collider beauty* (LHCb) oraz *A Large Ion Collider Experiment* (ALICE).




**Rys. 2.1.** Zespół akceleratorów cząstek w CERN [15].

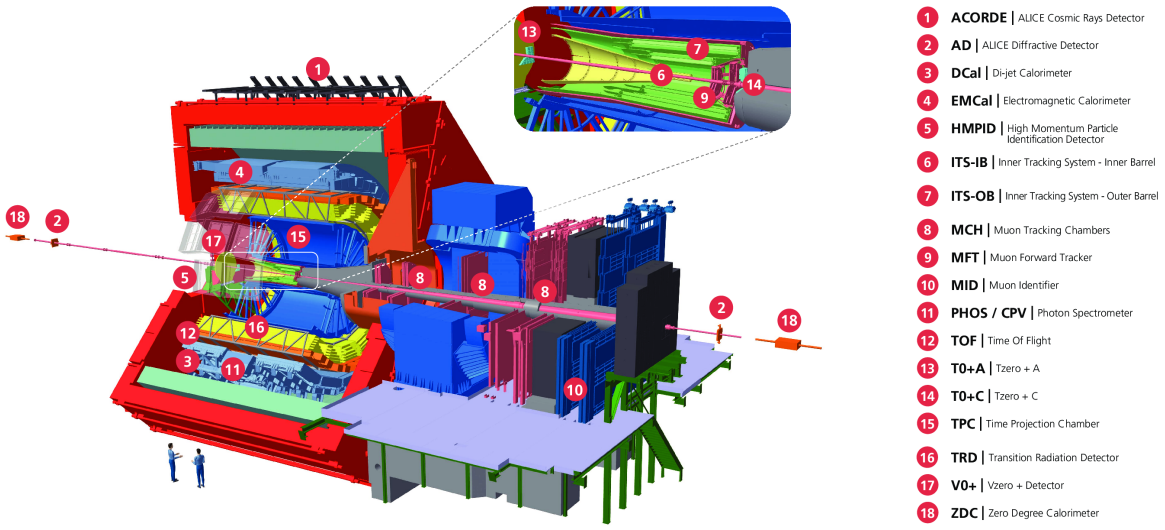
## 2.2. A Large Ion Collider Experiment

ALICE zlokalizowany jest na terenie miejscowości Saint Genis-Pouilly we Francji, w betonowej komorze umiejscowionej **56 m** pod ziemią. Rozmiary ALICE to ok. **16 m** wysokości, **26 m** długości i **16 m** szerokości. Maszyna, ważąca ok. **10000 t**, składa się z 18 systemów detekcyjnych, które dokonują pomiarów pozycji oraz energii cząstek powstających w zderzeniach.

Konstrukcję detektora przedstawia Rysunek 2.2. Główna część detektora to *Central Barrel*. Zawiera on większość systemów ALICE i zamknięty jest w cylindrycznym elektromagnesie solenoidalnym (L3), wytwarzającym stałe pole o maksymalnej sile **0.5 T** (na Rysunku 2.2 zaznaczony jest kolorem czerwonym, leży na nim bezpośrednio subde-

tektor ACORDE o numerze 1). W jego centrum (w środku subdetektora ITS, numer 6 na Rysunku 2.2) znajduje się *Interaction Point* (IP), gdzie wiązki pochodzące z LHC przecinają się, umożliwiając zderzenie. Aparatura badawcza ułożona jest w warstwach wokół IP, przez które sekwencyjnie przenikają cząstki pochodzące ze zderzeń. Najbliżej IP zainstalowane są systemy śledzenia trajektorii, następnie kalorymetry (elektromagnetyczny i hadronowy), a najdalej spektrometr fotonowy. System śledzenia trajektorii bada ruch cząstek, natomiast kalorymetry zatrzymują je i rejestrują ich energię.

Wyżej wymienione instrumenty nie są przystosowane do badań nad mionami, które mają wysokie pędy oraz słabo oddziałują z zainstalowaną wewnątrz L3 aparaturą. Aby je badać, przed cylindrem wybudowano dodatkową sekcję, zwaną Ramieniem Mionowym (*Muon Arm*). Połączenie między L3 i Ramieniem wykonane jest z materiału absorbującego prawie wszystkie cząstki nie będące mionami. Ramię zawiera drugi, silniejszy (0.7 T) elektromagnes dipolowy oraz dedykowany spektrometr do detekcji tych cząstek.




**Rys. 2.2.** Detektor ALICE oraz jego podzespoły [18].

Główną cechą odróżniającą ALICE od innych detektorów w CERN jest możliwość precyzyjnego śledzenia trajektorii lotu cząstek posiadających ładunek elektryczny. Ich tor lotu, zgodnie z prawem Lorentza, zakrzywiany jest przez wymienione wyżej elektromagnesy. Precyzja pomiaru takiej trajektorii pozwala na określenie z dużą dokładnością pędu oraz masy spoczynkowej cząstek. Przekłada się to ostatecznie na możli-

wość ustalenia ich typu. Pole generowane przez elektromagnesy nie jest idealnie równomierne, dlatego w analizie wyników przydatna byłaby możliwość jego wizualizacji. W ramach niniejszej pracy powstało rozwiązanie tego problemu, z którym nie zmierzyli się wcześniej naukowcy Kolaboracji.

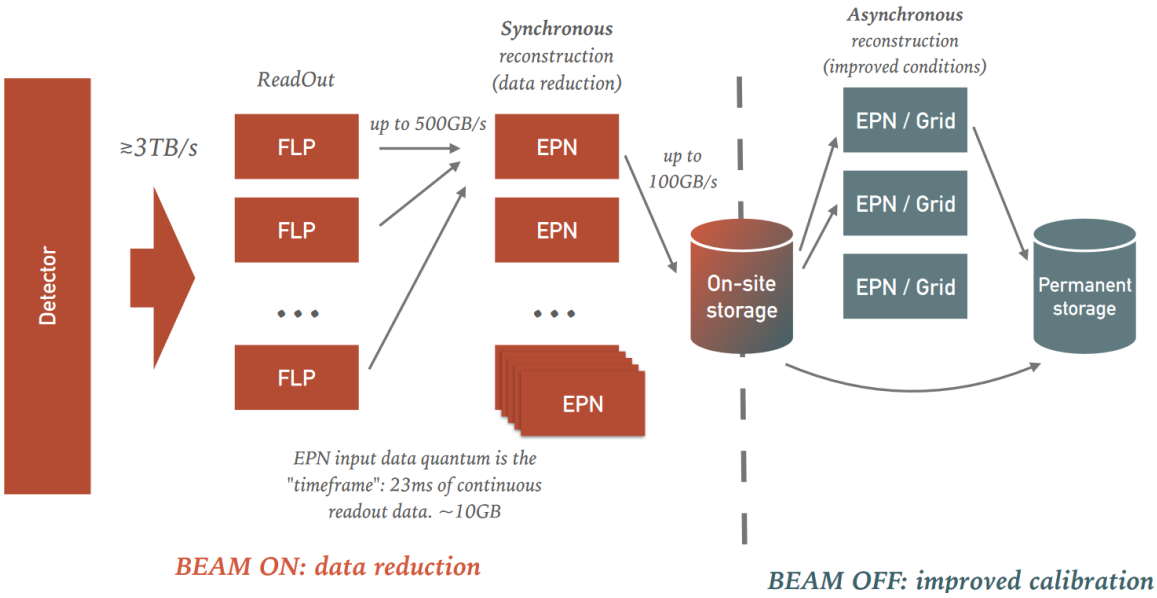
Detektor w swojej konstrukcji przystosowany jest do badań nad zderzeniami ciężkich jonów, które na ułamek sekundy wytwarzają w centrum detektora warunki, które według teorii panowały we Wszechświecie kilka mikrosekund po jego narodzinach, czyli stan plazmy kwarkowo - gluonowej [19]–[22].

Własności plazmy kwarkowo-gluonowej nie da się badać bezpośrednio. Są one poznawane drogą dedukcji — na podstawie porównywania efektów zderzeń zarejestrowanych przez instrumenty ALICE pomiędzy kolizjami ołów - ołów (gdzie plazma kwarkowo - gluonowa powstaje) i proton - proton (gdzie ta plazma nie powstaje) [23], [24].

Fizyka wysokich energii jest nauką opartą głównie o analizy statystyczne ze względu na nieprzewidywalność oraz niskie prawdopodobieństwo otrzymania interesujących rezultatów w wyniku zderzeń cząstek. Z tego powodu niezbędny jest nieustanny rozwój aparatury CERN w kierunku zwiększania częstotliwości zderzeń w LHC po to, aby przyspieszyć proces zbierania wystarczającej ilości próbek do potwierdzenia lub obalenia aktualnie rozpatrywanych teorii wyjaśniających funkcjonowanie Wszechświata.

W latach 2018 – 2022 CERN przerwał po raz drugi w historii placówki prowadzenie eksperymentów, a czas ten przeznaczył na modernizację instrumentów, przygotowując się do kolejnego okresu zbierania danych (zwanego *Run 3*), mającego potrwać do 2025 r. [25].

W trakcie oczekiwania na *Run 3* w ALICE, oprócz wymiany części podzespołów detektora, zastąpiony został w całości system akwizycji oraz przetwarzania danych. W obecnej formie obsługą detektora zajmują się dwa klastry obliczeniowe: *First Level Processors* (FPL) oraz *Event Processing Nodes* (EPN) [26]. Schemat systemu przedstawia Rysunek 2.3. Klaster FPL, zlokalizowany w pobliżu detektora pod ziemią, odbiera surowe dane pomiarowe z detektora oraz przeprowadza wstępną filtrację, rekonstrukcję oraz kompresję danych, redukując wymaganą przepustowość z ok. 3.4 TiB/s do ok. 500 GiB/s. Dzięki temu możliwe staje się przesłanie strumienia danych za pomocą




**Rys. 2.3.** Schemat przetwarzania danych od akwizycji w detektorze ALICE do miejsca przechowywania podczas *Run 3* [26].

dedykowanej sieci komputerowej do klastra EPN, zlokalizowanego w budynku serwerowni na powierzchni ziemi. Klaster EPN zajmuje się pełną rekonstrukcją danych, która polega na obliczaniu parametrów fizycznych wykrytych przez detektor cząstek (np. pierwotnej pozycji, ładunku elektrycznego, wektora pędu). Proces rekonstrukcji, zakończony kolejnym etapem kompresji, zapisuje dane na lokalnych dyskach twardych z prędkością 100 GiB/s. Kiedy detektor nie zbiera danych klaster EPN zajmuje się tworzeniem plików zawierających minimum informacji niezbędnych do analiz fizycznych (*Analysis Object Data*, AOD) oraz dystrybucją zawartości jego dysków na serwery na całym świecie, połączone w system GRID. Należy podkreślić, że pliki w ostatecznej formie zawierają jedynie parametry fizyczne wykrytych cząstek, natomiast trajektoria ich przelotu przez detektor, będąca informacją redundantną z punktu widzenia analiz fizycznych, nie jest zapisywana.

## 2.3. Oprogramowanie $O^2$

Zmiana fizycznej struktury systemu oraz konieczność zwiększenia wydajności przetwarzania danych wymagała stworzenia nowego oprogramowania do obsługi detektora [9], [27]. Wymaganiem, mającym fundamentalny wpływ na architekturę nowego systemu, było ujednolicenie sposobu obsługi procesów zachodzących podczas zbiera-

nia danych z detektora (nazywanym przez Kolaborację terminem zbiorczym *Online*) oraz sposobu przeprowadzania analiz fizycznych i poprawy kalibracji jego instrumentów (nazywanym przez Kolaborację terminem zbiorczym *Offline*). Ta cecha nowego oprogramowania była również źródłem inspiracji przy nadawaniu mu nazwy — skrót $O^2$, czyli *Online - Offline*, odnosi się do połączenia tych dwóch projektów, wcześniej rozwijanych niezależnie.

Na potrzeby dalszych części pracy (Rozdziałów 5, 7 i 6) w dokładniejszy sposób opisałem trzy elementy projektu $O^2$, czyli wspólny układ współrzędnych, model pola magnetycznego ALICE oraz algorytm rekonstrukcji trajektorii cząstek.

## 2.4. Wspólny układ współrzędnych ALICE

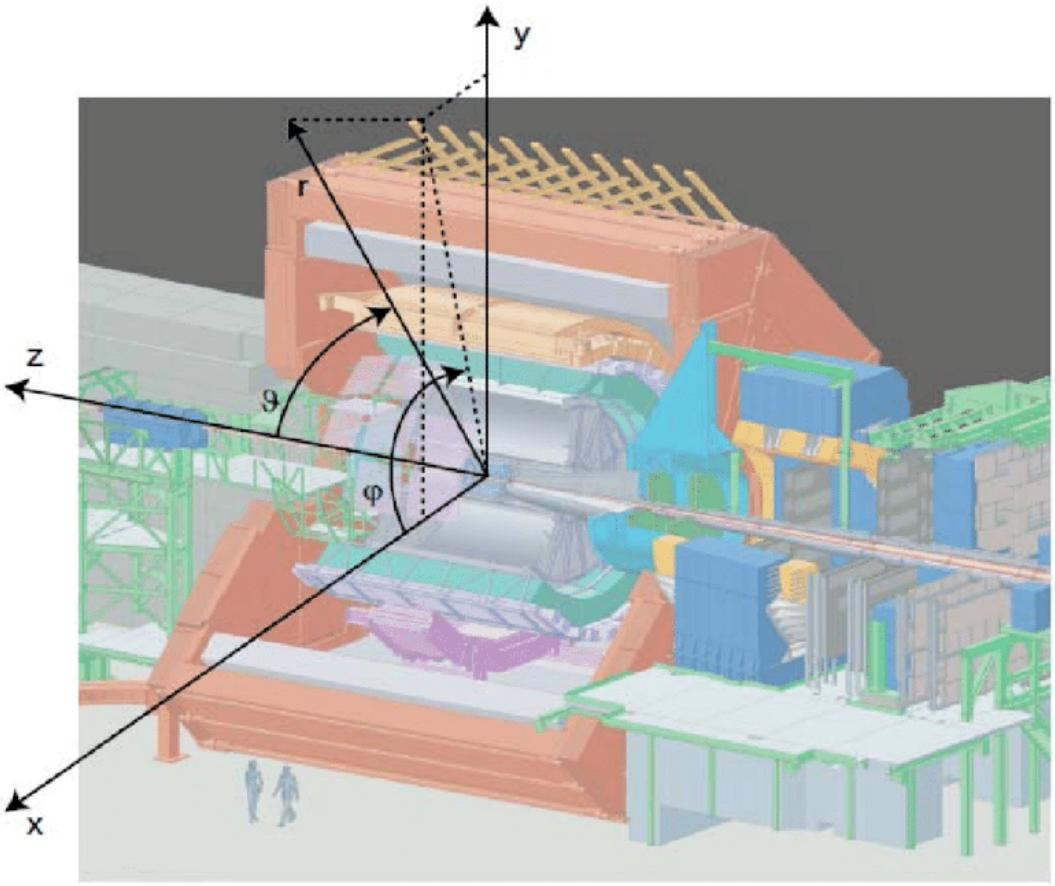
Aby ułatwić współpracę między zespołami fizyków i inżynierów odpowiedzialnych za różne części detektora, zdefiniowany został wspólny układ współrzędnych (Rysunek 2.4) w którym określona jest pozycja wszystkich jego elementów oraz wykrywanych w czasie pracy cząstek [28]. Koordynaty mogą być przedstawiane w układzie kartezjańskim albo w równoważnym układzie cylindrycznym. Jednostką odległości tego układu współrzędnych są centymetry.

W układzie kartezjańskim oś $z$ ma identyczny kierunek co oś wiązki LHC, natomiast zwrócona jest od Ramienia Mionowego w kierunku elektromagnesu L3. Oś $y$ skierowana jest w górę. Oś $x$ zdefiniowana jest w taki sposób, aby zachowana została reguła prawej dłoni.

Układ cylindryczny wykorzystuje tą samą oś $z$ co układ kartezjański. Współrzędna $r$ to odległość euklidesowa danego punktu od początku układu współrzędnych. Kąt $\phi$ zdefiniowany jest jako kąt między osią $x$ a rzutem danego punktu na płaszczyznę $(x, y)$.

## 2.5. Model pola magnetycznego

Charakterystyki pola magnetycznego generowanego przez obydwa elektromagnesy detektora ALICE zostały zmierzone w 2005 r. przez naukowców Kolaboracji [30], [31]. Zebrane dane pomiarowe zostały użyte do stworzenia modelu pola (oraz obsługującego go algorytmu) na potrzeby ówcześnie wykorzystywanego oprogramowania *AliROOT*.




**Rys. 2.4.** System współrzędnych detektora ALICE [29].

Algorytm ten (działający wyłącznie na procesorze komputera) został następnie przeniesiony do obecnie używanego oprogramowania $O^2$. Oryginalna implementacja jest dziełem Rubena Shahoyana [32]. Według specyfikacji [30], [31] algorytm zwraca wektor pola $\vec{B}$ z dokładnością w granicach od $2 \times 10^{-4}$ do $5 \times 10^{-3}$ kGauss, w zależności od lokalizacji żądanego punktu w objętości detektora (26 m na 16 m na 16 m). Punkt ten musi być podany we współrzędnych kartezjańskich ALICE. Dokładność przestrzenna modelu to 1 cm.

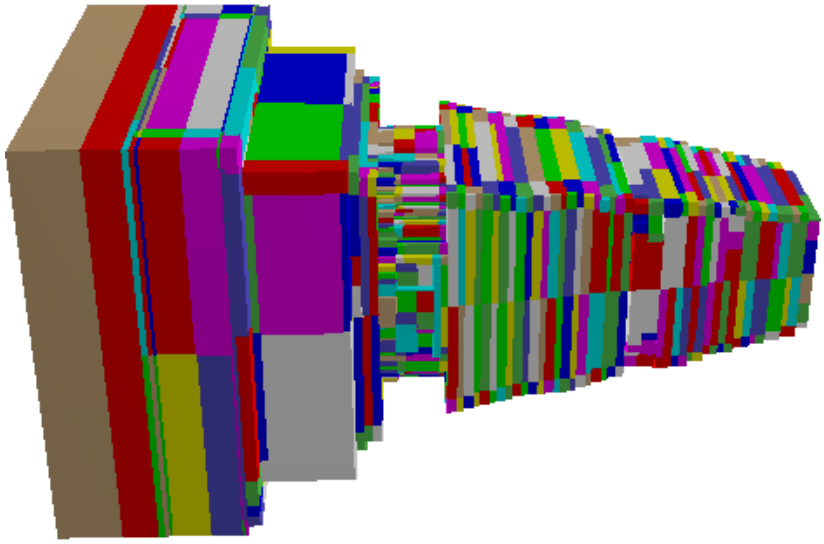
Obliczenia wykonywane są jedynie wtedy, gdy współrzędna $z$ żądanego punktu znajduje się w przedziale $[-1760\,\text{cm},\ 850\,\text{cm}]$ — w przeciwnym wypadku zwracany jest wektor zerowy. Oprogramowanie ALICE zawiera dwa modele pola (opisujące oddziaływanie każdego elektromagnesu oddzielnie), a odpowiedni z nich wybierany jest na podstawie współrzędnej $z$ w następujący sposób:

- $z \leq -550\,\text{cm}$, wtedy używany jest model elektromagnesu dipolowego,

- $z > -550\,\mathrm{cm}$, wtedy używany jest model elektromagnesu solenoidalnego.

### 2.5.1. Model elektromagnesu dipolowego

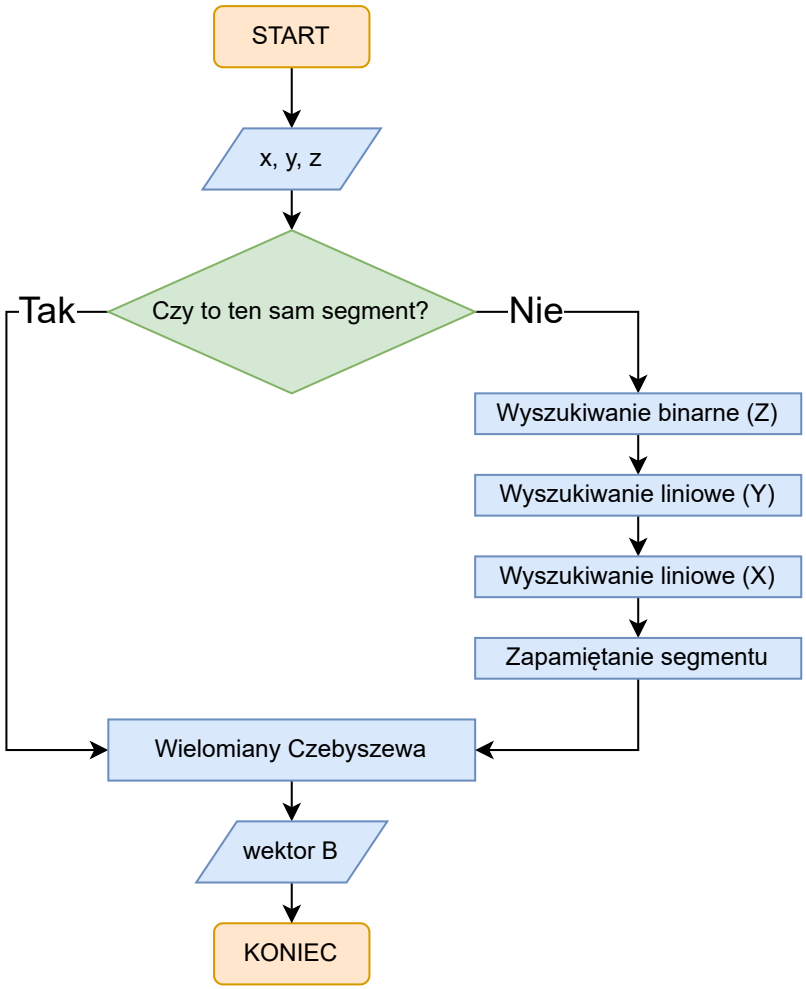
Model elektromagnesu dipolowego składa się z wielu prostopadłościennych segmentów w przestrzeni trójwymiarowej. Pole w każdym z segmentów przybliżone jest innym wielomianem Czebyszewa i o różnych stopniach. Wizualizację segmentów modelu przedstawia Rysunek 2.5.



**Rys. 2.5.** Wizualizacja systemu segmentów modelu elektromagnesu dipolowego [33].

Odpowiedni segment wyszukiwany jest za pomocą następującego algorytmu w $O^2$:

1. Sprawdź, czy żądany punkt leży w poprzednio znalezionym segmencie.
2. Jeżeli tak, to przypisany do niego wielomian użyj do obliczenia wektora pola. Koniec.
3. Jeżeli nie, odpowiedni rząd segmentów na osi $z$ znajdź przy pomocy wyszukiwania binarnego.
4. Następnie odpowiedni rząd segmentów na osi $y$ znajdź przy pomocy wyszukiwania liniowego.
5. Ostatecznie odpowiedni segment znajdź przy pomocy wyszukiwania liniowego względem osi $x$.
6. Znaleziony segment zapisz w pamięci podręcznej, a przypisany do niego wielomian użyj do obliczenia wektora pola. Koniec.




**Rys. 2.6.** Schemat algorytmu wyszukiwania segmentu.

Schemat algorytmu wyszukiwania segmentu przedstawiony jest na Rysunku 2.6. Pamięć podręczna poprawia wydajność algorytmu, gdy żądane punkty leżą blisko siebie. W takiej sytuacji pomijany jest krok wyszukiwania, a zamiast tego wielokrotnie ewaluowany jest wielomian tego samego segmentu.

### 2.5.2. Model elektromagnesu solenoidalnego

Model tego magnesu również składa się z segmentów. W tym wypadku jednak, aby wykorzystać symetrię kołową elektromagnesu, ułożone są one według ich pozycji w cylindrycznym układzie współrzędnych ALICE ($r$, $\phi$, $z$) oraz mają kształt fragmentów pierścienia. Algorytm wyszukiwania jest podobny do tego omówionego w Rozdziale 2.5.1. Różnica polega na tym, że szukanie oraz ewaluacja pola operuje na współrzędnych cylindrycznych (algorytm szuka segmentów kolejno po osiach $z$, $\phi$, $r$).

Z tego też powodu algorytm rozpoczyna działanie od dodatkowej konwersji współrzędnych kartezjańskich do cylindrycznych, a kończy dodatkową konwersją obliczonego wektora pola z systemu cylindrycznego na kartezjański.

### 2.5.3. Ewaluacja pola

Po zidentyfikowaniu odpowiedniego segmentu modelu żądany przez użytkownika punkt $\vec{P}$ (bez zmian w postaci kartezjańskiej w przypadku elektromagnesu dipolowego, po zmianie na postać cylindryczną w przypadku elektromagnesu solenoidalnego) konwertowany jest do "wewnętrznej" reprezentacji $\vec{P_{int}}$ w taki sposób, aby każda współrzędna $\vec{P_{int}}$ zawierała się w przedziale $[-1, 1]$. Umożliwia to użycie w kolejnych krokach algorytmu wartości tych współrzędnych jako parametrów wielomianów Czebyszewa. $\vec{P_{int}}$ otrzymywany jest za pomocą równania $\vec{P_{int}} = (\vec{P} - \vec{offsets}) \cdot \vec{scales}$, gdzie operator kropki to w tym wypadku mnożenie element po elemencie. Wektor translacji $\vec{offsets}$ i wektor skalowania $\vec{scales}$ są unikalne dla każdego segmentu.

W przypadku elektromagnesu dipolowego każdy element wynikowego wektora pola $\vec{B}$ (oznaczony na potrzeby tego wywodu jako $B^0$, $B^1$, $B^2$) jest obliczany przy użyciu różnych wielomianów Czebyszewa, używając współrzędnej $x$ pozycji jako ich parametr. Współczynniki tych wielomianów (oznaczone jako $c_i$) nie są przechowywane w danych segmentu bezpośrednio. Są one obliczane w locie za pomocą innego zestawu wielomianów Czebyszewa, używającego współrzędnej $y$ pozycji. Współczynniki tych wielomianów (oznaczone jako $c_{ij}$) również nie są przechowywane — są obliczane za pomocą finalnego zestawu wielomianów, używającego współrzędnej $z$ pozycji. Współczynniki tych ostatnich (oznaczone jako $c_{ijk}$) są przechowywane razem z danymi segmentu.

Dla przykładu, aby obliczyć element $B^0$, muszą zostać wykonane następujące operacje:

$$c_{ij}^0 = \sum_{k=0}^{K_j^0}, c_{ijk}^0 T_k(z),$$
$$c_i^0 = \sum_{j=0}^{J_i^0} c_{ij}^0 T_j(y),$$
$$B^0 = \sum_{i=0}^{I^0} c_i^0 T_i(x).$$

Funkcja $T_n$ to pojedynczy wielomian Czebyszewa rzędu $n$. $K$, $J$ i $I$ to maksymalne rzędy wielomianu używanego do obliczenia danego współczynnika, które mogą być za każdym razem inne. Ten sam zestaw działań (operujący na innym zestawie danych) musi zostać wykonany po to, aby uzyskać $B^1$ i $B^2$ i ukończyć obliczanie wektora $\vec{B}$ dla danej pozycji w przestrzeni.

W przypadku elektromagnesu solenoidalnego kroki potrzebne do obliczenia wektora pola są takie same z tą różnicą, że algorytm operuje na współrzędnych cylindrycznych $r$, $\phi$, $z$. Wynik kalkulacji jest również przedstawiony w tym układzie, więc ostatnim, dodatkowym krokiem jest jego konwersja z powrotem na układ kartezjański.

### 2.6. Rekonstrukcja trajektorii cząstek

Cząstki posiadające ładunek elektryczny w stałym polu magnetycznym poruszają się po helisie zgodnie z prawem Lorentza. Autorem algorytmu w $O^2$ wykorzystującego tą własność do rekonstrukcji trajektorii ich lotu jest Matevz Tadel oraz Alja Mrak-Tadel [34]. Wymaga on od użytkownika podania ładunku elektrycznego cząstki $q$, jej pierwszej znanej pozycji $\vec{V}$, jej wektora pędu $\vec{P}$ oraz wektora pola magnetycznego $\vec{B}$. Algorytm działa w sposób iteracyjny, określając kolejne pozycje cząstki w pętli.

Pierwszym krokiem algorytmu jest obliczenie wektorów lokalnego układu współrzędnych $\vec{E}_1$, $\vec{E}_2$ i $\vec{E}_3$, w którym ten pierwszy jest równoległy do wektora pola $\vec{B}$, a dwa pozostałe tworzą płaszczyznę do niego prostopadłą. Wektor $\vec{E}_1$ jest równy wektorowi pola $\vec{B}$ poddanemu normalizacji. Obliczenie wektorów $\vec{E}_2$ i $\vec{E}_3$ wymaga najpierw rozdzielenia wektora pędu $\vec{P}$ na jego składowe wzdłużną $\vec{P}_\parallel$ oraz prostopadłą $\vec{P}_\perp$. Aby uzyskać wektor $\vec{P}_\parallel$ należy najpierw obliczyć iloczyn skalarny $\vec{P}$ i $\vec{E}_1$ — otrzymamy w

ten sposób długość szukanego wektora. Sam wektor $\vec{P}_{\parallel}$ można otrzymać mnożąc wektor $\vec{E}_1$ przez obliczoną długość. Składową prostopadłą $\vec{P}_{\perp}$ można otrzymać odejmując od wektora $\vec{P}$ wektor $\vec{P}_{\parallel}$. Wektor $\vec{E}_2$ równy jest wektorowi $\vec{P}_{\perp}$ poddanemu normalizacji. Brakujący wektor $\vec{E}_3$ można otrzymać za pomocą iloczynu wektorowego $\vec{E}_1$ i $\vec{E}_2$. Jeżeli ładunek cząstki $q$ jest ujemny, należy odwrócić kierunek otrzymanego wektora $\vec{E}_3$.

Podsumowując, algorytm wykonuje następujące operacje, aby uzyskać wektory $\vec{E}_1$, $\vec{E}_2$ i $\vec{E}_3$:

$$\begin{aligned}
\vec{E}_1 &= norm(\vec{B}),\\
\vec{P}_{\parallel} &= (\vec{P}\cdot\vec{E}_1)*\vec{E}_1,\\
\vec{P}_{\perp} &= \vec{P}-\vec{P}_{\parallel},\\
\vec{E}_2 &= norm(\vec{P}_{\perp}),\\
\vec{E}_3 &= \vec{E}_1\times\vec{E}_2,\\
\vec{E}_3 &= sign(q)*\vec{E}_3.
\end{aligned}$$

Należy zwrócić uwagę na to, że kolejność wektorów w iloczynie wektorowym jest odwrotna niż we wzorze na siłę Lorentza, $\vec{F}=q\vec{v}\times\vec{B}$. Wynika to z faktu, że CERN stosuje odwróconą konwencję kierunku pola magnetycznego [35].

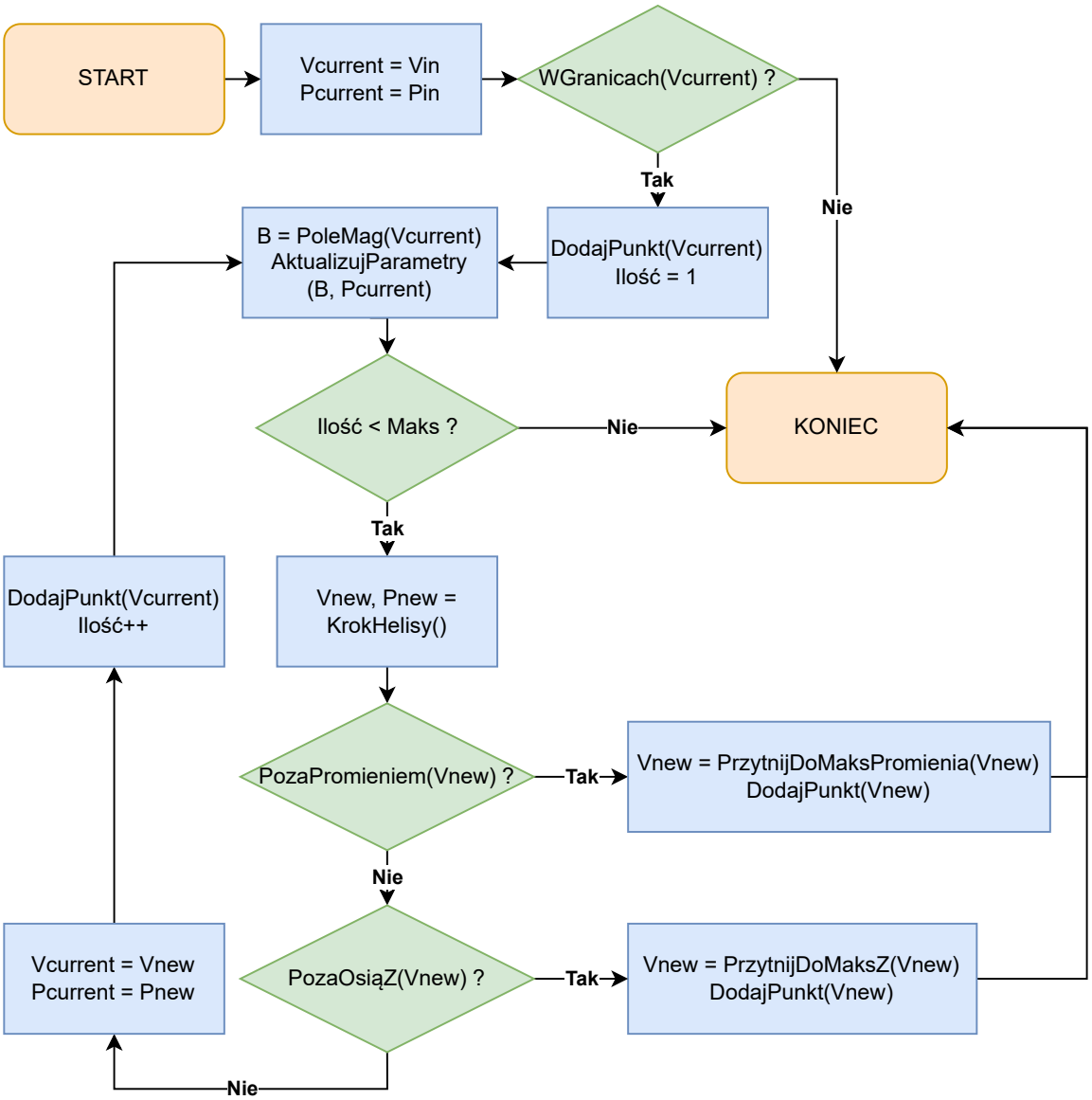
Nowa pozycja cząstki $\vec{V_{new}}$ oraz jej zaktualizowany wektor pędu $\vec{P_{new}}$ może zostać obliczony przy pomocy równania parametrycznego ruchu po helisie:

$$R = \frac{\left|\vec{P_\perp}\right|}{B2C * \left|\vec{B}\right| * \left|q\right|},$$

$$\begin{aligned} \vec{V_{new}} = \vec{V} &+ R\frac{\left|P_\parallel\right|}{|P_\perp|}\varphi * \vec{E_1} \\ &+ R\sin\varphi * \vec{E_2} \\ &+ (R * (1 - \cos\varphi)) * \vec{E_3}, \end{aligned}$$

$$\begin{aligned} \vec{P_{new}} = \vec{P_\parallel} &+ \left|\vec{P_\perp}\right|\cos\varphi * \vec{E_2} \\ &+ \left|\vec{P_\perp}\right|\sin\varphi * \vec{E_3}, \end{aligned}$$

gdzie $B2C$ to stała (konwersji pędu na krzywiznę), $R$ to promień helisy i $\varphi$ to kąt azymutalny, będący jednocześnie parametrem określającym długość kroku algorytmu. Kroki aktualizacji wektorów lokalnego układu współrzędnych oraz obliczania nowej pozycji cząstki wykonywane są naprzemiennie do momentu osiągnięcia maksymalnej ilości iteracji albo przekroczenia ustalonych przez użytkownika granic. Schemat algorytmu przedstawia Rysunek 2.7.



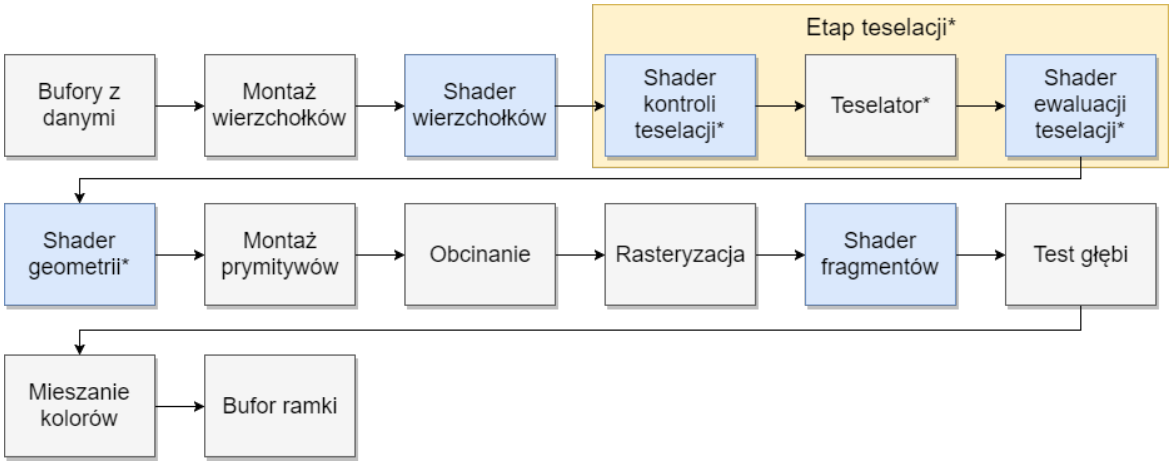

**Rys. 2.7.** Schemat algorytmu obliczania trajektorii cząstek. Funkcja `AktualizujParametry()` dokonuje aktualizacji wektorów układu współrzędnych. Funkcja `KrokHelisy()` oblicza nową pozycję oraz wektor pędu cząstki. Funkcje `PrzytnijDo*()` wymuszają granice.

Granice definiuje się jako maksymalną dopuszczalną wartość współrzędnej $z$ oraz maksymalną odległość euklidesową (promień) na płaszczyźnie $(x, y)$. Jeżeli punkt $\vec{V_{new}}$ osiąga lub przekracza wyznaczoną granicę na płaszczyźnie $(x, y)$, jego pozycja jest do niej przycinana. Operacja ta zaimplementowana jest jako liniowa interpolacja pomiędzy poprzednią pozycją $\vec{V}$, a następną $\vec{V_{new}}$ z wagą dobraną w taki sposób, aby wynikowy punkt leżał dokładnie na granicy. W podobny sposób punkt przycinany jest do granicy na osi $z$.

# 3. Generowanie geometrii na karcie graficznej

Rozdział ten przedstawia krótki opis potoków graficznych (klasycznego oraz shaderów siatki) ze szczególnym uwzględnieniem konkretnych ich etapów, które pozwalają na implementację algorytmów proceduralnej generacji geometrii. Na możliwości ich programowania opiera się część badawcza rozprawy. Opis został uzupełniony o listę przykładowych zastosowań danego etapu w dziedzinie wizualizacji, które wyszukałem w dostępnej literaturze naukowej.

## 3.1. Klasyczny potok graficzny




**Rys. 3.1.** Klasyczny potok graficzny. Gwiazdką zaznaczono etapy opcjonalne [36].

Współczesny potok graficzny (Rysunek 3.1) składa się z wielu etapów, z których część jest programowalna (zaznaczona kolorem niebieskim), a część nieprogramowalna (zaznaczona kolorem szarym). Etapy programowalne to takie, których funkcjonowanie zdefiniowane jest kodem pisanym przez programistę w języku *OpenGL Shading Language* (GLSL, używanym w OpenGL [37]) lub *High Level Shading Language* (HLSL, używanym w DirectX [38]). Etapy nieprogramowalne to takie, których funkcjonalność jest z góry zdefiniowana i zazwyczaj zaimplementowana na poziomie sprzętowym. Każdy etap nieprogramowalny posiada ograniczony zbiór ustawień, które można zmieniać. Tylko za ich pomocą możliwy jest wpływ na operacje w nich wykonywane.

To, czy dany etap jest programowalny, ściśle zależy od architektury karty graficznej. Wczesne karty posiadały wyłącznie etapy nieprogramowalne. Możliwość programowania pojawiła się wraz z zastąpieniem sprzętowych bloków procesorami wierzchołków i pikseli, a następnie została rozszerzona, kiedy również i te procesory zostały usunięte na rzecz rdzeni ogólnego przeznaczenia [39].

Wraz z ich wprowadzeniem pojawiła się również możliwość wykorzystania dużych możliwości kart do przetwarzania równoległego do pracy nad danymi innymi niż grafika za pomocą shaderów obliczeń. Shadery te mogą w razie potrzeby współdzielić wyniki obliczeń z potokiem graficznym, ale w ogólności traktować należy je jako część zupełnie oddzielnego, jednoetapowego potoku obliczeniowego [40].

Shadery wierzchołków operują na informacjach dotyczących poszczególnych wierzchołków (pozycja w przestrzeni, kolor, koordynaty na teksturze), które zostały połączone przez nieprogramowalny etap montażu wierzchołków z danych pochodzących z różnych buforów przekazanych karcie przez programistę. Ogólnie przyjętym zadaniem shadera wierzchołków jest obliczenie ich pozycji na ekranie za pomocą mnożenia przez macierz projekcji, ale może on wysyłać do dalszych etapów również inne dane (co jest przydatne, gdy zastosowana jest teselacja albo shader geometrii).

Teselacja to nieprogramowalny etap potoku, w którym prymitywy (linie lub trójkąty) dzielone są na mniejsze po to, aby w sposób proceduralny zwiększyć szczegółowość rysowanych obiektów. Shader kontroli teselacji pozwala na sterowanie opcjami etapu nieprogramowalnego (liczbą podziałów) na podstawie np. odległości obiektu od wirtualnej kamery. Etap teselacji generuje dodatkowe wierzchołki oraz przesyła o nich informacje wraz z ich współrzędnymi barycentrycznymi (w ramach oryginalnej figury), na podstawie których shader ewaluacji teselacji oblicza np. ich pozycję na ekranie. Teselację, mimo opcji produkowania dodatkowych wierzchołków, można wykorzystać do generowania nowej geometrii tylko w niektórych przypadkach. Jest to spowodowane tym, że instancje shadera ewaluacji teselacji wykonują się niezależnie dla każdego wynikowego wierzchołka. Uniemożliwia to implementację algorytmów generujących nową geometrię w sposób iteracyjny.

Shader geometrii [41] jako dane wejściowe otrzymuje wierzchołki pochodzące z

poprzednich etapów oraz informację, w jaki sposób są one połączone. Etap ten pozwala na zmianę kolejności wierzchołków, generowanie dodatkowych, a nawet zmianę rodzaju prymitywu, który przetwarzany jest w potoku graficznym. Ponieważ pojedyncza instancja shadera uruchamiana jest dla każdego prymitywu wejściowego, pozwala on na swobodną implementację różnych algorytmów generacji, w tym algorytmów iteracyjnych.

Shader fragmentów operuje na pikselach wygenerowanych w trakcie operacji rasteryzacji. Jego zadaniem jest wyliczenie koloru oraz głębi danego piksela — informacji niezbędnych do konstrukcji ostatecznego obrazu.

### 3.2. Shader geometrii w literaturze

Rozważania zawarte w niniejszej pracy skupiają się głównie na shaderze geometrii ze względu na jego duże możliwości w zakresie generacji dodatkowych elementów obrazu. Jego wszechstronność potwierdza liczba jego zastosowań w różnych dziedzinach nauki, która została opisana w dostępnej literaturze.

Shader geometrii został między innymi wykorzystany przy wizualizacji sieci przesyłu energii elektrycznej [42].

Prace [43], [44] wykorzystują shadery geometrii, OpenGL oraz potok graficzny do implementacji systemu przestrzennej bazy danych.

W [45] zaproponowany został algorytm bezpośredniej wokselizacji obiektów opisanych w reprezentacji powierzchniowej za pomocą tego shadera.

W pracy [46] shader ten został wykorzystany w kontekście medycznym do wizualizacji wyników traktografii istoty białej, która umożliwia uwidocznienie kierunku i ciągłości przebiegu włókien nerwowych (podobna praca wykorzystuje do tej wizualizacji shadery siatki [1] — Rozdział 3.4).

Prace [47] oraz [48] analizują jego zastosowanie jako sposób ewaluacji płatów Béziera (powierzchni parametrycznych stosowanych w modelowaniu geometrycznym).

W [49] posłużył on jako część systemu generacji zestawu danych do treningu dla sieci neuronowych, których celem jest detekcja oraz rekonstrukcja kształtu obiektów na podstawie materiałów wideo.

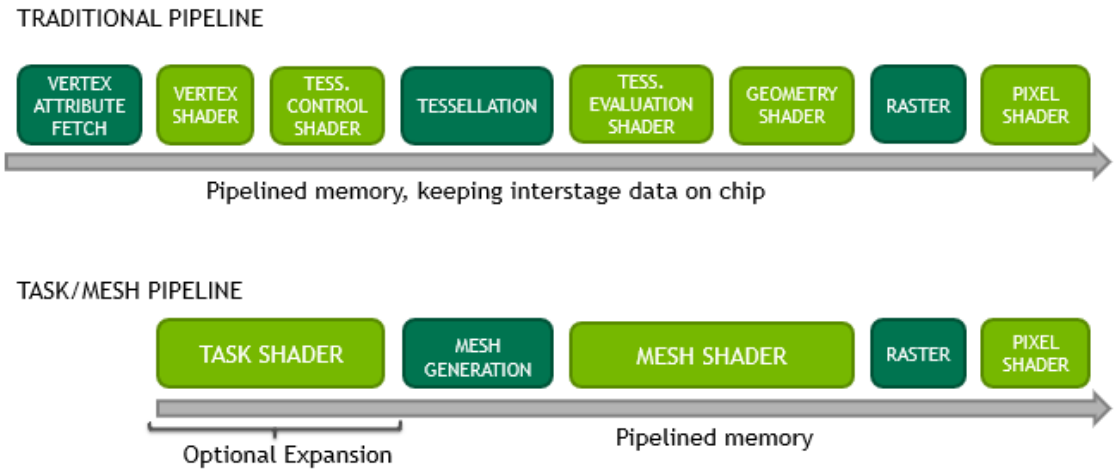
Praca [50] wykorzystuje shader geometrii w ramach techniki wizualizacji tekstury tkanin pluszowych oraz pokrytych wzorami (tkanin żakardowych).

W [51] został on wykorzystany w ramach silnika Unity do wizualizacji chmury punktów (pochodzących np. z pomiarów odległości systemem laserowym LIDAR) w przestrzeni wirtualnej rzeczywistości (*Virtual Reality*, VR).

Oprogramowanie Ragrug [52] wykorzystuje go do wizualizacji informacji kontekstowych nakładanych na obraz z kamery wideo w ramach systemu rzeczywistości rozszerzonej (*Augmented Reality*, AR).

Shader geometrii został wykorzystany do implementacji systemu cząsteczkowego na użytek gier komputerowych (często używanego w tej dziedzinie do wizualizacji ognia, dymu, wiatru) [53].

### 3.3. Potok shaderów siatki




**Rys. 3.2.** Porównanie etapów klasycznego potoku oraz potoku shaderów siatki [54].

Architektura klasycznego potoku graficznego oparta jest o jednowątkowe przetwarzanie niezależnych jednostek pracy (tzn. pojedynczego wierzchołka, pojedynczego prymitywu, pojedynczego piksela) — przyspieszenie osiągane jest przez wykonywanie obliczeń dla wielu jednostek na raz. Jest to pewnego rodzaju abstrakcja ułatwiająca programowanie, ponieważ w rzeczywistości rdzenie kart graficznych wykonują te same operacje na częściowo współdzielonych danych, tzw. *Single Instruction Multiple Thread* (SIMT). Projektowanie shaderów obliczeń dużo bardziej odzwierciedla

rzeczywisty sposób działania kart. Programista ma tu do dyspozycji zbiór wątków wykonujących identyczny kod — opracowywanie wydajnych algorytmów opiera się o umiejętne wykorzystanie identyfikatora wątku np. do adresowania komórek pamięci wykorzystywanych do obliczeń w taki sposób, aby sumarycznie cały zbiór wykonywał żądane zadanie. Potok ten nie jest przystosowany jednak do generowania grafiki.

W 2018 r. firma NVIDIA wprowadziła nową mikroarchitekturę kart graficznych Turing [7] wspierającą sprzętowy *ray tracing*. Zmiany w konstrukcji pozwoliły również na wprowadzenie nowego, alternatywnego potoku graficznego opartego o shadery zadań oraz siatki. Model programistyczny tych shaderów jest bardzo podobny do modelu shaderów obliczeń. Shader zadań to etap opcjonalny — za jego pomocą można dynamicznie dopasować ilość wywołań shadera siatki. Jego zadanie jest w pewnym sensie analogią do zadania shadera kontroli teselacji. Shader siatki jest bezpośrednio odpowiedzialny za przygotowanie listy wierzchołków tworzących prymitywy, które zostaną następnie wykorzystane przez rasteryzator.

Główną koncepcją stojącą za wprowadzeniem nowego potoku jest przyspieszenie generowania grafiki przez ograniczenie wczytywania oraz przesyłania między etapami danych o wierzchołkach trójkątów, które zostają później odrzucone ze względu na bycie niewidocznymi. Efektywne zastosowanie nowego potoku polega więc na wstępnym podzieleniu renderowanych modeli 3D na "strzępy" (*meshlets*) złożone z pewnej liczby trójkątów (na obecnej architekturze liczbą optymalną jest 126 [8]), które shader zadań powinien wstępnie ewaluować pod kątem widoczności. W ten sposób duża część trójkątów odrzucana jest już na samym początku, co pozwala na wykorzystanie przepustowości pamięci do przetwarzania danych, które faktycznie są widoczne na ekranie.

Rysunek 3.2 przedstawia porównanie schematów klasycznego potoku oraz potoku shaderów siatki. Etapy zbierania danych o wierzchołkach, shadera wierzchołków, teselacja oraz shader geometrii zastąpione zostały dwoma shaderami (zadań i siatki) oraz jednym nieprogramowalnym etapem generującym zadania.

### 3.3.1. Shader siatki

Podobnie jak shadery obliczeń, shadery siatki przetwarzają dane w lokalnych grupach wątków, których liczebność kontrolowana jest przez programistę za pomocą dyrektywy `layout(local_size_x=...)`). Nie mają również żadnych domyślnych zmiennych wejściowych dostarczanych przez potok. Wymaga to samodzielnego oprogramowania obsługi danych pochodzących z zewnętrznych buforów, ale pozwala na zastosowanie dowolnego ich formatu lub na implementację generacji takich danych proceduralnie, bezpośrednio w shaderze siatki. Cecha ta została wykorzystana w badaniach opisanych w Rozdziale 7.

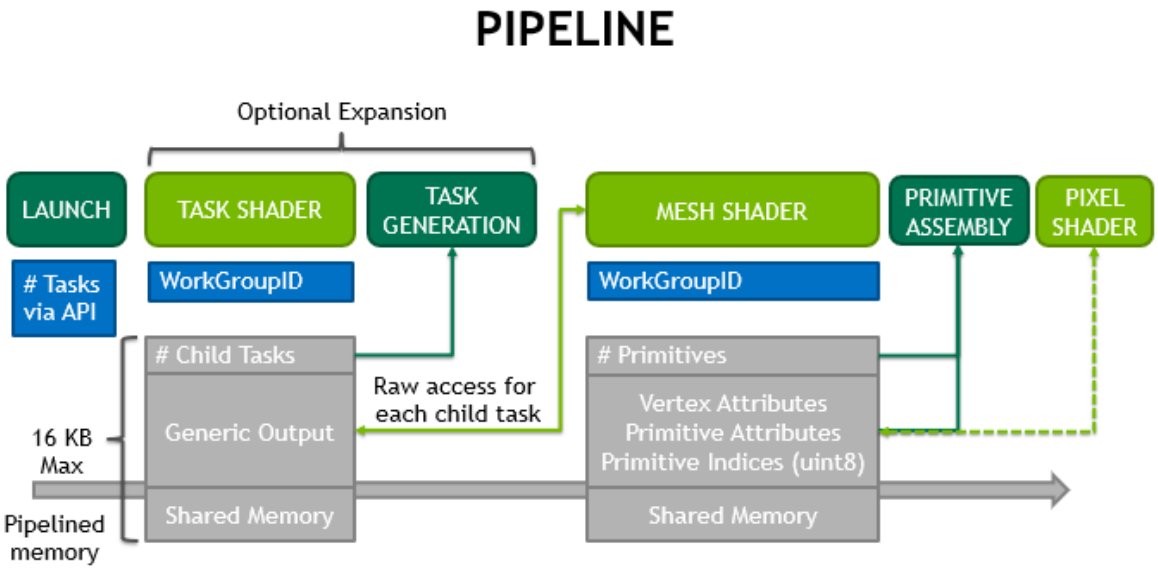
Informacjami wyjściowymi z shadera siatki jest typ generowanych prymitywów (punkty, linie lub wierzchołki), pozycje wierzchołków (zapisywane do specjalnej tablicy `gl_MeshVerticesNV`), dane o indeksach (zapisywane do `gl_PrimitiveIndicesNV`) oraz ostatecznie ilość wygenerowanych prymitywów (zapisywana do zmiennej `gl_PrimitiveCountNV`). Na potrzeby alokacji pamięci maksymalna potencjalna liczba generowanych wierzchołków oraz prymitywów musi zostać określona za pomocą dyrektywy `layout(max_vertices=...,max_primitives=...)`.

```
1  #version 450
2  #extension GL_NV_mesh_shader : require
3
4  layout(local_size_x=1) in;
5  layout(points, max_vertices=1, max_primitives=1) out;
6
7  layout(std140, binding = 0) uniform state_state {
8      mat4 MVP;
9  } state;
10
11 layout(std430, binding = 1) buffer vertices_vertices {
12     vec4 pos[];
13 } vertices;
14
15 out PerVertexData {
```

```
16      vec4 color;
17  } v_out[];
18
19  taskNV in Task {
20      uint vertexID;
21  } IN;
22
23  void main() {
24      const uint thread_id = gl_LocalInvocationID.x;
25      const vec4 position = vertices.pos[IN.vertexID];
26      gl_MeshVerticesNV[thread_id].gl_Position = state.MVP * position;
27      v_out[thread_id].color = vec4(1.0, 0.0, 0.0, 1.0);
28      gl_PrimitiveIndicesNV[thread_id] = 0;
29      gl_PrimitiveCountNV = 1;
30  }
```

**Listing 1.** Przykład najprostszego shadera siatki.

### 3.3.2. Shader zadań




**Rys. 3.3.** Schemat bloków pamięci współdzielonych pomiędzy shaderem zadań i shaderem siatki [54].

Tak jak wspomniano wyżej, celem wykorzystania shadera zadań jest dynamiczne określenie liczby shaderów siatki, które mają być uruchomione. Ma on jednak pewne dodatkowe możliwości przydatne w innych zastosowaniach — między innymi może on udostępnić wśród swoich wywołań shadera siatki niewielką ilość pamięci współdzielonej (maks. 16 KiB — Rysunek 3.3) w postaci bloku `taskNV`, gdzie zapisać można informacje, co powinien zrobić dany shader siatki. Ze względu na to, że shadery siatki mają jedynie dostęp do swojego lokalnego indeksu wywołania, funkcjonalność ta jest bardzo przydatna do przekazania im np. wartości przesunięcia w tablicy wierzchołków (czyli który wierzchołek wykorzystać do obliczeń). Cecha ta została wykorzystana w badaniach opisanych w Rozdziale 7.

```
1  #version 450
2  #extension GL_NV_mesh_shader : require
3
4  taskNV out Task {
5      uint vertexID;
6  } OUT;
7
8  void main() {
9      OUT.vertexID = gl_WorkGroupID.x;
10     gl_TaskCountNV = 1;
11 }
```

**Listing 2.** Przykład najprostszego shadera zadań.

### 3.4. Shader siatki w literaturze

Potok shaderów siatki jest obecnie dostępny w OpenGL wyłącznie pod postacią specyficznego dla firmy NVIDIA rozszerzenia tego standardu [55]. Według materiałów prasowych [56] potok ten ma być również wspierany przez mikroarchitekturę RDNA 2 firmy AMD, na której bazują układy graficzne najnowszych konsol do gier: Playstation 5 i Xbox Series S oraz X. Spodziewam się, że zainteresowanie tą technologią

wśród programistów oraz naukowców wzrośnie wówczas, gdy będzie ona dostępna w większym stopniu.

Ponieważ jest to relatywnie nowa technologia, w literaturze istnieje ograniczona liczba odnoszących się do niej publikacji. Pod tym względem rozważania zawarte w Rozdziale 7 można uznać za jedno z pierwszych zastosowań shaderów siatki w badaniach naukowych.

Wśród dostępnych materiałów znaleźć można sposób zastąpienia tym potokiem standardowej procedury teselacji trójwymiarowych terenów [57], implementacji wspieranego sprzętowo obcinania dla uzyskania płynnego poziomu szczegółowości (*Level of Detail*) [58], czy implementację testu widoczności dla oteksturowanych i animowanych szkieletowo brył [59]. Potok ten wykorzystany został również jako jeden ze sposobów wizualizacji danych pochodzących z traktografii istoty białej mózgu [1] (podobna praca wykorzystuje do tej wizualizacji shadery geometrii [46] — Rozdział 3.2).

# 4. Wizualizacja pól wektorowych w literaturze

Techniki wizualizacji pól wektorowych rozwijane są od wielu lat i posiadają bogaty zbiór artykułów naukowych oraz prac podsumowujących obecny stan wiedzy [4], [5], [60]–[66]. Były one pomocne w procesie powstawania niniejszego przeglądu literatury. Byłem jednak świadomy, że prace zbiorcze nie będą zawierać informacji o najnowszych osiągnięciach naukowych. Dlatego, o ile było to możliwe, starałem się uwzględnić w tej części rozprawy również nowsze, własnoręcznie wyszukane artykuły.

## 4.1. Metody wizualizacji

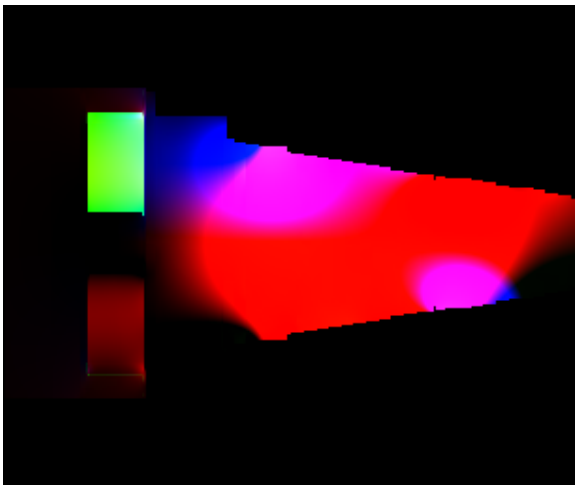
Społeczność naukowa [4] podzieliła metody wizualizacji pól wektorowych na cztery główne grupy: bezpośrednie, oparte o teksturę, oparte o cechy wizualizowanego pola oraz geometryczne.

### 4.1.1. Bezpośrednie

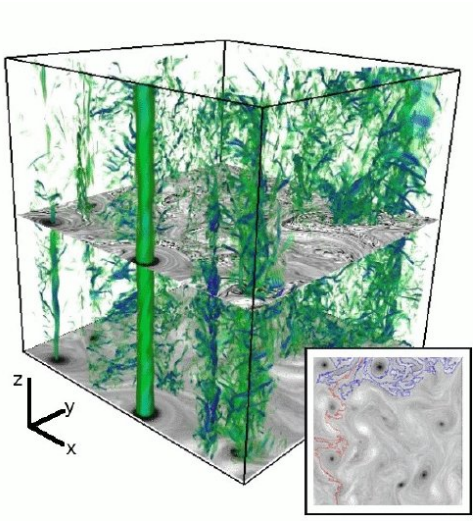
Metody bezpośrednie polegają na umieszczeniu w wybranych punktach wizualizowanej przestrzeni pewnego rodzaju znacznika.

Do najprostszych metod bezpośrednich do wizualizacji dwuwymiarowych pól skalarnych zaliczyć można kodowanie kolorem, gdzie pole przedstawione jest jako obraz złożony z czarno-białych lub kolorowych pikseli. Barwie każdego piksela przypisana jest pewna cecha danego pola. W sposób trywialny metodę tą można rozszerzyć do dwuwymiarowych pól wektorowych, mając na uwadze fakt, że kolor danego piksela ma zazwyczaj trzy składowe (czerwony, zielony, niebieski lub cyjan, magenta, żółć itd.), co pozwala przestawić w ten sposób wektory o maksymalnie 3 elementach [67] (Rysunek 4.1a). Zastosowanie składowych (barwa, nasycenie, jasność) pozwala na wyeksponowanie, za pomocą np. kontroli jasności, długości wektorów pola. Rozszerzenie tej metody do przestrzeni trójwymiarowej to technika *volume rendering* [68], [69] (Rysunek 4.1b). Polega ona na przypisaniu punktom w przestrzeni współczynników emisji oraz absorpcji światła. Są one następnie wykorzystywane w symulacji po to, aby uzyskać ostateczny obraz [68], [70]–[72]. Przykładem zastosowania tej metody w praktyce jest termografia trójwymiarowa [73]. Metodą pokrewną do *volume renderingu*

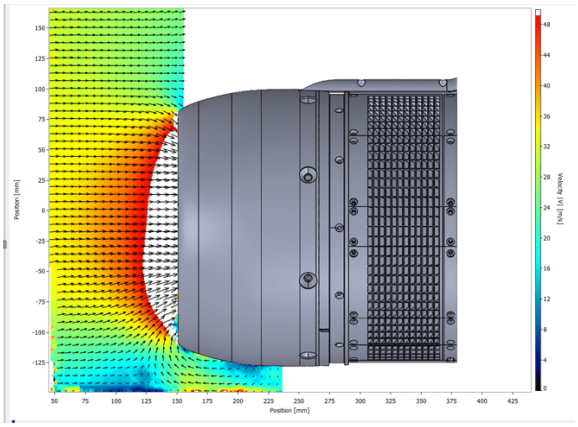
jest wizualizacja izopowierzchni, działająca na podstawie map odelgłości [74]. Kolejną wartą uwagi pracą jest [75], gdzie w ramach rozszerzonej rzeczywistości na obraz wideo rejestrujący cewkę na stole laboratoryjnym nakładana jest wizualizacja pola elektromagnetycznego otrzymanego w wyniku symulacji jej modelu.



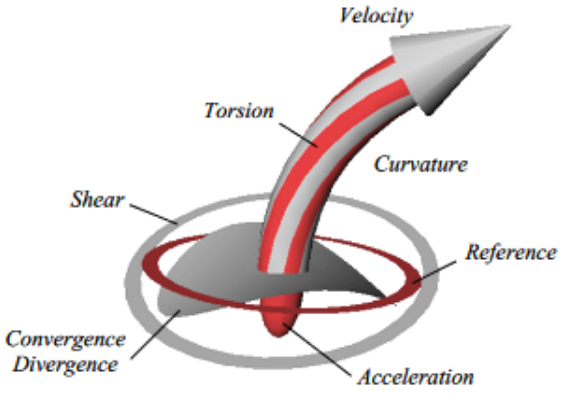
**(a)** Składowe wektorów jako kolor (Rozdział 5).




**(b)** *Volume rendering* [69].



**(c)** Znaczniki (strzałki) [2].




**(d)** Sonda [76].

**Rys. 4.1.** Przykłady wizualizacji wykorzystujących metody bezpośrednie.

Innym rodzajem metody bezpośredniej jest zastosowanie znaczników, np. strzałek, reprezentujących wektor pola w wybranych punktach przestrzeni. Przykładem zastosowania tej metody jest oprogramowanie do symulacji oraz wizualizacji przepływu powietrza przez silnik lotniczy [2] (Rysunek 4.1c). Dominującą rolę w zapewnieniu czytelności tego rodzaju wizualizacji ma rozmieszczenie rysowanych znaczników, dlatego najwięcej badań przeprowadzonych zostało w tym kierunku. Prace [77]–[79] pro-

ponują rozwiązania, które minimalizują obecność pustych przestrzeni bez znaczników jednocześnie unikając nakładania się na siebie symboli. Naturalnym rozszerzeniem tej metody jest zastosowanie znaczników przekazujących większą ilość informacji niż tylko kierunek wektorów pola, czyli tak zwanych sond. Tam, gdzie znacznik ma zostać umieszczony, pole badane jest pod kątem rotacji, dywergencji, skręcania, krzywizny itp., a następnie przygotowywany jest trójwymiarowy model przedstawiający te informacje [76], [80] (Rysunek 4.1d).

### 4.1.2. Oparte o teksturę

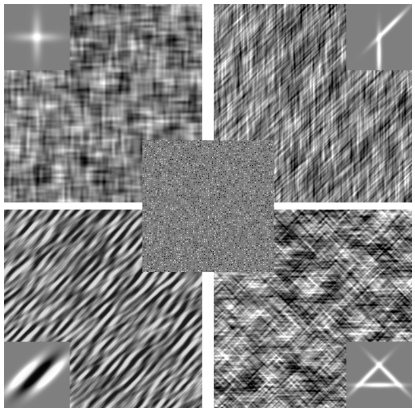
Wynikiem działania metod opartych o teksturę jest powstanie płaskiego obrazu lub tekstury na powierzchni obiektu trójwymiarowego, która została wygenerowana na podstawie informacji o kierunku wektorów wizualizowanego pola.

Jedną z pierwszych metod realizujących ten rodzaj wizualizacji jest szum kropkowy [81]. Polega ona na wielokrotnym nakładaniu na siebie kropek (tj. niewielkich obrazków przedstawiających np. figury geometryczne) dopasowując ich orientację oraz rozmiar tak, aby odpowiadały wektorom wizualizowanego pola. W późniejszych pracach rozwinięto tą metodę w kierunku stosowania kilku rodzajów kropek jednocześnie [82] (Rysunek 4.2a).

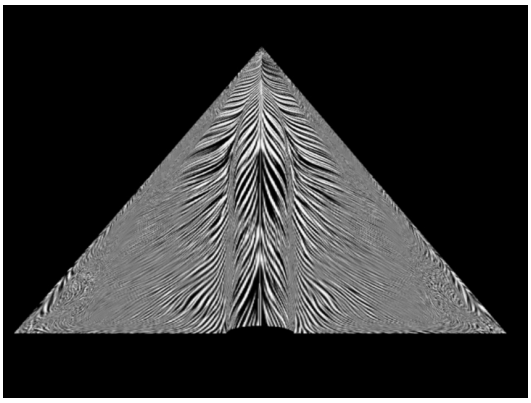
Alternatywną metodą, która zyskała dość dużą popularność (wnioskując na podstawie mnogości prac ją rozszerzających [5]), jest *Line Integral Convolution*. Została ona zaprezentowana po raz pierwszy w [83]. Polega ona na inicjalizacji tekstury pewnego rodzaju szumem, którego wartości w poszczególnych pikselach obrazu przetwarzane są za pomocą pewnego filtra konwolucyjnego wzdłuż kierunków wektorów pola. Daje to efekt wybiórczego "rozmazania" obrazu, który uwidacznia linie wizualizowanego pola. Praca ta dała początek całej rodzinie pokrewnych algorytmów [5].

Obrazy uzyskane oryginalną metodą pokazują jedynie kierunek, ale nie zwrot wektorów. Rozwiązanie tego problemu zostało zaproponowane w *Oriented LIC* (OLIC) [84], który wykorzystuje rzadszą teksturę szumu oraz anizotropowe jądro splotu do uzyskania tego efektu. Aby poprawić wydajność tego rozwiązania powstał *Fast Rendering*

*of OLIC* (FROLIC) [85], który aproksymuje kosztowny obliczeniowo splot za pomocą serii zanikających dysków.

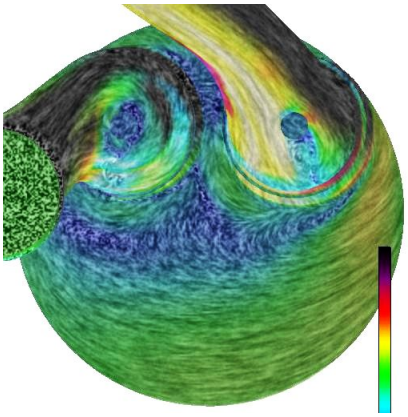


**(a)** Szum kropkowy [82].

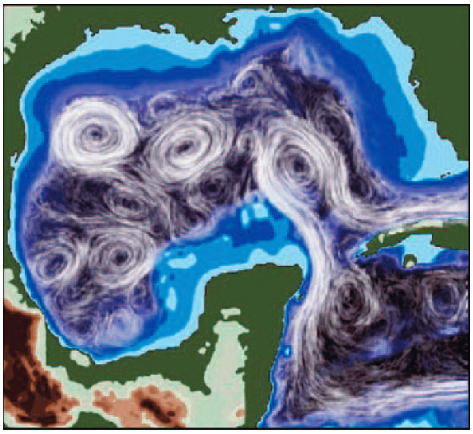


**(b)** *Unsteady Flow LIC* [86].

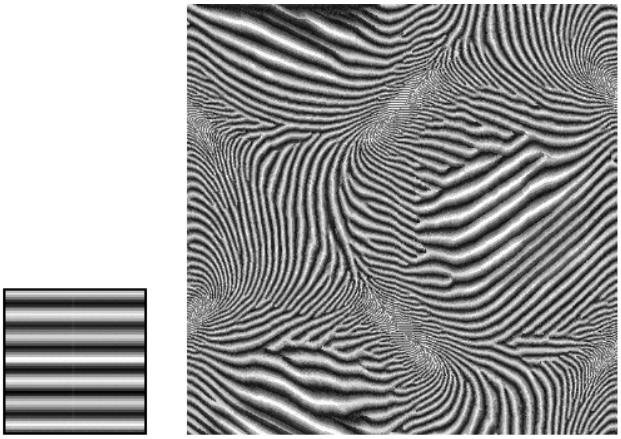
**(c)** *Dye injection* [5].



**(d)** *Texture advection* [87].



**(e)** *Lagrangian-Eulerian Advection* [88].



**(f)** *Markov Random Field* [89].

**Rys. 4.2.** Przykłady wizualizacji wykorzystujących metody oparte o teksturę.

Algorytm *Unsteady Flow LIC* (UFLIC) [86] (Rysunek 4.2b) lepiej radzi sobie z dynamicznie zmieniającymi się polami ze względu na ulepszony filtr konwolucyjny. Podobnie jak w przypadku OLIC, późniejsza praca przedstawiła wersję wydajniejszą obliczeniowo, AUFLIC [90]. Praca GPULIC [91] wykorzystuje OpenGL oraz procesowanie na karcie graficznej po to, aby dodatkowo przyspieszyć działanie algorytmu LIC.

Innym sposobem na wizualizację pola, które zachowuje informacje o kierunku jest algorytm *Dye injection* [92] (Rysunek 4.2c), który polega na symulacji oraz na animacji rozchodzenia się kolorowych plam barwnika zgodnie z kierunkiem pola.

Drugą rodziną algorytmów są techniki unoszenia tekstur (*texture advection*) [87], [93] (Rysunek 4.2d), w których bazowa tekstura zniekształcana jest zgodnie z kierunkami w polu. Istotnym faktem odróżniającym je od pozostałych metod jest fakt, że działają one wstecz — kolor danego piksela określany jest na podstawie obliczeń, który inny piksel (po przesunięciu) znalazłby się na obecnie rozważanej pozycji. Podyktowane jest to względami praktycznymi — symulacja w przód, w zależności od wizualizowanego pola, mogłaby powodować opuszczanie obrazu przez piksele, zostawiając "dziury".

Metoda *Lagrangian-Eulerian Advection* (LEA) [88], [94], [95] (Rysunek 4.2e) jest hybrydą podejścia unoszenia tekstur oraz symulacji ruchu cząstek. Algorytm w pętli inicjalizuje pozycję i kolor cząstek na podstawie danych obecnie przetwarzanego obrazu, które następnie są przesuwane, a z danych o cząstkach tworzony jest nowy obraz.

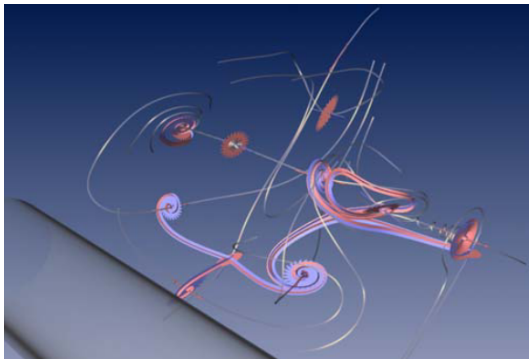
Technika *Image Based Flow Visualization* (IBFV) [96], [97] jest rodzajem techniki unoszenia działającej w przód — problem "znikających" pikseli obrazu rozwiązany jest za pomocą operacji mieszania kilku tekstur poddawanych unoszeniu ze sobą. "Zużyte" tekstury zastępowane są nowymi po to, aby zapewnić ciągłość wizualizacji. Metoda operuje na teksturach generowanych szumem.

Do syntezy tekstur wykorzystuje się również algorytmy uczenia maszynowego [98], [99]. Jednym z nich jest technika *Markov Random Field* (MRF), która pozwala na generowanie obrazów podobnych do tego dostarczonego przez użytkownika jako przykład. W kontekście wizualizacji pól wektorowych w tym temacie powstała praca [89], gdzie

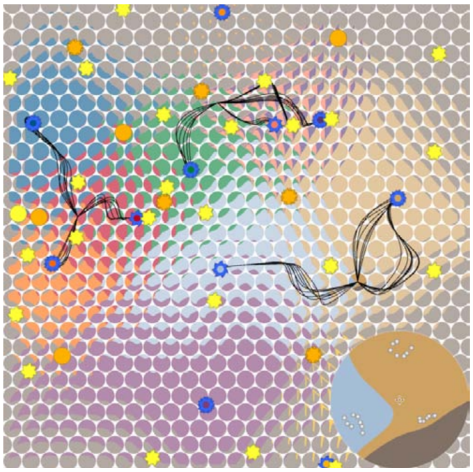
algorytm MRF został zmodyfikowany w taki sposób, aby generowane tekstury wykazywały podobieństwo zarówno do przykładowego obrazu, jak również do charakterystyki (kierunków, zwrotów itp. wektorów) wizualizowanego pola (Rysunek 4.2f).

### 4.1.3. Oparte o właściwości pola

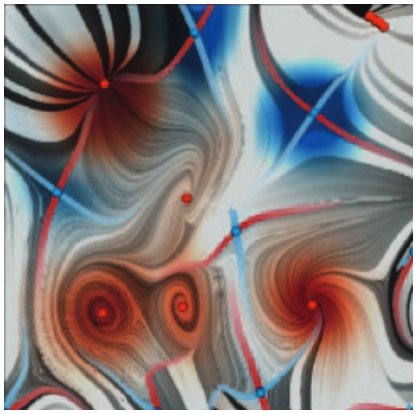
Metody wizualizujące właściwości pola polegają na wykorzystaniu metod topologicznych [100] do znalezienia *szkieletu topologicznego* pola, czyli punktów krytycznych (miejsc gdzie pole ma wartość zerową — są to źródła, wycieki, siodła itp.) oraz linii lub powierzchni oddzielających pewne cechy pola, na przykład prądy wirowe [101].



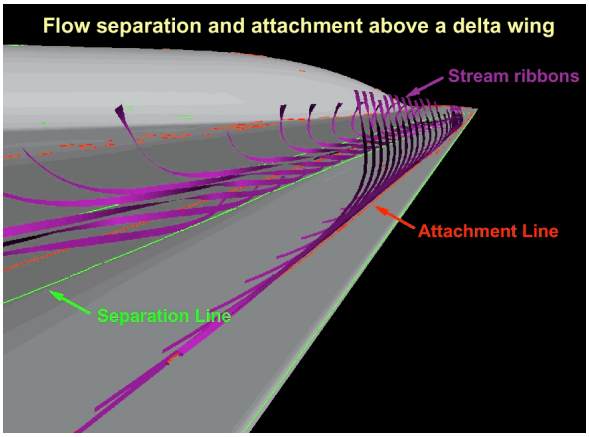
**(a)** Połączenia siodeł [102].



**(b)** Partycjonowanie [103].



**(c)** Niezmiennicze punkty krytyczne [104].




**(d)** Linie odrywania i przyłączania [105].

**Rys. 4.3.** Przykłady wizualizacji wykorzystujących metody oparte o właściwości pola.

W różnych dziedzinach nauki istotne są różne właściwości pól. Z tego powodu w tej

grupie metod spotyka się prace zarówno dziedzinowe jak i takie opisujące rozwiązania uniwersalne.

Do artykułów przedstawiających metody ogólne zaliczyć można [102] (Rysunek 4.3a), gdzie w zaproponowanym rozwiązaniu wykryte w danym polu siodła łączy się ze sobą za pomocą linii przepływu.

Artykuł [103] (Rysunek 4.3b) przedstawia partycjonowanie przestrzeni pola przez wizualizację obszarów wpływu punktów krytycznych pola (czyli takich obszarów, gdzie oddziaływanie danego punktu jest najsilniejsze).

Klasyczne metody wyszukiwania punktów krytycznych dla uzyskania zadowalającego rezultatu wizualizacji wymagają odpowiedniego zdefiniowana przyjętego układu odniesienia, ponieważ silnie od niego zależą. Praca [104] (Rysunek 4.3c) przedstawia sposób szukania niezmienniczych punktów krytycznych (*Galilean invariant critical point*), które nie zależą od wybranego układu odniesienia.

W zastosowaniach lotniczych przy tworzeniu elementów samolotów ważne jest uwidocznienie na symulacjach obszarów, w których prądy powietrza odrywają lub przyłączają się do projektowanych powierzchni. Detekcją tych punktów w modelach pól przepływu powietrza zajmuje się praca [105] (Rysunek 4.3d).

Jednym z celów badań meteorologicznych jest zrozumienie zjawisk, które doprowadzają do powstawania tornad. Dlatego w tej dziedzinie ważne jest śledzenie oraz wizualizacja prądów wirowych mas powietrza, np. nad Oceanem Atlantyckim [106].

W pracy [107] wizualizacja ruchu powietrza połączona jest z wizualizacją jego temperatury po to, aby uwidocznić sposób rozchodzenia się skupisk pyłu wulkanicznego z miejsca erupcji.

Najnowsze prace w zakresie wizualizacji cech pola prowadzone są w kierunku zastosowania sztucznej inteligencji do wyszukiwania interesujących obszarów [108]–[110].

### 4.1.4. Geometryczne

Metody geometryczne polegają w ogólności na rozrzuceniu po przestrzeni ziaren (punktów startowych) [111] od których, próbkując wizualizowane pole, konstruuje się obiekty geometryczne reprezentujące jego zachowanie. Metody geometryczne używane

są do wizualizacji linii przepływu (*streamlines*) — linii zakreślanych przez punkty w stałym polu (Rysunek 4.4f), linii ścieżki (*pathlines*) — linii zakreślanych przez punkty w zmiennym polu, smug (*streaklines*) — linii zakreślanych przez zbiór punktów oddalonych o określoną odległość od siebie symulowanych jednocześnie oraz linii czasu (*timelines*) — linii zakreślanych przez zbiór punktów symulowanych z jednego punktu startowego w odstępach czasu [112]. Tak uzyskane dane wejściowe można wizualizować bezpośrednio jako linie [113], ale również w postaci: a) wstążek przedstawiających użytkownikowi dodatkowe informacje jak skrętność czy rozchodzenie się linii przepływu [114]; b) tub przedstawiających skrętność oraz siłę [115]; c) powierzchnie przedstawiające ogólny kształt przepływu [116]. Rozwinięciem koncepcji tub są komety [117], które za pomocą strzałek przedstawiają dodatkowo kierunek przepływu. Do metod geometrycznych zalicza się również symulację dryfu bezmasowych cząsteczek w polu [118].

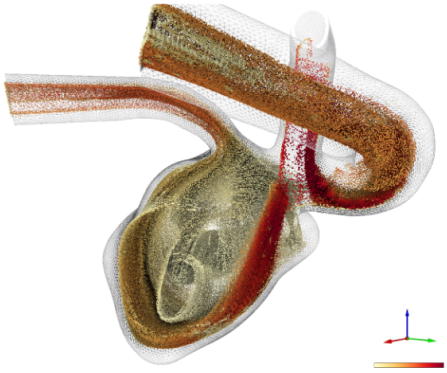
Przykładem pracy wykorzystującej wizualizację linii jest program *PedVis*, który wizualizuje, w jaki sposób grupy pasażerów poruszają się pomiędzy wejściami/wyjściami, przejściami podziemnymi, wiaduktami oraz peronami na dworcach kolejowych [119].

Artykuł [120] wykorzystuje wstążki jako sposób wizualizacji różnic pomiędzy liniami przepływu wygenerowanymi różnymi technikami obliczeń. Praca [114] (Rysunek 4.4b) wykorzystuje je jako sposób uproszczenia reprezentacji pola dla zwiększenia czytelności obrazu, kiedy konieczne jest wzięcie pod uwagę dużej liczby linii przepływu.

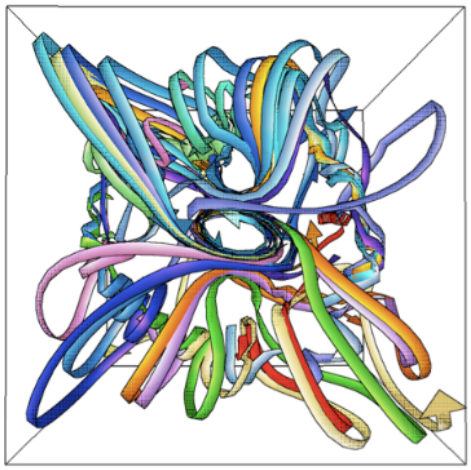
Artykuł [115] (Rysunek 4.4c) łączy wizualizację tub oraz technikę *ray tracing* do stworzenia wysokiej jakości reprezentacji włókien nerwowych tworzących mózg człowieka.

Praca [117] (Rysunek 4.4d) wykorzystuje komety do wizualizacji przepływu krwi przez patologicznie rozszerzone naczynia krwionośne (tętniaki).

Artykuł [116] (Rysunek 4.4e) przedstawia narzędzie pozwalające na optymalizację kształtu płaszcza wodnego silników spalinowych wykorzystując powierzchnie przepływu do wizualizacji zachowania płynnego chłodziwa wewnątrz tego elementu. Metoda powierzchni wykorzystana została również w [121] do wizualizacji struktur mózgu na podstawie danych rezonansu magnetycznego z kontrastem.

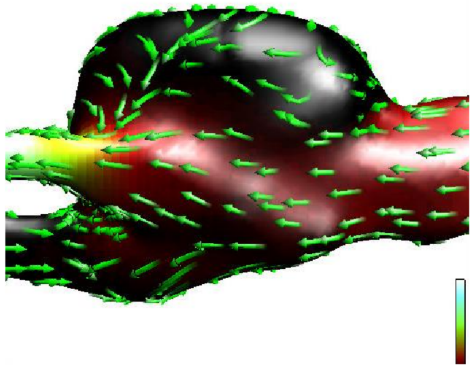


**(a)** Cząsteczki [118].

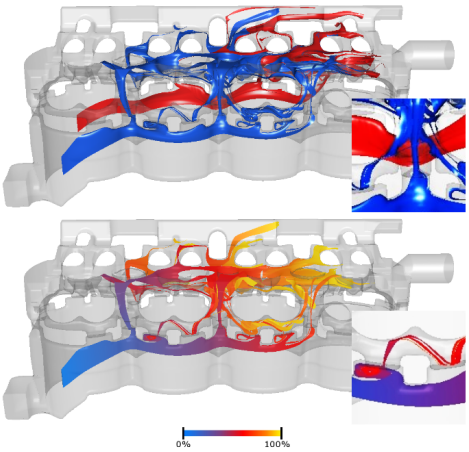


**(b)** Wstążki [114].

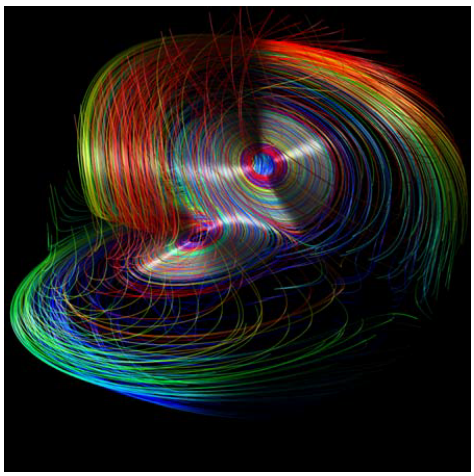
**(c)** Tuby [115].



**(d)** Komety [117].




**(e)** Powierzchnie [116].



**(f)** Linie przepływu [113].

**Rys. 4.4.** Przykłady wizualizacji wykorzystujących metody geometryczne.

Praca [118] (Rysunek 4.4a) wykorzystuje symulację punktów na przykładzie wi-

zualizacji przepływu cieczy wokół przeszkody oraz przepływu krwi wewnątrz naczyń krwionośnych. Autorzy przedstawili w niej algorytm usuwania punktów, które przestały się poruszać oraz strategię rozmieszczania nowych cząstek. Pozwala to na stworzenie wysokiej jakości animacji przepływu.

Praca [122] skupia się na problemach czytelności wizualizacji wykorzystujących linie przepływu. Jako możliwe rozwiązania zaproponowano tu wybór zbioru linii przepływu do wizualizacji względem hierarchii ich ważności dla użytkownika, metodę wokselizacji linii przepływu oraz ich wizualizację za pomocą techniki *ray tracing*, a ostatecznie wizualizację gęstości liczby linii przepływu za pomocą klastrowania.

## 4.2. Podsumowanie

Każda z przedstawionych grup metod wizualizacji ma swoje wady i zalety [4]. Metody bezpośrednie cechują się niską złożonością obliczeniową (co przekłada się na wysoką wydajność), ale wiążą się z problemem wzajemnego zasłaniania się znaczników i trudnością interpretacji złożonych pól.

Zaletą metod opartych o teksturę jest ich rozdzielczość pozwalająca na podkreślenie interesujących cech pola, takich jak źródła czy wycieki. Problematyczne natomiast jest zasłanianie pewnych cech pola przez inne, szczególnie przy próbach ich zastosowania w przestrzeni trójwymiarowej.

Metody oparte o cechy pola zwiększają czytelność wizualizacji dzięki wyświetlaniu jedynie pewnego ograniczonego zbioru elementów graficznych. Ich wadą jest duża złożoność obliczeniowa potrzebna do wyszukiwania szkieletu topologicznego. Dodatkowo nie każdy znaleziony element topologiczny jest istotny dla wizualizacji — część to szum pogarszający czytelność. Użytkownik musi posiadać wiedzę ekspercką po to, aby elementy nieistotne usunąć i uzyskać zadowalający efekt.

Zaletą metod geometrycznych jest możliwość ich wykorzystania zarówno w dwóch jak i trzech wymiarach oraz dla pól stałych i zmiennych. Wadą jest problem zasłaniania oraz konieczność odpowiedniego rozmieszczenie elementów po to, aby uzyskać dobrej jakości wizualizację przedstawiającą wszystkie istotne cechy pola wektorowego.

Tak jak wspomniałem już w Rozdziale 3, żadna z omówionych tutaj prac nie poru-

sza tematu wykorzystania shaderów siatki do wizualizacji pól wektorowych. W ramach prób opracowania wydajniejszych algorytmów reprezentacji pól ten temat podejmuje niniejsza praca doktorska. Ze względu na to, że shadery siatki przeznaczone są m.in. do generowania geometrii, to właśnie ten rodzaj wizualizacji został w niej przebadany, czerpiąc inspirację z przedstawionych w tym Rozdziale prac.

# 5. Dostęp do próbek pola wektorowego z poziomu pamięci karty graficznej

Funkcjonowanie algorytmów wizualizacji na karcie graficznej uwarunkowane jest możliwością dostępu do danych o polu wektorowym z poziomu kodu shaderów. Aby umożliwić badania nad wizualizacjami, problem tego typu musi zostać usunięty w pierwszej kolejności. Rozdział ten przedstawia zaproponowane przeze mnie metody jego rozwiązania. Zostały one przetestowane na przykładzie modelu pola magnetycznego detektora ALICE. Wyniki badań (wraz z testami wydajnościowymi oraz jakościowymi) zostały opublikowane w czasopiśmie *Computer Physics Communications* [11]. Kod stworzonego oprogramowania dostępny jest w serwisie GitHub [123].

## 5.1. Rozważane metody

Oprogramowanie $O^2$ zawiera algorytm dostępu do pełnego modelu pola magnetycznego na potrzeby przetwarzania danych detektora (opisany szczegółowo w Rozdziale 2.5). Jego implementacja przystosowana jest jednak wyłącznie do działania na procesorze komputera. Alternatywnie, istniejące w $O^2$ narzędzie do wizualizacji danych detektora Event Display [124] wykorzystuje najprostszy możliwy sposób reprezentacji pola detektora — idealnie jednorodne pole elektromagnesu L3 o sile **0.5 T** w objętości *Central Barrel*, nieuwzględniające wpływu elektromagnesu dipolowego. Te dwie (skrajne pod względem dokładności reprezentacji) metody obsługi informacji o polu posłużyły mi za inspirację przy opracowaniu algorytmów działających na karcie graficznej, które zostały opisane w poniższych podrozdziałach.

### 5.1.1. Stałe pole

Pierwsza zaimplementowana przeze mnie metoda (oparta o sposób opisany wyżej) zakłada obecność idealnego, stałego pola magnetycznego o sile **0.5 T** (Rozdział 2). Takie pole zgodne jest z rzeczywistością jedynie w objętości *Central Barrel*. Warunek ten sprawdzany jest przez kod shadera — jeżeli żądany punkt znajduje się na zewnątrz, zwracane jest zero.

```
const float SOL_MIN_Z = -550.f;
const float SOL_MAX_Z = 550.f;
const float SOL_MAX_R = 500.f;

vec3 CarttoCyl(vec3 pos) {
    return vec3(length(pos.xy), atan(pos.y, pos.x), pos.z);
}

vec3 Field(vec3 pos) {
    vec3 cyl = CarttoCyl(pos);
    if(cyl.z > SOL_MIN_Z && cyl.z < SOL_MAX_Z && cyl.x < SOL_MAX_R) {
        return vec3(0.0f, 0.0f, -5.0f);
    }

    return vec3(0.0f);
}
```

**Listing 3.** Model stałego pola w postaci kodu shadera.

Ta implementacja posłużyła przede wszystkim za punkt odniesienia przy interpretacji wyników pozostałych metod.

### 5.1.2. Shader Storage Buffer Object

Ze względu na to, że rozdzielczość oryginalnego modelu nie jest nieograniczona i wynosi 1 cm w każdej z osi (Rozdział 2.5), najmniej skomplikowanym rozwiązaniem byłoby przechowywanie wszystkich próbek pola (w postaci trójelementowych wektorów) na karcie graficznej w postaci tablicy. Nie jest to jednak możliwe ze względu na ilość wykorzystanej pamięci — zapisanie w ten sposób informacji z całej objętości detektora ALICE to ok. 2700 milionów wektorów zmiennoprzecinkowych (12 B na wektor), co daje w sumie ok. 32 GiB danych. Taką ilość pamięci RAM posiadają obecnie jedynie karty graficzne do zastosowań profesjonalnych, co mocno ograniczyłoby potencjalny zbiór odbiorców mojego rozwiązania.

Aby mimo wszystko być w stanie przetestować tą opcję, postanowiłem zmniejszyć

zapotrzebowanie na RAM karty poprzez zapisywanie jedynie co czwartej próbki w każdej z osi. W takiej sytuacji pamięci potrzeba 64 razy mniej, czyli ok. 500 MiB. Taki krok potencjalnie pogarsza wierność reprezentacji pola w porównaniu z oryginalnym modelem, dlatego część finalnych testów poświęcona jest pomiarom tej różnicy.

W OpenGL 4.6 istnieją dwa sposoby na udostępnienie shaderom danych o strukturze ustalonej przez programistę:

- `Uniform Buffer`,
- `Shader Storage Buffer`.

Potencjalnie szybszy bufor `Uniform` nie mógł zostać wykorzystany ze względu na ograniczenia rozmiaru pamięci — wg. specyfikacji może on przechowywać maksymalnie 16 KiB. `Shader Storage Buffer` wg. specyfikacji musi być w stanie pomieścić przynajmniej 128 MiB danych, natomiast definicja górnej granicy została powierzona producentowi [125]. Na sprzętach wykorzystanych do testów sterownik karty pozwolił zaalokować wymaganą ilość pamięci.

Zastosowanie `Shader Storage Buffer` wiąże się z dodatkowymi ograniczeniami na wyrównanie adresów w pamięci narzuconymi przez OpenGL, które musiały zostać uwzględnione w projekcie. Tzw. *layout* `std430` oferuje największe możliwe upakowanie danych, ale wciąż wymaga, aby wektory wyrównane były do 16 B. Oznacza to konieczność zastosowania wektorów czteroelementowych i "odpakowywanie" próbek za pomocą dodatkowego kodu (spowalniającego program) albo traktowanie wektorów 4D jako 3D, poświęcając dodatkową pamięć. Zdecydowałem się na drugą opcję, co zwiększyło zapotrzebowanie z 500 MiB do ok. 656 MiB.

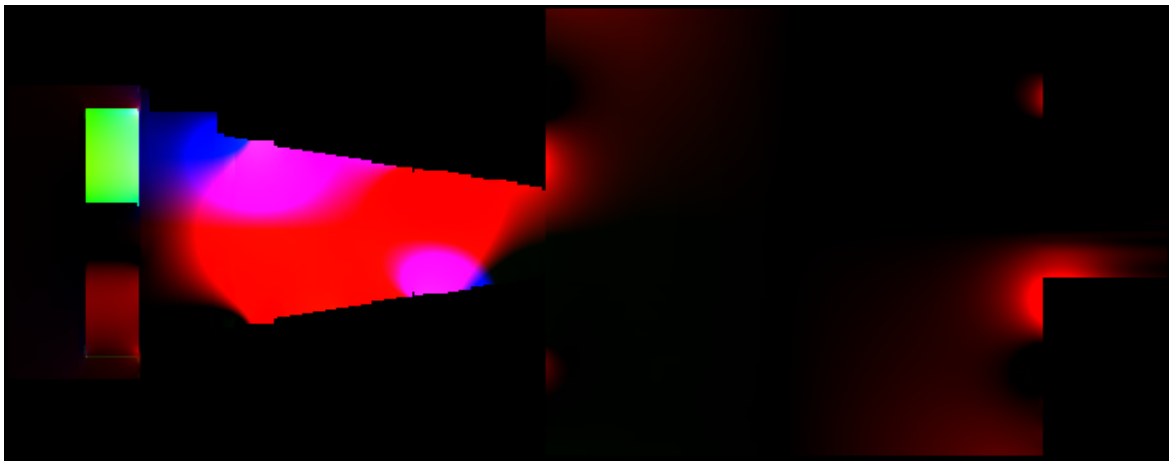
```
1 int world_to_index(vec3 position) {
2     ivec3 r = ivec3((position.zxy + offsets) / scales);
3
4     return (r.p * detector_dimensions_scaled.s *
5     detector_dimensions_scaled.t) +
6     r.t * detector_dimensions_scaled.s + r.s;
7 }
8
```

```
9  vec3 Field(vec3 pos) {
10     int index = world_to_index(pos);
11     return data[index].xyz;
12 }
```

**Listing 4.** Obsługa próbek pola w shaderze przy zastosowaniu `Shader Storage Buffer`.

Próbki pola magnetycznego `data` ułożone są w postaci trójwymiarowej tablicy spłaszczonej do jednego wymiaru. Funkcja `world_to_index()` odpowiedzialna jest za przeliczenie pozycji w układzie współrzędnych ALICE na indeks najbliższej pasującej próbki obecnej w tablicy. Należy zwrócić uwagę na to, że próbki zapisane są w sposób oszczędzający pamięć: środkiem tablicy jest geometryczny środek detektora (który nie jest symetryczny). Nie jest to zgodne z układem współrzędnych ALICE, dlatego konieczne jest przesunięcie (za pomocą wektora `offsets`). Dzielenie przez wektor `scales` wprowadza korekcję pozycji ze względu na opisany wyżej fakt czterokrotnego zmniejszenia ilości próbek w każdej z osi. Po to, aby ostatecznie wyliczyć indeks próbki, algorytm musi znać wymiary tablicy przed spłaszczeniem — zawarta jest ona w wektorze `detector_dimensions_scaled`.

### 5.1.3. Tekstura 3D



**Rys. 5.1.** Przekrój przez środek tekstury 3D zawierającej próbki pola magnetycznego.

Próbki pola magnetycznego można również przechowywać na karcie graficznej w postaci tekstury 3D, o ile dana karta graficzna pozwala na stworzenie takiej o od-

powiednim rozmiarze. Jednym z formatów pikseli oferowanym przez OpenGL jest `GL_RGB_32F`, dzięki któremu próbki pola można przechowywać w sposób upakowany, co jest pierwszą przewagą tej metody nad metodą z buforem. Metoda wykorzystująca teksturę może być również wydajniejsza ze względu na jej sprzętową obsługę przez dedykowane układy na czipie karty (*texture unit*). Dodatkową zaletą jest wbudowana interpolacja liniowa, poprawiająca reprezentację pola pogorszoną omijaniem próbek. Przekrój przez środek tekstury 3D przedstawia Rysunek 5.1 (próbki pola potraktowane są jako kolor pikseli).

Punkt w przestrzeni w układzie współrzędnych ALICE można w prosty sposób przekalkulować na współrzędne tekstury $u$, $v$, $w$, jeżeli wykonamy ten sam krok translacji i skalowania co w poprzedniej metodzie, a następnie podzielimy tak skorygowaną pozycję przez rozmiar tekstury w pikselach (równy wymiarom tablicy 3D poprzedniej metody) przechowywany w wektorze `detector_dimensions_scaled`. Aby karta graficzna zwróciła nieprzetworzone dane, teksturę próbkujemy bez mipmappingu przy pomocy funkcji `textureLod()`.

```
1 vec3 world_to_uvw(vec3 position) {
2     const vec3 pos = (position.zxy + offsets) / scales;
3     return pos / detector_dimensions_scaled;
4 }
5
6 vec3 Field(vec3 pos) {
7     vec3 uv = world_to_uvw(pos);
8     return textureLod(fieldTexture, uv, 0).xyz;
9 }
```

**Listing 5.** Obsługa próbek pola w shaderze przy zastosowaniu tekstury 3D.

### 5.1.4. Rzadka tekstura 3D

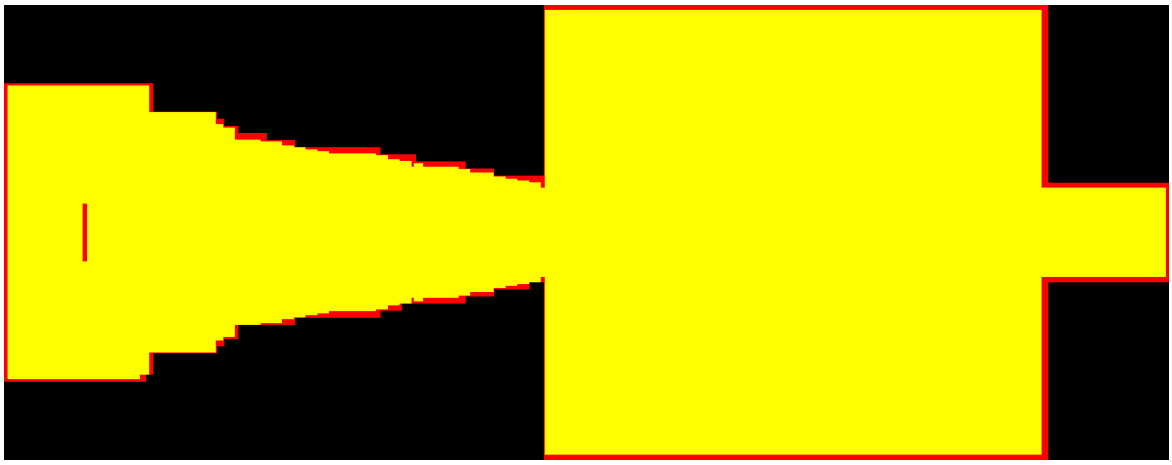
Dużą część omówionej wyżej tekstury zajmują próbki o wartości zero, co widoczne jest na Rysunku 5.1, gdzie odwzorowane są one za pomocą czarnego koloru. Zapotrzebowanie na pamięć karty poprzedniej metody spadłoby drastycznie, gdyby możliwe

było pominięcie ich zapisywania. W OpenGL możliwe jest to przy zastosowaniu tekstury rzadkiej, *Sparse Texture*. Zastosowanie tego obiektu pozwala na rozdzielenie adresacji pamięci karty (co przekłada się na rozmiar tekstury) od faktycznej alokacji dostępnych zasobów [40]. Operacje zapisu do niezaalokowanych fragmentów tekstury rzadkiej są zawsze ignorowane. Jeżeli karta graficzna wspiera rozszerzenie OpenGL `ARB_sparse_texture2`, to operacje odczytu niezaalokowanych fragmentów tekstury rzadkiej zwracają zero — jest to pożądana cecha (brak pola to wektor zerowy), którą wykorzystałem.

Kontrola programisty nad tym, które fragmenty tektury są zaalokowane, a które nie, ma pewną minimalną rozdzielczość. Zależy ona od modelu karty graficznej. Na sprzęcie testowym dla tekstury 3D był to blok 16 na 16 na 16 pikseli. Aby wykorzystać teksturę rzadką musiałem więc przygotować algorytm, który przekalkuluje listę segmentów tworzącą model pola (Rozdział 2.5) na listę bloków, które powinny być zaalokowane w teksturze.

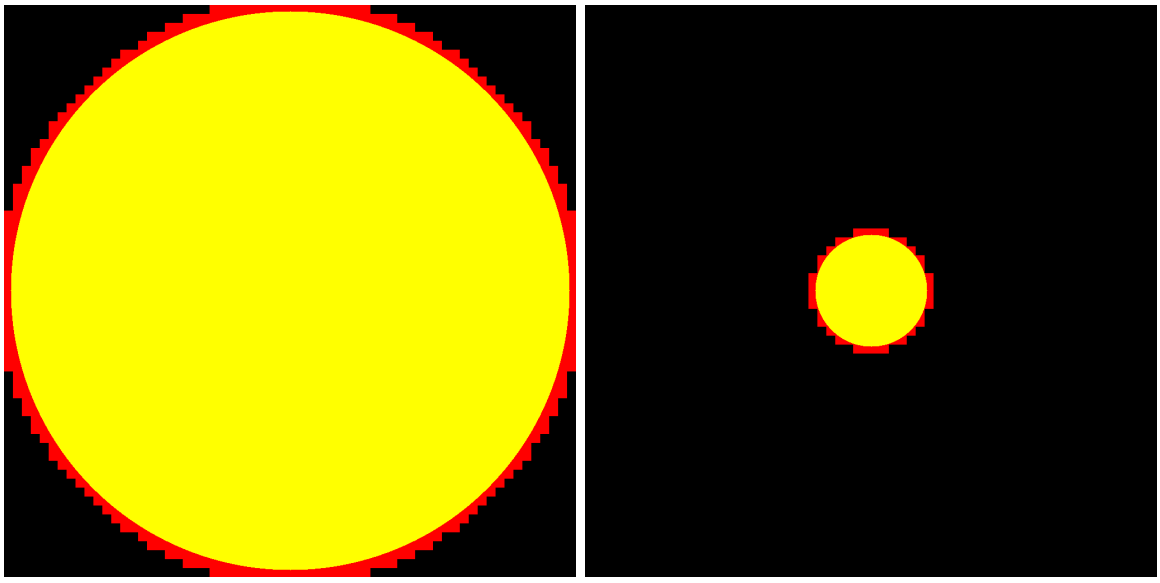
Dla elektromagnesu dipolowego lista segmentów mogła zostać przetworzona bezpośrednio, ponieważ składa się z prostopadłościanów. Algorytm dla każdego segmentu pobiera pozycję jego lewego dolnego rogu oraz prawego górnego rogu, koryguje je, aby wyrównane były do wielokrotności 16 (współrzędne dolnego rogu w dół, współrzędne górnego rogu w górę). Następnie alokuje on wszystkie bloki tekstury znajdujące się wewnątrz tych współrzędnych. Zaokrąglanie współrzędnych w omówiony sposób gwarantuje, że alokowany fragment tekstury będzie równy albo większy od analizowanego segmentu (i nie zostaną utracone żadne dane).

Dla elektromagnesu solenoidalnego (o kształcie cylindra) rozwiązaniem prostszym od procesowania segmentów było zastosowanie algorytmu Bresenhama (rasteryzacji okręgów), za pomocą którego wygenerowany został odpowiedni kształt (lista bloków do alokacji).



**Rys. 5.2.** Wizualizacja przekroju przez teksturę 3D pokazująca, które obszary tekstury zostały zaalokowane w pamięci karty graficznej.

Rysunek 5.2 przedstawia ten sam przekrój, co Rysunek 5.1. Kolorem czerwonym zaznaczony jest obszar, gdzie zaalokowana jest tekstura. Kolorem żółtym zaznaczony jest obszar, gdzie w modelu istnieje niezerowe pole (i gdzie tekstura jest również zaalokowana). Rysunek 5.3 przedstawia przekrój przez *Central Barrel* oraz przekrój przez fragment rury LHC (widoczny na Rysunku 5.2 po prawej stronie), które różnią się tylko średnicą.



**Rys. 5.3.** Wizualizacja przekrojów przez teksturę 3D pokazujących, które obszary tekstury zostały zaalokowane w pamięci karty graficznej.

W wyniku działania mojego algorytmu powstaje lista bloków, która ciasno przylega

do kształtu detektora, poświęcając minimalną możliwą ilość pamięci karty. Według moich obliczeń zastosowanie tekstury rzadkiej pozwoliło na redukcję zajmowanej przestrzeni o ok. 49%, czyli do ok. 250 MiB. Ponieważ tekstura rzadka jest również teksturą, kod shadera do jej obsługi jest taki sam jak dla poprzedniej metody.

### 5.1.5. Implementacja modelu pola w języku GLSL

Jako ostatnią metodę rozważyłem reimplementację oryginalnego algorytmu (Rozdział 2.5), która mimo dużo większego stopnia skomplikowania kodu shadera (przekładającego się na niższą wydajność) ma dużo mniejsze zapotrzebowanie na pamięć (dane segmentów i współczynniki wielomianów zajmują ok. 2.5 MiB) oraz najwyższą wierność wektorów pola w stosunku do oryginalnego modelu. Oryginalny algorytm jest jednowątkowy, w związku z czym zmienna przechowująca ostatnio używany segment również jest tylko jedna. Zmuszony zostałem do opracowania innego rozwiązania, ponieważ na karcie graficznej algorytm ten wykonywany jest przez wiele wątków jednocześnie, potencjalnie pracujących na różnych segmentach.

```
1  vec3 SolDipField(vec3 pos) {
2      if(pos.z>SOL_MIN_Z) {
3          vec3 rphiz = CarttoCyl(pos);
4
5          int segID = segment_cache.SolSegCache[gl_VertexID];
6
7          if(segID >= 0 && IsInsideSol(segID, rphiz)) {
8              vec3 brphiz = EvalSol(segID, rphiz);
9              return CyltoCartCylB(rphiz, brphiz);
10         }
11
12         segID = findSolSegment(rphiz);
13         if(segID >=0 && IsInsideSol(segID, rphiz)) {
14             vec3 brphiz = EvalSol(segID, rphiz);
15             segment_cache.SolSegCache[gl_VertexID] = segID;
16             return CyltoCartCylB(rphiz, brphiz);
17         }
```

```
18      }
19
20      int segID = segment_cache.DipSegCache[gl_VertexID];
21      if(segID >= 0 && IsInsideDip(segID, pos)) {
22          return EvalDip(segID, pos);
23      }
24
25      segID = findDipSegment(pos);
26      if(segID >= 0 && IsInsideDip(segID, pos)) {
27          segment_cache.DipSegCache[gl_VertexID] = segID;
28          return EvalDip(segID, pos);
29      }
30
31      return vec3(0);
32  }
33
34  vec3 Field(vec3 pos) {
35      if(pos.z > MIN_Z && pos.z < MAX_Z) {
36          return SolDipField(pos);
37      }
38      return vec3(0);
39  }
```

**Listing 6.** Obsługa próbek pola w shaderze przy zastosowaniu implementacji modelu w GLSL.

Zamiast jednej zmiennej, ostatnio używanego segment (tj. jego indeks) może być przechowywany w tablicy i adresowany w kodzie shadera za pomocą dostarczanego przez OpenGL indeksu obecnie przetwarzanego wierzchołka `gl_VertexID` albo przetwarzanej figury `gl_PrimitiveID`. Taka tablica musi mieć wystarczający rozmiar po to, aby można było zaadresować każdy wierzchołek oraz musi być wypełniona ustaloną wartością, symbolizującą brak zapamiętanego segmentu. W moich rozważaniach wybrałem na tą wartość `-1` ze względu na to, że numery faktycznych segmentów zaczynają się od zera.

Po to, aby sprawdzić czy w warunkach pracy na karcie graficznej uzasadnione jest wprowadzanie tablicy segmentów (co zużywa dodatkową pamięć), przetestowałem również wersję algorytmu która jej nie posiada. Wersja ta za każdym razem szuka odpowiedniego segmentu.

### 5.1.6. Zużycie pamięci

Zużycie pamięci karty graficznej przez poszczególne metody przechowywania na niej danych o polu magnetycznym detektora ALICE podsumowałem w celu łatwiejszego porównania w Tabeli 5.1. W pierwszej kolumnie odnotowałem zużycie w warunkach w których metody były testowane (czyli przy zastosowanym zmniejszeniu rozmiaru danych, opisanym na początku Rozdziału) natomiast w drugiej kolumnie przedstawione jest teoretyczne zużycie pamięci bez tego kompromisu.

| | Zużycie pamięci karty graficznej | |
|---|---|---|
| Algorytm | aktualne (model zmniejszony) | teoretyczne (pełny model) |
| Stałe pole | — | — |
| Shader Storage Buffer | 656 MiB | 42 GiB |
| Tekstura 3D | 500 MiB | 32 GiB |
| Rzadka tekstura 3D | 500 MiB | 32 GiB |
| GLSL (bez pam. pod.) | — | 2.5 MiB |
| GLSL (pam. pod.) | — | 3.26 MiB* |

**Tabela 5.1.** Tabela zużycia pamięci karty graficznej przy zastosowaniu przedstawionych metod. * - wartość obliczona dla 200000 jednoczesnych ewaluacji modelu.

## 5.2. Metodologia badań

Przygotowałem dwa scenariusze testów:

- *eval*, w którym dla zbioru $N$ punktów w objętości detektora pobierany jest wektor pola magnetycznego,
- *fieldline*, w którym na zbiorze $N/100$ punktów przeprowadzana jest symulacja ich dryfu w polu (pobierany jest wektor pola, punkt jest przesuwany o jego wartość, powtórzone 100 razy).

W obu scenariuszach ilość odwołań do modelu pola jest taka sama i wynosi $N$. Algorytmy zostały przetestowane dla wartości $N$: 100, 200, 500, 1000, 2000, 5000, 10000,

20000, 50000, 100000, 200000. Ilość kroków symulacji w scenariuszu *fieldline* została dobrana arbitralnie.

Wybrałem trzy źródła punktów dla scenariuszy opisanych wyżej:

(a) wygenerowane losowo w całej objętości detektora,

(b) wygenerowane losowo w objętości elektromagnesu L3,

(c) punkty pochodzące z trajektorii cząstek uzyskane w skutek symulacji zderzeń z użyciem oprogramowania ALICE.

Ponieważ stosowanie metody z Rozdziału 5.1.1 ma sens jedynie w objętości elektromagnesu L3, dla testów wydajności został wykorzystany tylko sposób (b) generowania punktów (Rysunek 5.4 i Rysunek 5.5).

W przypadku scenariusza *eval* losowość pozycji — (a) i (b) — pozwala na zbadanie najgorszego przypadku dla działania algorytmów, ponieważ mechanizm pamięci podręcznej segmentów nie może zostać efektywnie wykorzystany. Z drugiej strony scenariusz (c) oferuje podobne do rzeczywistych warunki korzystania z modelu pola, gdzie algorytmy z pamięcią podręczną powinny działać szybciej.

W podobny sposób zachowuje się scenariusz *fieldline*. Dla testów wydajności dla tego scenariusza wykorzystałem sposób (b), aby móc przetestować wszystkie metody.

### 5.2.1. Odczyt wyników z karty graficznej

Jednym z założeń scenariuszy testowych jest to, że powinny się one kończyć w momencie, gdy dane pochodzące z obliczeń będą dostępne w głównej pamięci komputera. Przy testach algorytmu $O^2$, który wykonuje się na procesorze komputera, jest on spełniony bezpośrednio. Dla algorytmów działających na karcie graficznej potrzebne jest opracowanie sposobu pobrania wyników.

Aby tego dokonać wykorzystałem funkcję *Transform Feedback* OpenGL, która umożliwia skopiowanie wierzchołków wygenerowanych przez shader wierzchołków lub geometrii do dodatkowego bufora, którego zawartość można później odczytać. Ze względu na to, że chcę mierzyć tylko czas wykonywania się kodu przetwarzającego wierzchołki dodatkowo wyłączyłem również rasteryzację (przy pomocy wywołania funk-

cji `glEnable(GL_RASTERIZER_DISCARD)`) po to, aby pominąć (na tyle na ile jest to możliwe) przetwarzanie danych w dalszych etapach potoku graficznego.

Na potrzeby scenariusza *eval* benchmark do przetwarzania pozycji w przestrzeni użyłem shadera wierzchołków; na potrzeby scenariusza *fieldline* użyłem shadera geometrii do generowania listy punktów.

### 5.2.2. Sposób pomiaru wydajności

Przy dokonywaniu pomiarów pominięty został czas potrzebny na inicjalizację danych dla algorytmów — mierzony jest tylko czas spędzony w głównej pętli programu, podczas ewaluacji modeli. Upływ czasu mierzony jest za pomocą zegara oferującego najwyższą precyzję w bibliotece standardowej C++ (`std::chrono::high_resolution_clock`).

Dla algorytmów działających na karcie graficznej zmierzyłem czas spędzony pomiędzy dodaniem do kolejki rysowania kolejnej komendy (za pomocą funkcji `glDrawArrays()`), a momentem ukończenia kopiowania wyników do pamięci komputera (za pomocą funkcji `glGetNamedBufferSubData()`).

```
1  glBeginTransformFeedback(GL_POINTS);
2
3  auto const start = std::chrono::high_resolution_clock::now();
4
5  glDrawArrays(...);
6  glEndTransformFeedback();
7
8  glGetNamedBufferSubData(...);
9
10 auto const end = std::chrono::high_resolution_clock::now();
```

**Listing 7.** Schemat pomiaru czasu wykonania dla algorytmów na karcie graficznej.

W przypadku algorytmu $O^2$ zmierzyłem czas spędzony w pętli programu procesującej punkty w przestrzeni.

```
1  auto const start = std::chrono::high_resolution_clock::now();
2
```

```
3  for (std::size_t i = 0; i < TOTAL_SAMPLES; ++i) {
4      field->Field(points_in[i], points_out[i]);
5  }
6
7  auto const end = std::chrono::high_resolution_clock::now();
```

**Listing 8.** Schemat pomiaru czasu wykonania dla algorytmu $O^2$.

### 5.2.3. Sposób pomiaru wierności

Do oszacowania wierności uzyskanych wektorów pola na karcie graficznej w stosunku do implementacji w $O^2$ zastosowałem pomiar błędu średniokwadratowego (*Root Mean Square Error*) dla każdej osi oddzielnie.

```
1  double RMSEx = 0.0, RMSEy = 0.0, RMSEz = 0.0;
2
3  for (std::size_t i = 0; i < TOTAL_SAMPLES; ++i) {
4      glm::dvec3 v1 = mag->Field(points[i]);
5      glm::dvec3 v2 = points_result[i];
6
7      const auto diff = v1 - v2;
8
9      RMSEx += diff.x*diff.x;
10     RMSEy += diff.y*diff.y;
11     RMSEz += diff.z*diff.z;
12 }
13
14 RMSEx = glm::sqrt(RMSEx/TOTAL_SAMPLES);
15 RMSEy = glm::sqrt(RMSEy/TOTAL_SAMPLES);
16 RMSEz = glm::sqrt(RMSEz/TOTAL_SAMPLES);
```

**Listing 9.** Schemat pomiaru błędu średniokwadratowego między wynikami zaproponowanych metod i oryginalnym modelem pola w $O^2$.

W przypadku scenariusza *eval* błąd ten liczony jest na różnicy składowych wektorów pola, w związku z czym jego jednostka to kilogaus. W przypadku scenariusza

*fieldline* błąd ten liczony jest na różnicy pozycji punktów, więc jego jednostką jest centymetr.

## 5.3. Wyniki

Testy przeprowadziłem na następujących maszynach:

- ThinkPad X1 Extreme — NVIDIA GeForce GTX 1050 Ti Max-Q Design (4 GiB VRAM), Intel Core i7-8850H 2.6GHz,
- HPE ProLiant SL270s — NVIDIA Tesla K40m (12 GiB VRAM), Xeon E5-2670 v2 2.5 GHz.

Na komputerze przenośnym testy przeprowadziłem na systemie operacyjnym Ubuntu 20.04 (z zainstalowanym sterownikiem NVIDIA w wersji 460.39). Na maszynie serwerowej testy przeprowadziłem na systemie operacyjnym CentOS 6 Scientific Linux (z zainstalowanym sterownikiem NVIDIA w wersji 387.26). Przedstawione poniżej wyniki to średnie z 10 niezależnych uruchomień programu testującego.

### 5.3.1. Scenariusz *eval*

Tabele 5.2 i 5.3 przedstawiają pomiary błędu średniokwadratowego (*Root Mean Square Error*) dla maksymalnej testowanej ilości punktów pomiarowych (200000) w scenariuszu *eval*.

W przypadku losowego rozmieszczenia punktów w objętości detektora oraz dla punktów pochodzących z trajektorii cząstek prawdopodobne jest, że żądany punkt znajdzie się w obszarze zerowego pola magnetycznego, co zostanie odnotowane jako idealna zgodność z modelem referencyjnym. Sytuacja taka nie zachodzi w przypadku rozmieszczenia ograniczonego do objętości elektromagnesu solenoidalnego — z tego powodu wartość błędu dla tego testu jest w ogólności większa niż w pozostałych przypadkach.

| Algorytm | RMSE (x) | RMSE (y) | RMSE (z) |
|---|---|---|---|
| Losowe rozmieszczenie punktów (objętość detektora) | | | |
| Shader Storage | 8.80e-01 | 1.83e-01 | 2.82e+00 |
| Tekstura 3D | 1.41e-01 | 4.24e-02 | 2.77e-01 |
| Rzadka tekstura 3D | 1.41e-01 | 4.24e-02 | 2.77e-01 |
| GLSL (pam. pod.) | 3.67e-08 | 2.35e-08 | 5.38e-08 |
| GLSL (bez pam. pod.) | 3.67e-08 | 2.35e-08 | 5.38e-08 |
| Losowe rozmieszczenie punktów (objętość elektromagnesu L3) | | | |
| Shader Storage | 1.16e-01 | 1.16e-01 | 4.91e+00 |
| Tekstura 3D | 3.92e-03 | 3.97e-03 | 6.19e-03 |
| Rzadka tekstura 3D | 3.92e-03 | 3.97e-03 | 6.19e-03 |
| GLSL (bez pam. pod.) | 9.71e-05 | 7.48e-05 | 8.19e-05 |
| GLSL (pam. pod.) | 9.71e-05 | 7.49e-05 | 8.21e-05 |
| Stałe pole | 1.16e-01 | 1.16e-01 | 2.38e-01 |

**Tabela 5.2.** Wartości błędu średniokwadratowego dla maksymalnej testowanej (200000) ilości punktów pomiarowych. Wartości w kilogausach.

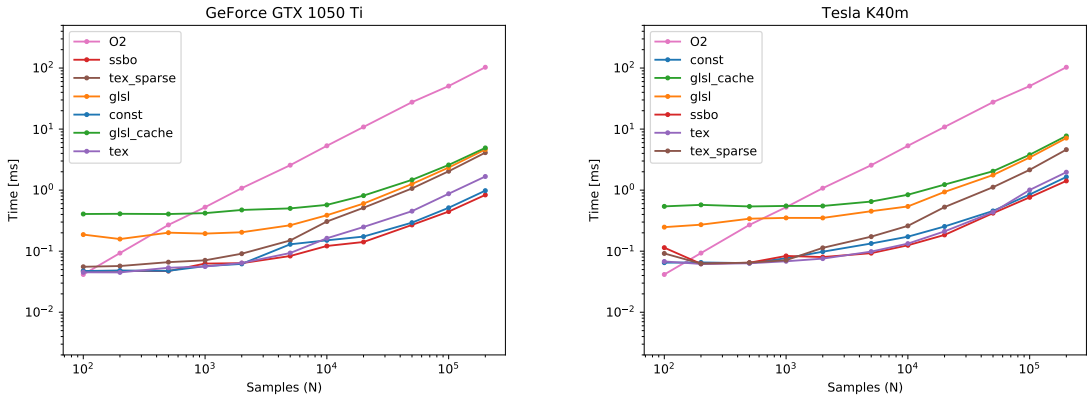
| Algorytm | RMSE (x) | RMSE (y) | RMSE (z) |
|---|---|---|---|
| Punkty pochodzące z trajektorii | | | |
| Shader Storage | 4.50e-02 | 4.56e-02 | 4.90e+00 |
| Tekstura 3D | 1.85e-03 | 1.91e-03 | 3.56e-03 |
| Rzadka tekstura 3D | 1.85e-03 | 1.91e-03 | 3.56e-03 |
| GLSL (bez pam. pod.) | 1.10e-08 | 1.06e-08 | 6.32e-08 |
| GLSL (pam. pod.) | 1.10e-08 | 1.06e-08 | 6.32e-08 |
| Stałe pole | 4.50e-02 | 4.56e-02 | 1.87e-01 |

**Tabela 5.3.** Wartości błędu średniokwadratowego dla maksymalnej testowanej (200000) ilości punktów pomiarowych. Wartości w kilogausach.

Rozwiązaniem, które najbardziej odbiega od modelu referencyjnego jest *Shader Storage Buffer*. We wszystkich przypadkach jest ono gorsze lub na podobnym poziomie co metoda ze stałym polem. Oba podejścia wykorzystujące teksturę znalazły się w tym rankingu pośrodku, oferując o rząd wielkości mniejszy błąd niż stałe pole. Najwierniejszym rozwiązaniem okazała się implementacja oryginalnego algorytmu w języku GLSL (z pamięcią podręczną i bez).

Rysunek 5.4 przedstawia czas wykonywania się algorytmów w funkcji liczby punktów testowych $N$. Można zauważyć, że metoda ze stałym polem, *Shader Storage Buffer*

oraz ze zwykłą teksturą mają podobną wydajność i są najszybsze. Implementacja metody z teksturą rzadką jest od nich wolniejsza, co pozwala przypuszczać że nie jest ona w pełni akcelerowana sprzętowo. Ze względu na losowy dobór punktów pomiarowych metoda GLSL z pamięcią podręczną osiągnęła gorsze wyniki niż wersja bez niej.




**Rys. 5.4.** Czas wykonywania się kodu algorytmów (w milisekundach) dla scenariusza *eval*.

Metoda ze stałym polem, *Shader Storage Buffer* oraz implementacje oparte o teksturę okazały się szybsze niż algorytm na CPU przy liczbie punktów 200 lub większej. Metoda GLSL bez pamięci podręcznej okazała się szybsza od niego przy 500, a z pamięcią przy 1000 punktów.

### 5.3.2. Scenariusz *fieldline*

Tabela 5.4 przedstawia pomiary błędu średniokwadratowego (*Root Mean Square Error*) dla maksymalnej testowanej ilości punktów pomiarowych (200000) w scenariuszu *fieldline*. Należy zwrócić uwagę na fakt, że w tym scenariuszu występuje efekt kumulacji błędu — drobne różnice w wartości wektorów pola na początku symulacji zmieniają drogę całej ścieżki punktu ze względu na iteracyjny sposób symulacji. Z tego powodu wartości błędu średniokwadratowego są w ogólności wyższe niż w scenariuszu *eval*.

Ranking metod pod względem wielkości błędu wygląda podobnie jak w poprzednim scenariuszu. Metody GLSL oferują najniższy błąd tak jak poprzednio, natomiast tym razem najgorszą okazała się implementacja ze stałym polem. W tym przypadku

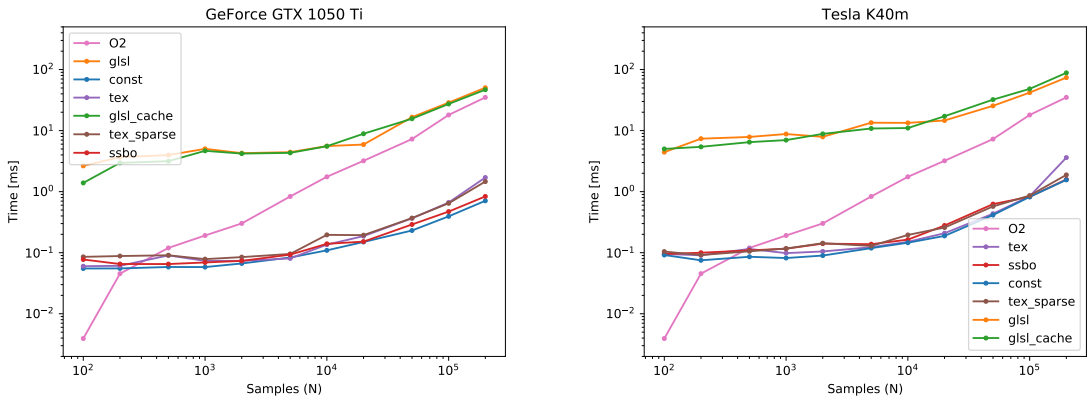
*Shader Storage Buffer* oraz metody z teksturą osiągnęły porównywalny wynik — o rząd wielkości lepszy niż metoda ze stałem polem.

| Algorytm | RMSE (x) | RMSE (y) | RMSE (z) |
|---|---|---|---|
| Losowe rozmieszczenie punktów (objętość elektromagnesu L3) | | | |
| Shader Storage | 1.83e-02 | 1.85e-02 | 2.81e-02 |
| Tekstura 3D | 1.72e-02 | 1.75e-02 | 2.72e-02 |
| Rzadka tekstura 3D | 1.72e-02 | 1.75e-02 | 2.72e-02 |
| GLSL (bez pam. pod.) | 9.27e-06 | 1.26e-05 | 1.03e-05 |
| GLSL (pam. pod.) | 9.27e-06 | 1.26e-05 | 1.03e-05 |
| Stałe pole | 5.70e-01 | 5.54e-01 | 1.13e+00 |

**Tabela 5.4.** Wartości błędu średniokwadratowego dla maksymalnej testowanej (200000) ilości punktów pomiarowych. Wartości w centymetrach.

Rysunek 5.5 przedstawia czas wykonywania się algorytmów w funkcji liczby punktów testowych $N$. Ten eksperyment pokazuje, że wielokrotna ewaluacja implementacji GLSL powoduje duży spadek wydajności. Tym razem obie wersje algorytmu osiągnęły podobne wyniki do siebie, ale okazały się wolniejsze od algorytmu na CPU (w najgorszym przypadku o rząd wielkości).

Pozostałe metody osiągnęły podobną wydajność jak w poprzednim scenariuszu. Ponieważ tym razem algorytm na CPU wykonywał się wydajniej, były w stanie osiągnąć od niego lepszy wynik dopiero przy 500 próbkach.




**Rys. 5.5.** Czas wykonywania się kodu algorytmów (w milisekundach) dla scenariusza *fieldline*.

## 5.4. Podsumowanie

Wizualizacja danych o polu na karcie graficznej wymaga opracowania sposobów dostępu do nich z poziomu karty. Istniejące dotychczas rozwiązania w ramach oprogramowania $O^2$ [32], [124], przystosowane do działania na procesorze komputera, nie umożliwiały takiego ich zastosowania. Rozdział ten przedstawia stworzone przeze mnie metody: stałego pola, bufora (Shader Storage Buffer Object), tekstury 3D, rzadkiej tekstury 3D i implementacji GLSL. Metoda stałego pola oraz implemetacja GLSL to metody wzorowane na istniejących rozwiązaniach; pozostałe zostały wymyślone oraz opracowane w całości przeze mnie. Wszystkie działają w postaci kodu shaderów oraz stanowią rozwiązanie przedstawionego problemu.

Metody przetestowałem w dwóch scenariuszach: a) *eval*, w którym pole ewaluowane jest w różnych punktach wewnątrz objętości detektora i b) *fieldline*, w którym przeprowadzany jest symulowany dryf tych punktów w polu magnetycznym.

Dla maksymalnej testowanej liczby punktów (200000) algorytm na CPU wykonał zadanie *eval* w **103 ms**. Dla porównania, najwolniej działająca metoda wykonała to zadanie w **4.92 ms** na karcie 1050 Ti oraz w **7.72 ms** na karcie K40m, co daje 13–20-krotne przyspieszenie. Taki czas wykonania przekłada się na ok. 129 – 203 klatek na sekundę.

W takich samych warunkach w przypadku scenariusza *fieldline*, algorytm na CPU wykonał zadanie w **35 ms**. Reimplementacja tego algorytmu w języku GLSL zrobiła to samo w **50 ms** na 1050 Ti i **88 ms** na K40m (20 i 11 klatek na sekundę). Pozostałe metody wypadły w tym teście dużo lepiej — metoda z teksturą rzadką wykonała się w **1.46 ms** (684 klatek na sekundę) na 1050 Ti, a w **1.88 ms** (531.91 klatek na sekundę) na K40m, więc 18 – 23 razy szybciej niż algorytm na CPU. Testy dowodzą, że ewaluacja modelu, przy zapewnieniu odpowiedniej minimalnej liczby punktów do przetworzenia, działa szybciej niż algorytm na CPU.

Wybór odpowiedniej metody stanowi kompromis pomiędzy wykorzystaną pamięcią wideo, wydajnością, a poziomem dokładności odwzorowania pola i zależy od konkretnego zastosowania. Przeprowadzone badania oraz analiza zużycia pamięci wideo dostarczają danych, na podstawie których można podjąć tą decyzję.

Dla dalszych rozważań najważniejszym faktem jest to, że udało się osiągnąć wydajność wystarczającą do tego, aby na postawie wprowadzonych przeze mnie metod zbudować aplikację do wizualizacji pola magnetycznego działającą w czasie rzeczywistym. Opracowane metody mogą zostać wykorzystane również w innych warunkach, na przykład przy prezentacji danych detektora LHCb, gdzie naukowcy wykorzystują podobny sposób modelowania pola magnetycznego [126].

# 6. Propagacja trajektorii cząstek

Jednym z zastosowań modeli pól wektorowych jest symulacja zjawisk fizycznych, na przykład zakrzywiania się w polu magnetycznym toru lotu cząstek posiadających ładunek elektryczny. Istniejące implementacje wizualizacji tego typu [9], [127], [128] wykorzystują do symulacji algorytm działający na procesorze komputera, a obliczone w ten sposób trajektorie (w postaci listy punktów w przestrzeni) przesyłane są do pamięci karty graficznej. Rozdział ten przedstawia zaproponowane przeze mnie ulepszone rozwiązanie, w którym wszystkie wymagane obliczenia przeprowadzane są bezpośrednio na karcie. Opisany tu generator proceduralny wymaga przekazania do karty graficznej jedynie kilku parametrów fizycznych dla każdej cząstki, co redukuje ilość przesyłanych danych. Wstępne wyniki badań zostały przeze mnie przedstawione na konferencji *Quark Matter* [14]. Artykuł zawierający pełny zestaw badań oczekuje na recenzję w czasopiśmie *Computer Physics Communications* (preprint [12]). Kod stworzonego oprogramowania dostępny jest w serwisie GitHub [129].

## 6.1. Rekonstrukcja trajektorii cząstek

Na podstawie informacji, w jaki sposób działa oryginalna, przeznaczona dla procesora komputera metoda propagacji trajektorii cząstek (Rozdział 2.6) opracowałem podobny algorytm przystosowany do działania na karcie graficznej w języku GLSL. Ze względu na specyfikę zadania (generowanie trajektorii z pozycji początkowych cząstek), zaimplementowany on został w postaci shadera geometrii. Można go skonfigurować w taki sposób, aby na ekranie pojawiały się linie (`line_strip`, Rysunek 6.2 — docelowy sposób wykorzystywany podczas wizualizacji) albo punkty (`points` — zastosowany podczas testów).

Sekwencja kroków wykonywana przez shader jest zbliżona do oryginalnej metody. Działanie rozpoczyna on od pobrania wektora pozycji, pędu oraz ładunku elektrycznego ze zmiennych wejściowych (`gl_Position`, `inMomentum`, `inCharge`). Następnie przeprowadzany jest test przekazanej przez użytkownika pozycji na obecność w dozwolonych granicach propagacji za pomocą funkcji `IsOutsideBounds(position, MAXRSQ, MAXZ)`,

gdzie stała `MAXRSQ` to kwadrat maksymalnej dozwolonej odległości rozważanego punktu od początku układu współrzędnych, a `MAXZ` to jego maksymalna odległość na osi $z$. Jeżeli wynik testu jest negatywny (czyli punkt leży w dozwolonych granicach), staje się on pierwszym punktem trajektorii (tj. mnożony jest przez macierz projekcji, a potem wysłany do dalszych etapów potoku graficznego za pomocą wywołania funkcji `EmitVertex()`). Następnie inicjalizowane są wymagane zmienne globalne potrzebne do obliczeń trajektorii helisy (Rozdział 2.6) za pomocą funkcji `Update()` oraz wywoływana jest funkcja `LoopToBounds()`, która w pętli generuje kolejne punkty żądanej trajektorii.

```
1  void main(void) {
2      const vec3 iVertex = gl_in[0].gl_Position.xyz;
3      const vec3 iMomentum = inMomentum[0].xyz;
4      const float iCharge = inCharge[0];
5
6      if (!IsOutsideBounds(iVertex, MAXRSQ, MAXZ)) {
7          gl_Position = state.MVP * vec4(iVertex.xyz, 1);
8          EmitVertex();
9
10         Update(iMomentum, Field(iVertex), iCharge);
11         LoopToBounds(iVertex, iMomentum, iCharge);
12     }
13     EndPrimitive();
14 }
```

**Listing 10.** Implementacja głównej procedury shadera geometrii.

Funkcja `Update()` inicjalizuje wektory lokalnego układu współrzędnych $\vec{E}_1$, $\vec{E}_2$ i $\vec{E}_3$, długości poprzecznego oraz wzdłużnego wektora pędu $\left|\vec{P}_{\parallel}\right|$ i $\left|\vec{P}_{\perp}\right|$ oraz promień helisy $R$ zgodnie z wzorami pochodzącymi z O[2] przedstawionymi w Rozdziale 2.6.

```
1  float PlMag, PtMag, R;
2  vec3 Pl, Pt, E1, E2, E3;
3
4  void Update(const vec3 p, const vec3 b, const float charge) {
```

```
5       E1 = normalize(b);
6
7       PlMag = dot(p, E1);
8       Pl = E1 * PlMag;
9       Pt = p - Pl;
10      PtMag = length(Pt);
11
12      E2 = normalize(Pt);
13      E3 = cross(E1, E2);
14
15      if (charge < 0) {
16          E3 = -E3;
17      }
18
19      //B2C is a momentum-to-curvature conversion constant
20      R = abs(PtMag / (B2C * length(b) * abs(charge)));
21  }
```

**Listing 11.** Implementacja funkcji `Update()`.

Funkcja `LoopToBounds()` implementuje logikę głównej pętli generatora, której schemat działania przedstawiony został na Rysunku 2.7. Produkuje ona w sposób iteracyjny kolejne punkty na helisie za pomocą funkcji `StepHelix()` tak długo, aż nie zostanie osiągnięty limit liczby punktów (reprezentowany przez stałą `NMAX`), nie zostaną osiągnięte warunki brzegowe na dozwoloną odległość rozważanego punktu od początku układu współrzędnych (porównanie `RforwVsq` z `MAXRSQ`) lub na odległość na osi $z$ (porównanie wartości bezwzględnej `forwV.z` z `MAXZ`). W przypadku przekroczenia dozwolonych granic punkt uzyskany z równania helisy jest do nich przycinany za pomocą interpolacji liniowej pomiędzy poprzednim i obecnym punktem, a algorytm kończy działanie. W przeciwnym wypadku obliczony punkt staje się punktem obecnym oraz następuje aktualizacja wymaganych parametrów za pomocą funkcji `Update()`.

```
1   int number_points = 0;
2
```

```
3  void StepHelix(vec4 v, vec3 p, out vec4 vOut, out vec3 pOut) {
4      const vec3 d = E2 * (R * sin(PhiStep))
5                    + E3 * (R * (1 - cos(PhiStep)))
6                    + E1 * R * PhiStep * (Pl Mag / PtMag);
7
8      vOut = v + vec4(d, abs(R * PhiStep * (Pl Mag / PtMag)));
9      pOut = Pl + E2 * (PtMag * cos(PhiStep))
10               + E3 * (PtMag * sin(PhiStep));
11 }
12
13 void LoopToBounds(vec3 vtx, vec3 momentum, float charge) {
14     vec4 currV = vec4(vtx, 0);
15     vec3 currP = momentum;
16     vec4 forwV = vec4(vtx, 0);
17     vec3 forwP = vec3(currP);
18
19     while (number_points < NMAX) {
20         StepHelix(currV, currP, forwV, forwP);
21
22         const float RforwVsq = dot(forwV.xy, forwV.xy);
23
24         // radius bound
25         if (RforwVsq > MAXRSQ) {
26             const float RcurrV = length(currV.xy);
27             const float RforwV = sqrt(RforwVsq);
28
29             const float t = (MAXR - RcurrV) /
30                             (RforwV - RcurrV);
31
32             const vec3 d = mix(currV.xyz, forwV.xyz, t);
33
34             gl_Position = state.MVP * vec4(d, 1);
35
36             EmitVertex();
```

```
37              return;
38
39          // z distance bound
40          } else if (abs(forwV.z) > MAXZ) {
41              const float t = (MAXZ - abs(currV.z)) /
42                              abs(forwV.z - currV.z);
43
44              const vec3 d = mix(currV.xyz, forwV.xyz, t);
45
46              gl_Position = state.MVP * vec4(d, 1);
47
48              EmitVertex();
49              return;
50          }
51
52          currV = forwV;
53          currP = forwP;
54          Update(currP, Field(currV.xyz), charge);
55
56          gl_Position = state.MVP * vec4(currV.xyz, 1);
57
58          EmitVertex();
59          ++number_points;
60      }
61  }
```

**Listing 12.** Implementacja funkcji `StepHelix()` oraz `LoopToBounds()`.

Funkcja `Field()` użyta w implementacjach procedur `main()` oraz `LoopToBounds()` reprezentuje metodę dostępu shadera do danych o polu magnetycznym detektora. W rezultacie badań opisanych w Rozdziale 5 powstało kilka jej rozwiązań, na których opisywany algorytm propagatora może pracować. Są to:

- *Shader Storage Buffer* (SSBO) — Rozdział 5.1.2,
- tekstura 3D — Rozdział 5.1.3,

- rzadka tekstura 3D — Rozdział 5.1.4,
- GLSL (implementacja oryginalnego modelu pola na karcie graficznej) — Rozdział 5.1.5
- stałe pole w objętości detektora oraz w objętości tylko elektromagnesu L3 — Rozdział 5.1.1.

Metoda SSBO wykorzystuje tablicę z próbkami pola jako sposób przechowywania modelu w pamięci karty graficznej. Metoda z teksturą wykorzystuje trójwymiarową teksturę do przechowywania próbek pola. Metoda z rzadką teksturą optymalizuje zużycie pamięci, pomijając obszary detektora w których nie ma pola. Metoda GLSL implementuje model pola w taki sposób, w jakim jest on zaimplementowany w $O^2$.

Implementację propagatora trajektorii przetestowałem w kombinacji z każdą implementacją modelu pola.

## 6.2. Metodologia badań

Dane o zderzeniach wykorzystane w testach wygenerowane zostały przy pomocy oprogramowania ALICE do symulacji zderzeń za pomocą metody Monte Carlo — GEANT4 [130]. Wyniki przedstawione niżej powstały w skutek uśrednienia rezultatów 10 niezależnych uruchomień eksperymentów.

### 6.2.1. Odczyt wyników z karty graficznej

W projekcie scenariusza testów zrobiłem podobne założenia dotyczące finalnej lokalizacji wyniku działania algorytmów co w Rozdziale 5.2.1. Z tego powodu po raz kolejny wykorzystałem funkcję `Transform Feedback` po to, aby zapisać generowane przez propagator punkty do bufora oraz `GL_RASTERIZER_DISCARD` po to, aby pominąć kroki prowadzące do wyświetlenia elementów na ekranie.

### 6.2.2. Sposób pomiaru wydajności

Podobnie jak zrobiłem to w Rozdziale 5.2.2, wydajność propagatora zmierzyłem jako czas spędzony pomiędzy zakolejkowaniem nowego zadania dla karty za pomocą wywołania funkcji `glDrawArrays()`, a ukończeniem kopiowania wygenerowanych punktów z karty za pomocą wywołania funkcji `glGetNamedBufferSubData()`. Czas zmie-

rzony został za pomocą zegara o największej dostępnej w bibliotece standardowej C++ precyzji (`std::chrono::high_resolution_clock`).

Należy zwrócić uwagę na to, że metoda ta ma pewne wady. W normalnych warunkach karty graficzne (dla maksymalnego wykorzystania dostępnych zasobów) pracują w sposób strumieniowy, przetwarzając kilka zakolejkowanych klatek animacji (na różnym etapie ukończenia) jednocześnie [37]. Proces ten jest zaburzony, jeśli wymagana jest synchronizacja z CPU, np. jeśli potrzebny jest odczyt danych za pomocą `glGetNamedBufferSubData()`. Z tego powodu przeprowadzony przeze mnie test wydajności mierzy czas wykonywania się algorytmu, ale w najgorszym możliwym przypadku — w rzeczywistych warunkach wydajność powinna być większa (gdzie strumieniowanie częściowo kompensuje czas potrzebny na wygenerowanie klatki animacji).

### 6.2.3. Sposób pomiaru różnic między trajektoriami

Po to, aby w sposób ilościowy określić różnice w działaniu algorytmu propagacji na karcie graficznej oraz oryginalnej implementacji wykorzystałem błąd średniokwadratowy (*Root Mean Square Error*), obliczany dla wszystkich punktów składających się na daną trajektorię. Błąd liczony jest dla każdej osi oddzielnie.

Algorytm propagacji posiada dwa kryteria stopu: osiągnięcie maksymalnej liczby iteracji albo osiągnięcie granic propagacji (Rozdział 2.6). Powoduje to, że wygenerowane trajektorie składają się z różnej liczby punktów pośrednich. Rodzi to problem dla interpretacji danych pochodzących z karty graficznej, gdzie wszystkie wygenerowane wierzchołki umieszczone są jeden po drugim w buforze odczytywanym z jej pamięci. Uniemożliwia to znalezienie początku oraz końca danej trajektorii. W celu rozwiązania tego problemu kod shadera zmodyfikowałem, aby zawsze produkował maksymalną liczbę punktów (po przekroczeniu granic, wektory zerowe). W ten sposób w odczytywanym buforze początki oraz końce trajektorii mają regularne indeksy. Ponieważ stanowi to dla shadera dodatkową pracę (która może spowodować spadek wydajności), opcję tą zastosowałem tylko i wyłącznie podczas pomiarów różnic w generowanych trajektoriach (przy testach wydajności została ona wyłączona). Dla zwiększenia czytelności

ten szczegół implementacyjny został pominięty w przedstawionych w Rozdziale 6.1 fragmentach kodu — wersja kompletna dostępna jest w serwisie GitHub [129].

## 6.3. Wyniki

Testy przeprowadziłem na następującym sprzęcie:

- Komputer stacjonarny — NVIDIA GeForce RTX 2080 Ti (11 GiB VRAM), AMD Ryzen Threadripper 1920X 3.5 GHz,
- Laptop ThinkPad X1 Extreme — NVIDIA GeForce GTX 1050 Ti with Max-Q Design (4 GiB VRAM), Intel Core i7-8850H 2.6 GHz.

Na komputerze przenośnym zainstalowany był system Windows 10 oraz sterownik graficzny NVIDIA 460.39. Na komputerze stacjonarnym zainstalowany był system Ubuntu 20.04.4 LTS oraz sterownik graficzny NVIDIA 470.103.

Rysunek 6.1 pokazuje średni czas rysowania klatek przez kartę graficzną dla rosnącej liczby propagowanych trajektorii od 50 do 7250. Średnia liczba cząstek zawartych w pojedynczym zderzeniu w plikach symulacji to ok. 4000.

Można zauważyć, że wyniki propagatora zebrane na starszej karcie (GTX 1050 Ti) zachowują podobny trend, co wyniki testów samych metod ewaluacji pola magnetycznego opisanych w Rozdziale 5.3. Przy 7250 cząstkach:

1. metoda stałego pola oraz *Shader Buffer Storage Object* okazała się najszybsza (7–10 ms, co przekłada się na 100–143 klatek na sekundę),
2. następne były metody z teksturą (25–28 ms, co przekłada się na 35–40 klatek na sekundę),
3. najwolniejsza okazała się metoda GLSL (67 ms, co przekłada się na 15 klatek na sekundę).

Przy 4000 trajektorii metoda ze stałym polem osiągnęła 250 klatek na sekundę, metoda z buforem 175 klatek na sekundę, metody z teksturą 50 klatek na sekundę, a metoda GLSL 19 klatek na sekundę.

Na karcie 2080 Ti, przy 7250 cząstkach:

1. metoda stałego pola oraz *Shader Buffer Storage Object* osiągnęła 9 ms, czyli 111

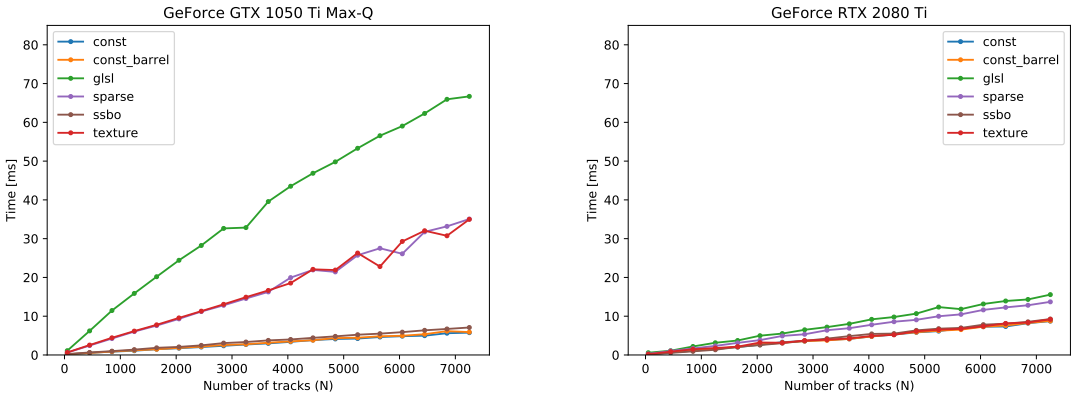
klatek na sekundę. Wynik ten jest bardzo podobny do dużo słabszej 1050 Ti i sugeruje, że w tym wypadku limitem wydajności nie jest moc samej karty,

2. metoda ze zwykłą teksturą osiągnęła tą samą wydajność co wyżej wymienione,
3. metoda GLSL okazała się tym razem dużo szybsza i o podobnej wydajności co metoda z teksturą rzadką (13–16 ms, co przekłada się na 62–77 klatek na sekundę).

Przy 4000 trajektorii metoda ze stałym polem, zwykłej tekstury oraz z buforem osiągnęła 208 klatek na sekundę. Metoda z teksturą rzadką osiągnęła 128 klatek na sekundę, a metoda GLSL osiągnęła 109 klatek na sekundę.

Tabela 6.1 przedstawia wartość błędu średniokwadratowego na trajektorię, obliczoną przy maksymalnej (7250) testowanej ich liczbie. Przy zastosowaniu metody stałego pola, co czyni algorytm wykonywany na karcie graficznej identycznym do tego na CPU błąd ten jest bliski zeru. Potwierdza to poprawne działanie algorytmu. Identyczna wartość błędu przypada dla pola ograniczonego do objętości elektromagnesu L3. Pozostałe metody osiągnęły podobny, ale dużo wyższy poziom błędu. W tym akurat przypadku jest to pożądane — oznacza to, że różnica między stosowaniem modelu naiwnego (stałe pole), a dokładnego nie jest pomijalna.

Aby wizualnie sprawdzić wynik działania propagatora, typ generowanych figur przestawiony został z punktów na linie, a rasteryzacja została z powrotem włączona. Rysunek 6.2 przedstawia przykładową wizualizację 500 cząstek (dla zachowania przejrzystości) dla propagatora ze stałym polem (w kolorze czerwonym) oraz dla propagatora z metodą GLSL (w kolorze niebieskim). W obszarze zderzenia (środek obrazka) można zauważyć drobne, ale dostrzegalne różnice w pozycjach oraz kątach zakrzywiania się cząstek. Wyraźniejszym efektem jest zakrzywienie “niebieskich” cząstek będących mionami po lewej stronie obrazka. Zakrzywienie nie występuje dla cząstek “czerwonych”, bo metoda stałego pola nie modeluje wpływu elektromagnesu dipolowego na trajektorię. Zakrzywienie “niebieskich” cząstek po prawej stronie obrazka wynika ze szczątkowego pola magnetycznego w rurze LHC, który jest zawarty w modelu (Rysunek 5.2). Ujęcie tego samego zestawu trajektorii przy innym ustawieniu kamery przedstawia Rysunek 6.3.




**Rys. 6.1.** Średni czas rysowania klatki w milisekundach dla liczby cząstek od 50 do 7250, dla każdej implementacji pola magnetycznego na karcie graficznej.

| Algorytm | RMSE (x) | RMSE (y) | RMSE (z) |
|---|---|---|---|
| Stałe pole | 1.407e−3 | 1.381e−3 | 1.070e−3 |
| Stałe pole - elektromagnes L3 | 1.407e−3 | 1.381e−3 | 1.070e−3 |
| SSBO | 3.957 | 4.126 | 6.571 |
| Tekstura 3D | 3.956 | 4.124 | 6.562 |
| Tekstura rzadka 3D | 3.956 | 4.124 | 6.562 |
| GLSL (bez pam. pod.) | 3.956 | 4.136 | 6.569 |

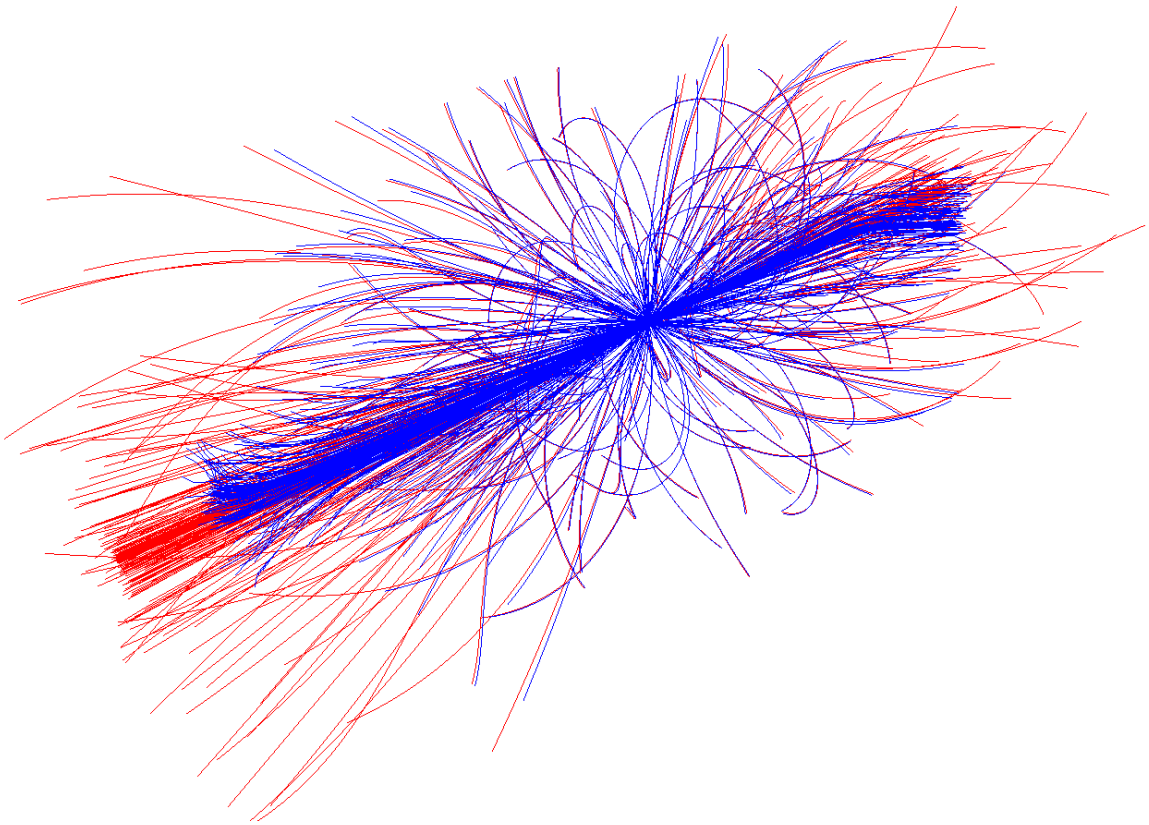
**Tabela 6.1.** Podsumowanie różnic w generowanych trajektoriach między algorytmem na CPU (z stałym modelem pola), a implementacjami na GPU dla 7250 trajektorii. Wartości w centymetrach.

## 6.4. Podsumowanie

W trakcie badań zaproponowałem metodę przeprowadzenia propagacji trajektorii cząstek na karcie graficznej przy użyciu shadera geometrii.

Działanie implementacji zostało porównane z propagatorem działającym na CPU w ramach oprogramowania $O^2$. Przy identycznych ustawieniach (pole magnetyczne, granice propagacji) obydwa rozwiązania generują praktycznie identyczne (Tabela 6.1) trajektorie, co dowodzi, że moje dzieło działa w poprawny sposób oraz może zastąpić oryginalną metodę.

W ramach kolejnego testu w shaderze uproszczony model pola magnetycznego wymieniony został na model dokładny (wynik badań opisanych w Rozdziale 5) po to, aby

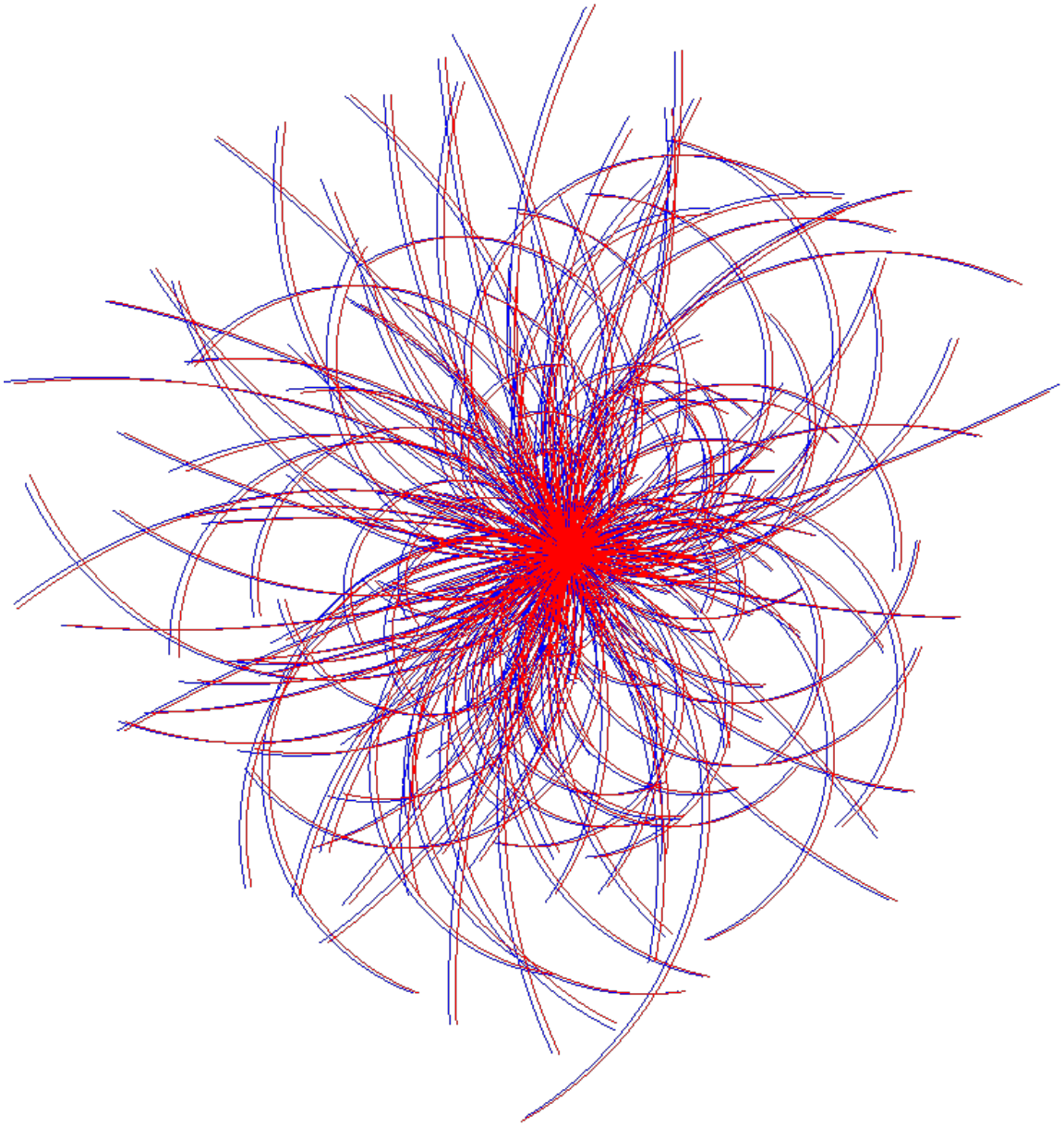


**Rys. 6.2.** Przykładowy wynik propagacji oraz wizualizacji 500 trajektorii cząstek (widok pod kątem). Trajektorie w kolorze czerwonym zostały wygenerowane przy użyciu modelu stałego pola, natomiast w kolorze niebieskim modelu dokładnego (GLSL).

sprawdzić, w jaki sposób wpłynie on na kształt trajektorii. Wyniki testu przedstawione są w sposób ilościowy w Tabeli 6.1 oraz w sposób jakościowy na Rysunkach 6.2 oraz 6.3. Test ten dowodzi, że trajektorie obliczone z wykorzystaniem modelu uproszczonego oraz dokładnego różnią się od siebie. Znacząca różnica w trajektorii występuje dla mionów, których ścieżki powinny być zakrzywione z powodu oddziaływania z elektromagnesem dipolowym, a w uproszczonym modelu nie są (Rysunek 6.2).

Ranking wydajności propagatora przy użyciu różnych technik przechowywania modelu pola wygląda następująco (na karcie GTX 1050 Ti):

- metoda ze stałym polem osiągnęła 250 klatek na sekundę,
- metoda z buforem osiągnęła 175 klatek na sekundę,



**Rys. 6.3.** Przykładowy wynik propagacji oraz wizualizacji 500 trajektorii cząstek (widok na wprost detektora). Trajektorie w kolorze czerwonym zostały wygenerowane przy użyciu modelu stałego pola, natomiast w kolorze niebieskim modelu dokładnego (GLSL).

- metody z teksturą osiągnęły 50 klatek na sekundę,
- metoda GLSL osiągnęła 19 klatek na sekundę.

Uważam, że wszystkie metody oprócz ostatniej nadają się do wykorzystania w aplikacji wizualizacji trajektorii cząstek działającej w czasie rzeczywistym. Zastosowanie ostatniej metody (której nie odrzucałbym ze względu na to, że oferuje najlepsze odwzoro-

wanie pola) ograniczyłbym do sytuacji nie wymagających interakcji z użytkownikiem, np. przy zapisywaniu obrazów i filmów, kiedy aplikacja tylko odtwarza ustawienia określone wcześniej przez użytkownika.

Wyniki dla karty RTX 2080 są następujące:

- metoda ze stałym polem, z buforem oraz ze zwykłą teksturą osiągnęła 208 klatek na sekundę,
- metoda z teksturą rzadką osiągnęła 128 klatek na sekundę,
- metoda GLSL osiągnęła 109 klatek na sekundę.

Nowsza oraz mocniejsza karta graficzna była w stanie przeprowadzić propagację trajektorii z liczbą klatek na sekundę przewyższającą 60. Dla użytkownika aplikacji oznacza to gwarancję zachowania interaktywności aplikacji bez względu na wybrany model pola magnetycznego.

Metoda stworzona na potrzeby tego Rozdziału jest w stanie w pełni zastąpić podstawową funkcjonalność programu Event Display w ALICE. Nie istnieją jednak żadne ograniczenia, które mogłyby przeszkadzać w jej wykorzystaniu w ogólniejszym kontekście np. do ulepszenia podobnych do Event Display programów używanych w pozostałych eksperymentach w CERN [127], [128]. Przeprowadzanie propagacji trajektorii cząstek na karcie graficznej, otwiera również dodatkowe możliwości np. prezentacji wpływu pola magnetycznego na cząstki przez dynamiczną zmianę siły pola. Innym rozwinięciem tej metody mogłoby być wyświetlanie dodatkowych informacji na reprezentacjach trajektorii np. wektora prędkości czy wektora pola.

Propagacja trajektorii cząstek na karcie graficznej pozwala na tworzenie atrakcyjnych wizualizacji, którymi zainteresowani mogą być również organizatorzy studenckich warsztatów Masterclass promujących naukę, które prowadzone są przez CERN [131]–[133].

# 7. Wizualizacja pól wektorowych z wykorzystaniem shaderów siatki

Przeprowadzona przeze mnie analiza najnowszych sposobów realizacji proceduralnego generowania geometrii na kartach graficznych (Rozdział 3) oraz istniejących algorytmów wizualizacji pól wektorowych (Rozdział 4) dowodzi, że zagadnienie wykorzystania potoku shaderów siatki (Rozdział 3.3) do tego rodzaju wizualizacji nie zostało jeszcze do tej pory przebadane przez środowisko naukowe. Nowy potok graficzny łączy w sobie zalety potoku graficznego i obliczeniowego. Z tego powodu jest interesujący dla dziedziny ze względu na potencjalne możliwości optymalizacyjne (a co za tym idzie, poprawę wydajności algorytmów) w tym zastosowaniu.

Rozdział ten zawiera opis opracowanych przeze mnie w ramach badań algorytmów wizualizacji pól wektorowych. Ich implementacje zostały zawarte w stworzonej przeze mnie aplikacji *FieldView*, która pozwala użytkownikowi na eksplorację na ekranie komputera dowolnego pola wektorowego. Program posłużył w badaniach do przetestowania wydajności zaproponowanych algorytmów. Krótki opis jego obsługi znajduje się w Załączniku A. Artykuł zawierający wyniki badań oczekuje na recenzję w czasopiśmie *SoftwareX* (preprint [13]). Rezultaty badań zostały również wdrożone w praktyce, w postaci implementacji wizualizacji pola magnetycznego detektora ALICE w ramach oprogramowania $O^2$. Kod źródłowy dostępny jest w serwisie GitHub [134].

## 7.1. Metody wizualizacji

W trakcie badań zaprojektowałem cztery sposoby wizualizacji pola wektorowego: za pomocą linii, wstążek, tub oraz podziałów powierzchni. Ogólny pomysł na wizualizację pola za pomocą trzech pierwszych wymienionych sposobów nie jest nowy i zaczerpnięty został z literatury (Rozdział 4.1.4). Pomysł na ich realizację za pomocą shaderów siatki (a w szczególności opracowane algorytmy generujące odpowiednie elementy graficzne) są w całości mojego autorstwa. Ostatnia metoda, wykorzystująca podziały powierzchni, jest moim dziełem zarówno od strony koncepcyjnej jak i realizacyjnej.

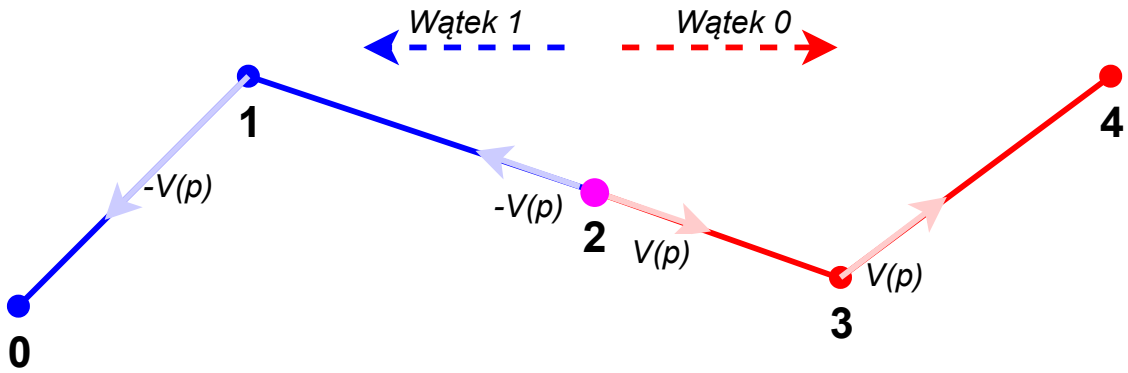
### 7.1.1. Linie

Rysunek 7.3 przedstawia przykładową wizualizację wykorzystującą linie przepływu dla pola wektorowego określonego wzorem $f = (\sin(x+z), z, y)$. Ta metoda wizualizacji oferuje trzy parametry konfiguracyjne:

1. *Instances*, który kontroluje liczbę linii na ekranie,
2. *Segments*, który kontroluje liczbę segmentów, z których składa się pojedyncza linia,
3. *Step*, który kontroluje rozmiar kroku wykorzystywany w obliczaniu punktów linii przepływu.

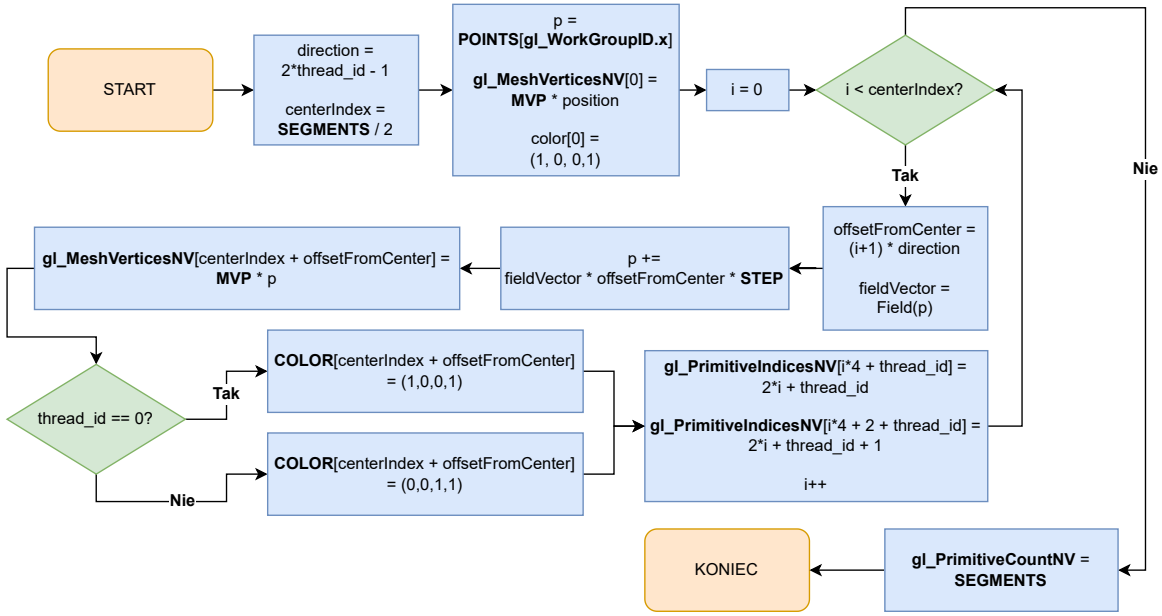
Algorytm generowania linii przepływu (diagram na Rysunku 7.2) tworzy ją zaczynając od punktu wskazanego przez użytkownika. Schemat jej budowy przedstawia Rysunek 7.1. Do pracy wykorzystywane są dwa wątki, które generują punkty w przeciwnych kierunkach.

Pierwszym krokiem jest zapisanie wejściowej pozycji pod środkowym indeksem tablicy wierzchołków. Teoretycznie mógłby dokonać tego tylko jeden z wątków, jednak jednym z podstawowych sposobów optymalizacji kodu shaderów jest unikanie rozgałęzień (instrukcji `if`). W architekturze kart graficznych powodują one sekwencyjne wykonywanie obu ścieżek programu przez dany zbiór wątków [135], obniżając wydajność. Z tego powodu punkt ten zapisują oba wątki jednocześnie — jest to w tym wypadku dozwolone ze względu na identyczność danych.




**Rys. 7.1.** Schemat budowy linii przepływu w shaderze siatki na przykładzie czterech segmentów.

Następnie w pętli oba wątki generują kolejne pozycje $p$ poprzez ewaluację pola $V$ w $p$ i przesunięcie $p$ o uzyskany w ten sposób wektor pola. Długość kroku (kontrolowana przez parametr *Step*) wykorzystywana jest jako mnożnik wektora. Wątek 1 wykorzystuje ujemne wartości kroku (dzięki czemu buduje linię pola w przeciwnym kierunku). Wątek 0 zapisuje punkty w tablicy wierzchołków od indeksu środkowego w przód. Wątek 1 zapisuje punkty od indeksu środkowego w tył.




**Rys. 7.2.** Diagram algorytmu generowania linii w shaderze siatki. Zmienne, których nazwy zostały pogrubione to zmienne globalne.

```
1  void main() {
2      const uint thread_id = gl_LocalInvocationID.x;
3      const int direction = int((thread_id * 2) - 1);
4      const uint centerIndex = (settings.segmentsCount / 2);
5      vec4 position = vertices.pos[IN.vertexID];
6      gl_MeshVerticesNV[centerIndex].gl_Position = state.MVP * position;
7      v_out[centerIndex].color = vec4(1.0, 0.0, 0.0, 1.0);
8      for (uint i = 0; i < centerIndex; ++i) {
9          const int offsetFromCenter = int(i+1) * direction;
10         const vec3 fieldVector = Field(position.xyz);
11         position.xyz += direction * fieldVector * settings.step;
12         gl_MeshVerticesNV[centerIndex + offsetFromCenter].gl_Position =
```

```
13              state.MVP * position;
14          if (thread_id == 0) {
15              v_out[centerIndex + offsetFromCenter].color =
16                  vec4(1.0, 0.0, 0.0, 1.0);
17          } else {
18              v_out[centerIndex + offsetFromCenter].color =
19                  vec4(0.0, 0.0, 1.0, 1.0);
20          }
21          gl_PrimitiveIndicesNV[i*4 + thread_id] = i*2 + thread_id;
22          gl_PrimitiveIndicesNV[i*4 + 2 + thread_id] = i*2 + thread_id + 1;
23      }
24      gl_PrimitiveCountNV = settings.segmentsCount;
25  }
```

**Listing 13.** Implementacja funkcji `main()` shadera siatki generującego linie pola.

W tej samej pętli jednocześnie wypełniania jest tablica indeksów wierzchołków na potrzeby procesora prymitywów, która musi zawierać indeks punktu początkowego oraz końcowego dla każdego segmentu linii. Dla przykładu z Rysunku 7.1 poprawna jest następująca kolejność wierzchołków: $0,1-1,2-2,3-3,4$. Ponieważ przejść przez pętlę jest dwa razy mniej niż generowanych segmentów, oba wątki muszą zapisać w każdym z nich po dwa indeksy. Wątek 0 zapisuje indeksy początków linii, a wątek 1 indeksy końców linii. Dlatego:

- wątek 0 powinien wpisywać do tablicy `gl_PrimitiveIndicesNV` pod indeksami $0,2,4,6$ wartości $0,1,2,3$,
- wątek 1 powinien wpisywać do tablicy `gl_PrimitiveIndicesNV` pod indeksami $1,3,5,7$ wartości $1,2,3,4$.

We wspomnianej wyżej pętli, gdzie $i$ jest iteratorem, aby uzyskać odpowiednie ciągi liczb:

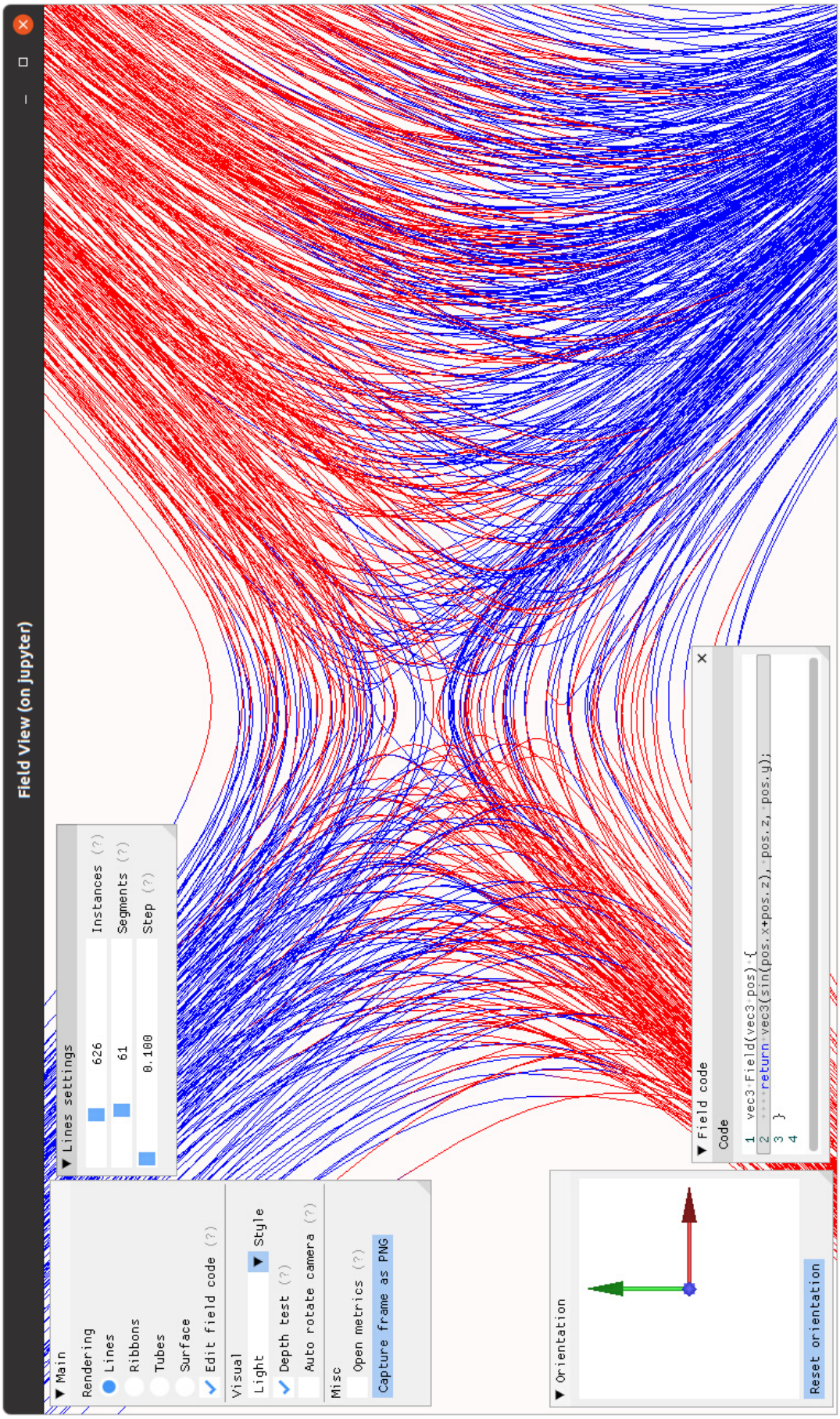
- wątek 0 powinien wpisywać pod indeksem $i*4$ wartość $i*2$, a pod indeksem $i*4+2$ wartość $i*2+1$,

- wątek 1 powinien wpisywać pod indeksem $i*4+1$ wartość $i*2+1$, a pod indeksem $i*4+2$ wartość $i*2+2$.

Aby uniknąć instrukcji `if` w kodzie shadera wykorzystałem numer wątku $t_{id}$ do połączenia powyższych przypadków w dwa wzory na indeks oraz dwa wzory na wartość, przedstawione poniżej:

$$idx1 = i*4+t_{id},$$
$$var1 = i*2+t_{id},$$
$$idx2 = i*4+2+t_{id},$$
$$var2 = i*2+1+t_{id}.$$

Wygenerowane wartości wpisywane są do tablicy `gl_PrimitiveIndicesNV`: pod `idx1` wartość `var1`, pod `idx2` wartość `var2`. Ostatnim krokiem algorytmu jest ustawienie wartości zmiennej `gl_PrimitiveCountNV` na liczbę renderowanych segmentów linii.

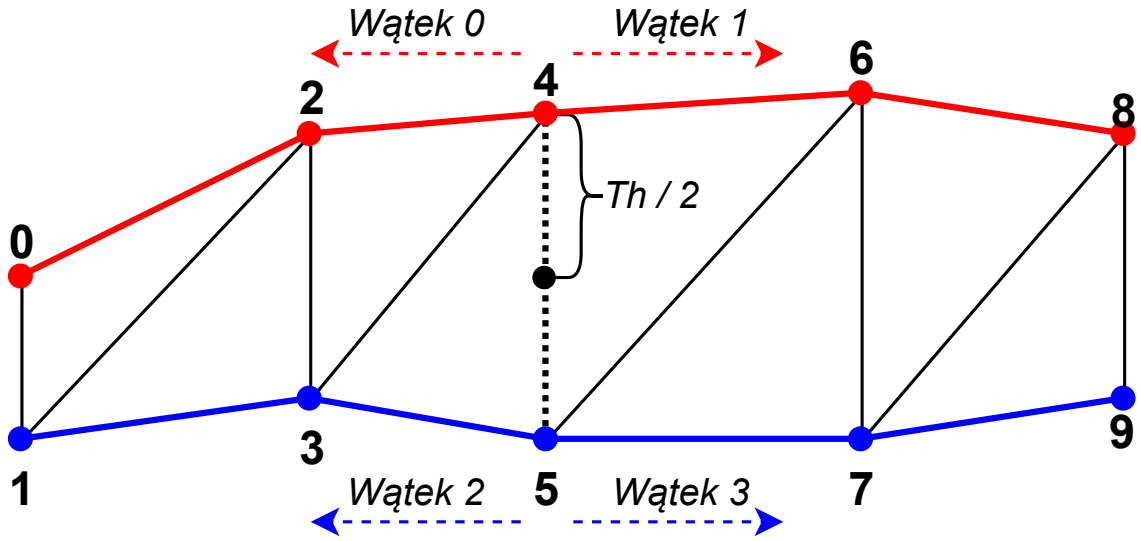



**Rys. 7.3.** Przykładowa wizualizacja pola wektorowego o wzorze $f = (\sin(x+z), z, y)$ za pomocą linii.

### 7.1.2. Wstążki

Rysunek 7.6 przedstawia przykładową wizualizację wykorzystującą wstążki dla pola wektorowego określonego wzorem $f = (-y, -z, x)$. Ta metoda wizualizacji oferuje cztery parametry konfiguracyjne:

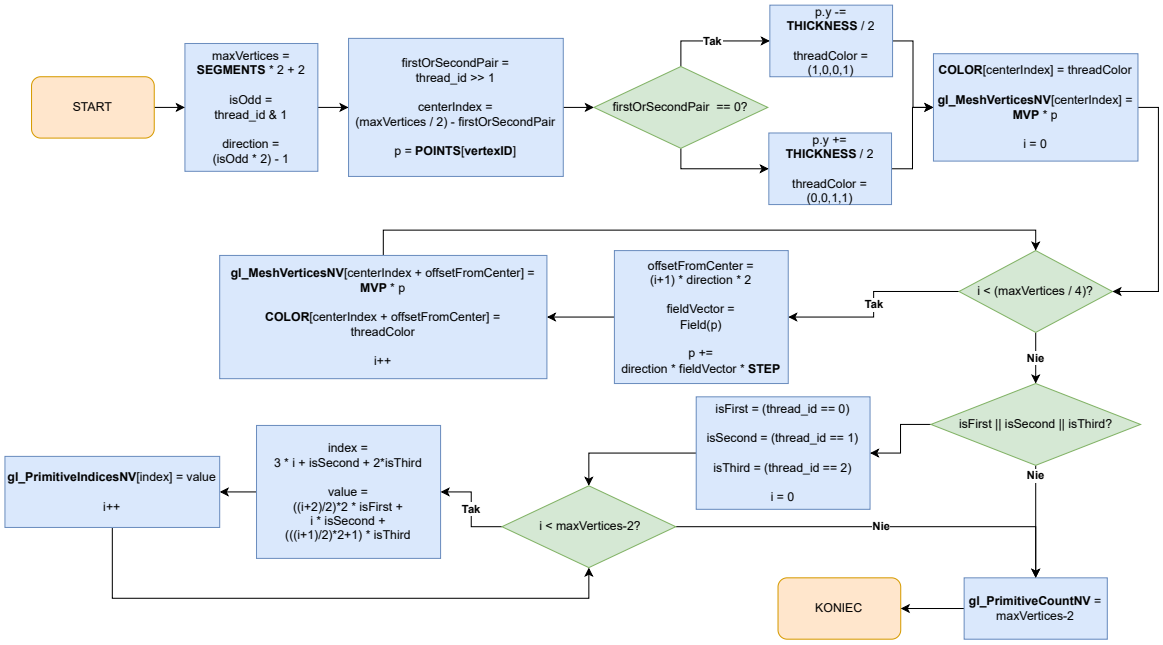
- *Instances*, który kontroluje liczbę wstążek na ekranie,
- *Segments*, który kontroluje liczbę segmentów, z których składa się pojedyncza wstążka,
- *Thickness*, który kontroluje grubość wstążki w jej centrum (punkcie startowym algorytmu),
- *Step*, który kontroluje rozmiar kroku wykorzystywany w obliczaniu punktów linii przepływu.




**Rys. 7.4.** Schemat budowy wstążki w shaderze siatki na przykładzie czterech segmentów.

Algorytm generowania geometrii (diagram na Rysunku 7.5) tworzy wstążkę z punktów leżących na dwóch liniach przepływu, które połączone są w trójkąty. Schemat budowy wstążki przedstawia Rysunek 7.4. Pozycja początkowa wstążki (wskazana przez użytkownika) zaznaczona jest czarną kropką. Algorytm rozpoczyna działanie od inicjalizacji punktów startowych linii przepływu przez rozsunięcie punktu startowego w przeciwnych kierunkach na osi $y$ o połowę wartości zmiennej `Th` (*Thickness*), która

kontrolowana jest przez użytkownika i określa początkową grubość wstążki. Punkty leżące na liniach przepływu, tak jak w Rozdziale 7.1.1, generowane są przez dwa wątki — cały shader siatki wykorzystuje ich więc cztery. Pierwsza para wątków (0 i 1) zapisuje generowane przez siebie punkty na parzystych pozycjach w buforze wierzchołków; druga para wątków (2 i 3) zapisuje je na pozycjach nieparzystych. Wątki, zgodnie ze schematem, zapisują punkty zaczynając od środka tablicy (w przykładzie jest to indeks 4 i 5). Liczba generowanych punktów zależy od liczby segmentów kontrolowanej przez parametr *Segments*. Pojedynczy segment to czworobok złożony z dwóch trójkątów (np. z wierzchołków 4,5,6,7 na Rysunku 7.4).




**Rys. 7.5.** Diagram algorytmu generowania wstążek w shaderze siatki. Zmienne, których nazwy zostały pogrubione to zmienne globalne.

```
1  void main() {
2      const uint maxVertices = settings.segmentsCount * 2 + 2;
3      const uint thread_id = gl_LocalInvocationID.x;
4      const uint isOdd = thread_id & 1u;
5      const uint firstOrSecondPair = thread_id >> 1;
6      const int direction = int((isOdd * 2) - 1);
7      const uint centerIndex = (maxVertices / 2) - firstOrSecondPair;
8      vec4 position = vertices.pos[IN.vertexID];
```

```
9       vec4 color;
10      if (firstOrSecondPair == 0) {
11          position.y -= settings.thickness / 2;
12          color = vec4(1.0, 0.0, 0.0, 1.0);
13      } else {
14          position.y += settings.thickness / 2;
15          color = vec4(0.0, 0.0, 1.0, 1.0);
16      }
17      gl_MeshVerticesNV[centerIndex].gl_Position = state.MVP * position;
18      v_out[centerIndex].color = color;
19      for (uint i = 0; i < (maxVertices / 4); ++i) {
20          const int offsetFromCenter = int(i+1) * direction * 2;
21          const vec3 fieldVector = Field(position.xyz);
22          position.xyz += direction * fieldVector * settings.step;
23          gl_MeshVerticesNV[centerIndex + offsetFromCenter].gl_Position =
24              state.MVP * position;
25          v_out[centerIndex + offsetFromCenter].color = color;
26      }
27      const uint isFirst = uint(thread_id == 0);
28      const uint isSecond = uint(thread_id == 1);
29      const uint isThird = uint(thread_id == 2);
30      if (isFirst || isSecond || isThird) {
31          for (uint i = 0; i < (maxVertices-2); ++i) {
32              const uint index = 3*i + isSecond + 2*isThird;
33              const uint value = ((i+2)/2) * 2 * isFirst +
34                  i * isSecond + (((i+1)/2)*2+1) * isThird;
35              gl_PrimitiveIndicesNV[index] = value;
36          }
37      }
38      gl_PrimitiveCountNV = maxVertices - 2;
39  }
```

**Listing 14.** Implementacja funkcji `main()` shadera siatki generującego wstążki.

Po obliczeniu pozycji punktów kolejnym krokiem algorytmu jest wygenerowanie bu-

fora indeksów wierzchołków trójkątów dla procesora prymitywów. Dla uproszczenia obliczeń w tym etapie biorą udział tylko trzy wątki (0, 1 i 2), z których każdy zapisuje po jednym indeksie dla każdego trójkąta. Zgodnie z domyślną konwencją w OpenGL [37] wierzchołki trójkątów powinny być przedstawiane w kolejności przeciwnej do ruchu wskazówek zegara. Dla przykładu z Rysunku 7.4 poprawna jest następująca kolejność wierzchołków: $2,0,1-2,1,3-4,2,3-4,3,5-6,4,5-6,5,7\ldots$

Z tego wynika, że:

- wątek 0 powinien wpisywać do tablicy `gl_PrimitiveIndicesNV` pod indeksami $0,3,6,9,12,15\ldots$ wartości $2,2,4,4,6,6\ldots$,
- wątek 1 powinien wpisywać do tablicy `gl_PrimitiveIndicesNV` pod indeksami $1,4,7,10,13,16\ldots$ wartości $0,1,2,3,4,5\ldots$,
- wątek 2 powinien wpisywać do tablicy `gl_PrimitiveIndicesNV` pod indeksami $2,5,8,11,14,17\ldots$ wartości $1,3,3,5,5,7\ldots$.

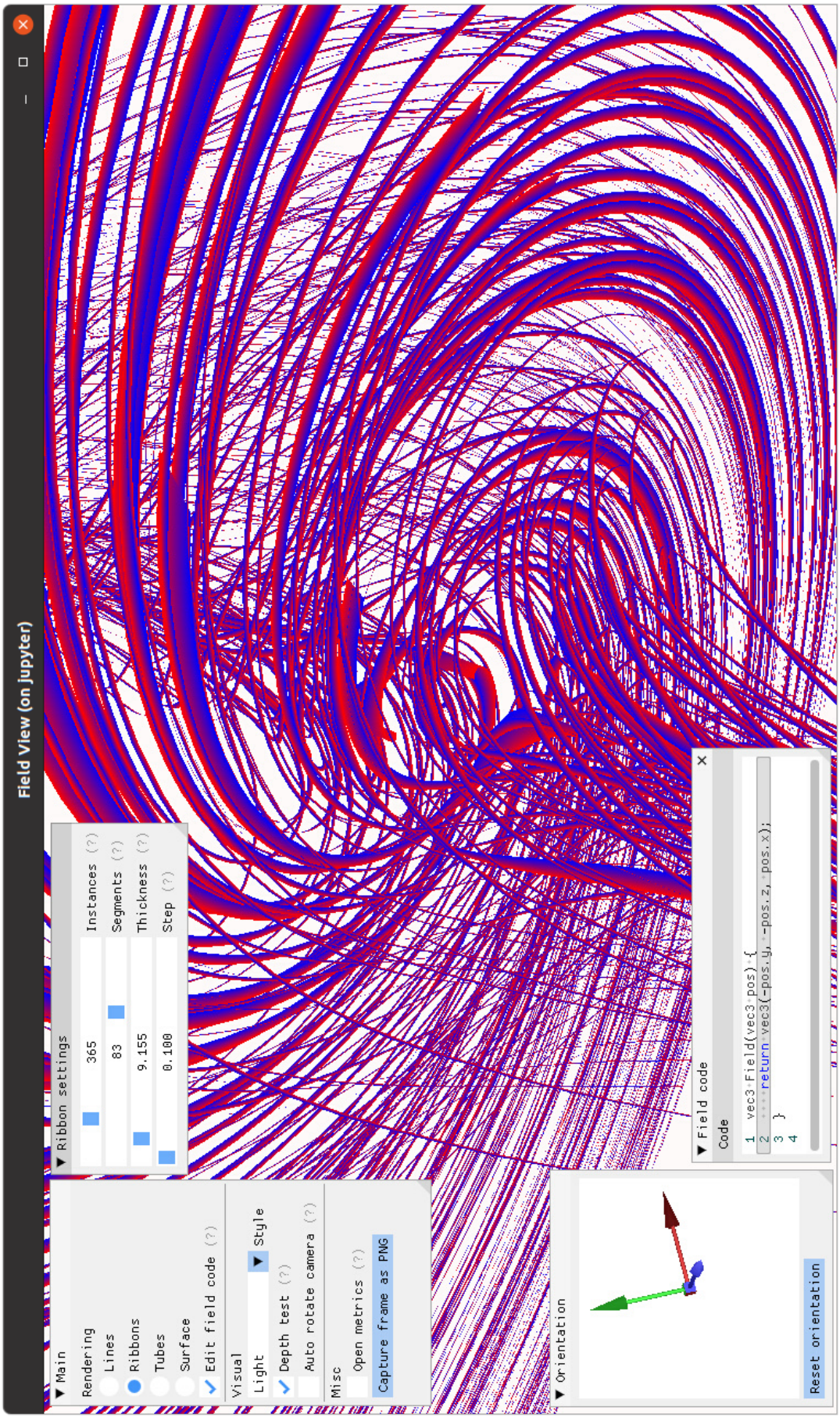
Algorytm wypełnia tablicę w pętli, gdzie iteratorem $i$ jest numer obecnie przetwarzanego trójkąta. Aby uzyskać odpowiednie ciągi liczb:

- wątek 0 powinien wpisywać pod indeksem $3*i$ wartość $2*\lfloor\frac{i+2}{2}\rfloor$,
- wątek 1 powinien wpisywać pod indeksem $3*i+1$ wartość $i$,
- wątek 2 powinien wpisywać pod indeksem $3*i+2$ wartość $2*\lfloor\frac{i+1}{2}\rfloor+1$.

Aby ponownie uniknąć wykorzystania instrukcji `if` wprowadziłem trzy dodatkowe stałe binarne $t_0, t_1, t_2$. Dla każdego wątku tylko jedna z nich ma wartość 1, dwie pozostałe mają wartość 0. Przy ich pomocy wzory cząstkowe przedstawione wyżej połączyłem w jeden wzór na indeks oraz jeden wzór na wartość, przedstawione poniżej:

$$idx = 3*i + t_1 + 2*t_2,$$
$$var = 2*\lfloor\frac{i+2}{2}\rfloor * t_0 + i*t_1 + (2*\lfloor\frac{i+1}{2}\rfloor+1)*t_2.$$

Tak obliczony indeks oraz wartość wstawiane są do tablicy, `gl_PrimitiveIndicesNV[idx] = var`. Ostatnim krokiem algorytmu jest ustawienie wartości zmiennej `gl_PrimitiveCountNV` na liczbę renderowanych trójkątów (równą dwukrotności liczby segmentów).



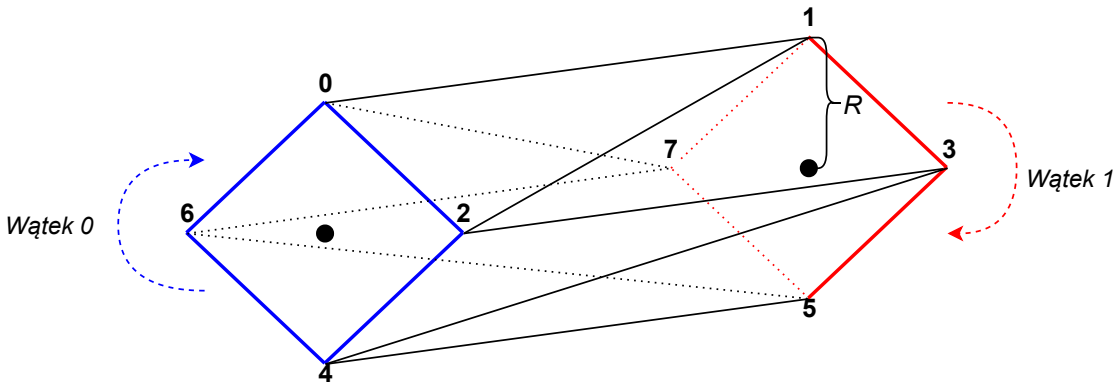

**Rys. 7.6.** Przykładowa wizualizacja pola wektorowego o wzorze $f = (-y, -z, x)$ za pomocą wstążek.

### 7.1.3. Tuby

Rysunek 7.6 przedstawia przykładową wizualizację wykorzystującą tuby dla modelu pola magnetycznego detektora ALICE. Ta metoda wizualizacji oferuje cztery parametry konfiguracyjne:

- *Instances*, który kontroluje liczbę tub na ekranie,
- *Segments*, który kontroluje liczbę segmentów, z których składa się pojedyncza tuba,
- *Circle LOD (Level Of Detail)*, który określa z ilu wierzchołków skład się przekrój tuby (im więcej tym bardziej przypomina on okrąg),
- *Radius*, który określa maksymalny promień tuby,
- *Step*, który kontroluje rozmiar kroku wykorzystywany w obliczaniu punktów linii przepływu.

Proces generowania tuby można przedstawić jako formę wytłaczania kształtu koła wzdłuż linii przepływu pola. W moim przypadku do wygenerowania bazowej linii przepływu wykorzystałem shader zadań (w podobny sposób jak opisałem to w Rozdziale 7.1.1). Wygenerowane wierzchołki współdzielone są z shaderami siatki za pomocą bloku `taskNV`. Shader zadań uruchamia liczbę shaderów siatki równą parametrowi liczby segmentów. Każdy shader siatki odpowiedzialny jest za wygenerowanie pojedynczego segmentu, czyli bryły podobnej do cylindra. Diagram algorytmu przedstawia Rysunek 7.8.




**Rys. 7.7.** Schemat budowy segmentu tuby w shaderze siatki.

Ponieważ shader siatki ma za zadanie generować trójkąty, wykorzystuje on w sumie

trzy wątki. Pierwszym krokiem algorytmu jest wygenerowanie wierzchołków tworzących dwie podstawy cylindra. Zajmują się tym dwa wątki (trzeci jest nieaktywny). Wierzchołki, które należy wygenerować, leżą na dwóch płaszczyznach prostopadłych do kierunku pola wektorowego w punktach przekazanych przez shader zadań. Aby obliczyć ich pozycje, potrzebujemy dwóch lokalnych układów współrzędnych, gdzie każdy z nich tworzony jest przez wektor styczny $\hat{t}$ (wzdłużny do wektora pola), oraz dwa wektory tworzące interesującą nas płaszczyznę: normalny $\hat{n}$ oraz binormalny $\hat{b}$.

Wektor styczny $\hat{t}$ równy jest znormalizowanemu wektorowi pola. Wektor normalny $\hat{n}$ równy jest znormalizowanemu wektorowi rotacji pola. Wektor binormalny $\hat{b}$ można obliczyć za pomocą iloczynu wektorowego $\hat{t}$ i $\hat{n}$. Powyższe obliczenia w formie równań są następujące:

$$\hat{t} = \frac{Field(\vec{pos})}{|Field(\vec{pos})|}, \hat{n} = \frac{Curl(\vec{pos})}{|Curl(\vec{pos})|}, \hat{b} = \frac{\hat{n} \times \hat{t}}{|\hat{n} \times \hat{t}|},$$

gdzie $\vec{pos}$ to pozycja punktu, *Field* to funkcja obliczająca wektor pola, a *Curl* to funkcja obliczajaca wektor rotacji. Tak obliczone wektory można wykorzystać w równaniu parametrycznym dla okręgu:

$$\vec{p} = \vec{pos} + \hat{b} * R * sin(\alpha) + \hat{n} * R * cos(\alpha),$$

gdzie $\vec{pos}$ to pozycja punktu, $R$ to promień okręgu (o wartościach od 0 do parametru *Radius*, wprost proporcjonalny do siły pola), a $\alpha$ to kąt. Schemat konstrukcji bryły przedstawia Rysunek 7.7. Liczba generowanych wierzchołków dla każdej podstawy (minimum 3, aby tuba była trójwymiarowym obiektem) zależy od parametru *Circle LOD*. Wierzchołki generowane są w pętli — podstawiając jako wartość kąta $\alpha = \frac{2\pi}{lod} * i$, gdzie $i$ jest iteratorem pętli, a $lod$ to wartość parametru *Circle LOD*. Wierzchołki pierwszej podstawy cylindra zapisywane są pod parzyste indeksy w tablicy wierzchołków. Wierzchołki drugiej podstawy zapisywane są pod nieparzyste indeksy.

```
1 void main() {
2     const uint thread_id = gl_LocalInvocationID.x;
3     const int direction = int((thread_id * 2) - 1);
```

```
4       const uint centerIndex = ((settings.segmentsCount + 1) / 2);
5       vec4 position = vertices.pos[gl_WorkGroupID.x];
6       vec3 fieldVector = Field(position.xyz);
7       OUT.vertices[centerIndex] = position;
8       for (uint i = 0; i < centerIndex; ++i) {
9           const int offsetFromCenter = int(i+1) * direction;
10          position.xyz += direction * fieldVector * settings.step;
11          fieldVector = Field(position.xyz);
12          OUT.vertices[centerIndex + offsetFromCenter] = position;
13      }
14      gl_TaskCountNV = settings.segmentsCount;
15  }
```

**Listing 15.** Implementacja funkcji `main()` shadera zadań generującego tuby.

```
1   void main() {
2       const float angle_chunk = 2 * PI / float(settings.circleVertices);
3       const uint thread_id = gl_LocalInvocationID.x;
4       if (thread_id == 0 || thread_id == 1) {
5           const uint vertex_index = gl_WorkGroupID.x;
6           const vec4 position = IN.vertices[vertex_index + thread_id];
7           const vec3 fieldVector = Field(position.xyz);
8           const vec4 tangent = vec4(normalize(fieldVector), 0.0f);
9           const vec4 normal = vec4(normalize(Curl(position.xyz)), 0.0f);
10          const vec4 binormal =
11              vec4(normalize(cross(normal.xyz, tangent.xyz)), 0.0f);
12          const float radius = length(fieldVector) * settings.radius;
13          for (uint i = 0; i < settings.circleVertices; ++i) {
14              const float angle = angle_chunk * i;
15              const vec4 pointPosition = position +
16                  binormal*radius*sin(angle) + normal*radius*cos(angle);
17              if (thread_id == 0) {
18                  v_out[2 * i + thread_id].color = vec4(1.0, 0.0, 0.0, 1.0);
19              } else {
20                  v_out[2 * i + thread_id].color = vec4(0.0, 0.0, 1.0, 1.0);
```

```
21              }
22              gl_MeshVerticesNV[2 * i + thread_id].gl_Position =
23                  state.MVP * pointPosition;
24          }
25      }
26      const uint isFirst = uint(thread_id == 0);
27      const uint isSecond = uint(thread_id == 1);
28      const uint isThird = uint(thread_id == 2);
29      for (uint i = 0; i < 2*settings.circleVertices-1; ++i) {
30          const uint index = 3*i + isSecond + 2*isThird;
31          const uint value = i * isFirst +
32              ((i+2)/2)*2 * isSecond + (((i+1)/2)*2+1) * isThird;
33          gl_PrimitiveIndicesNV[index] = value;
34      }
35      const uint value1 = (2 * settings.circleVertices - 2) * isFirst +
36          (2 * settings.circleVertices - 1) * isThird;
37      const uint value2 = (2 * settings.circleVertices - 1) * isFirst +
38          1 * isThird;
39      gl_PrimitiveIndicesNV[3 * (2 * settings.circleVertices - 2) +
40          isSecond + 2 * isThird] = value1;
41      gl_PrimitiveIndicesNV[3 * (2 * settings.circleVertices - 1) +
42          isSecond + 2 * isThird] = value2;
43      gl_PrimitiveCountNV = 2 * settings.circleVertices;
44  }
```

**Listing 16.** Implementacja funkcji `main()` shadera siatki generującego tuby.

Kolejnym krokiem algorytmu jest wygenerowanie bufora indeksów dla procesora prymitywów. Do tego zadania wykorzystywane są wszystkie trzy wątki. Odbywa się to w taki sam sposób jak w rozważaniach z Rozdziału 7.1.2 dla wszystkich trójkątów oprócz dwóch ostatnich, które "domykają" segment tuby. Indeksy ich wierzchołków nie pasują do użytego wcześniej ciągu. Dla załączonego przykładu na Rysunku 7.7 są to trójkąty $0, 6, 7 - 0, 7, 1$. Dla nich indeksy muszą zostać wygenerowane oddzielnym wzorem:

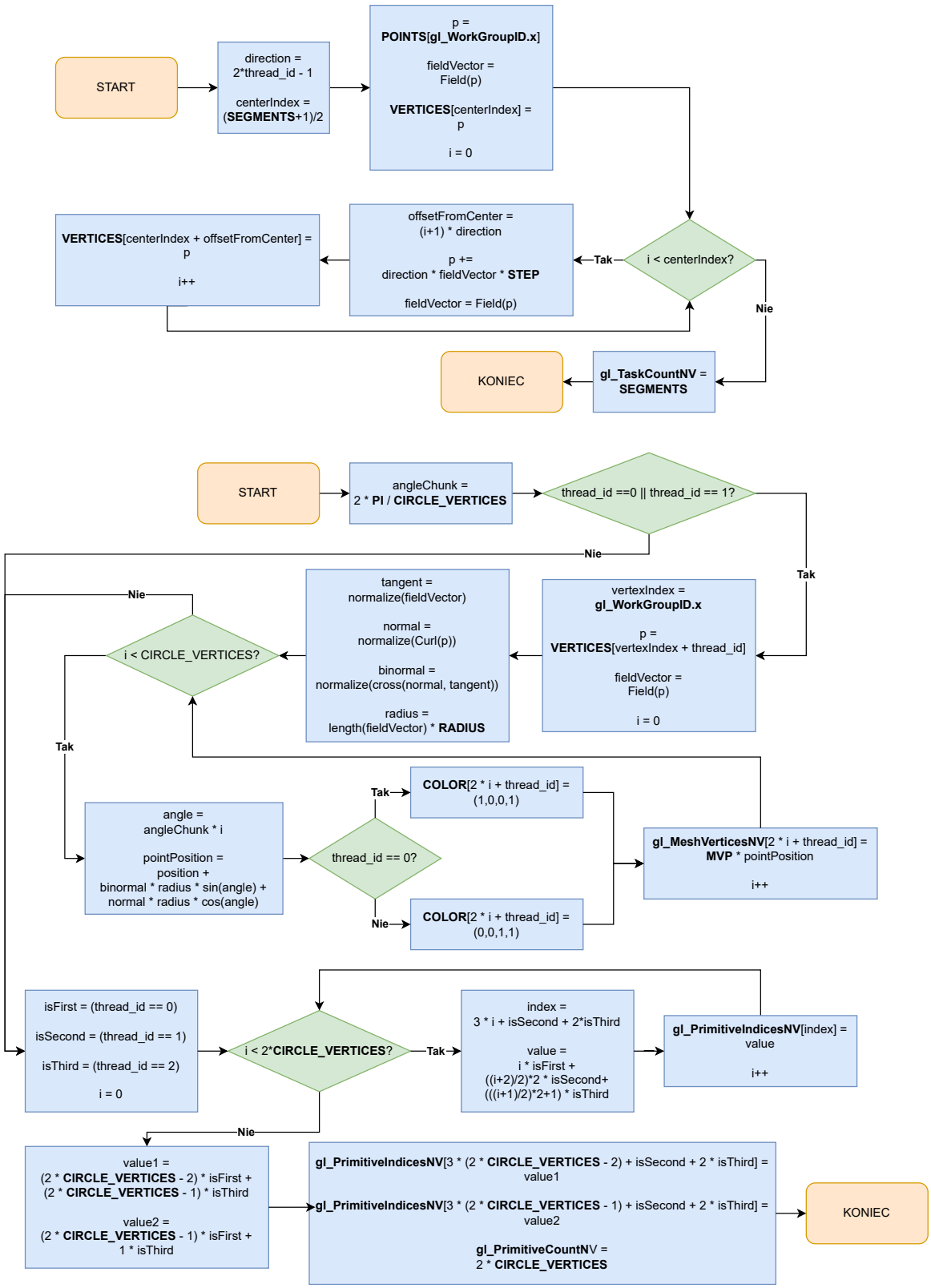
$$idx1 = 3 * (2 * lod - 2) + t_1 + 2 * t_2,$$

$$var1 = (2 * lod - 2) * t_1 + (2 * lod - 1) * t_2,$$

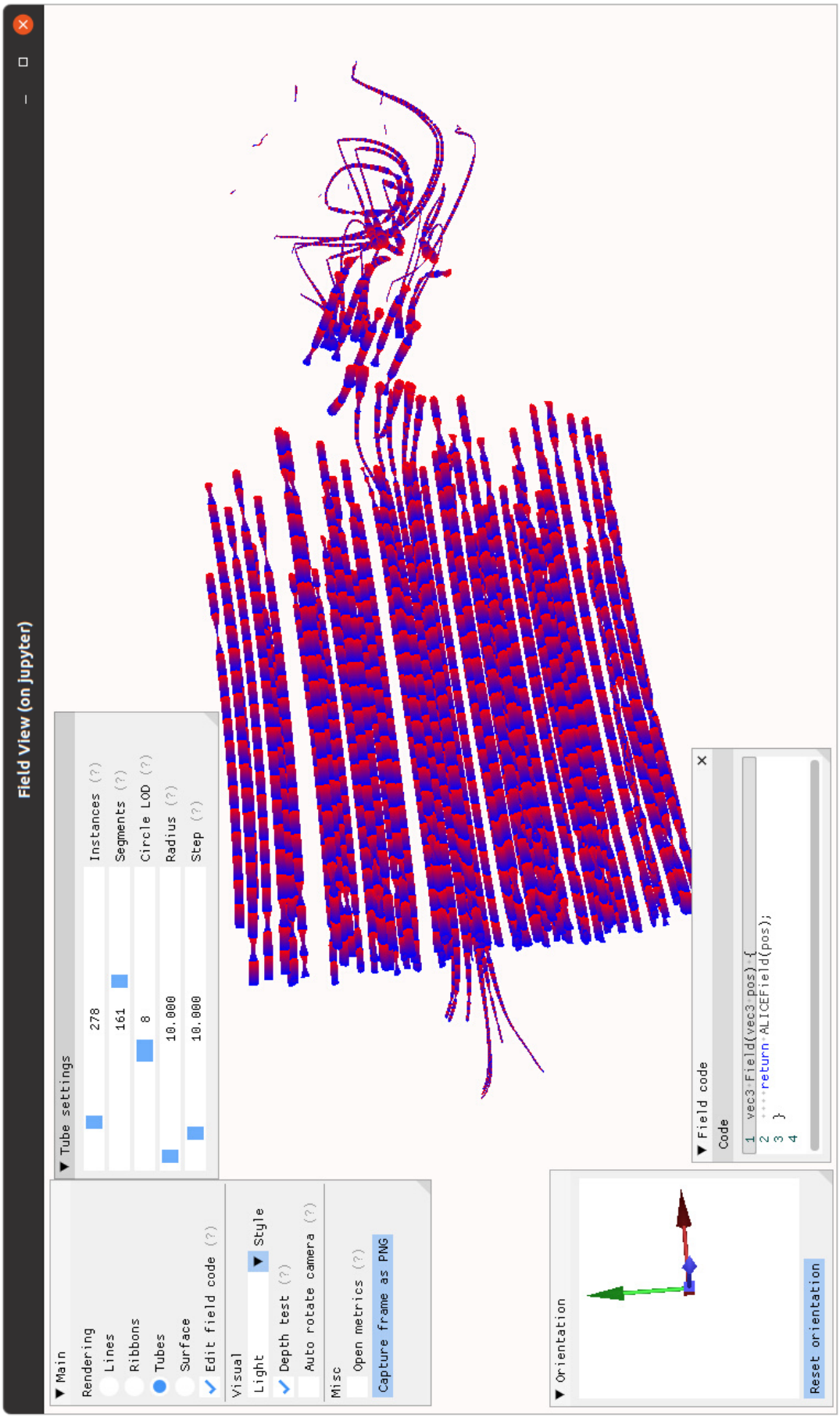
$$idx2 = 3 * (2 * lod - 1) + t_1 + 2 * t_2,$$

$$var2 = (2 * lod - 1) * t_1 + t_2,$$

gdzie parametr ***lod*** to liczba wierzchołków przekroju tuby (czyli *Circle LOD*). Wygenerowane wartości wpisywane są do tablicy `gl_PrimitiveIndicesNV`: pod `idx1` wartość `var1`, pod `idx2` wartość `var2`. Ostatnim krokiem algorytmu jest ustawienie wartości zmiennej `gl_PrimitiveCountNV` na liczbę renderowanych trójkątów (równą dwukrotności liczby wierzchołków przekroju tuby).




**Rys. 7.8.** Diagram algorytmu generowania tub w shaderach zadań (górny) oraz siatki (dolny). Zmienne, których nazwy zostały pogrubione to zmienne globalne.




**Rys. 7.9.** Przykładowa wizualizacja pola wektorowego detektora ALICE za pomocą tub.

### 7.1.4. Powierzchnia

Rysunek 7.12 przedstawia przykładową wizualizację wykorzystującą powierzchnię dla modelu pola magnetycznego detektora ALICE. Ta metoda wizualizacji oferuje pięć parametrów konfiguracyjnych:

- Funkcję `curiosityLevel()`, którą implementuje użytkownik. Ma ona za zadanie określić poziom ciekawości pola (w postaci wartości skalarnej) w danym punkcie przestrzeni,
- *Threshold*, który określa granicę poziomu ciekawości, poniżej którego reprezentacja pola jest upraszczana,
- Punkty *P1*, *P2*, *P3*, *P4*, które określają granice wizualizowanej powierzchni,
- *Offset*, który pozwala przesunąć płaszczyznę wzdłuż każdej z osi bez indywidualnej modyfikacji pozycji punktów.

Wizualizacja za pomocą płaszczyzny to opracowana przeze mnie metoda łącząca aspekty metody geometrycznej i opartej o cechy pola. Działa ona w następujący sposób:

1. Użytkownik definiuje cztery punkty w przestrzeni, pomiędzy którymi rozłożona będzie wizualizowana powierzchnia.
2. Algorytm w równomierny sposób rozkłada punkty na płaszczyźnie. Dla każdego punktu wywołuje zdefiniowaną przez użytkownika funkcję `curiosityLevel()`, za pomocą której szacowany jest poziom zainteresowania widza danym wycinkiem płaszczyzny.
3. Rozpoczyna się proces upraszczania reprezentacji pola. Wartości poziomu zainteresowania każdych czterech sąsiadujących obszarów są uśredniane, a następnie porównywane z poziomem *Threshold*. Jeżeli obszary są mniej "ciekawe" niż ustalona granica, łączone są w jeden większy. Jeżeli bardziej, to pozostawiane są bez zmian.
4. Poprzedni krok jest powtarzany, za każdym razem porównywane są coraz większe obszary.

5. Powtarzanie kończy się, kiedy algorytmowi zostają do porównania cztery obszary, które sumarycznie pokrywałyby obszar całej płaszczyzny.
6. Wynik działania algorytmu przedstawiany jest w postaci prostokątów o rozmiarach odpowiadających obszarom pozostałym po procesie upraszczania.

W moim algorytmie za kroki 2–5 odpowiedzialny jest shader zadań, natomiast krok 6 wykonuje shader siatki (działanie shaderów przedstawiają diagramy na Rysunkach 7.10 oraz 7.11). Shader zadań do działania potrzebuje dwóch tablic: `samples`, w której tymczasowo przechowywane są wartości "ciekawości" pola oraz `keep_levels`, w której przechowywana jest informacja, czy dany obszar na danym poziomie został uproszczony. Tablice `mesh_index` oraz `mesh_level` służą do przekazania do shadera siatki informacji, które oraz jakiego rozmiaru obszary powinny zostać zwizualizowane. Tablica `keep_levels` musi być zerowana na zewnątrz shadera na początku rysowania każdej klatki animacji.

Shader zadań wykorzystuje tyle wątków, ile pobieranych jest próbek pola. Praktycznym czynnikiem limitującym liczbę próbek jest maksymalna liczba wątków na shader wspierana przez kartę — na testowanych kartach graficznych (Rozdział 7.3) limit wątków to 256, co przekłada się na 16 x 16 próbek. Pierwszym zadaniem shadera jest inicjalizacja tablicy `samples`. Wymaga to od każdego wątku obliczenia odpowiedniego punktu na płaszczyźnie na podstawie punktów *P1*, *P2*, *P3*, *P4* oraz numeru wątku `thread_id`. Odbywa się to w dwóch etapach. Pierwszym krokiem jest zmapowanie jednowymiarowej wartości, jaką jest numer wątku, na dwuwymiarową współrzędną w taki sposób, aby równomiernie próbkować całą powierzchnię. Można to zrobić za pomocą jednej z krzywych wypełniających przestrzeń, np. krzywej Peana, Hilberta [136] lub za pomocą kodu Mortona [137]. Wybrałem ten ostatni ze względu na prostotę jego implementacji dla zastosowań informatycznych (wymaga jedynie kilku operacji przesuwania, koniunkcji oraz alternatywy rozłącznej na bitach). Z indeksu `thread_id` za jego pomocą wyliczana jest para współrzędnych (`mX`, `mY`). Następnie za pomocą trzech operacji interpolacji liniowej (w sposób identyczny do procesu filtrowania biliniowego) z punktów krańcowych płaszczyzny *P1*, *P2*, *P3*, *P4* oraz współrzędnych (`mX`, `mY`) uzyskiwany jest żądany punkt na płaszczyźnie, czyli `center`. Punkt ten przekazywany jest

do funkcji `curiosityLevel()`, którą implementuje użytkownik, a zwracana wartość zapisywana jest pod odpowiedni indeks w tablicy `samples`.

Po inicjalizacji tablicy próbek algorytm przetwarza po kolei poziomy upraszczania reprezentacji. Dla 256 próbek są to poziomy od 0 do 3. Do obliczeń na poziomie 0 wykorzystywane są wszystkie 256 wątki, na poziomie 1 wykorzystywane są 64 wątki, na poziomie 2 wykorzystywanych jest 16 wątków, a na ostatnim poziomie 4 wątki. Wątki wywołują funkcję `process_level`, która wykonuje następujące kroki:

1. Jeżeli przetwarzany jest inny poziom niż zerowy:
   a) Sprawdź w tablicy `keep_levels` czy wszystkie cztery analizowane obecnie obszary zostały uproszczone w poprzedniej iteracji (wartości są zerami).
   b) Jeśli tak, to kontynuuj od punktu 2.
   c) Jeśli jeden lub więcej nie zostało uproszczonych, to tej czwórki na obecnym poziomie nie można uprościć. Koniec.
2. Oblicz średnią wartość ciekawości analizowanych obszarów (ciekawość przechowywana jest na pozycji pierwszego w kolejności obszaru podrzędnego).
3. Jeżeli wypada ona poniżej wartości *Threshold*, to:
   a) Zaktualizuj wartość ciekawości pierwszego w kolejności analizowanego obszaru na obliczoną średnią w tablicy `samples`; wyzeruj ją dla pozostałych.
   b) Cztery analizowane obszary nie będą już istnieć. Nadpisz w tablicy `keep_levels` informację o nich (wstaw zera).
   c) Zapisz w tablicy `keep_levels` informację, że istnieje obszar powstały po obecnym uproszczeniu (wstaw jedynkę). Koniec.
4. Jeżeli wypada ona powyżej wartości *Threshold* i przetwarzany jest poziom inny niż zerowy, to koniec.
5. Jeżeli wypada ona powyżej wartości *Threshold* i przetwarzany jest poziom zerowy (początek algorytmu), to odpowiednie pozycje w tablicy `keep_levels` należy w tym momencie zainicjować. Umieść w niej informację, że cztery analizowane obszary istnieją (wstawiając jedynki).

Ostatnim krokiem algorytmu, który wykonuje jeden wątek, jest wygenerowanie

informacji dla shadera siatki o tym, które i jakiego rozmiaru obszary należy zwizualizować (tablice `mesh_index` oraz `mesh_level`). Aby tego dokonać wystarczy zrobić przegląd wszystkich elementów tablicy `keep_levels` w poszukiwaniu jedynek. Zliczana jest również liczba napotkanych jedynek — będzie to liczba shaderów siatki, które należy uruchomić (zapisywana do zmiennej `gl_TaskCountNV`).

Zadaniem shadera siatki jest wygenerowanie pojedynczego czworoboku, który odpowiada jednemu obszarowi. Wykorzystuje on cztery wątki — po jednym wątku dla każdego wierzchołka. Pierwszym krokiem algorytmu jest wyciągnięcie informacji z tablic `mesh_index` oraz `mesh_level`, adresując je numerem uruchomionego shadera (`gl_WorkGroupID.x`). Następnie, w podobny sposób jak w shaderze zadań, indeks z `mesh_index` oraz poziom uproszczenia z `mesh_level` zamieniany jest na współrzędne za pomocą kodu Mortona. Aby uzyskać cztery wierzchołki, należy rozsunąć otrzymany punkt w czterech kierunkach: $(-1,-1),(-1,1),(1,1),(1,-1)$, co realizuje każdy z wątków oddzielnie. Otrzymane wierzchołki poddawane są (tak jak w shaderze zadań) interpolacji pomiędzy punktami krańcowymi powierzchni oraz zapisane do tablicy wierzchołków (pod indeksami zgodnymi z numerami wątków).

Kolejnym krokiem algorytmu jest wygenerowanie bufora indeksów dla procesora prymitywów. Ponieważ czworobok to dwa trójkąty, w tym etapie wykorzystałem tylko trzy wątki. Biorąc pod uwagę kolejność wierzchołków opisaną wyżej, poprawna kolejność indeksów to $0,1,2-0,2,3$. Te ciągi można wygenerować (unikając rozgałęzień tak jak w poprzednich przypadkach) następującymi wzorami:

$$idx1 = t_{id},$$
$$var1 = t_{id},$$
$$idx2 = 3 + t_{id},$$
$$var2 = t_{id} + t_{1\vee 2},$$

gdzie $t_{id}$ to numer wątku, a $t_{1\vee 2}$ to stała binarna określająca czy obecny wątek to wątek 1 lub 2. Wygenerowane wartości wpisywane są do tablicy `gl_PrimitiveIndicesNV`:

pod `idx1` wartość `var1`, pod `idx2` wartość `var2`. Ostatnim krokiem algorytmu jest ustawienie wartości zmiennej `gl_PrimitiveCountNV` na liczbę renderowanych trójkątów, czyli 2.

```
1  void process_level(float threshold, uint level, uint i) {
2      const uint coord = i * (1 << ((level + 1) << 1));
3      const uint currentLevelIndex = levels_index(level);
4      const uint nextLevelIndex = levels_index(level + 1);
5      bool stopConcatenate = false;
6
7      if (level != 0) {
8          // don't try to concatenate if one
9          // or more smaller squares were not concatenated
10         for (uint j = 0; j < 4; ++j) {
11             const uint idx = currentLevelIndex + i * 4 + j;
12             if (keep_levels.keep_levels[idx] == 0) {
13                 stopConcatenate = true;
14                 break;
15             }
16         }
17     }
18
19     if (!stopConcatenate) {
20         float average = 0.0f;
21         const uint multiplier = 1 << (level << 1);
22         for (uint j = 0; j < 4; ++j) {
23             average += samples.samples[coord + j * multiplier];
24         }
25         average /= 4;
26         // not attractive enough, concatenate
27         if (average < threshold) {
28             samples.samples[coord] = average;
29             uint levelIdx = i * multiplier;
30             if (level != 0) {
```

```
31                  const uint idx = currentLevelIndex + levelIdx;
32                  keep_levels.keep_levels[idx] = 0;
33              } else {
34                  levelIdx = coord;
35              }
36              for (uint j = 1; j < 4; ++j) {
37                  samples.samples[coord + j * multiplier] = 0;
38                  const uint idx = currentLevelIndex + levelIdx + j;
39                  keep_levels.keep_levels[idx] = 0;
40              }
41              const uint idx = nextLevelIndex + i;
42              keep_levels.keep_levels[idx] = 1;
43          // nothing to do if attractive enough
44          // except for first level - mark squares for keeping
45          } else if (level == 0) {
46              for (uint j = 0; j < 4; ++j) {
47                  const uint idx = currentLevelIndex + coord + j;
48                  keep_levels.keep_levels[idx] = 1;
49              }
50          }
51      }
52  }
53
54  // find curiosity level for given sample position
55  float curiosity_level_for(uint sample_idx) {
56      const uint mX = morton(sample_idx);
57      const uint mY = morton(sample_idx >> 1);
58      const float frac = 1.0f / (2.0f * RESOLUTION);
59      const float xFrac = frac * (mX * 2 + 1);
60      const float yFrac = frac * (mY * 2 + 1);
61      const vec4 vl = mix(vertices.pos[0], vertices.pos[1], yFrac);
62      const vec4 vr = mix(vertices.pos[2], vertices.pos[3], yFrac);
63      const vec3 center = mix(vl, vr, xFrac).xyz;
64      // provided by the user
```

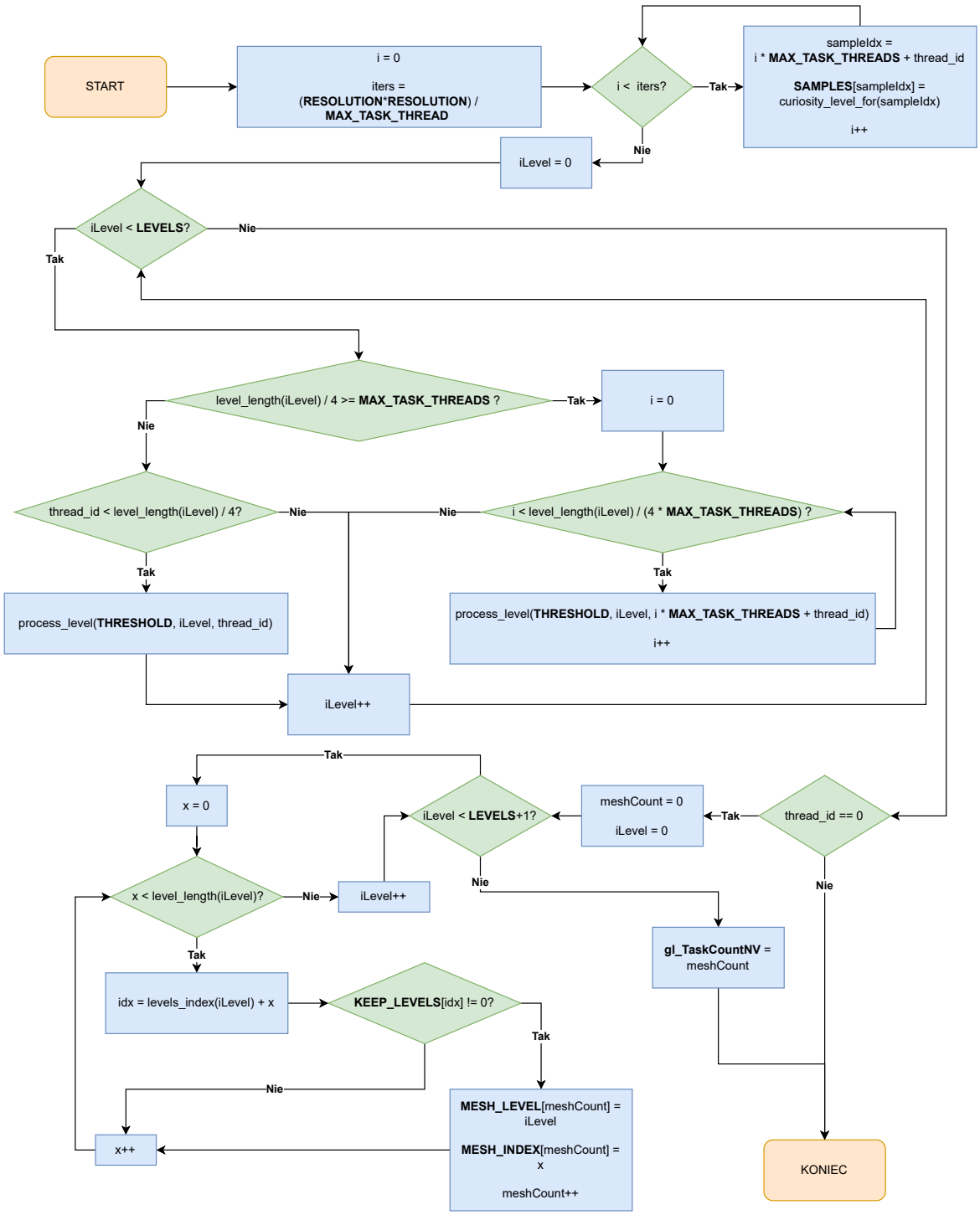
```
65      return curiosityLevel(center);
66  }
67
68  void main() {
69      const uint thread_id = gl_LocalInvocationID.x;
70      const uint iters = (RESOLUTION*RESOLUTION)/MAX_TASK_THREADS;
71      for (uint i = 0; i < iters; ++i) {
72          const uint sampleIdx = i * MAX_TASK_THREADS + thread_id;
73          samples.samples[sampleIdx] = curiosity_level_for(sampleIdx);
74      }
75
76      const float threshold = settings.threshold;
77      for (uint iL = 0; iL < LEVELS; ++i_l) {
78          if ((level_length(iL)/4) >= MAX_TASK_THREADS) {
79              const uint itersLevel = level_length(i_l)/(4*MAX_TASK_THREADS);
80              for (uint i = 0; i < itersLevel; ++i) {
81                  process_level(threshold, iL, i*MAX_TASK_THREADS + thread_id);
82              }
83          } else if (thread_id < (level_length(iL)/4)) {
84              process_level(threshold, iL, thread_id);
85          }
86      }
87
88      if (thread_id == 0) {
89          uint meshCount = 0;
90          for(uint iL = 0; iL < LEVELS+1; ++iL) {
91              for(uint x = 0; x < level_length(iL); ++x) {
92                  const uint idx = levels_index(iL) + x;
93                  if (keep_levels.keep_levels[idx] != 0) {
94                      mesh_level.mesh_level[meshCount] = iL;
95                      mesh_index.mesh_index[meshCount] = x;
96                      meshCount++;
97                  }
98              }
```

```
99          }
100         gl_TaskCountNV = meshCount;
101     }
102 }
```

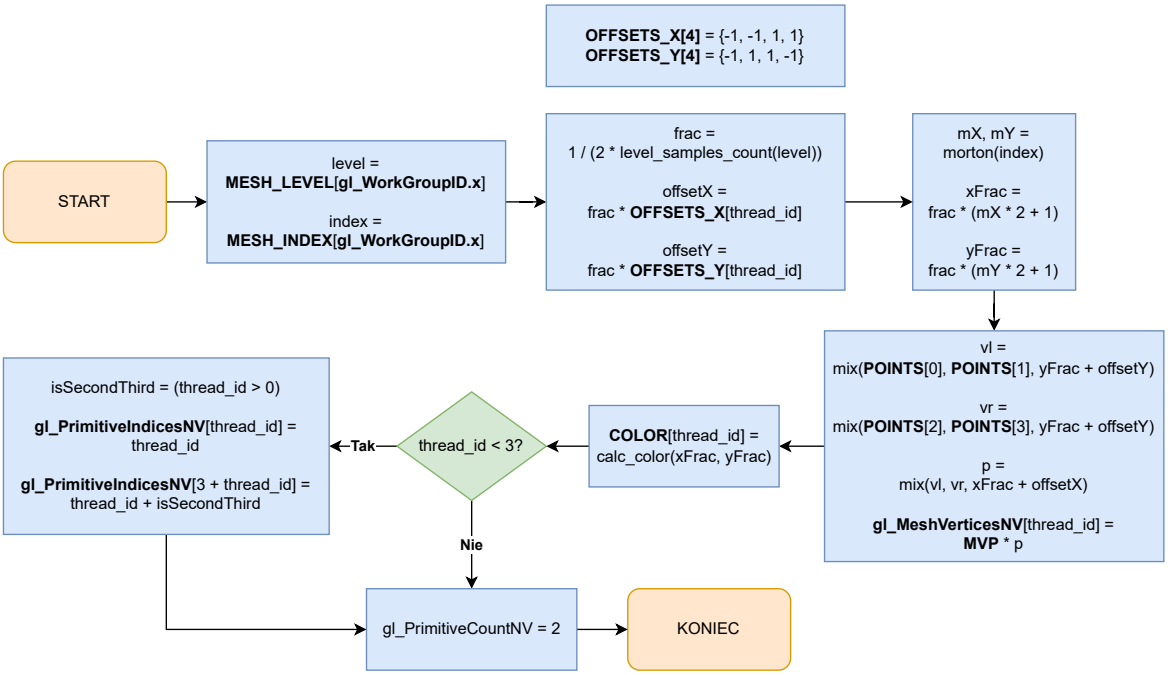
**Listing 17.** Implementacja funkcji `process_level()`, `curiosity_level_for()` oraz `main()` shadera zadań generującego powierzchnię.

```
1  const float offsets_x[] = {-1.0f, -1.0f, 1.0f,  1.0f};
2  const float offsets_y[] = {-1.0f,  1.0f, 1.0f, -1.0f};
3  void main() {
4      const uint thread_id = gl_LocalInvocationID.x;
5      const uint group_id = gl_WorkGroupID.x;
6      const uint level = mesh_level.mesh_level[group_id];
7      const uint index = mesh_index.mesh_index[group_id];
8      const float frac = 1.0f / (2.0f * level_samples_count(level));
9      const uint mX = morton(index);
10     const uint mY = morton(index >> 1);
11     const float offsetX = frac * offsets_x[thread_id];
12     const float offsetY = frac * offsets_y[thread_id];
13
14     const vec4 vl = mix(vertices.pos[0], vertices.pos[1], yFrac + offsetY);
15     const vec4 vr = mix(vertices.pos[2], vertices.pos[3], yFrac + offsetY);
16     gl_MeshVerticesNV[thread_id].gl_Position =
17         state.MVP * mix(vl, vr, xFrac + offsetX)
18     // "rainbow" color pallete based on position on the surface
19     v_out[thread_id].color = calc_color(xFrac, yFrac);
20     if (thread_id < 3) {
21         const uint isSecondThird = uint(thread_id > 0);
22         gl_PrimitiveIndicesNV[thread_id] = thread_id;
23         gl_PrimitiveIndicesNV[3 + thread_id] = thread_id + isSecondThird;
24     }
25     gl_PrimitiveCountNV = 2;
26 }
```

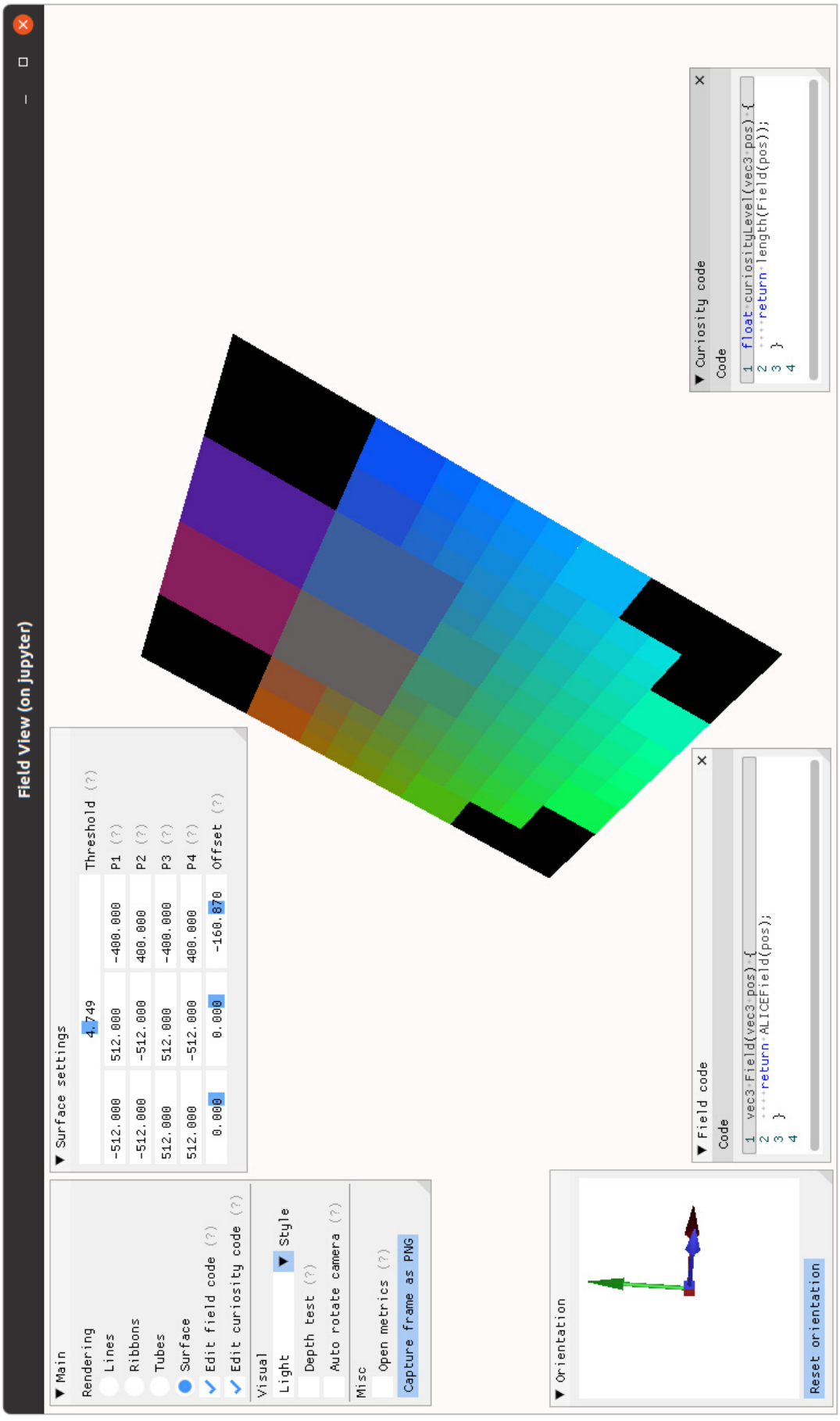
**Listing 18.** Implementacja funkcji `main()` shadera siatki generującego pojedynczy czworobok.




**Rys. 7.10.** Diagram algorytmu generowania listy obszarów w shaderze zadań. Zmienne, których nazwy zostały pogrubione to zmienne globalne.




**Rys. 7.11.** Diagram algorytmu generowania czworoboków w shaderze siatki. Zmienne, których nazwy zostały pogrubione to zmienne globalne.




**Rys. 7.12.** Przykładowa wizualizacja pola wektorowego detektora ALICE za pomocą powierzchni.

### 7.2. Metodologia badań

Do testów wydajnościowych wykorzystałem model pola magnetycznego detektora ALICE. Model ten przetestowałem we wszystkich czterech wariantach implementacji, podobnie jak Rozdziale 6. Większość parametrów metod wizualizacji określiłem arbitralnie, starając się uzyskać w każdym wypadku obraz zbliżony (pod względem pokrycia elementami graficznymi obu elektromagnesów modelu) do Rysunku 7.9. Ustawienia te to:

- linie — *Segments*: 61, *Step*: 10.0,
- wstążki — *Segments*: 61, *Step*: 10.0, *Thickness*: 18.0,
- tuby —- *Segments*: 61, *Step*: 10.0, *Radius*: 10.0, *Circle LOD*: 8,
- powierzchnia — `curiosityLevel()` jako długość wektora pola, powierzchnia jako kwadrat zaczynający się w punkcie ($-512\,\text{cm}$, $-512\,\text{cm}$, $-100\,\text{cm}$), a kończący w punkcie ($512\,\text{cm}$, $512\,\text{cm}$, $100\,\text{cm}$), *Threshold*: 1.0.

Parametrem, który zmieniany był w trakcie eksperymentów była liczba elementów graficznych (*Instances*): 50, 100, 250, 500, 750, 1000, 1250, 1500, 1750, 2000, 2250, 2500, 3000, 3500, 4000, 5000, 6000, 7000, 9000, 10000. Wyjątkiem jest metoda z powierzchnią, która zakłada rysowanie tylko jednego elementu graficznego.

W przeciwieństwie do poprzednich rozważań (Rozdział 5.2.2 i 6.2.2) program ten nie odczytuje żadnych danych z karty graficznej. Z tego powodu pomiar czasu wykonania samej funkcji `glDrawMeshTasksNV()` byłby niemiarodajny ze względu na asynchroniczność procedur rysujących w OpenGL [37]. Aby rozwiązać ten problem wykorzystałem funkcję `glFlush()`, za pomocą której można wymusić natychmiastowe narysowanie klatki oraz opróżnienie bufora zadań dla karty graficznej. Tak jak w poprzednich rozważaniach wiąże się to z nieoptymalnym wykorzystaniem zasobów karty — mierzona jest wydajność w najgorszym przypadku. Czas, podobnie jak wcześniej, zmierzony został za pomocą zegara o największej precyzji dostępnej w bibliotece standardowej C++ (`std::chrono::high_resolution_clock`). Tak jak w poprzednio przeprowadzonych eksperymentach, przedstawione niżej wyniki to średnie z 10 pomiarów.

```
1  const auto start = std::chrono::high_resolution_clock::now();
```

```
2
3  gl::glDrawMeshTasksNV(...);
4  gl::glFlush();
5
6  const auto end = std::chrono::high_resolution_clock::now();
```

**Listing 19.** Schemat pomiaru czasu wykonania dla algorytmów na karcie graficznej.

### 7.3. Wyniki

Testy przeprowadziłem na następujących maszynach:

- Komputer stacjonarny — NVIDIA GeForce RTX 2080 Ti (11 GiB VRAM), AMD Ryzen Threadripper 1920X 3.5 GHz,
- Serwer — NVIDIA GeForce RTX 3060 (12 GiB VRAM), AMD Ryzen 7 3700X 4.4 GHz.

Na komputerze stacjonarnym zainstalowany był system Ubuntu 20.04.4 LTS oraz sterownik graficzny NVIDIA 470.103. Na maszynie serwerowej zainstalowany był system 20.04.1 LTS oraz sterownik graficzny NVIDIA 470.141.

Rysunek 7.13 przedstawia średni czas rysowania klatki w milisekundach dla wszystkich zaproponowanych metod wizualizacji (oprócz metody z powierzchnią), przetestowanych z wszystkimi implementacjami modelu pola magnetycznego na obu kartach graficznych. Można zauważyć, że wykresy dla wszystkich wariantów są stosunkowo płaskie. Oznacza to, że zaproponowane algorytmy shaderów siatki są na tyle wydajne, że obie karty wciąż mają zapas mocy obliczeniowej nawet przy rysowaniu 10000 elementów graficznych (co jest liczbą przekraczającą praktyczne zastosowanie ze względu na nieczytelność obrazu). Pierwsza połowa wszystkich wykresów (czyli dla liczby elementów graficznych mniejszej niż ok. 4000) zawiera większe wahania wyników niż druga połowa (od ok. 4000 do 10000). Jest to prawdopodobnie spowodowane faktem, że dla małych rozmiarowo zadań na ostateczny wynik większy wpływ ma sposób ich rozłożenia na karcie graficznej — zajmuje się tym sterownik w nieznany (a więc z punktu widzenia eksperymentów do pewnego stopnia losowy) sposób.

Tabela 7.1 przedstawia uzyskaną liczbę klatek na sekundę dla maksymalnej (10000) testowanej liczby elementów graficznych dla wszystkich kombinacji metod wizualizacji oraz implementacji modelu pola. We wszystkich przypadkach metoda z buforem (*Shader Storage Buffer*) okazała się najszybsza. Pozostałe trzy metody w rankingu wydajności zamieniają się miejscami w zależności od konkretnego przypadku. Osiągnięte wyniki kilkukrotnie przekraczają konieczną do w pełni komfortowego korzystania z aplikacji przez użytkownika (60 klatek na sekundę) granicę szybkości generowania klatek. Zestawienie wyników pokazuje również, że karta RTX 2080 Ti jest wydajniejsza od RTX 3060, co zgadza się z zewnętrznym rankingiem kart graficznych [138]. Potwierdza to, że eksperymenty zostały przeprowadzone przez mnie w prawidłowy sposób.

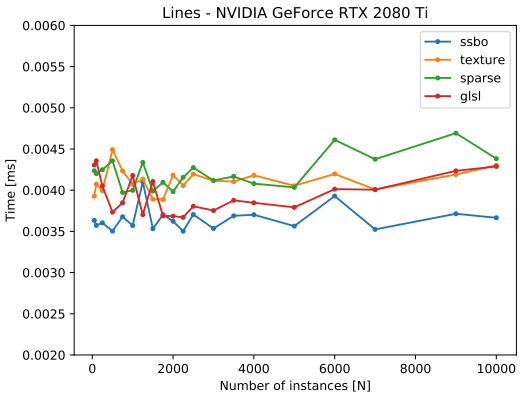
| | RTX 2080 Ti | | | RTX 3060 | | |
|---|---|---|---|---|---|---|
| Algorytm | Linie | Wstążki | Tuby | Linie | Wstążki | Tuby |
| SSBO | 273 | 251 | 260 | 221 | 245 | 238 |
| Tekstura 3D | 232 | 220 | 227 | 235 | 217 | 216 |
| Tekstura rzadka 3D | 228 | 231 | 234 | 213 | 221 | 215 |
| GLSL | 233 | 270 | 221 | 214 | 226 | 221 |

**Tabela 7.1.** Liczba klatek na sekundę dla każdej metody wizualizacji oraz implementacji pola magnetycznego na karcie graficznej NVIDIA GeForce RTX 2080 Ti oraz karcie graficznej NVIDIA GeForce RTX 3060 przy 10000 elementów graficznych.

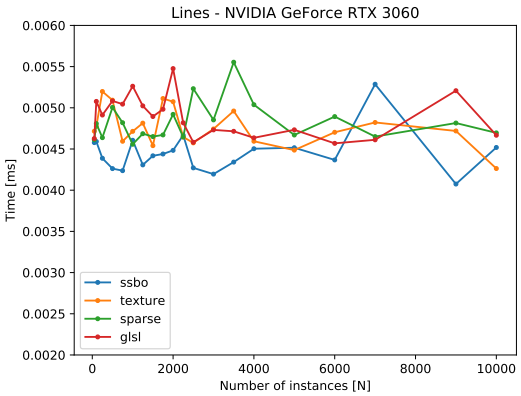
Wyniki wydajności dla metody z powierzchnią przedstawia Tabela 7.2. Dla niej, podobnie jak dla poprzednich, *Shader Storage Buffer* okazał się szybszy o ok. 5%. Wynik pozostałych metod jest zbliżony do siebie. Metoda *Shader Storage Buffer* osiągnęła 251 klatek na sekundę dla karty 2080, natomiast 222 klatek na sekundę dla karty 3060. Metody z teksturą osiągnęły 225 – 226 klatek na sekundę dla karty 2080, a 201 – 204 klatek na sekundę dla karty 3060. Metoda wykorzystująca algorytm w języku GLSL osiągnęła 240 klatek na sekundę dla karty 2080, a 205 dla karty 3060.

| Algorytm | RTX 2080 Ti | RTX 3060 |
|---|---|---|
| SSBO | 251 | 222 |
| Tekstura 3D | 226 | 204 |
| Tekstura rzadka 3D | 225 | 201 |
| GLSL | 240 | 205 |

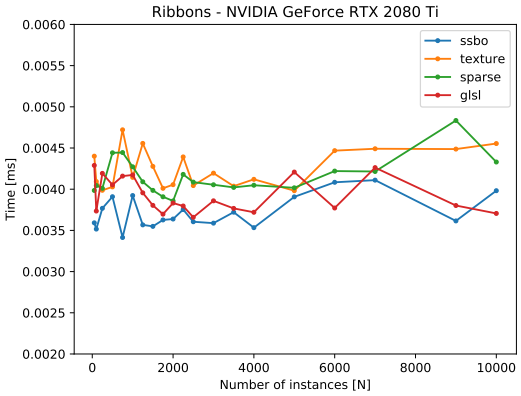
**Tabela 7.2.** Liczba klatek na sekundę dla metody z powierzchnią dla każdej implementacji pola magnetycznego na karcie graficznej NVIDIA GeForce RTX 2080 Ti oraz karcie graficznej NVIDIA GeForce RTX 3060.



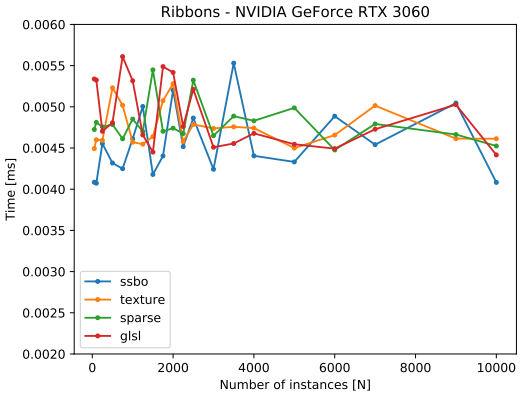

**(a)** Linie — RTX 2080 Ti



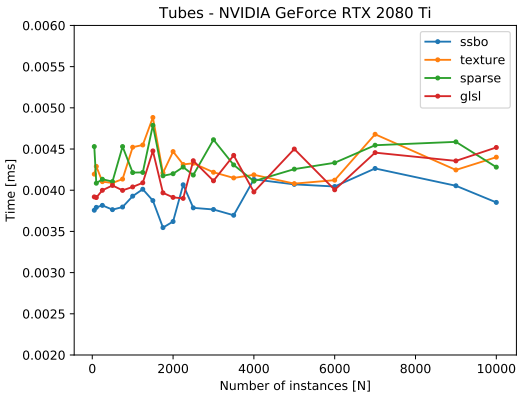

**(b)** Linie — RTX 3060



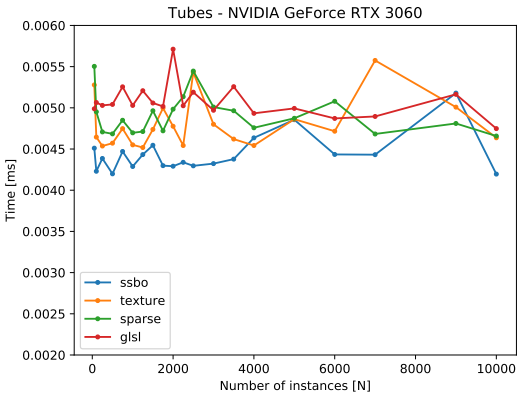

**(c)** Wstążki — RTX 2080 Ti




**(d)** Wstążki — RTX 3060




**(e)** Tuby — RTX 2080 Ti




**(f)** Tuby — RTX 3060

**Rys. 7.13.** Średni czas rysowania klatki w milisekundach dla liczby elementów graficznych od 50 do 10000, dla każdej implementacji pola magnetycznego na karcie graficznej NVIDIA GeForce RTX 2080 Ti (lewa kolumna) oraz karcie graficznej NVIDIA GeForce RTX 3060 (prawa kolumna).

## 7.4. Podsumowanie

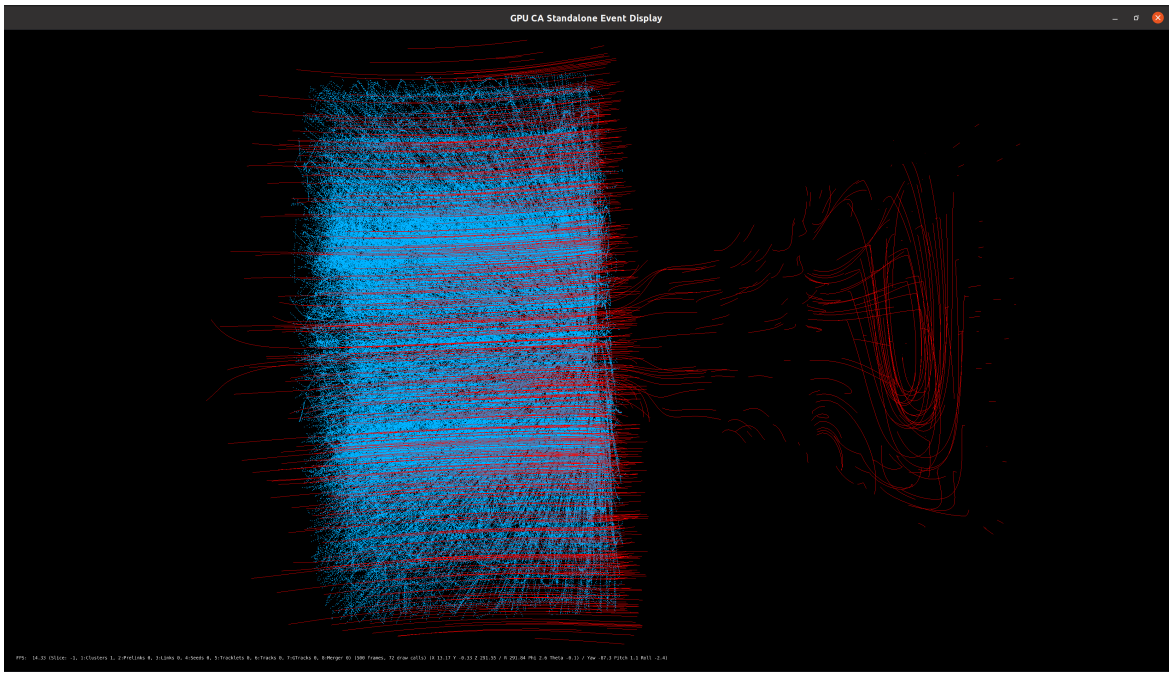
W trakcie badań zaproponowałem cztery metody wizualizacji pól wektorowych wykorzystujące proceduralną generację geometrii za pomocą shaderów siatki. Implementacje metod zostały połączone w jeden program z graficznym interfejsem umożliwiający wizualizację dowolnego pola na podstawie wzoru matematycznego podanego przez użytkownika. Zaproponowane metody należą do grupy geometrycznych metod wizualizacji pól wektorowych. Są to linie, wstążki, tuby oraz metoda uwypuklająca wybrane przez użytkownika cechy pola na wybranej przez użytkownika powierzchni. Metody zostały przetestowane przy użyciu modelu pola magnetycznego ALICE ze względu na jego strukturę oraz możliwość przetestowania wydajności metod przy różnym stopniu skomplikowania obliczeń (dzięki kilku dostępnym implementacjom modelu opisanych w Rozdziale 5).

Wydajność została przetestowana przy różnej liczbie rysowanych elementów graficznych od 50 do 10000. Górny limit dobrany został eksperymentalnie na podstawie czytelności wizualizacji (gęstość rozmieszczenia elementów staje się na tyle duża, że tworzą one na ekranie zwartą bryłę, co uniemożliwia analizę charakterystyki pola). W każdym przypadku liczba klatek na sekundę na obu testowanych kartach graficznych przekroczyła kilkukrotnie granicę komfortowego korzystania z aplikacji (60 klatek na sekundę). Oznacza to, że testowany sprzęt oferuje pewien zapas wydajności, dzięki któremu program można w przyszłości rozwinąć o dodatkowe elementy, np. o filtrowanie cech pola za pomocą jednego z przeznaczonych do tego celu algorytmów opisanych w literaturze naukowej (Rozdział 4.1.3).

## 7.5. Wdrożenie w ALICE

Wizualizację pola magnetycznego detektora ALICE, która powstała na skutek zastosowania metod opracowanych w tym Rozdziale, dostosowałem w taki sposób, aby możliwa była jej integracja z oprogramowaniem $O^2$ (Rozdział 2). We wdrożonej wersji programu dane o polu nie są wczytywane z zewnętrznego pliku, a ładowane bezpośrednio z bazy danych kalibracji detektora — *Condition and Calibration Data Base*

(CCDB). Ważnym dodatkiem w tej implementacji jest możliwość wyboru konfiguracji modelu odpowiadającej warunkom panującym w danym momencie w detektorze. W zależności od przeprowadzanego eksperymentu fizycznego elektromagnes solenoidalny (Rozdział 2) może być wyłączony, zasilany prądem o natężeniu 12 kA (wtedy jego siła wynosi 0.2 T) lub 30 kA (wtedy jego siła wynosi 0.5 T), natomiast elektromagnes dipolowy może być (niezależnie od elektromagnesu solenoidalnego) włączony (6 kA, siła 0.7 T) lub nie. Ponadto możliwa jest zmiana kierunku oddziaływania pola przez odwrócenie polaryzacji elektromagnesów. Przy takiej liczbie możliwych kombinacji istotnym było więc opracowanie dostępu do danych kalibracji w taki sposób, aby ustawienia wizualizacji pola pokrywały się z ustawieniami, z jakimi rekonstruowane są trajektorie cząstek.




**Rys. 7.14.** Wizualizacja pola magnetycznego detektora ALICE na tle zarejestrowanych pozycji trajektorii cząstek (klastrów) w ramach programu Event Display.

Reprezentacja pola magnetycznego jest od tej pory dostępna w $O^2$ w programie Event Display, który służy do wizualizacji trajektorii zderzeń cząstek (Rysunek 7.14). Źródła zaakceptowanej propozycji implementacji [139] dostępne są w serwisie GitHub, gdzie przechowywany jest kod całego oprogramowania $O^2$.

# 8. Podsumowanie

Głównym celem badań przeprowadzonych w ramach niniejszej pracy był rozwój algorytmów proceduralnej generacji geometrii na karcie graficznej oraz ich praktyczne zastosowanie w kontekście wizualizacji pól wektorowych. Rozprawa powstawała we współpracy z Kolaboracją ALICE z CERN, która udostępniła odpowiednie dane oraz zaoferowała pomoc w zrozumieniu sposobu działania istniejącego kodu (Rozdział 2). Był on punktem wyjścia przy opracowaniu moich oryginalnych algorytmów. Współpraca ta uwieńczona została jednym z kluczowych osiągnięć pracy, jakim jest bezpośrednie, praktyczne zastosowanie jej wyników w postaci wdrożenia wizualizacji pola magnetycznego w pakiecie oprogramowania detektora ALICE.

Wyniki z każdej części badawczej pracy opracowane zostały w postaci artykułów naukowych (jeden opublikowany [11], a dwa pozostałe z publicznie dostępnymi preprintami [12], [13], oczekujące na recenzję w czasopismach), w których w każdym jestem głównym autorem. Przedstawiają one szczegółowo wymienione wyżej zagadnienia, metodykę badawczą, rezultaty oraz wnioski.

Działanie algorytmów wizualizacji na karcie graficznej uwarunkowane jest możliwością dostępu do danych o polu wektorowym z poziomu kodu shaderów, co było pierwszym problemem podjętym w niniejszej pracy. Moje propozycje jego rozwiązania opisałem w Rozdziale 5. Przedstawione metody stanowią różnego poziomu kompromis pomiędzy wykorzystaną pamięcią wideo, wydajnością, a poziomem dewiacji uzyskiwanych wektorów pola w stosunku do implementacji referencyjnej zawartej w $O^2$. Ich zachowanie porównałem zarówno między sobą jak również z rozwiązaniem tego problemu stosowanym do tej pory po to, aby określić silne oraz słabe strony moich propozycji.

Jednym z zastosowań modeli pól wektorowych jest symulacja zjawisk fizycznych, na przykład toru lotu cząstek w polu magnetycznym posiadających ładunek elektryczny. Istniejące implementacje wizualizacji tego typu wykorzystują do symulacji algorytm działający na procesorze komputera. Rozdział 6 przedstawia rozwiązanie zagadnienia propagacji trajektorii cząstek na potrzeby wizualizacji za pomocą algorytmu procedu-

ralnej generacji bezpośrednio na karcie graficznej. Moje rozwiązanie redukuje rozmiar danych wysyłanych do karty (bo wymaga przesłania jedynie informacji o punkcie początkowym trajektorii oraz o kilku parametrach fizycznych), daje możliwość dynamicznej zmiany pola w czasie rzeczywistym (po to, aby obserwować kształt trajektorii w różnych symulowanych warunkach) oraz pozwala na implementację dodatkowych elementów np. wskaźników sił oddziałujących na cząsteczkę w każdym punkcie obliczanej trajektorii.

Ostatnia część pracy zajmuje się algorytmami proceduralnej generacji geometrii na karcie graficznej (linii, wstążek, tub oraz powierzchni) w zastosowaniu wizualizacji pól wektorowych. Opracowane metody zostały opisane w Rozdziale 7. Algorytmy zostały zrealizowane w nowatorski sposób wykorzystujący potok graficzny shaderów siatki. Przedstawione przeze mnie wyniki badań potwierdzają zasadność ich zastosowania — uzyskana wydajność wielokrotnie przekracza granicę interaktywnego korzystania z aplikacji wizualizacji przez użytkownika. Shadery siatki reprezentują najnowszą technologię oferowaną jedynie na kartach graficznych z rodziny RTX firmy NVIDIA. Niewielka liczba opublikowanych prac wykorzystujących tą technologię sugeruje, że w momencie pisania pracy jest to temat jeszcze niedostatecznie poznany i warty jest uwagi. Na podstawie przeprowadzonych badań powstała wizualizacja pola magnetycznego detektora (która została wdrożona w ramach oprogramowania $O^2$) oraz niezależny program *FieldView* do wizualizacji dowolnych pól wektorowych.

Opracowane przeze mnie metody oraz algorytmy, mimo że testowane były na danych pochodzących z ALICE, są uniwersalne i mogą być zaadoptowane do innych warunków pracy. Zostało to udowodnione w ramach Rozdziału 7, który zawiera przykładowe wizualizacje kilku różnych pól wektorowych przygotowanych za pomocą zaproponowanych metod.

Podsumowując, wszystkie określone na początku cele pracy zostały osiągnięte.

### 8.1. Wkład w rozwój dyscypliny

Oryginalnymi osiągnięciami opisanymi w niniejszej pracy są:

- opracowanie czterech sposobów na przechowywanie oraz na dostęp do danych o

polu wektorowym w shaderach (tj. na karcie graficznej), co umożliwiło kontynuację badań w kierunku jego zastosowania w wizualizacjach. Mimo koncentracji prac na ALICE, przedstawione metody można również zastosować dla innych pól wektorowych. W badaniach porównano szybkość obliczeń oraz poziom dewiacji uzyskiwanych wektorów pola w stosunku do modelu referencyjnego zawartego w oprogramowaniu ALICE dla wszystkich metod. Temat ten, opublikowany w [11], został przedstawiony w Rozdziale 5.

- opracowanie metody obliczania trajektorii cząstek oraz jej wizualizacja na karcie graficznej w oparciu o shader geometrii. Metoda może zostać użyta do wizualizacji danych z dowolnego detektora, który zakrzywia tor lotu cząstek. W badaniach porównano wpływ wybranego modelu pola magnetycznego na szybkość działania algorytmu oraz różnice kształtu trajektorii w porównaniu z modelem uproszczonym. Temat ten został zaprezentowany we wstępnym stadium rozwoju na konferencji [14], a następnie przygotowany do publikacji w postaci artykułu naukowego [12]. Pełny opis metody przedstawiony został w Rozdziale 6.
- opracowanie sposobów wizualizacji pól wektorowych za pomocą linii pola, wstążek, tub i powierzchni w oparciu o nowy potok graficzny wykorzystujący shadery siatki. Metody zostały przedstawione w przygotowanym do publikacji artykule [13]. Zaproponowane metody zostały zaimplementowane w ramach programu *FieldView*. Pełny opis metod zawarty został w Rozdziale 7.
- wdrożenie wizualizacji pola magnetycznego detektora ALICE jako części Event Display, trójwymiarowej aplikacji służącej do graficznej inspekcji jakości zbieranych przez detektor danych. Szczegóły wdrożenia zostały przedstawione w Rozdziale 7.5.

## 8.2. Wnioski końcowe

Zaproponowane oraz zaimplementowane metody dostępu do modelu pola wektorowego z poziomu pamięci karty graficznej porównałem między sobą oraz między rozwiązaniem referencyjnym (działającym tylko na procesorze komputera) zawartym w oprogramowaniu ALICE. Testy wydajnościowe wykazały, że zrównoleglenie ewaluacji pola

za pomocą karty graficznej daje przewagę wydajności nad implementacją referencyjną pod warunkiem, że otrzyma ona odpowiednio dużą ilość danych do przetworzenia.

Zastosowanie dokładnego modelu pola magnetycznego do wizualizacji trajektorii cząstek znacząco poprawiło zgodność uzyskanych symulacji z rzeczywistością. Jest to szczególnie istotne dla grupy cząstek, na których tor lotu największy wpływ ma jeden z elektromagnesów detektora, a którego obecność pomijana jest przez używany do tej pory uproszczony model. Zaimplementowany algorytm działa na karcie graficznej w czasie rzeczywistym i otwiera możliwości uzupełniania wizualizacji o dodatkowe elementy, takie jak wektory sił oddziałujących na reprezentowane cząstki.

Potok shaderów siatki okazał się bardzo dobrym sposobem generowania geometrii bezpośrednio na karcie graficznej. Pozwolił on na uzyskanie w testowanych warunkach wydajności renderowania wielokrotnie przekraczającej granicę interaktywnego korzystania z aplikacji przez użytkownika (60 klatek na sekundę). Okazało się to prawdą również przy zastosowaniu dokładnego modelu pola, którego zastosowanie w poprzednim rozważanym zagadnieniu w ramach shadera geometrii wiązało się ze spadkiem wydajności.

## 8.3. Możliwości dalszego rozwoju

Zagadnieniami, które obecnie cieszą się dużą popularnością w dziedzinie grafiki komputerowej są rzeczywistość rozszerzona oraz rzeczywistość wirtualna. Osiągnięcia w tej dziedzinie można łączyć z tematem wizualizacji pól dla uzyskania większej immersji użytkownika oraz ciekawszego efektu, czego przykładem jest praca [140]. Wartym uwagi zagadnieniem byłoby przetestowanie uzyskanych przeze mnie wyników w takim właśnie środowisku.

W pracy wykorzystałem najprostszą metodę numerycznej aproksymacji linii przepływu jaką jest bezpośrednia metoda Eulera. W celu poprawy jakości generowanych linii tematem wartym rozważenia byłoby zastąpienie jej bardziej zaawansowaną metodą, np. Rungego-Kutty.

Kwestią będącą częścią dziedziny wizualizacji pól za pomocą metod geometrycznych jest odpowiednie rozmieszczenie punktów-ziaren na podstawie wymagań użyt-

kownika, charakterystyki pola oraz czytelności obrazu [111]. W pracy (w której nacisk położyłem na stworzenie algorytmów wizualizacji w nowej technologii) wykorzystałem do tego celu generator liczb losowych, który okazał się wystarczający do moich zastosowań. Ciekawą kwestią byłaby analiza wyników przy zastąpieniu go jednym z bardziej zaawansowanych algorytmów generacji.

Jeden ze sposobów wizualizacji pól wektorowych — *volume rendering* [68] — wykorzystuje kosztowną obliczeniowo technikę *ray tracing* do generowania obrazów. Obecnie na rynku kart graficznych pojawiają się modele oferujące sprzętową akcelerację tej technologii. Wartym rozważenia byłoby przeprowadzenie badań w kierunku wykorzystania nowych możliwości dla poprawy wydajności opisywanych w literaturze algorytmów.

# Bibliografia


[1] J. Schertzer, C. Mercier, S. Rousseau i T. Boubekeur, "Fiblets for Real-Time Rendering of Massive Brain Tractograms", *Computer Graphics Forum (Proc. EUROGRAPHICS 2022)*, t. 41, nr. 2, s. 447–460, 2022.

[2] A. Preece, H. Bdeiwi i A. Ciarella, "The Thrust Reverser Unit flow Visualisation Project (TRUflow)", czer. 2022. DOI: 10.2514/6.2022-3688.

[3] M. M. InMeteo i M. Prantl, *Ventusky*, Dostęp: 2022-08-01, 2022. adr.: https://www.ventusky.com.

[4] T. McLoughlin, R. S. Laramee, R. Peikert, F. H. Post i M. Chen, "Over Two Decades of Integration-Based, Geometric Flow Visualization", *Computer Graphics Forum*, t. 29, nr. 6, s. 1807–1829, wrz. 2010. DOI: 10.1111/j.1467-8659.2010.01650.x.

[5] R. Laramee, H. Hauser, H. Doleisch, B. Vrolijk, F. Post i D. Weiskopf, "The State of the Art in Flow Visualization: Dense and Texture-Based Techniques", *Computer Graphics Forum*, t. 22, sty. 2003. DOI: 10.1111/j.1467-8659.2004.00753.x.

[6] H. Schäfer, M. Nießner, B. Keinert, M. Stamminger i C. Loop, "State of the Art Report on Real-time Rendering with Hardware Tessellation", kw. 2014.

[7] J. Burgess, "RTX on—The NVIDIA Turing GPU", *IEEE Micro*, t. 40, nr. 2, s. 36–44, 2020. DOI: 10.1109/MM.2020.2971677.

[8] P. Mours, *Mesh Shaders in Turing*, Accessed: 2021-08-15, list. 2018. adr.: http://on-demand.gputechconf.com/gtc-eu/2018/pdf/e8515-mesh-shaders-in-turing.pdf.

[9] P. Buncic, M. Krzewicki i P. V. Vyvre, "Technical Design Report for the Upgrade of the Online-Offline Computing System", CERN, spraw. tech., kw. 2015. adr.: https://cds.cern.ch/record/2011297.

[10] ALICE Collaboration, *Dane o polu magnetycznym detektora ALICE*, Dostęp: 2022-08-01, 2022. adr.: https://github.com/AliceO2Group/AliceO2/tree/dev/Common/maps.

[11] P. Nowakowski, P. Rokita i Ł. Graczykowski, "Distributed simulation and visualization of the ALICE detector magnetic field", *Computer Physics Communications*, t. 271, s. 108 206, 2022, ISSN: 0010-4655. DOI: 10.1016/j.cpc.2021.108206.

[12] P. Nowakowski, P. Rokita i Ł. Graczykowski, *GPU propagation and visualisation of particle collisions with ALICE magnetic field model*, 2022. DOI: 10.48550/ARXIV.2212.03627.

[13] P. Nowakowski i P. Rokita, *FieldView: An interactive software tool for exploration of three-dimensional vector fields*, 2022. DOI: 10.48550/ARXIV.2212.11813.

[14] P. Nowakowski, Ł. Graczykowski i P. Rokita, "Propagation of tracks using accurate model of ALICE detector magnet system for event visualisation", Quark Matter 29th International Conference On Ultra Relativistic Nucleus - Nucleus Collisions, 2022. adr.: https://indico.cern.ch/event/895086/contributions/4721755/attachments/2418718/4139697/PN_QuarkMatterPres2022.pdf.

[15] C. Lefèvre, *Complexe des accélérateurs du CERN*, Dostęp: 2022-08-01, 2008. adr.: https://cds.cern.ch/record/2066992.

[16] L. Evans i P. Bryant, "LHC Machine", *Journal of Instrumentation*, t. 3, nr. 08, S08001–S08001, sierp. 2008. DOI: 10.1088/1748-0221/3/08/s08001.

[17] CERN, *LHC Machine Outreach*, Dostęp: 2022-08-01, 2022. adr.: http://lhc-machine-outreach.web.cern.ch/lhc-machine-outreach-faq.htm.

[18] C. Hadjidakis, D. Kikoła i J. L. et al., "A fixed-target programme at the LHC: Physics case and projected performances for heavy-ion, hadron, spin and astroparticle studies", *Physics Reports*, t. 911, s. 1–83, 2021, A Fixed-Target Programme at the LHC: Physics Case and Projected Performances for Heavy-Ion, Hadron, Spin and Astroparticle Studies, ISSN: 0370-1573. DOI: 10.1016/j.physrep.2021.01.002.

[19] K. Adcox i in., "Formation of dense partonic matter in relativistic nucleus-nucleus collisions at RHIC: Experimental evaluation by the PHENIX collaboration", *Nucl. Phys. A*, t. 757, s. 184–283, 2005.

[20] J. Kapusta, B. Muller i J. Rafelski, *Quark-Gluon Plasma: Theoretical Foundations*. Elsevier, 2003, ISBN: 978-0-444-51110-2.

[21] U. W. Heinz i J. Maurice, "Evidence for a new state of matter: An Assessment of the results from the CERN lead beam program", sty. 2000. arXiv: `nucl-th/0002042`.

[22] J. Rafelski, "Melting Hadrons, Boiling Quarks", *Eur. Phys. J. A*, t. 51, nr. 9, s. 114, 2015.

[23] P. Foka i M. A. Janik, "An overview of experimental results from ultra-relativistic heavy-ion collisions at the CERN LHC: Bulk properties and dynamical evolution", *Rev. Phys.*, t. 1, s. 154–171, 2016. arXiv: `1702.07233 [hep-ex]`.

[24] P. Foka i M. A. Janik, "An overview of experimental results from ultra-relativistic heavy-ion collisions at the CERN LHC: Hard probes", *Rev. Phys.*, t. 1, s. 172–194, 2016. arXiv: `1702.07231 [hep-ex]`.

[25] CERN, *Longer term LHC schedule*, Dostęp: 2022-08-01, 2022. adr.: `https://lhc-commissioning.web.cern.ch/schedule/LHC-long-term.htm`.

[26] G. Eulisse, P. Konopka, M. Krzewicki, M. Richter, D. Rohr i S. Wenzel, "Evolution of the ALICE Software Framework for Run 3", *EPJ Web of Conferences*, t. 214, s. 05 010, sty. 2019.

[27] M. Richter, "A design study for the upgraded ALICE O2 computing facility", *Journal of Physics: Conference Series*, t. 664, s. 082 046, grud. 2015.

[28] L. Betev i P.Chochula, "Definition of the ALICE Coordinate System and Basic Rules for Subdetector Components Numbering", ALICE, spraw. tech., mar. 2003.

[29] C. Cavicchioli, "Development and Commissioning of the Pixel Trigger System for the ALICE Experiment at the CERN Large Hadron Collider", prac. dokt., Florence U., 2011.

[30] R. Shahoyan, "Summary of the L3 magnet field analysis", ALICE, spraw. tech., maj 2007.

[31] R. Shahoyan i A. Morsch, "Summary of the ALICE dipole magnet field analysis", ALICE, spraw. tech., lip. 2008.

[32] R. Shahoyan, *Algorytm obsługi modelu pola magnetycznego detektora ALICE*, Dostęp: 2022-08-01, 2022. adr.: `https://github.com/AliceO2Group/AliceO2/blob/dev/Common/Field/src/MagneticField.cxx`.

[33] S. Yamasaki, "Fast Magnetic Field Query Algorithm for the ALICE O2 Project", prac. mag., Hiroshima University, Japan, 2018.

[34] M. Tadel i A. Mrak-Tadel, *Algorytm propagacji trajektorii cząstek*, Dostęp: 2022-08-01, 2022. adr.: `https://github.com/root-project/root/blob/master/graf3d/eve/inc/TEveTrackPropagator.h`.

[35] CERN, *Odwrócona konwencja kierunku pola magnetycznego*, Dostęp: 2022-08-01, 2022. adr.: `https://github.com/root-project/root/blob/master/graf3d/%20eve/src/TEveTrackPropagator.cxx%5C#L55`.

[36] P. Nowakowski, "Wizualizacja danych Wielkiego Zderzacza Hadronów", prac. mag., Politechnika Warszawska, 2018.

[37] G. Sellers, R. S. Wright i N. Haemel, *OpenGL Superbible: Comprehensive Tutorial and Reference*, 7th. Addison-Wesley Professional, 2015, ISBN: 0672337479.

[38] F. Luna, *Introduction to 3D Game Programming with DirectX 12*. Mercury Learning & Information, 2016, ISBN: 1942270062.

[39] E. Lindholm, J. Nickolls, S. Oberman i J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture", *IEEE Micro*, t. 28, nr. 2, s. 39–55, 2008. DOI: `10.1109/MM.2008.31`.

[40] M. Segal i K. Akeley, *The OpenGL® Graphics System: A Specification*, Dostęp: 2022-08-01, 2019. adr.: `https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf`.

[41] R. Mukundan, "The Geometry Shader", w mar. 2022, s. 73–89, ISBN: 978-3-030-81353-6. DOI: `10.1007/978-3-030-81354-3_4`.

[42] J. N. Rodriguez, M. C. Canosa i E. H. Pereira, "Improving Electrical Power Grid Visualization Using Geometry Shaders", w *2011 Eighth International Conference Computer Graphics, Imaging and Visualization*, 2011, s. 177–182.

[43] H. Doraiswamy i J. Freire, "SPADE: GPU-Powered Spatial Database Engine for Commodity Hardware", w *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, 2022, s. 2669–2681. DOI: 10.1109/ICDE53745.2022.00245.

[44] H. Doraiswamy i J. Freire, *GPU-Powered Spatial Database Engine for Commodity Hardware: Extended Version*, 2022. DOI: 10.48550/ARXIV.2203.14362.

[45] H.-H. Chang, Y.-C. Lai, C.-Y. Yao, K.-L. Hua, Y. Niu i F. Liu, "Geometry-shader-based real-time voxelization and applications", *The Visual Computer*, t. 30, nr. 3, s. 327–340, 2014.

[46] A. Köhn, J. Klein, F. Weiler i H.-O. Peitgen, "A GPU-based fiber tracking framework using geometry shaders", w *Medical Imaging 2009: Visualization, Image-Guided Procedures, and Modeling*, M. I. Miga i K. H. Wong, red., International Society for Optics i Photonics, t. 7261, SPIE, 2009, s. 508–517. DOI: 10.1117/12.812219.

[47] R. Concheiro, M. Amor i M. Bóo, "Synthesis of Bézier Surfaces on the GPU.", sty. 2010, s. 110–115.

[48] R. Concheiro, M. Amor, M. Gil, E. Padrón i X. Martorell, "Rendering of Bézier Surfaces on Handheld Devices", w *Conference: 21st International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG 2013)*, t. 21, czer. 2013.

[49] Y.-T. Hu, J. Wang, R. A. Yeh i A. G. Schwing, "SAIL-VOS 3D: A Synthetic Dataset and Baselines for Object Detection and 3D Mesh Reconstruction From Video Data", w *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, czer. 2021, s. 1418–1428.

[50] A. Zhang, X. Li, G. Jiang, Z. Dong i C. Honglian, "3-D Simulation of Double-Bar Plush Fabrics with Jacquard Patterns", *Autex Research Journal*, t. 18, lut. 2018. DOI: 10.1515/aut-2017-0029.

[51] D. Garrido, R. Rodrigues, A. Augusto Sousa, J. Jacob i D. Castro Silva, "Point Cloud Interaction and Manipulation in Virtual Reality", w *2021 5th International Conference on Artificial Intelligence and Virtual Reality (AIVR)*, ser. AIVR 2021, Kumamoto, Japan: Association for Computing Machinery, 2021, s. 15–20, ISBN: 9781450384148. DOI: 10.1145/3480433.3480437.

[52] P. Fleck, A. Sousa Calepso, S. Hubenschmid, M. Sedlmair i D. Schmalstieg, "RagRug: A Toolkit for Situated Analytics", *IEEE Transactions on Visualization and Computer Graphics*, s. 1–1, 2022. DOI: 10.1109/TVCG.2022.3157058.

[53] F. I. Wiryadi i R. Kosala, "Particle rendering using geometry shader", w *2016 1st International Conference on Game, Game Art, and Gamification (ICGGAG)*, 2016, s. 1–6.

[54] C. Kubisch, *Introduction to Turing Mesh Shaders*, Dostęp: 2022-08-01, wrz. 2018. adr.: https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/.

[55] P. B. Christoph Kubisch, *GLSL Mesh Shader Extension*, Accessed: 2022-08-01, wrz. 2018. adr.: https://www.khronos.org/registry/OpenGL/extensions/NV/NV_mesh_shader.txt.

[56] A. Blake-Davies, *Next-Generation Gaming with AMD RDNA 2 and DirectX 12 Ultimate*, Dostęp: 2022-08-01, lut. 2020. adr.: https://community.amd.com/t5/gaming/next-generation-gaming-with-amd-rdna-2-and-directx-12-ultimate/ba-p/427032.

[57] B. Santerre, M. Abe i T. Watanabe, "Improving GPU Real-Time Wide Terrain Tessellation Using the New Mesh Shader Pipeline", w *2020 Nicograph International (NicoInt)*, czer. 2020, s. 86–89. DOI: 10.1109/NicoInt50878.2020.00025.

[58] M. Englert, "Using Mesh Shaders for Continuous Level-of-Detail Terrain Rendering", sierp. 2020, s. 1–2. DOI: 10.1145/3388767.3407391.

[59] J. Unterguggenberger, B. Kerbl, J. Pernsteiner i M. Wimmer, "Conservative Meshlet Bounds for Robust Culling of Skinned Meshes", *Computer Graphics Forum*, t. 40, s. 57–69, paź. 2021. DOI: 10.1111/cgf.14401.

[60] R. Bujack i A. Middel, "State of the art in flow visualization in the environmental sciences", *Environmental Earth Sciences*, t. 79, sty. 2020. DOI: 10.1007/s12665-019-8800-4.

[61] R. Laramee, H. Hauser, L. Zhao i F. Post, "Topology-Based Flow Visualization, The State of the Art", w maj 2007, s. 1–19, ISBN: 978-3-540-70822-3. DOI: 10.1007/978-3-540-70823-0_1.

[62] L. Chao i W. Lingda, "Vector Field Visualization Review and Prospects", w *2014 International Conference on Virtual Reality and Visualization*, 2014, s. 43–49. DOI: 10.1109/ICVRV.2014.42.

[63] Y. A. Yusoff, F. Mohamad, M. S. Sunar i A. Selamat, "Flow Visualization Techniques: A Review", w *Trends in Applied Knowledge-Based Systems and Data Science*, Springer International Publishing, 2016, s. 527–538. DOI: 10.1007/978-3-319-42007-3_46.

[64] F. H. Post, B. Vrolijk, H. Hauser, R. S. Laramee i H. Doleisch, "The State of the Art in Flow Visualisation: Feature Extraction and Tracking", *Computer Graphics Forum*, t. 22, nr. 4, s. 775–792, grud. 2003. DOI: 10.1111/j.1467-8659.2003.00723.x.

[65] M. Edmunds, R. S. Laramee, G. Chen, N. Max, E. Zhang i C. Ware, "Surface-based flow visualization", *Computers & Graphics*, t. 36, nr. 8, s. 974–990, grud. 2012. DOI: 10.1016/j.cag.2012.07.006.

[66] A. Pobitzer, R. Peikert, R. Fuchs i in., "The State of the Art in Topology-Based Visualization of Unsteady Flow", *Computer Graphics Forum*, t. 30, s. 1789–, wrz. 2011. DOI: 10.1111/j.1467-8659.2011.01901.x.

[67] D. Stalling, "Fast Texture-Based Algorithms for Vector Field Visualization", prac. dokt., Konrad-Zuse-Zentrum f˙ur Informationstechnik, 1998.

[68] S. Weiss i R. Westermann, "Differentiable Direct Volume Rendering", *IEEE Transactions on Visualization and Computer Graphics*, t. PP, s. 1–1, wrz. 2021. DOI: 10.1109/TVCG.2021.3114769.

[69] S. Thalabard, D. Rosenberg, A. Pouquet i P. Mininni, "Conformal Invariance in Three-Dimensional Rotating Turbulence", *Physical review letters*, t. 106, s. 204 503, maj 2011. DOI: 10.1103/PhysRevLett.106.204503.

[70] T. Fruhauf, "Raycasting vector fields", w *Proceedings of Seventh Annual IEEE Visualization '96*, 1996, s. 115–120. DOI: 10.1109/VISUAL.1996.567780.

[71] D. Ebert, R. Yagel, J. Scott i Y. Kurzion, "Volume Rendering Methods for Computational Fluid Dynamics Visualization", sty. 1994, s. 232–239. DOI: 10.1109/VISUAL.1994.346314.

[72] R. A. Katkar, R. M. Taft i G. T. Grant, "3D Volume Rendering and 3D Printing (Additive Manufacturing)", *Dental Clinics of North America*, t. 62, nr. 3, s. 393–402, lip. 2018. DOI: 10.1016/j.cden.2018.03.003.

[73] R. Schmoll, S. Schramm, T. Breitenstein i A. Kroll, "Method and experimental investigation of surface heat dissipation measurement using 3D thermography", *Journal of Sensors and Sensor Systems*, t. 11, s. 41–49, lut. 2022. DOI: 10.5194/jsss-11-41-2022.

[74] M. Imre, J. Tao i C. Wang, "Efficient GPU-accelerated computation of isosurface similarity maps", w *2017 IEEE Pacific Visualization Symposium (PacificVis)*, 2017, s. 180–184. DOI: 10.1109/PACIFICVIS.2017.8031592.

[75] B. Jalal, V. Blakaj, S. Greedy i P. Evans, "Real-Time Electromagnetic Visualisation for Large 3D Accelerated Models", czer. 2022, s. 1–7. DOI: 10.1109/COMPEL53829.2022.9830033.

[76] W. De Leeuw i J. Wijk, "A Probe for Local Flow Field Visualization", sty. 1993, s. 39–45. DOI: 10.1109/VISUAL.1993.398849.

[77] K. Lu, "Distribution-based Exploration and Visualization of Large-scale Vector and Multivariate Fields", prac. dokt., The Ohio State University, 2017.

[78] D. Dovey, "Vector plots for irregular grids", w *Proceedings Visualization '95*, 1995, s. 248–253. DOI: 10.1109/VISUAL.1995.480819.

[79] G. Turk i D. Banks, "Image-Guided Streamline Placement", w *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*,

ser. SIGGRAPH '96, New York, NY, USA: Association for Computing Machinery, 1996, s. 453–460, ISBN: 0897917464. DOI: 10.1145/237170.237285.

[80] R. T. Happe i M. Rumpf, "Characterizing global features of simulation data by selected local icons", w *Eurographics Workshop on Virtual environments and scientific visualization'96*, Springer, 1996, s. 234–242.

[81] J. J. van Wijk, "Spot Noise Texture Synthesis for Data Visualization", *SIGGRAPH Comput. Graph.*, t. 25, nr. 4, s. 309–318, lip. 1991, ISSN: 0097-8930. DOI: 10.1145/127719.122751.

[82] N. Pavie, G. Gilet, J.-M. Dischler i D. Ghazanfarpour, "Procedural Texture Synthesis by Locally Controlled Spot Noise", w *WSCG 2016*, maj 2016.

[83] B. Cabral i L. C. Leedom, "Imaging Vector Fields Using Line Integral Convolution", w *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '93, Anaheim, CA: Association for Computing Machinery, 1993, s. 263–270, ISBN: 0897916018. DOI: 10.1145/166117.166151.

[84] R. Wegenkittl, E. Groller i W. Purgathofer, "Animating flow fields: rendering of oriented line integral convolution", w *Proceedings. Computer Animation '97 (Cat. No.97TB100120)*, 1997, s. 15–21. DOI: 10.1109/CA.1997.601035.

[85] R. Wegenkittl i E. Groller, "Fast oriented line integral convolution for vector field visualization via the Internet", w *Proceedings. Visualization '97 (Cat. No. 97CB36155)*, 1997, s. 309–316. DOI: 10.1109/VISUAL.1997.663897.

[86] H.-W. Shen i D. Kao, "UFLIC: a line integral convolution algorithm for visualizing unsteady flows", w *Proceedings. Visualization '97 (Cat. No. 97CB36155)*, 1997, s. 317–322. DOI: 10.1109/VISUAL.1997.663898.

[87] R. Laramee, B. Jobard i H. Hauser, "Image space based visualization of unsteady flow on surfaces", w *IEEE Visualization, 2003. VIS 2003.*, 2003, s. 131–138. DOI: 10.1109/VISUAL.2003.1250364.

[88] B. Jobard, G. Erlebacher i M. Hussaini, "Lagrangian-Eulerian advection of noise and dye textures for unsteady flow visualization", *IEEE Transactions*

*on Visualization and Computer Graphics*, t. 8, nr. 3, s. 211–222, 2002. DOI: 10.1109/TVCG.2002.1021575.

[89] F. Taponecco i M. Alexa, "Vector Field Visualization using Markov Random Field Texture Synthesis", w *Eurographics / IEEE VGTC Symposium on Visualization*, G.-P. Bonneau, S. Hahmann i C. D. Hansen, red., The Eurographics Association, 2003, ISBN: 3-905673-01-0. DOI: 10.2312/VisSym/VisSym03/195-202.

[90] Z. Liu i R. J. M. II, "AUFLIC: An Accelerated Algorithm For Unsteady Flow Line Integral Convolution", w *Eurographics / IEEE VGTC Symposium on Visualization*, D. Ebert, P. Brunet i I. Navazo, red., The Eurographics Association, 2002, ISBN: 1-58113-536-X. DOI: 10.2312/VisSym/VisSym02/043-052.

[91] V. Interrante i C. Grosch, "Strategies for effectively visualizing 3D flow with volume LIC", w *Proceedings. Visualization '97 (Cat. No. 97CB36155)*, 1997, s. 421–424. DOI: 10.1109/VISUAL.1997.663912.

[92] H.-W. Shen, C. Johnson i K.-L. Ma, "Visualizing vector fields using line integral convolution and dye advection", w *Proceedings of 1996 Symposium on Volume Visualization*, 1996, s. 63–70. DOI: 10.1109/SVV.1996.558044.

[93] J. Huang, Z. Pan, G. Chen, W. Chen i H. Bao, "Image-Space Texture-Based Output-Coherent Surface Flow Visualization", *IEEE Transactions on Visualization and Computer Graphics*, t. 19, nr. 9, s. 1476–1487, 2013. DOI: 10.1109/TVCG.2013.62.

[94] B. Jobard, G. Erlebacher i M. Yousuff Hussaini, "Lagrangian-Eulerian advection for unsteady flow visualization", w *Proceedings Visualization, 2001. VIS '01.*, 2001, s. 53–541. DOI: 10.1109/VISUAL.2001.964493.

[95] D. Weiskopf, G. Erlebacher, M. Hopf i T. Ertl, "Hardware-Accelerated Lagrangian-Eulerian Texture Advection for 2D Flow Visualization", 2002.

[96] J. J. van Wijk, "Image Based Flow Visualization", *ACM Trans. Graph.*, t. 21, nr. 3, s. 745–754, lip. 2002, ISSN: 0730-0301. DOI: 10.1145/566654.566646.

[97] J. van Wijk, "Image based flow visualization for curved surfaces", w *IEEE Visualization, 2003. VIS 2003.*, 2003, s. 123–130. DOI: 10.1109/VISUAL.2003.1250363.

[98] C. Li i M. Wand, "Combining Markov Random Fields and Convolutional Neural Networks for Image Synthesis", *CoRR*, t. abs/1601.04589, 2016. arXiv: 1601.04589. adr.: http://arxiv.org/abs/1601.04589.

[99] L.-Y. Wie, S. Lefebvre, V. Kwatra i G. Turk, "State of the Art in Example-based Texture Synthesis", w *Eurographics 2009 - State of the Art Reports*, M. Pauly i G. Greiner, red., The Eurographics Association, 2009. DOI: 10.2312/egst.20091063.

[100] T. Günther i I. Baeza, "Introduction to Vector Field Topology", w sty. 2021, s. 289–326, ISBN: 978-3-030-83499-9. DOI: 10.1007/978-3-030-83500-2_15.

[101] T. Günther i H. Theisel, "The State of the Art in Vortex Extraction", *Computer Graphics Forum*, t. 37, nr. 6, s. 149–173, sty. 2018. DOI: 10.1111/cgf.13319.

[102] H. Theisel, T. Weinkauf, H.-C. Hege i H.-P. Seidel, "Saddle connectors - an approach to visualizing the topological skeleton of complex 3D vector fields", w *IEEE Visualization, 2003. VIS 2003.*, 2003, s. 225–232. DOI: 10.1109/VISUAL.2003.1250376.

[103] T. Günther i H. Theisel, "Inertial Steady 2D Vector Field Topology", *Computer Graphics Forum*, t. 35, s. 455–466, maj 2016. DOI: 10.1111/cgf.12846.

[104] R. Bujack, M. Hlawitschka i K. I. Joy, "Topology-inspired Galilean invariant vector field analysis", w *2016 IEEE Pacific Visualization Symposium (PacificVis)*, 2016, s. 72–79. DOI: 10.1109/PACIFICVIS.2016.7465253.

[105] D. Kenwright, C. Henze i C. Levit, "Feature extraction of separation and attachment lines", *IEEE Transactions on Visualization and Computer Graphics*, t. 5, nr. 2, s. 135–144, 1999. DOI: 10.1109/2945.773805.

[106] I. Ari Sadarjoen i F. H. Post, "Detection, quantification, and tracking of vortices using streamline geometry", *Computers & Graphics*, t. 24, nr. 3, s. 333–341, 2000, ISSN: 0097-8493. DOI: 10.1016/S0097-8493(00)00029-7.

[107] F. Sauer, H. Yu i K.-L. Ma, "Trajectory-Based Flow Feature Tracking in Joint Particle/Volume Datasets", *IEEE Transactions on Visualization and Computer Graphics*, t. 20, s. 2565–2574, grud. 2014. DOI: 10.1109/TVCG.2014.2346423.

[108] M. Raissi, A. Yazdani i G. E. Karniadakis, "Hidden fluid mechanics: Learning velocity and pressure fields from flow visualizations", *Science*, t. 367, nr. 6481, s. 1026–1030, 2020. DOI: 10.1126/science.aaw4741.

[109] M. Raissi, A. Yazdani i G. E. Karniadakis, "Hidden Fluid Mechanics: A Navier-Stokes Informed Deep Learning Framework for Assimilating Flow Visualization Data", *CoRR*, t. abs/1808.04327, 2018. arXiv: 1808.04327. adr.: http://arxiv.org/abs/1808.04327.

[110] A. Kumar, S. Ridha, M. Narahari i S. U. Ilyas, "Physics-guided deep neural network to characterize non-Newtonian fluid flow for optimal use of energy resources", *Expert Systems with Applications*, t. 183, s. 115 409, 2021, ISSN: 0957-4174. DOI: 10.1016/j.eswa.2021.115409.

[111] S. Sane, R. Bujack, C. Garth i H. Childs, "A Survey of Seed Placement and Streamline Selection Techniques", *Computer Graphics Forum*, t. 39, nr. 3, s. 785–809, czer. 2020. DOI: 10.1111/cgf.14036.

[112] W. S. Janna, *Introduction to Fluid Mechanics*, 6th. CRC Press, 2020, ISBN: 9780367341275.

[113] O. Mallo, R. Peikert, C. Sigg i F. Sadlo, "Illuminated lines revisited", w *VIS 05. IEEE Visualization, 2005.*, 2005, s. 19–26. DOI: 10.1109/VISUAL.2005.1532772.

[114] C.-K. Chen, S. Yan, H. Yu, N. Max i K.-L. Ma, "An Illustrative Visualization Framework for 3D Vector Fields", *Computer Graphics Forum*, t. 30, nr. 7, s. 1941–1951, 2011. DOI: 10.1111/j.1467-8659.2011.02064.x.

[115] T. McGraw, "High-quality real-time raycasting and raytracing of streamtubes with sparse voxel octrees", w *2020 IEEE Visualization Conference (VIS)*, 2020, s. 21–25. DOI: 10.1109/VIS47514.2020.00011.

[116] R. Laramee, C. Garth, H. Doleisch, J. Schneider, H. Hauser i H. Hagen, "Visual analysis and exploration of fluid flow in a cooling jacket", w *VIS 05. IEEE Visualization, 2005.*, 2005, s. 623–630. DOI: 10.1109/VISUAL.2005.1532850.

[117] R. S. Laramee i H. Hauser, "Geometric flow visualization techniques for CFD simulation data", w *Proceedings of the 21st spring conference on Computer graphics - SCCG '05*, ACM Press, 2005. DOI: 10.1145/1090122.1090158.

[118] W. Engelke, K. Lawonn, B. Preim i I. Hotz, "Autonomous Particles for Interactive Flow Visualization", *Computer Graphics Forum*, t. 38, nr. 1, s. 248–259, 2019. DOI: 10.1111/cgf.13528.

[119] J. Schmid, H. Klingemann, A. Scheuermann, J. Buehling, N. Bernasconi i M. Flückiger, "PedVis: Pedestrian Flow Visualisations", w grud. 2016, s. 369–376, ISBN: 978-3-319-33481-3. DOI: 10.1007/978-3-319-33482-0_47.

[120] V. Verma i A. Pang, "Comparative flow visualization", *IEEE Transactions on Visualization and Computer Graphics*, t. 10, nr. 6, s. 609–624, 2004. DOI: 10.1109/TVCG.2004.39.

[121] M. Ankele i T. Schultz, "DT-MRI Streamsurfaces Revisited", *IEEE Transactions on Visualization and Computer Graphics*, t. PP, s. 1–1, sierp. 2018. DOI: 10.1109/TVCG.2018.2864845.

[122] M. Kanzler, "Interactive Visualization of Large 3D Line Sets", prac. dokt., Technische Universität München, 2020.

[123] P. Nowakowski, *Distributed Simulation And Visualization of The ALICE Detector Magnetic Field*, Dostęp: 2022-08-01. adr.: https://github.com/pnwkw/distributed_field.

[124] J. Niedziela i B. Haller, "Event visualisation in ALICE - current status and strategy for Run 3", *Journal of Physics: Conference Series*, t. 898, s. 072 008, paź. 2017. DOI: 10.1088/1742-6596/898/7/072008.

[125] Khronos, *ARB Shader Storage Buffer Object*, Dostęp: 2022-08-01. adr.: https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_shader_storage_buffer_object.txt.

[126] A. Keune, "Parameterization of the LHCb Magnetic Field Map", *Nuclear Physics B - Proceedings Supplements*, t. 197, nr. 1, s. 163–166, 2009, 11th Topical Seminar on Innovative Particle and Radiation Detectors (IPRD08), ISSN: 0920-5632. DOI: 10.1016/j.nuclphysbps.2009.10.058.

[127] A. Tadel, M. Tadel, A. Yagil, D. Kovalskyi i S. Linev, "EVE-7 and FireworksWeb: The next generation event visualization tools for ROOT and CMS", *EPJ Web of Conferences*, t. 245, s. 08 027, sty. 2020.

[128] M. Tadel, "Overview of EVE – the event visualization environment of ROOT", *Journal of Physics: Conference Series*, t. 219, nr. 4, s. 042 055, kw. 2010. DOI: 10.1088/1742-6596/219/4/042055.

[129] P. Nowakowski, *Visualisation of tracks using accurate model of ALICE detector magnets*, Dostęp: 2022-08-01. adr.: https://github.com/pnwkw/gpu_propagator.

[130] S. Agostinelli i J. A. et al., "Geant4—a simulation toolkit", *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, t. 506, nr. 3, s. 250–303, 2003, ISSN: 0168-9002. DOI: 10.1016/S0168-9002(03)01368-8.

[131] Ł. Graczykowski, P. Nowakowski i P. Foka, "New developments for ALICE MasterClasses and the new Particle Therapy MasterClass", *EPJ Web Conf.*, t. 245, s. 08 011, 2020. DOI: 10.1051/epjconf/202024508011.

[132] A. Trisovic, "Measuring the D0 lifetime at the LHCb Masterclass", *Nuclear and Particle Physics Proceedings*, t. 273-275, s. 1215–1220, kw. 2016.

[133] K. Cecire i T. McCauley, "The QuarkNet CMS masterclass: bringing the LHC to students", *Nuclear and Particle Physics Proceedings*, t. 273-275, s. 1261–1264, kw. 2016.

[134] P. Nowakowski, *FieldView: An interactive software tool for exploration of three dimensional vector fields*, Dostęp: 2022-08-01. adr.: https://github.com/pnwkw/field_view.

[135] NVIDIA, *NVIDIA Volta GPU Architecture Whitepaper*, Dostęp: 2022-08-01, 2017. adr.: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[136] E. Estevez-Rams, D. Estevez-Moya, Y. Martinez Camejo, D. Gómez-Gómez i B. Aragon-Fernandez, "Hilbert curves in two dimensions", *Revista Cubana de Fisica*, t. 34, s. 9–18, lip. 2017.

[137] R. Hunter, B. White, R. Patel i J. Ballard, "Using Morton Codes to Partition Faceted Geometry: An Architecture for Terabyte-Scale Geometry Models", Information Technology Laboratory, spraw. tech., kw. 2020. DOI: 10.21079/11681/36516.

[138] Z. UserBenchmark, *Ranking karty graficznej RTX 2080 Ti kontra RTX 3060*, Dostęp: 2022-08-01. adr.: https://gpu.userbenchmark.com/Compare/Nvidia-RTX-3060-Ti-vs-Nvidia-RTX-2080-Ti/4090vs4027.

[139] P. Nowakowski, *Wdrożenie wizualizacji pola magnetyczego ALICE w pakiecie O2*, Dostęp: 2022-08-01. adr.: https://github.com/AliceO2Group/AliceO2/commit/78fe2a22fe9c4a00836405c696d11d32451a3cb6.

[140] X. Liu, C. Wang, J. Huang, Y. Liu i Y. Wang, "Study on Electromagnetic Visualization Experiment System Based on Augmented Reality", w lip. 2019, s. 699–712, ISBN: 978-981-13-9916-9. DOI: 10.1007/978-981-13-9917-6_65.

[141] P. Nowakowski, P. Żórawski, K. Cabaj i W. Mazurczyk, "Study of the Error Detection and Correction Scheme for Distributed Network Covert Channels", w *The 16th International Conference on Availability, Reliability and Security*, ser. ARES 2021, Vienna, Austria: Association for Computing Machinery, 2021, ISBN: 9781450390514. DOI: 10.1145/3465481.3470087.

[142] P. Nowakowski, P. Żórawski, K. Cabaj i W. Mazurczyk, "Detecting Network Covert Channels using Machine Learning, Data Mining and Hierarchical Organisation of Frequent Sets", *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, t. 12, s. 20–43, 2021.

[143] P. Nowakowski, P. Zórawski, K. Cabaj i W. Mazurczyk, "Network Covert Channels Detection Using Data Mining and Hierarchical Organisation of Frequent

Sets: An Initial Study", w *Proceedings of the 15th International Conference on Availability, Reliability and Security*, ser. ARES '20, Virtual Event, Ireland: Association for Computing Machinery, 2020, ISBN: 9781450388337. DOI: 10.1145/3407023.3409217.

[144] P. Nowakowski, P. Zórawski, K. Cabaj, M. Gregorczyk, M. Purski i W. Mazurczyk, "Distributed Packet Inspection for Network Security Purposes in Software-Defined Networking Environments", ser. ARES '20, Virtual Event, Ireland: Association for Computing Machinery, 2020, ISBN: 9781450388337. DOI: 10.1145/3407023.3409210.

[145] M. Gregorczyk, P. Żórawski, P. Nowakowski, K. Cabaj i W. Mazurczyk, "Sniffing Detection Based on Network Traffic Probing and Machine Learning", *IEEE Access*, t. 8, s. 149 255–149 269, 2020.

[146] K. Cabaj, P. Żórawski, P. Nowakowski, M. Purski i W. Mazurczyk, "Efficient distributed network covert channels for Internet of things environments†", *Journal of Cybersecurity*, t. 6, nr. 1, grud. 2020, ISSN: 2057-2085. DOI: 10.1093/cybsec/tyaa018. eprint: https://academic.oup.com/cybersecurity/article-pdf/6/1/tyaa018/34893203/tyaa018.pdf.

[147] P. Nowakowski, P. Zorawski, K. Cabaj i W. Mazurczyk, "Securing Modern Network Architectures with Software Defined Networking", w *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2019, s. 235–238.

[148] K. Cabaj, M. Gregorczyk, W. Mazurczyk, P. Nowakowski i P. Żórawski, "Sniffing Detection within the Network: Revisiting Existing and Proposing Novel Approaches", w *Proceedings of the 14th International Conference on Availability, Reliability and Security*, ser. ARES '19, Canterbury, CA, United Kingdom: Association for Computing Machinery, 2019, ISBN: 9781450371643. DOI: 10.1145/3339252.3341494.

[149] K. Cabaj, W. Mazurczyk, P. Nowakowski i P. Żórawski, "Fine-Tuning of Distributed Network Covert Channels Parameters and Their Impact on Undetectability", w *Proceedings of the 14th International Conference on Availabi-*

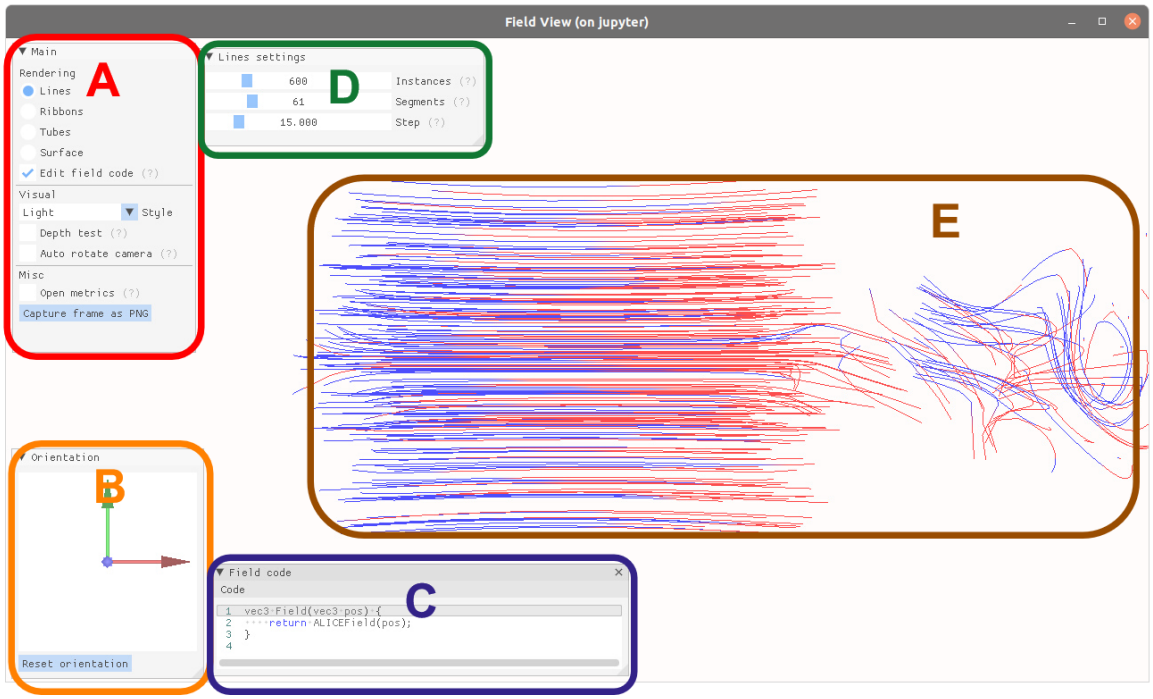
*lity, Reliability and Security*, ser. ARES '19, Canterbury, CA, United Kingdom: Association for Computing Machinery, 2019, ISBN: 9781450371643. DOI: 10.1145/3339252.3341489.

[150] K. Cabaj, M. Gregorczyk, W. Mazurczyk, P. Nowakowski i P. Żórawski, "Network Threats Mitigation Using Software-Defined Networking for the 5G Internet of Radio Light System", *Security and Communication Networks*, t. 2019, 2019, ISSN: 1939-0114. DOI: 10.1155/2019/4930908.

[151] K. Cabaj, W. Mazurczyk, P. Nowakowski i P. Żórawski, "Towards Distributed Network Covert Channels Detection Using Data Mining-Based Approach", w *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ser. ARES 2018, Hamburg, Germany: Association for Computing Machinery, 2018, ISBN: 9781450364485. DOI: 10.1145/3230833.3233264.

[152] K. Cabaj, M. Gregorczyk, W. Mazurczyk, P. Nowakowski i P. Żórawski, "SDN-Based Mitigation of Scanning Attacks for the 5G Internet of Radio Light System", w *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ser. ARES 2018, Hamburg, Germany: Association for Computing Machinery, 2018, ISBN: 9781450364485. DOI: 10.1145/3230833.3233248.

# A. Opis programu *FieldView*




**Rys. A.1.** Główne okno programu *FieldView*.

Uruchomienie programu powoduje pokazanie się głównego okna, przedstawionego na Rysunku A.1. Sekcja A (zatytułowana *Main*) kontroluje ogólne aspekty wizualizacji. Tutaj użytkownik może wybrać metodę (linie, wstążki, tuby, powierzchnia) oraz wyświetlić lub ukryć edytor wzoru matematycznego wykorzystywanego do opisu pola (zaznaczonego jako sekcja C). Poniżej znajduje się grupa ustawień kosmetycznych: stylu programu (jasny lub ciemny), włączenia/wyłączenia testu głębi (poprawiającego w niektórych przypadkach efekt wizualny) oraz automatycznego obrotu kamery wokół początku układu współrzędnych. Na końcu znajduje się opcja wyświetlająca/ukrywająca informacje diagnostyczne (liczba wierzchołków na ekranie, wydajność itp.) oraz przycisk stworzenia zrzutu ekranu.

Sekcja B (zatytułowana *Orientation*) daje użytkownikowi możliwość dopasowania pozycji kamery. Można to zrobić przytrzymując lewy przycisk myszy w zaznaczonym kwadratowym obszarze, a następnie przesuwając kursor. Pod tym obszarem znajduje się przycisk resetu pozycji do stanu początkowego.

Sekcja C to edytor równania pola. Użytkownik może wpisać tu dowolną formułę, która zostanie użyta do wizualizacji pola. Wpisana formuła wkompilowywana jest w locie w kod shadera w języku GLSL. Z tego powodu użytkownik może wykorzystać dowolną z funkcji wbudowanych w język GLSL, takich jak funkcje trygonometryczne, iloczyny skalarny i wektorowy, min i maks, potęgi itd. Użytkownik ma również dostęp do funkcji `ALICEField()`, która pozwala na wizualizację pola magnetycznego detektora. Została ona użyta w formule domyślnej, dostępnej po uruchomieniu programu.

Sekcja D zawiera ustawienia specyficzne dla danego sposobu wizualizacji. W poniższych podrozdziałach, gdzie opisane są szczegółowo algorytmy, wymieniona jest lista ustawień wraz z opisem funkcji.

Sekcja E przedstawia główną część programu, gdzie wizualizowane jest pole.

# B. Dorobek naukowy

Badania przedstawione w niniejszej pracy opublikowane zostały w następujących czasopismach naukowych oraz na konferencjach:

- Piotr Nowakowski, Przemysław Rokita, Łukasz Graczykowski, *Distributed simulation and visualization of the ALICE detector magnetic field*, Computer Physics Communications, 2022 [11],
- Piotr Nowakowski, Łukasz Graczykowski, Przemysław Rokita, *Propagation of tracks using accurate model of ALICE detector magnet system for event visualisation*, Quark Matter 29th International Conference On Ultra Relativistic Nucleus - Nucleus Collisions, 2022 [14].

Prace przedstawione poniżej zostały przygotowane do publikacji (i umieszczone w archiwum preprintów), ale w momencie pisania pracy znajdowały się wciąż na etapie recenzji:

- Piotr Nowakowski, Przemysław Rokita, Łukasz Graczykowski, *GPU propagation and visualisation of particle collisions with accurate model of ALICE detector magnetic field*, Computer Physics Communications [12],
- Piotr Nowakowski, Przemysław Rokita, Łukasz Graczykowski, *FieldView: An interactive software tool for exploration of three dimensional vector fields*, SoftwareX [13].

Udział w projektach badawczych:

- Wykrywanie ukrytej komunikacji sieciowej (*Covert Communication Detection* — CoCoDe); 2017–2020,
  stanowisko: konstruktor,
  zakres obowiązków: badania oraz implementacja systemu wykrywania ukrytej komunikacji sieciowej w środowisku *Software Defined Network* (SDN),
  grant udzielony przez: Air Force Office of Scientific Research,
- Internet Przyszłości w oparciu o sieci radiowe oraz komunikację światła widzialnego (*Internet of Radio Light* — IoRL); 2018–2020,
  stanowisko: konstruktor,

zakres obowiązków: badania oraz implementacja systemu ochrony sieci komputerowej przed atakami *Distributed Denial of Service* (DDoS) w środowisku *Software Defined Network* (SDN),

grant udzielony przez: Horizon 2020,

- Wykrywanie fałszywych wiadomości na platformach społecznościowych (*Detection of fake newS on SocIal MedIa pLAtfoRms* — DISSIMILAR); 2021–2024,

  stanowisko: konstruktor,

  zakres obowiązków: badania oraz implementacja rozproszonego systemu zbierania oraz analizy (pod kątem obecności złośliwego oprogramowania) materiałów multimedialnych z sieci Internet,

  grant udzielony przez: CONCERT-Japan

- Popularyzacja nauki, działań CERN i Eksperymentu ALICE (*ALICE Masterclass*); 2014–2018.

  stanowisko: samodzielny fizyk,

  zakres obowiązków: rozwój oprogramowania ALICE Masterclass, w tym port istniejącego kodu na system operacyjny Windows oraz implementacja przeglądarkowej wersji programu

  grant udzielony przez: MatFizChem.

Pozostały dorobek autora, niebędący bezpośrednim przedmiotem niniejszej rozprawy:

- Łukasz Graczykowski, Piotr Nowakowski, Panagiota Foka, *New developments for ALICE MasterClasses and the new Particle Therapy MasterClass*, EPJ Web of Conferences, 2020 [131],
- Piotr Nowakowski, Piotr Żórawski, Krzysztof Cabaj, Wojciech Mazurczyk, *Study of the Error Detection and Correction Scheme for Distributed Network Covert Channels*, ARES 2021: Proceedings of the 16th International Conference on Availability, Reliability and Security, 2021 [141],
- Piotr Nowakowski, Piotr Żórawski, Krzysztof Cabaj, Wojciech Mazurczyk, *Detecting Network Covert Channels using Machine Learning, Data Mining and*

*Hierarchical Organisation of Frequent Sets*, Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA), 2021 [142],

- Piotr Nowakowski, Piotr Żórawski, Krzysztof Cabaj, Wojciech Mazurczyk, *Network covert channels detection using data mining and hierarchical organisation of frequent sets: an initial study*, ARES 2020: Proceedings of the 15th International Conference on Availability, Reliability and Security, 2020 [143],
- Piotr Nowakowski, Piotr Zórawski, Krzysztof Cabaj, Marcin Gregorczyk, Maciej Purski, Wojciech Mazurczyk, *Distributed packet inspection for network security purposes in software-defined networking environments*, ARES 2020: Proceedings of the 15th International Conference on Availability, Reliability and Security, 2020 [144],
- Marcin Gregorczyk, Piotr Żórawski, Piotr Nowakowski, Krzysztof Cabaj, Wojciech Mazurczyk, *Sniffing Detection Based on Network Traffic Probing and Machine Learning*, IEEE Access, 2020 [145],
- Krzysztof Cabaj, Piotr Żórawski, Piotr Nowakowski, Maciej Purski, Wojciech Mazurczyk, *Efficient distributed network covert channels for Internet of things environments*, Journal of Cybersecurity, 2020 [146],
- Piotr Nowakowski, Piotr Zorawski, Krzysztof Cabaj, Wojciech Mazurczyk, *Securing Modern Network Architectures with Software Defined Networking*, 2019 International Conference on Computational Science and Computational Intelligence (CSCI), 2019 [147],
- Krzysztof Cabaj, Marcin Gregorczyk, Wojciech Mazurczyk, Piotr Nowakowski, Piotr Żórawski, *Sniffing Detection within the Network: Revisiting Existing and Proposing Novel Approaches*, ARES 2019: Proceedings of the 14th International Conference on Availability, Reliability and Security, 2019 [148],
- Krzysztof Cabaj, Wojciech Mazurczyk, Piotr Nowakowski, Piotr Żórawski, *Fine-tuning of Distributed Network Covert Channels Parameters and Their Impact on Undetectability*, ARES 2019: Proceedings of the 14th International Conference on Availability, Reliability and Security, 2019 [149],
- Krzysztof Cabaj, Marcin Gregorczyk, Wojciech Mazurczyk, Piotr Nowakowski,

Piotr Żórawski, *Network Threats Mitigation Using Software-Defined Networking for the 5G Internet of Radio Light System*, Security and Communication Networks, 2019 [150],

- Krzysztof Cabaj, Wojciech Mazurczyk, Piotr Nowakowski, Piotr Żórawski, *Towards Distributed Network Covert Channels Detection Using Data Mining-based Approach*, ARES 2018: Proceedings of the 13th International Conference on Availability, Reliability and Security, 2018 [151],
- Krzysztof Cabaj, Marcin Gregorczyk, Wojciech Mazurczyk, Piotr Nowakowski, Piotr Żórawski, *SDN-based Mitigation of Scanning Attacks for the 5G Internet of Radio Light System*, ARES 2018: Proceedings of the 13th International Conference on Availability, Reliability and Security, 2018 [152].

# C. Podziękowania od Kolaboracji ALICE

ALICE Collaboration
https://alice-collaboration.web.cern.ch

Barbara Erazmus
ALICE Experiment Deputy Spokesperson
EP Department - CERN
CH-1211 GENEVE 23

Tel. direct: +41 22 766 4490
Tel.Secretariat: +41 22 766 2525
Email: Barbara.Erazmus@cern.ch

Our reference: ALICE/BE/jc/21-12-2022

Geneva, 21 December 2022

To whom it may concern

Since 2011 the Masterclass Team from the Warsaw University of Technology has worked on developing the software for the ALICE Masterclasses - Looking For Strange Particles. This work has allowed to organize more than 200 ALICE Masterclasses all over the world. In 2019 the Warsaw team developed a new, unified version, consolidating other ALICE exercises. Since 2020, thanks to the Warsaw team's efforts, a web-based version is also available, allowing for easy organization of remote masterclasses and using the software in places with limited infrastructure. Piotr Nowakowski was the main person responsible for the development from scratch of this new version of the software as well as its improvement and maintenance during the following years. This web-based version was particularly useful during the years of the COVID pandemic, when masterclasses were mainly held online. It also greatly expanded the possibilities of utilizing the program, giving the opportunity to students in remote places, who do not have the opportunity to travel to Universities, to participate in online masterclasses.

In the four years between 2018 and 2022 the ALICE Collaboration performed significant upgrades of the detector and the data acquisition system in preparation for the next period of data taking with increased luminosity. The team from the Warsaw University of Technology developed the Event Display, a collision data visualization tool required for the day to day operation of the experiment. Piotr Nowakowski had a key contribution to the development, debugging and maintenance of the Event Display over the period of his PhD studies. He also supervised the visualization system during the months leading to a successful launch of ALICE during the restart of the LHC in July 2022 and the first ion collision test in November 2022.

We thank Piotr Nowakowski for his invaluable contribution to both the ALICE masterclass software and the ALICE event display, for his hard work and dedication to both projects.

Your sincerely,

Barbara Erazmus
ALICE Collaboration Deputy Spokesperson

Despina Hatzifotiadou
ALICE Collaboration Outreach Coordinator

Istituto Nazionale di Fisica Nucleare
DIVISIONE RICERCA
GRUPPO III – ESPERIMENTO ALICE

tel. + 39 06 9403 2312
tel. + 39 06 9403 8112
email: Federico.Ronchetti@lnf.infn.it

To whom it may concern
Geneva, 12 December 2022

Subject : Statement to support Piotr Nowakowski

In the four years between 2018 and 2022 ALICE Collaboration performed significant upgrades of the hardware and software of the detector in preparation for the next period of data taking. The Team from Warsaw University of Technology worked on developing the Event Display, an online collision data visualization tool running continuously required for day to day operation of the experiment in the ALICE Control Room.

We would like to thank Piotr Nowakowski for contributions to the Event Display over the period of his PhD studies, especially for his dedication for the project and direct supervision of the visualization system in months leading to a successful launch of ALICE during the first restart of LHC in July 2022 and the first ion collision test in November 2022.

**Federico Ronchetti**
ALICE Run Coordinator (2014-15) and (2021- 2022)
Senior Staff researcher at INFN-FRASCATI
Scientific Associate at CERN

*I.N.F.N. / L.N.F.* – Via E. Fermi, 40 – I - 00044 Frascati (RM) – Italy