Uniwersytet Warszawski
Wydział Fizyki

Raport

Zredukowany model wzrostu kanałów i struktur dendrytycznych

część 3.

Oleg Kmechak

Warszawa, Kwiecień 2019.

Treść

Wstęp

Już w poprzednim raporcie były zaimplementowane wszystkie metody zastosowane w celu rozwiązania problemu. W szczególności zostały zaimplementowane klasy, które generowały geometrie rzeki i mogły zostać w łatwy sposób modyfikowane, brzegi symulacji i sama klasa dla rozwiązywania PDE.
Okazało się jednak, że z płynem czasu, w miarę powiększania się kodu programu, błędów zaczęło przybywać. Stopień skomplikowania kodu spowodował, że głównym problemem stało się to, że ciężko było znaleźć przyczynę poszczególnych błędów.

Refactoring

CMake

Skrypty CMake chociaż nie miały błędów i generowały binarny plik programu poprawnie (static albo dynamic linking i debug albo wersje release). Jednak miały w sobie sporo ograniczeń na konkretną konfigurację systemu.
Po przeczytaniu oficjalnej dokumentacji zostały poprawione cmake skrypty w celu większej krosplatformowości i zostało dodanych kilka opcji w konfiguracji (włączanie/wyłączanie generacji kodu, dokumentacji i testów).
Chociaż krosplatformowość nie jest priorytetem w projekcie, jednak już na przykład dokumentacja była generowana na platformę Windows bez żadnych modyfikacji pod ten konkretny system.

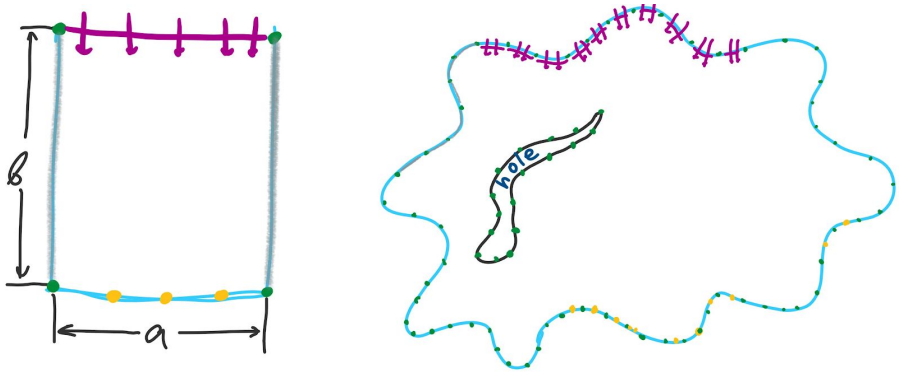
Modyfikacja programu - przeprojektowanie

Jak już zaznaczono we wstępie, w miarę rozwoju programu zaczęły się pojawiać niedoskonałości zaprojektowanych klas.

Z tego powodu jeszcze raz zrobiono dokładną analizę koncepcji programu i dokonano istotnych zmian i modyfikacji tej koncepcji. Wprowadzenie tych zmian bardzo istotnie pomogło ustabilizować i zoptymalizować strukturę programu.

Klasa Border

Na poniższym rysunku została zaprezentowana klasa Border.
Teraz, za wyjątkiem prostokątnego regionu symulacji, można będzie zadać dowolny format msh wersji 2 - z pomocą pliku (wiecej szczegółów dotyczących tego zagadnienia będzie na stronie internetowej projektu w dokumentacji do f-cji Border::ReadFromFile())

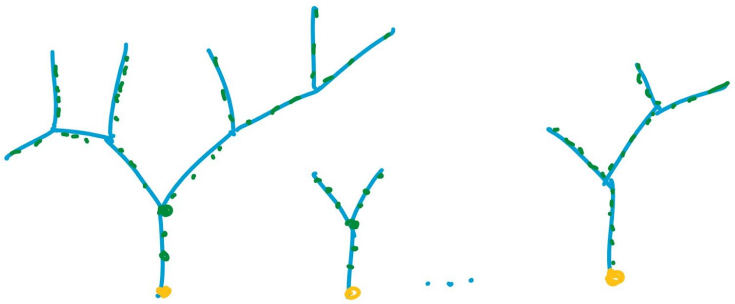


Zielone punkty - punkty na których jest budowany mesh.
Żółte - punkty źródeł dendrytów.
Szara linia - adiabatyczne warunki brzegowe,
Niebieskie linie - warunek brzegowy że stałym znaczeniem.
Fioletowe linie - stały strumień.
Hole - dziury w regionie.

Klasy Branch, Tree i fcja Boundary Generator

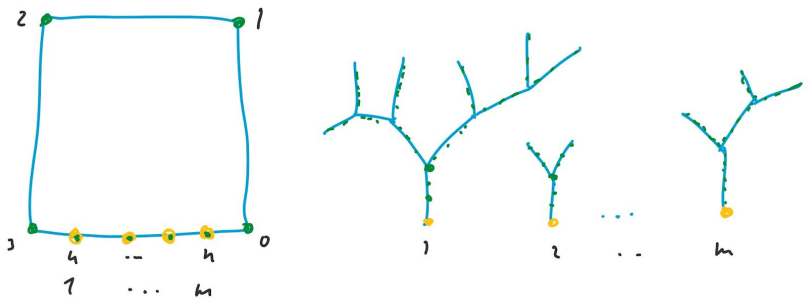
Te klasy były przeprojektowane w podobny sposób jak i ich poprzednia wersja

1.Ten wzór odpowiada klasie **Branch,** która w sumie jest obwolutą wektora.



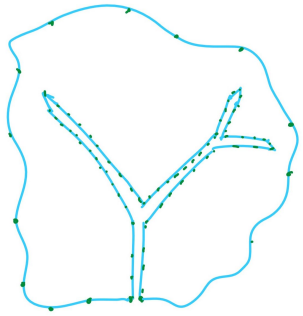
2.Dalej objekty **Branch** są łączane pomiedzy sobą w drzewa z pomocą klasy **Tree.** Drzew może być wiele.

3. Ostatnim etapem jest kombinowanie razem klasy **Border** i **Tree** z pomocą f-cji **BoundaryGenerator().**



Każde pojedyncze drzewo ma swój numer ID, tak samo każdy punkt źródła klasy Border ma też swój ID. Dzięki temu wiadomo, jak łączyć te geometrie.

Końcowy wynik wygląda tak:



Jedna zamknięta linia,
która obiega wokół cały region.

Test Driven Development. BOOST.Tests.

Chociaż przeprojektowanie kodu pomogło uprościć program, to jako główny atut zastosowanego rozwiązania należy wskazać możliwość korzystania z **testów.**

Test Driven Development(TDD) zaleca, że testy powinny być napisane przed implementacja kodu, a kod należy pisać tylko w zakresie niezbędnym do pokrycia zaplanowanych testów.
To zaś powoduje, że te klasy powinny być zaprojektowane i zaplanowane na wcześniejszym etapie prac.
Korzystanie z tej metody w praktyce daje wysoką skuteczność, a powstały kod - swoją niezawodność.
To powoduje, że etap projektowania staje się jednym z najważniejszych w całym procesie.
Zgodnie z książką “Idealny Programista” projektowanie powinno zająć połowę czasu trwania

projektu,, a samo programowanie (implementacja) - tylko 20% i powinno być prawie że rutynowe.

BOOST.Tests. był wybrany jako framework do napisania testów. Ponieważ już wcześniej projekt miał powiązanie z Boost.Program_Options i ponieważ Boost jest przedłużeniem de-facto STL. toteż BOOST.Test już był podłączony do projektu z pomocą CMake jeszcze w 2 części raportu. Ale wtedy brakowało szczegółowej wiedzy praktycznej, jak z niego skutecznie korzystać.

Po zapoznaniu się z dokumentacją okazało się, że jest bardzo przyjazdny w użyciu i umożliwia wiele użytecznych opcji, na przykład:

- Generacja różnych danych wejściowych.
- Testowanie wyjątkowych (exception/throw) sytuacji.
- Porównywanie wyników do plików
- Porównywanie float liczb
- Bardzo prosty interfejs: BOOST_TEST().

Przykład programu:

```
BOOST_AUTO_TEST_CASE( constructor_and_methods,
   *utf::tolerance(eps))
{
   //empty borderMesh
   tethex::Mesh border_mesh;
   Border border(border_mesh);

   BOOST_TEST(border.GetHolesId().empty());
}
```

Po kompilacji programu wystarczy w konsoli wprowadzić
> **make test**

W danym momencie wszystkie zaprojektowane klasy i funkcje są pokryte testami.

Doxygen. Dokumentacja. Strona Internetowa. UML diagramy.

W drugiej części raportu Doxygen już był podłączony do projektu z pomocą CMake, ale brakowało wiedzy praktycznej, jak z niego korzystać. Po zapoznaniu się z dokumentacją została wygenerowana dokumentacja programu.

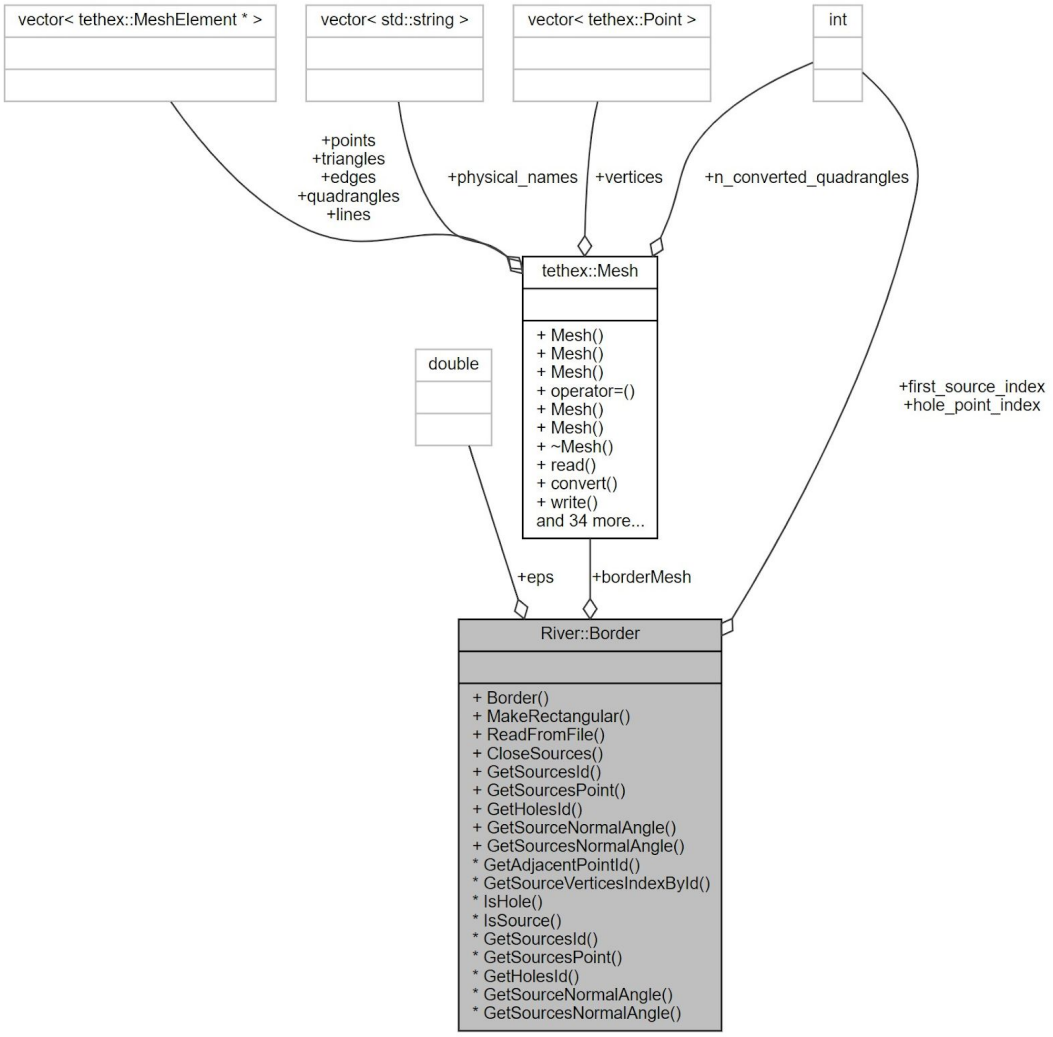
Na początku składa się ona z jednej strony, która zawiera ogólne założenia i opis projektu, a dalej - ze stron, które pokrywają kod, takie jak Klasy, Namespaces, Pliki, Funkcje.
Te wszystkie opisy mieszczą się jako komentarze programu, na przykład:

```
/**
 * Add border line to each source points.
 *
 * As each source consist of two points with gap in between,
 * this function will fill this gap by adding line.
 * \param boundary_id of additional lines.
 * Used only for test purposes.
 */
 Border& CloseSources(int boundary_id);
```

Strona internetowa
GitHub dla każdego repozytorium proponuje hosting dla statycznych stron web. Wykorzystując tę możliwość wystarczy tylko umieścić stronę w folderze *docs* w repozytorium i zmienić kilka opcji w ustawieniach projektu na githubie.

Link na strone: https://okmechak.github.io/RiverSim/

Diagramy UML
Doxygen też pozwala wygenerować UML diagrame, Call Diagram i inne. Wszystkie one zostały wygenerowane na stronie projektu, na przykład diagram klasy Border.



Linia z rombem na końcu oznacza, że klasa Border mieści w sobie tą klasę, na przykład klasę tethex::Mesh.

- \+ Wskazuje że pole jest w statusie *public*.
- Gwiazdka - że w statusie *private*.