Scientific Bulletin of Chełm
Section of Mathematics and
Computer Science
No. 1/2009

ZASTOSOWANIE TECHNOLOGII XNA DO RENDEROWANIA MAP CYFROWYCH

MATEUSZ KAPKA

Uniwersytet Marii Curie-Skłodowskiej w Lublinie

Streszczenie. Niniejsza praca porusza zagadnienie efektywnego i wydajnego renderowania map cyfrowych. W celu rozwiązania tego problemu napisałem aplikację prezentującą dane przestrzenne. Aby uzyskać dużą wydajność mapa rednerowana jest przy użyciu technologii XNA. Kolejnym rozwiązaniem poprawiającym wydajność jest buforowanie danych. Ostatnim użytym prze mnie mechanizmem jest asynchroniczne pobieranie danych dla każdej z warstw.

1. Wstęp

Prezentowana praca porusza temat renderowania i przechowywania w pamięci danych przestrzennych (map geograficznych). Temat ten ma bardzo duże zastosowanie we wszelakiego rodzaju systemach typu GIS. Systemami typu GIS nazywamy aplikacje prezentujące i analizujące dane przestrzenne.

Przedstawiony tu materiał dotyczy aspektu prezentacji danych przestrzennych w systemach GIS. Podstawowym, poruszanym tutaj problemem, jest szybkość i płynność całego procesu wyświetlania. O ile ma to mniejsze znaczenie w przypadku aplikacji webowych (chociaż i w ich przypadku czas, który zabiera stworzenie obrazu danej mapy jest ważny), o tyle zaczyna być bardzo ważną kwestią w przypadku aplikacji desktopowych. Najczęściej stosowane podejście, czyli renderowanie mapy przy wykorzystaniu bibliotek GDI+ firmy Microsoft, posiada wiele wad. Po pierwsze renderowanie mapy w taki sposób nigdy nie będzie do końca płynne. Biblioteki te są dość wolne i nie zostały zaprojektowane do dynamicznego tworzenia szybko zmieniającego się obrazu. Drugą wadą jest wykorzystanie do procesu renderowania tylko procesora głównego (CPU). Jest to, samo w sobie, bardzo wolne, ponieważ procesor CPU nie jest stworzony z myślą o przetwarzaniu grafiki, a ponadto obciąża cały system, przez co całość aplikacji działa dużo wolniej.

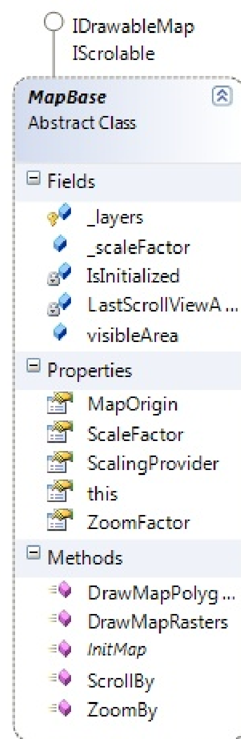
Prezentowana aplikacja opiera się na innym rozwiązaniu. Zastosowałem w niej technologię XNA, będącą następcą technologii DirectX. XNA jest w zasadzie biblioteką obudowującą technologię DirectX o wiele dodatkowych funkcjonalności oraz ułatwiającą użycie funkcjonalności już istniejących. Podstawowym powodem, który zadecydował

Treść artykułu była prezentowana w czasie *VIII Konferencji Informatyki Stosowanej* (Chełm 29 - 30 maja 2009 r.)

o użyciu właśnie XNA jest fakt wykorzystania przez nią do renderowania obrazu karty graficznej. Likwiduje to podstawową wadę technologii GDI+, czyli wykorzystanie do renderowania obrazu procesora CPU. Dzięki wykorzystaniu mechanizmów karty graficznej proces renderowania mapy odbywa się błyskawicznie i nie obciąża procesora, dzięki czemu jest on w stanie szybciej analizować, pobierać i przetwarzać dane przestrzenne. Dodatkowo, dzięki zastosowaniu buforowania danych, udało mi się uzyskać płynne przesuwanie, przybliżanie oraz oddalanie mapy (bez efektu czekania ułamka sekundy na doczytanie danych).

2. Renderowanie danych przestrzennych

Na początku chciałbym przedstawić sposób renderowania danych przestrzennych przy wykorzystaniu technologii Microsoft XNA.

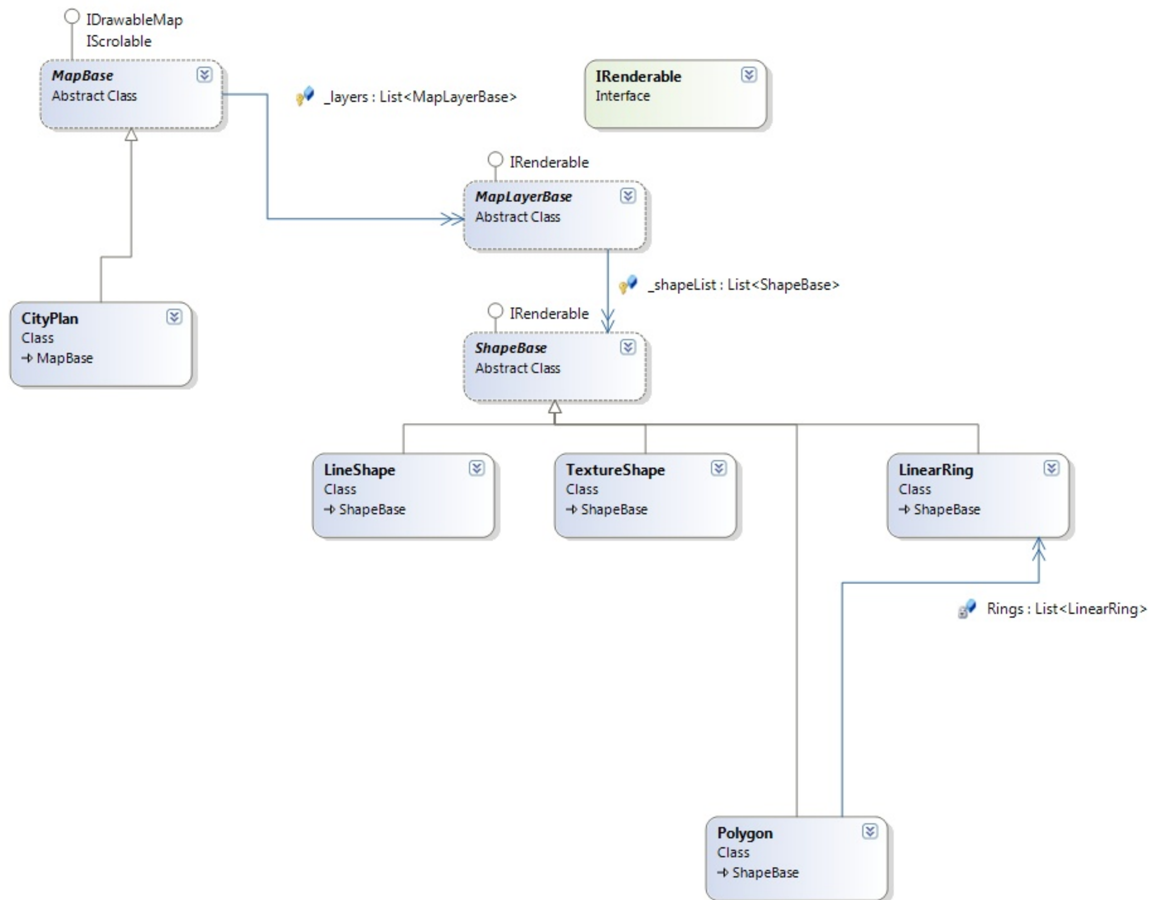


Rysunek 1. Klasa "MapBase"

Postaram się opisać sposób przechowywania danych przestrzennych w pamięci. Wszystkie dane przestrzenne pogrupowane są w warstwy. Warstwy te są całkowicie od siebie niezależne i przedstawiają odrębne i logicznie pogrupowane kolekcje obiektów. Przykładem może tutaj być np. mapa składająca się z warstw: dróg, budynków i opisów ulic. Każda warstwa z kolei składa się z wielu obiektów. W przypadku mapy rastrowej obiektami takimi będą bitmapy (np. zdjęcia satelitarne czy zdjęcia budynków z lotu ptaka). W przypadku map wektorowych obiektami, z których składają się poszczególne warstwy, będą kształty geometryczne. Wszystkie współrzędne obiektów przechowywanych w pamięci są przechowywane w jednostkach geograficznych, i dopiero podczas procesu renderowania konwertowane są na współrzędne ekranowe. Dochodzi tutaj także problem

indeksowania przestrzennego, tzn. wyboru tylko tych obiektów, które są w danym momencie widoczne. Szczegółowy mechanizm pobierania, indeksowania i wyboru obiektów buforowanych w pamięci przedstawię w następnym rozdziale.

W następnej części tego rozdziału postaram się przedstawić, w jaki sposób moja aplikacja przechowuje oraz wyświetla dane. Na Rys. 2 prezentuję diagram klas przedstawiający rysowanie mapy.



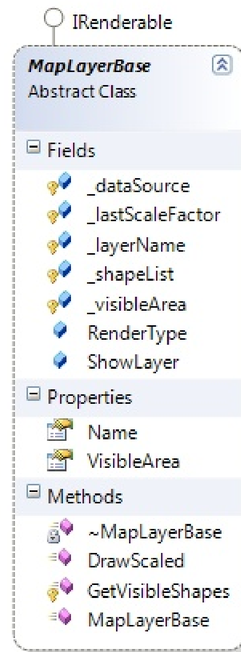
Rysunek 2. Diagram klas mechanizmu renderowania mapy

Klasą centralną w mojej aplikacji jest abstrakcyjna klasa "MapBase". Zawiera ona w sobie całą wysokopoziomową logikę renderowania, przesuwania i manipulacji skalą mapy. Klasa ta jest abstrakcyjna, ponieważ sama w sobie nie posiada definicji warstw biorących udział w renderowaniu. Aby je określić, należy stworzyć klasę dziedziczącą po klasie "MapBase" i w jej konstruktorze zainicjalizować wszystkie wymagane warstwy.

Podstawowymi metodami klasy "MapBase" są metody "DrawMapPolygons" oraz "DrawMapRasters". Odpowiadają one za renderowanie warstw rastrowych i wektorowych. W ich środku delegowane są wywołania metod rysujących do poszczególnych warstw. Konieczne jest zastosowanie dwóch metod, ponieważ ze specyfiki technologii XNA wynika konieczność renderowania grafiki czysto wektorowej oraz grafiki rastrowej w dwóch oddzielnych przejściach (tzw. efektach). Dwiema kolejnymi ważnymi metodami

są metody "ScrollBy" oraz "ZoomBy". Pierwsza z nich przesuwa widok mapy o określoną wartość, natomiast druga zmienia skalę, w jakiej prezentowana jest mapa. Logika tych metod polega na zmianie określonych atrybutów obiektu klasy "ScaleFactor", przechowującego właściwości obecnego widoku mapy (takie jak stopień przybliżenia mapy oraz obecna pozycja).

Każda mapa składa się z wielu warstw. Rolę warstw w mojej aplikacji odzwierciedla klasa "MapLayerBase":



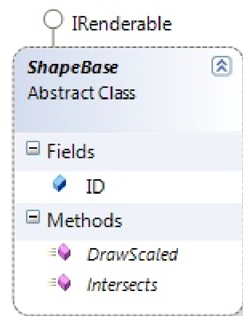
Rysunek 3. Klasa "MapLayerBase"

Klasa ta jest odpowiedzialna za rysowanie wszystkich należących do niej obiektów oraz za inicjowanie procesu pobierania danych ze źródła danych. Tak jak w przypadku klasy "MapBase" jest to klasa abstrakcyjna. Powodem zastosowania takiego podejścia jest skalowalność całego rozwiązania. Tak jak w przypadku "MapBase", to klasy dziedziczące określają szczegółowe cechy danego obiektu. W przypadku klasy "MapLayerBase" cechami tymi jest nazwa warstwy (wykorzystywana później przy pobieraniu danych) oraz jej rodzaj (rastrowy czy wektorowy). Dzięki zastosowaniu takiego wzorca projektowego cała faktyczna logika przetwarzania i renderowania warstw dostępna jest w jednej klasie, a jednocześnie dodawanie nowych typów warstw (czyli *de facto* skalowanie rozwiązania) jest bardzo proste i intuicyjne. Główną metodą klasy "MapLayerBase" jest metoda "DrawScaled". Jest ona implementacją interfejsu "IRenderable" i jej zadaniem jest faktyczne wyrenderowanie wszystkich kształtów należących do danej warstwy. Do jej odpowiedzialności należy także inicjowanie procesu asynchronicznego pobierania danych przestrzennych. Pola "RenderType" oraz "Name" określają nazwę warstwy oraz jej rodzaj. Rodzaj warstwy jest sprawdzany podczas wywoływania metod "DrawMapPolygons" oraz "DrawMapRasters" klasy "MapBase" i określa on czy dana warstwa jest rastrowa czy wektorowa, co z kolei determinuje podczas wywołania której funkcji

zostanie ona wyrenderowana. Pole "ShowLayer" określa z kolei czy dana warstwa powinna być rednerowana czy nie (dzięki niemu można określić, które warstwy mają być w danym momencie prezentowane).

Klasa "MapLayerBase" służy także jako kontener obiektów graficznych należących do danej warstwy. Obiekty te (tak rastrowe, jak i wektorowe) reprezentowane są przez wszystkie klasy dziedziczące po klasie abstrakcyjnej "ShapeBase".

Cały kod renderujący dany obiekt na ekranie skupiony jest w klasach potomnych po klasie "ShapeBase". Za proces renderowania pojedynczego obiektu odpowiedzialna jest tutaj metoda "DrawScaled". Jest to metoda abstrakcyjna, a jej implementacja w klasach potomnych faktycznie renderuje dany obiekt. Uzyskałem w ten sposób niezależność logiki zarządzania warstwami od implementacji renderowania konkretnych obiektów. Dodawanie nowych obiektów do całego rozwiązania jest bardzo proste i wymaga jedynie dwóch rzeczy. Po pierwsze należy stworzyć nową klasę dziedziczącą po klasie "ShapeBase", implementującą logikę przechowywania atrybutów oraz renderowania danego obiektu. Dodatkowo należy zaimplementować tworzenie nowego obiektu danej klasy w obiekcie providera danych, który używany jest do pobierania danych przestrzennych.



Rysunek 4. Klasa "ShapeBase"

Wybór technologii XNA narzucił kilka rozwiązań w architekturze projektu. Aplikacje napisane w technologii XNA działają na trochę innej zasadzie niż zwykłe aplikacje windowsowe. Każda typowa aplikacja wykorzystująca WinAPI działa na zasadzie obsługiwania zdarzeń *(Event-driven)*. Każda taka aplikacja posiada tzw. kolejkę zdarzeń *(Message pipe)*, do której system operacyjny podaje np. zdarzenie ruchu myszki czy zdarzenie kliknięcia myszką gdzieś na ekranie. Faktyczne działanie aplikacji polega na przechwytywaniu tych zdarzeń i ich obsłudze. XNA z kolei wykorzystuje mechanizm odpytywania się urządzeń o ich stan *(pooling)*. Aplikacja XNA działa w nieskończonej pętli, gdzie każde przejście generuje następną klatkę wyświetlaną na ekranie. Podczas każdego przejścia wywoływane są dwie metody z klasy nadrzędnej aplikacji:

- Update - metoda odpowiadająca za odpytywanie poszczególnych urządzeń o ich stan (np. które klawisze zostały wciśnięte czy też jaka jest aktualna pozycja myszki na ekranie) i przetwarzająca logikę aplikacji,
- Draw - metoda renderująca daną klatkę.

Zastosowanie takiego mechanizmu umożliwia dodatkowe zwiększenie płynności, ponieważ mamy pełną kontrolę nad tym, co w każdej klatce prezentowane jest na ekranie.

Tak jak wspominałem wcześniej renderowanie odbywa się w dwóch przejściach. Kod renderujący mapę przedstawiam poniżej:

```
graphics.GraphicsDevice.Clear(Color.White);
graphics.GraphicsDevice.VertexDeclaration = new VertexDeclaration(
graphics.GraphicsDevice, VertexPositionColor.VertexElements);
spriteBatch.Begin(SpriteBlendMode.AlphaBlend);
basicEffect.Begin();
foreach (EffectPass pass in basicEffect.CurrentTechnique.Passes)
{
pass.Begin();
mapa.DrawMapPolygons(drawContainer);
pass.End();
}
basicEffect.End();
GraphicsDevice.VertexDeclaration = new VertexDeclaration(GraphicsDevice,
VertexPositionTexture.VertexElements);
TexturingEffect.Begin();
foreach (EffectPass pass in TexturingEffect.CurrentTechnique.Passes)
{
pass.Begin();
mapa.DrawMapRasters(drawContainer);
pass.End();
}
TexturingEffect.End();
base.Draw(gameTime);
```

LISTING 1. Kod renderujący mapę

W powyższym kodzie widać dwa przejścia renderujące. Pierwsze renderuje obiekty wektorowe, natomiast drugie renderuje obiekty rastrowe. Dla każdego z przejść najpierw tworzona jest deklaracja rodzaju używanych Vertex-ów, a następnie wywoływana jest odpowiednia metoda renderująca z obiektu "mapa", który jest obiektem klasy "MapBase".

Poniżej przedstawiam przykładowy kod renderujący pojedynczą krzywą (metodę "DrawScaled" klasy "LineBase"):

```
VertexPositionColor[] pointVertexes =
new VertexPositionColor[points.Length];
int I = 0;
foreach (Point point in points)
{
ViewArea viewArea = scaleFactor.ConvertWorldCoord2View(new
ViewArea(point));
pointVertexes[i] = new VertexPositionColor(
new Vector3((float) (viewArea.X - graphicContainer.ScreenArea.Width/2.0),
(float) (viewArea.Y - graphicContainer.ScreenArea.Height/2.0),
0),
Color.Red);    i++;
}
```

```
graphicContainer.device.DrawUserPrimitives<VertexPositionColor>(
PrimitiveType.LineStrip,
pointVertexes,
0,
pointVertexes.Length - 1);
```

Listing 2. Kod renderujący pojedynczą krzywą

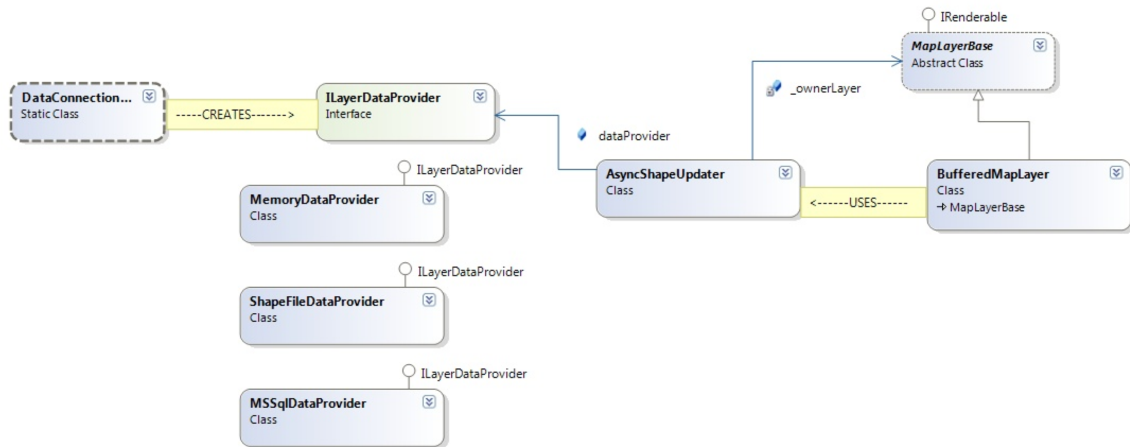
W kodzie przedstawionym powyżej obiekt "scaleFactor" zawiera w sobie wszystkie atrybuty identyfikujące widok mapy aktualnie prezentowany użytkownikowi. Metoda "ConvertWorldCoord2View" zamienia koordynaty w jednostkach geograficznych (ponieważ w takim formacie trzymane są one w pamięci) na współrzędne ekranowe. Następnie tworzony jest obiekt klasy "VertexPositionColor" będący reprezentacją danego punktu w przestrzeni, przystosowaną do przetwarzania przez kartę graficzną. Wszystkie powyższe kroki wykonywane są w pętli dla każdego punktu składającego się na dany obiekt. Następnie wywoływana jest metoda "DrawUserPrimitives" na rzecz klasy "GraphicalDevice" reprezentowanego przez obiekt "device", która renderuje dany zbiór punktów jako prostą. Obiekt "GraphicalDevice" jest obiektem wbudowanym w biblioteki XNA. Jego zadaniem jest bezpośredni dostęp do karty graficznej w celu faktycznego renderowania grafiki. Obiekt "graphicContainer" jest kontenerem zawierającym w sobie wszystkie obiekty wykorzystywane w procesie renderowania.

3. Pobieranie i buforowanie danych

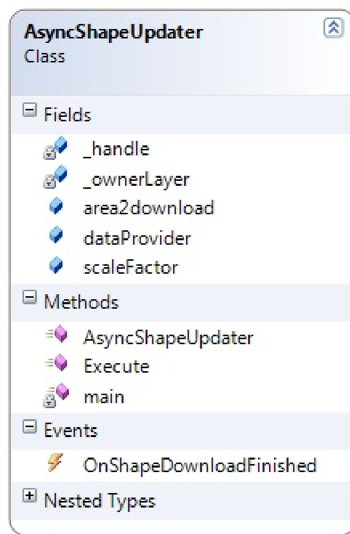
Kolejną bardzo istotną kwestią, jeśli chodzi o wydajność renderowania mapy systemu GIS, jest pobieranie danych i odpowiednie ich buforowanie. Jest to o tyle istotne, że bez odpowiedniego systemu asynchronicznego pobierania danych oraz dostatecznego ich buforowania, nie ma tak naprawdę znaczenia, jak wydajny będzie sam mechanizm renderowania. Wąskim gardłem całego procesu będzie zawsze konieczność oczekiwania na pobranie danych podczas przesuwania czy przeskalowywania mapy. W mojej aplikacji zastosowałem mechanizm konfigurowalnego buforowania danych oraz asynchronicznego pobierania danych dla każdej z warstw w oddzielnym wątku. Zapewnia to wysoką płynność procesu renderowania oraz dodatkowo umożliwia pobieranie każdej z warstw z zupełnie innego źródła (np. jedna warstwa może pochodzić z bazy danych, druga z pliku, a trzecia np. z Web serwisu). Na Rys. 5 przedstawiłem diagram klas całego rozwiązania. Diagram ten rozszerza diagram przedstawiony na Rys. 2 o klasę "BufferedMapLayer" dziedziczącą po klasie "MapLayerBase". To właśnie ta klasa jest głównym obiektem odpowiadającym za pobieranie i buforowanie danych. Wykorzystując mechanizm dziedziczenia zachowuje wszystkie funkcjonalności klasy "MapLayerBase", rozszerzając ją o mechanizm buforowania danych. Za pobieranie danych odpowiedzialna jest klasa "AsyncShapeUpdater".

Klasa ta obudowuje wątek reprezentowany przez metodę "main" umożliwiając łatwą jego inicjalizację oraz uruchomienie. Wykorzystywany jest tutaj mechanizm technologii .NET tzw. "ThreadPool". Znacząco przyśpiesza on tworzenie nowych wątków, ponieważ nie inicjalizuje za każdym razem nowego wątku, tylko wykorzystuje wątki wcześniej utworzone, przechowywane w odpowiedniej puli. Klasa "AsyncShapeUpdater" posiada dodatkowo mechanizm kolejkujący zapytania. Uniemożliwia on odpalenie więcej niż jednego wątku pobierającego dla każdej z warstw. Zapobiega to niepotrzebnemu utworzeniu wielu wątków podczas przesuwania bądź przeskalowywania map. Podczas inicjalizacji

klasy "AsyncShapeUpdater" ustawiane jest pole "dataProvider" będące referencją na interfejs "IlayerDataProvider" odpowiadający za pobieranie danych ze źródła danych.



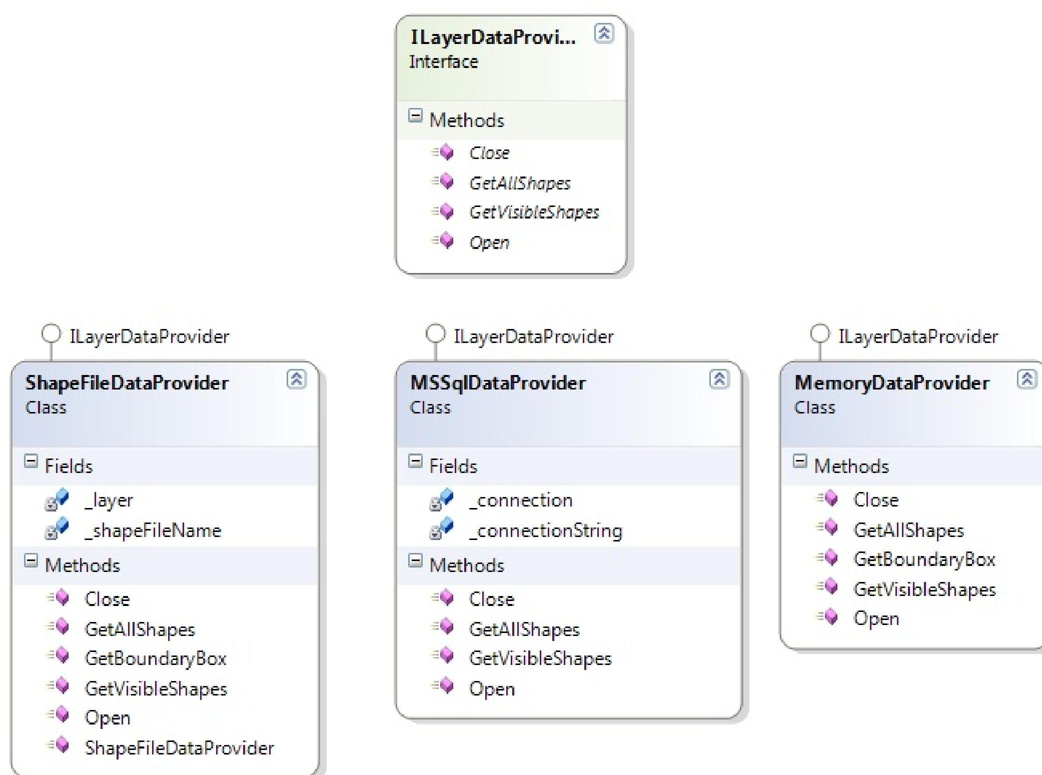
Rysunek 5. Diagram klas mechanizmu pobierania danych



Rysunek 6. Klasa "AsyncShapeUpdater"

Za faktyczne pobieranie danych odpowiadają oczywiście poszczególne implementacje interfejsu. Każda z nich reprezentuje inny rodzaj źródła danych. W tym momencie w aplikacji dostępne są trzy rodzaje źródeł:

- ShapeFileDataProvider - pobieranie danych z pliku binarnego zapisanego w standardzie "OGC",
- MSSqlDataProvider - pobieranie danych z bazy danych MS SQL Server,
- MemoryDataProvider - pobieranie danych ze strumienia umieszczonego w pamięci operacyjnej.



Rysunek 7. Rodzina klas "IlayerDataProvider"

Za przyporządkowanie odpowiedniego providera do konkretnej warstwy odpowiadają ustawienia zapisane w pliku konfiguracyjnym aplikacji, a za faktyczne tworzenie odpowiedniego obiektu providera odpowiada statyczna klasa "DataConnectionFactory". Interfejs "IlayerDataProvider" udostępnia metody implementowane przez wszystkie klasy potomne. Metody "Open" oraz "Close" odpowiadają za otwieranie oraz zamykanie połączenia do źródła danych. W przypadku pliku jest to otwarcie strumienia, a z kolei w przypadku bazy danych jest to nawiązanie i rozłączenie połączenia z bazą danych. Najważniejszą metodą tej klasy jest metoda "GetVisibleShapes". Odpowiada ona za faktyczne pobieranie danych ze źródła danych. Jako parametr przyjmuje ona obiekt klasy "ScaleFactor" reprezentujący obecnie prezentowany widok mapy. Metoda zwraca listę obiektów "ShapeBase" reprezentujących wszystkie widoczne w tym momencie kształty.

Za buforowanie mapy odpowiada opisana wcześniej klasa "BufferedMapLayer". Samo buforowanie polega na pobieraniu ze źródła danych większej liczby kształtów niż jest w danym momencie wyświetlane i przechowywanie nadmiarowych kształtów w pamięci. Ilość pobieranych nadmiarowych danych zależy od ustawień w pliku konfiguracyjnym aplikacji.

4. Wnioski

Mechanizmy przedstawione w tym artykule umożliwiły stworzenie silnika renderowania mapy systemu GIS działającego szybko i płynnie. Udało mi się wyeliminować podstawowe mankamenty desktopowych systemów GIS wynikające z używania do renderowania mapy bibliotek GDI+. Zastosowanie technologii XNA umożliwiło odciążenie procesora,

który więcej mocy obliczeniowej może poświęcać na analizę danych przestrzennych, oraz wyeliminowanie przekłamań renderowanej mapy. Dodatkowo mechanizm buforowania i asynchronicznego pobierania danych umożliwił bardzo płynne i szybkie przesuwanie oraz przeskalowywanie mapy. Asynchroniczne pobieranie danych dodatkowo umożliwiło wykorzystanie oddzielnych źródeł danych dla każdej z warstw, przez co możliwe stało się jednoczesne wykorzystanie wielu rodzajów źródeł danych.

APPLICATION OF XNA TECHNOLOGY TO DIGITAL MAPS RENDERING

MATEUSZ KAPKA

Abstract. The subject of this article is to develop efficient way of rendering digital map data. In order to achieve this goal I have created an application presenting digital maps. To perform rendering process more efficiently I have used XNA technology. Additionally, the application has embedded mechanism of data buffering. The last mechanism presented in this article is asynchronous data loading for each individual layer.