WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Katedra Informatyki i Automatyki

Paweł Dziedzic

Post Mortem

Generator tekstur szumów - NoiseTex

Rzeszów, 2018

1. Podsumowanie projektu

Projekt można uznać za zakończony sukcesem, pomimo braku niektórych funkcjonalności określonych w założeniach, które jednak nie wpływałyby w znacznym stopniu na funkcjonalność programu. Udało się zrealicować aplikację z prostym interfejsem graficznym, możliwością generowania tekstur a także wpływania w znacznym stopniu na parametry algorytmu generującego. Aplikacja pozwala także na zapis wygenerowanej tekstury.

2. Elementy, które się udały

W tym rozdziale znajduje się lista elementów programu, które zostały uznane za udane.

2.1 Interfejs użytkownika

Niewątpliwą zaletą stworzonego interfejsu użytkownika jest jego prostota. Pozwala na wygodną parametryzacje tekstury przy użyciu wygodnych suwaków lub przez wpisanie wartości bezpośrednio w widoczne pole tekstowe. Program dba także, aby podane parametry miały sens, np. nie można ustalić ilości oktaw na wartość mniejszą lub równą 0. Dzięki użyciu technologii WPF i języka XAML interfejs aplikacji bardzo dobrze się skaluje.

2.2 Architektura aplikacji

Aplikacje napisano w oparciu o wzorzec projektowy MVVM dzięki czemu dalsze rozwijanie aplikacji będzie znacznie ułatwione. Wzorzec projektowy to nie jedyny mechanizm, który pomaga w zachowaniu porządku w projekcie. Każdy klasa zawierająca algorytm generowania tekstury musi implementować interfejs INoise oraz posiadać atrybut Noise. Takie rozwiązanie pozwala na zastosowanie mechanizmu refleksji do wyodrębniania tych klas, więc dodawanie kolejnych algorytmów wymaga jedynie napisania klasy, reszta systemów zostaje bez zmian.

2.3 Generator

Aktualnie aplikacja pozwala na tworzenie dwóch rodzajów szumów: szumu wartościowego (Value Noise) i szumu Perlina (Perlin Noise). Dodatkowo implementacja algorytmu fBm (Fractional Brownian Motion) oraz jego szeroka parametryzacja pozwoliła na tworzenie dość kompleksowych rezultatów.

3. Elementy, które się nie udały

W tym rozdziale znajduje się lista elementów, które się nie udały. Część z z nich może być naprawiona w kolejnych wersjach aplikacji, pozostałe wynikają z nieodpowiedniego doboru technologii bądź niewystarczającej wiedzy.

3.1 Optymalizacja

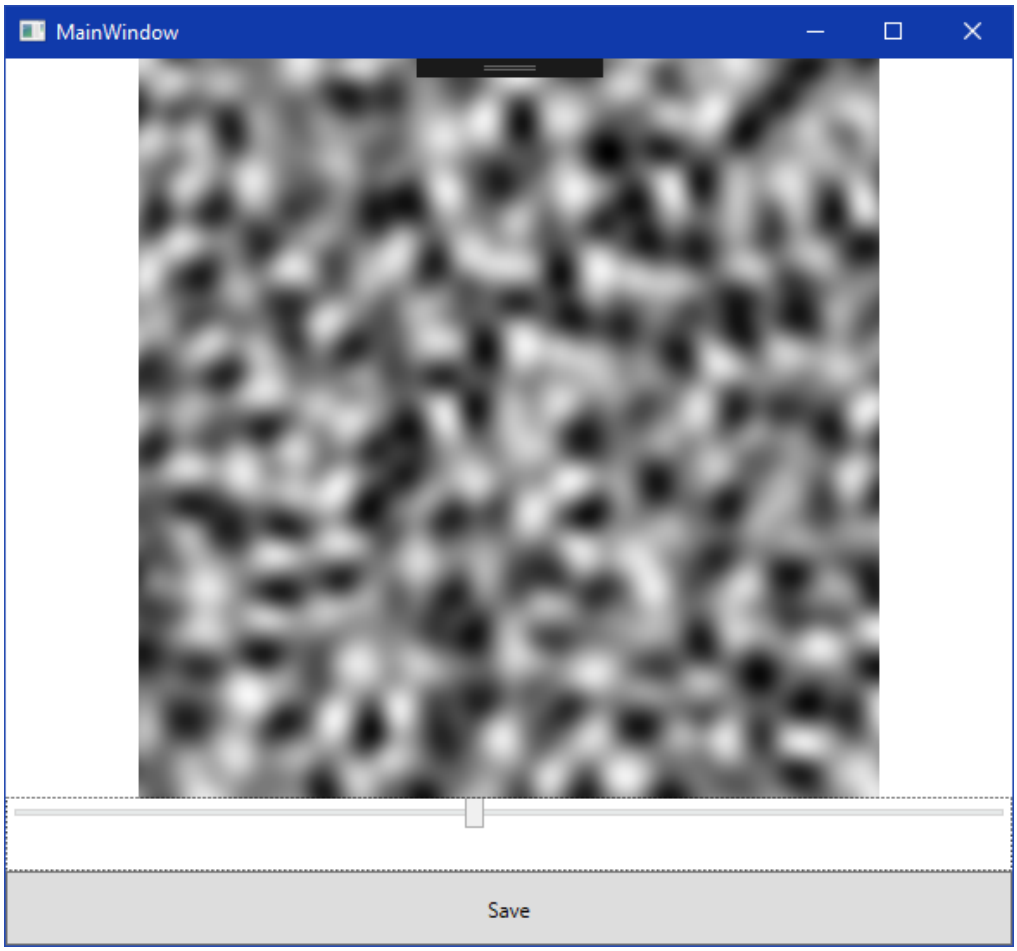
Dla małej rozdzielczości obrazu aplikacja działa wystarczająco szybko, aby można było widzieć rezultat mniej więcej w czasie rzeczywistym, jednak dla większych wymiarów (powyżej 1024x1024 pikseli) spowolnienie działania jest znaczące. Dla zachowania responsywności interfejsu zaimplementowano prostą wielowątkowość przy użyciu dostępnych w WPF mechanizmów. Sposób jej implementacji pozostawia jednak dużą ilość śmieci, które następnie musi uprzątnąć Garbage Collector platformy .NET. Rozwiązaniem problemu mogłoby być przerzucenie kosztownych obliczeń na kartę graficzną (o czym więcej w punkcie 3.3). W aktualnej wersji programu zastosowano obejście w postaci pola wyboru, gdy jest nieaktywne nowa tekstura jest generowana tylko w momencie naciśnięcia przycisku „Refresh”.

3.2 Obliczenia na GPU

Jednym ze sposobów rozwiązania problemu długiego czasu generowania tekstury jest użycie karty graficznej i języka shaderów. Problem pojawił się już na etapie wyboru technologii. Directx jest to API napisane w języku C++, żeby je wykorzystać w C# trzeba użyć mechanizmu marshallingu oraz kontrolki D3DImage, która jest źle udokumentowana. Lepszym rozwiązaniem jest użycie ImageEffect, który pozwala na wykonanie kodu HLSL bezpośrednio w WPF, ale i to rozwiązanie nie jest wolne od wad. Po pierwsze do klasy efektu trzeba dostarczyć skompilowany

kod HLSL, a Microsoft nie udostępnia kompilatora, który można użyć w kodzie programu, co jest bardzo uciążliwe na etapie testowania. Co prawda środowisko Visual Studio automatycznie kompiluje pliki .hlsl w trakcie budowania projektu, ale tylko dla projektów w języku C++. Rozwiązaniem tego problemu jest użycie zewnętrznych wrapperów, takich jak SharpDX lub SlimDX, które taką funkcjonalność udostępniają, a następnie zapisać skompilowany kod do strumienia w pamięci.

Dzięki użyciu wrappera możliwe jest wyświetlenie wygenerowanego szumu w zwykłej kontrolce obrazu WPF. Kolejne problemy pojawiają się podczas próby zapisu takiego obrazu. Z poziomu aplikacji nie ma dostępu do bufora tak wygenerowanego obrazu, jedyną możliwością jest użycie klasy RenderTargetBitmap, która pozwala na wyrenderowanie kontrolek do bitmapy. Takie renderowanie odbywa się jednak na procesorze i każdy ImageEffect renderowany jest przez emulator, który obsługuje maksymalnie Pixel Shader 2.0 (ps_2_0), który natomiast posiada ograniczenie ilości instrukcji do 96, co jest niewystarczające dla niektórych typów szumów. Dodatkowo RenderTargetBitmap nie renderuje w rzeczywistej rozdzielczości obrazu, ale aktualnej wielkości kontrolki, przez co ten sposób zapisu staje się bezużyteczny.



Rysunek 1. Testowa aplikacja generująca obraz przy użyciu karty graficznej

Powyższe problemy zostały zidentyfikowane w aplikacji testowej widocznej na rysunku 1. Generuje ona obraz szumu Simplex przy użyciu ImageEffect i Pixel Standard 3.0. Taka konfiguracja, przy próbie zapisu, generuje po prostu czarny prostokąt.

3.3 Ilość rodzajów szumów

W tym momencie aplikacja obsługuje 2 rodzaje najpopularniejszych szumów. Niestety nie wystarczyło czasu na implementacje kolejnych, takich jak szum Simplex oraz szum komórkowy (Cellural noise). Ich implementacja rozszerzyłaby funkcjonalność programu.

4. Dalszy rozwój aplikacji

Dzięki zastosowanym mechanizmom oraz wzorcom projektowym dalsze rozwijanie aplikacji nie jest dużym problemem. Pierwsze co należałoby zrobić to zastanowić się jak zwiększyć wydajność generatora bitmapy, który sprawia najwięcej kłopotów podczas użytkowania aplikacji.

Kolejne modyfikacje algorytmu fBm mogłyby zwiększyć różnorodność generowanych tekstur, chodzi tu o, przede wszystkim, losowość przesunięć poszczególnych oktaw oraz rotacja punktów próbkowania. Ważnym elementem jest także dodanie kafelkowości szumów, który jest wymagany w pewnych zastosowaniach. Można to zrealizować na kilka sposobów, najprostszy, polegający na mieszaniu dwóch tekstur w zależności od odległości od krawędzi, może prowadzić do niepoprawnych wyników i widocznych przerw na teksturze. Bardziej skomplikowany wymaga implementacji algorytmów dla 4 wymiarów co może doprowadzić do znaczącego spowolnienia obliczeń.

Innych usprawnień można doszukiwać się w interfejsie aplikacji. Są to, m. in. dodanie możliwości zapisu obrazu w innych formatach oraz odpowiednie grupowanie parametrów.