WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Katedra Informatyki i Automatyki

Paweł Dziedzic

Dokumentacja

Generator tekstur szumów - NoiseTex

Rzeszów, 2018

1. Temat projektu

Tematem projektu było stworzenie aplikacji pozwalającej na algorytmiczne generowanie obrazów szumów, takich jak szum Perlina, czy szum wartościowy, a następnie ich zapis na dysk. Podczas tworzenia wykorzystano język programowania C# oraz technologię Windows Presentation Foundasion (w skrócie WPF), która pozwala na tworzenie złożonego interfejsu użytkownika za pomocą języka znaczników XAML.

Jako licencje aplikacji użyto licencji MIT. Jest to najprostsza licencja chroniąca prawa autorskie, ale jednocześnie pozwala na swobodne wykorzystanie kodu nawet w projektach komercyjnych. Dodatkową zaletą tej licencji jest brak gwarancji i odpowiedzialności autora co pozwala na swobodę rozwoju.

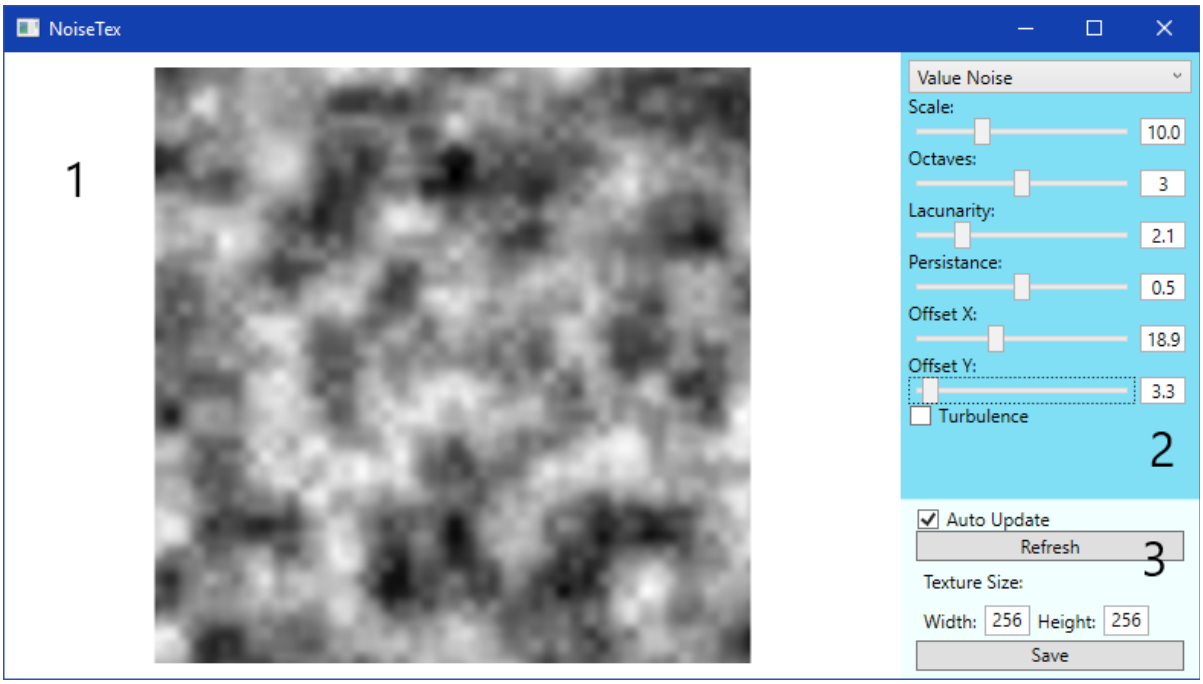
2. Architektura aplikacji

Architekturę aplikacji oparto o wzorzec projektowy MVVM, który pozwala zachować porządek w kodzie oraz w pełni wykorzystać możliwości technologii WPF. Sama aplikacja jest stosunkowo małym projektem, dlatego też zdefiniowano dla niej tylko jeden widok oraz jeden ViewModel. Nie stworzono także modeli w klasycznym tego słowa znaczeniu, rolę tej warstwy zastępują klasy zawierające algorytmy generowania szumów.

Oprócz zwykłych klas zdefiniowanych we wzorcu projektowym zdefiniowano także klasy pomocnicze. Są to klasy statyczne zawierające funkcje, które nie pasują do innych klas, ale są przydatne w procesie tworzenia metod. Przykładem takiej klasy jest klasa MathHelper zawierająca pewne funkcje matematyczne, niezaimplementowane w standardowej bibliotece języka C#.

3. Interfejs aplikacji

Na rysunku 1 przedstawiono finalny wygląd aplikacji.



Rysunek 1. Interfejs aplikacji

Interfejs aplikacji został podzielony na trzy obszary, zostały one podpisane na rysunku 1 kolejnymi cyframi. Obszar z numerem 1 jest to obszar tzw. roboczy, wyświetla się na nim aktualny wygląd tekstury. Obszar 2 służy do ustalania parametrów aktualnie generowanego szumu. Obszar 3 to w głównej mierze opcje programu, zawiera wymiary obrazu, opcje odświeżenia i wyłączenia automatycznego odświeżania obrazu oraz przycisk zapisu na dysk.

Od strony programowej interfejs jest napisany przy użyciu języka znaczników XAML. Wbudowane kontrolki pozwoliły na zbudowanie funkcjonalnego interfejsu bez potrzeby implementowania własnych kontrolek.

4. Generowanie szumów

Aby aplikacja rozpoznała daną klasę jako klasę zawierającą algorytm szumu musi ona implementować interfejs INoise oraz być oznaczona atrybutem Noise, dzięki temu jej instancja zostanie automatycznie umieszczona w ViewModelu dzięki mechanizmowi refleksji. Interfejs

INoise posiada jedną funkcje, GetValue(), która przyjmuje jeden parametr typu Vector2, który jest punktem próbkowania szumu, oraz zwraca wartość typu float, która jest wartością szumu w danym punkcie.

Zadaniem generowania bitmapy zajmuje się pomocnicza klasa BitmapRenderer, implementuje ona algorytm fBm i enkapsuluje inne funkcjonalności związane z tym zadaniem.

4.1 Szum wartościowy

Szum wartościowy jest jednym z najprostszych rodzajów szumów wykorzystywanych w grafice komputerowej. Charakteryzuje się wyraźnymi sygnaturami kwadratów na wygenerowanej teksturze. Wersję dwuwymiarową, użytą w projekcie, tworzy się wyznaczając wartości haszy dla czterech wierzchołki kwadratu w którym znajduje się próbkowany punkt. Wierzchołki kwadratu są to wektory $(x,y),(x+1,y),(x,y+1),(x+1,y+1)$, gdzie x,y są to wartości całkowite współrzędnych punktu. Części dziesiętne współrzędnych próbkowanego punktu wygładza się specjalnym wielomianem, w programie użyto wielomianu $f(t)=6t^5-15t^4+10t^3$, którego pierwsza i druga pochodna przyjmują wartości 0 dla argumentów 0 i 1. Następnie wyznaczone hasze interpoluje się liniowo poprzez wygładzone części dziesiętne współrzędnych, ten wynik jest zwracany przez funkcje. Rysunek 2 przedstawia wygenerowany w ten sposób obraz.

Rysunek 2. Szum wartościowy wygenerowany przez program NoiseTex

W programie wykorzystuje się statyczną tablicę haszy. Zawiera ona wartości całkowite od 0 do 255, rozłożone w sposób losowy. Innym sposobem jest użycie funkcji zwracającej wartości pseudolosowe, takie rozwiązanie stosuje się często w implementacjach algorytmów w językach shaderów.

Listing 1 przedstawia implementacje tego szumu w języku C# (bez dodatkowych pól klasy). Warto zwrócić uwagę, że zwracana wartość jest przeskalowywana do zakresu od -1 do 1. Jest to wymagane aby efekt turbulencji wyświetlany był poprawnie.

Listing 1. Kod funkcji algorytmu szumu wartościowego

```
public float GetValue(Vector2 position)
{
int ix0 = (int)Math.Floor(position.X);
int iy0 = (int)Math.Floor(position.Y);
float tx = position.X - ix0;
float ty = position.Y - iy0;
ix0 &= hashMask;
iy0 &= hashMask;
int ix1 = ix0 + 1;
int iy1 = iy0 + 1;

int h0 = hash[ix0];
int h1 = hash[ix1];
int h00 = hash[h0 + iy0];
int h10 = hash[h1 + iy0];
int h01 = hash[h0 + iy1];
int h11 = hash[h1 + iy1];

tx = MathHelper.Smooth(tx);
ty = MathHelper.Smooth(ty);
return MathHelper.Lerp(MathHelper.Lerp(h00, h10, tx),
MathHelper.Lerp(h01, h11, tx), ty) * (2f / hashMask) - 1;
}
```

4.2 Szum Perlina

Szum Perlina został stworzony przez Kena Perlina w 1985 roku na potrzeby efektów specjalnych do filmu „Tron”. Jest modyfikacją poprzedniego szumu wartościowego, jednak w tym przypadku interpolujemy gradienty zamiast konkretnych wartości. Kierunki gradientów zapisano, podobnie jak wartości haszy, w tablicy. Zastosowanie tej techniki pozwala na uzyskanie wzoru o bardziej naturalnym wyglądzie, co przekłada się też na szersze spektrum zastosowań. Rysunek 3 przedstawia wygenerowany obraz z szumem Perlina.

Rysunek 3. Szum Perlina wygenerowany w programie NoiseTex

Listing 2. Kod funkcji szumu Perlina

```
public float GetValue(Vector2 position)
{
int ix0 = (int)Math.Floor(position.X);
int iy0 = (int)Math.Floor(position.Y);

float tx0 = position.X - ix0;
float ty0 = position.Y - iy0;
float tx1 = tx0 - 1f;
float ty1 = ty0 - 1f;
ix0 &= hashMask;
iy0 &= hashMask;
int ix1 = ix0 + 1;
int iy1 = iy0 + 1;

int h0 = hash[ix0];
int h1 = hash[ix1];
Vector2 g00 = gradients2D[hash[h0 + iy0] & gradientsMask2D];
Vector2 g10 = gradients2D[hash[h1 + iy0] & gradientsMask2D];
Vector2 g01 = gradients2D[hash[h0 + iy1] & gradientsMask2D];
Vector2 g11 = gradients2D[hash[h1 + iy1] & gradientsMask2D];

float v00 = MathHelper.Dot(g00, tx0, ty0);
float v10 = MathHelper.Dot(g10, tx1, ty0);
float v01 = MathHelper.Dot(g01, tx0, ty1);
float v11 = MathHelper.Dot(g11, tx1, ty1);
```

```
float tx = MathHelper.Smooth(tx0);
float ty = MathHelper.Smooth(ty0);
return MathHelper.Lerp(MathHelper.Lerp(v00, v10, tx),
      MathHelper.Lerp(v01, v11, tx), ty) * sqr2;
}
```

Listing 2 przedstawia kod algorytmu generowania szumu Perlina. Warto zauważyć, że do wyznaczenia użytego gradientu wciąż używamy tej samej tablicy haszy co w przypadku szumu wartościowego. W tym przypadku algorytm zwraca wartości w zakresie od -1 do 1, dlatego też nie trzeba jej skalować, jedynie przemnożyć przez pierwiastek z 2 (normalizacja szumu).

4.3 Fraktalny ruch Browna (fBm)

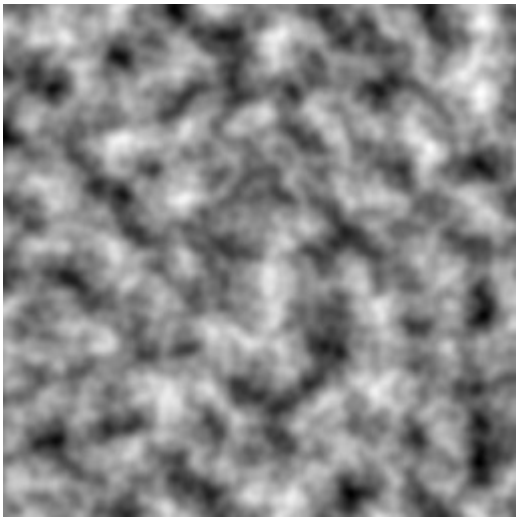
Zwykłe algorytmy szumu nie dostarczają wystarczającej ilości szczegółów dla niektórych zastosowań, dlatego wykorzystuje się fraktalne ruchy Browna (w skrócie fBm) jako rozwiązanie tego problemu. Pomimo faktu, że za fBm stoi skomplikowana teoria matematyczna, implementacja w programie jest bardzo prosta. Algorytmem sterują trzy wartości, ilość oktaw, trwałość i lakunarność. Ilość oktaw wpływa na liczbę szczegółów, trwałość na amplitudę kolejnych oktaw, czyli na to jak bardzo te szczegóły są widoczne, zaś lakunarność wpływa na częstotliwość oktaw, czyli na wielkość szczegółów.

Listing 3. Implementacja fBm

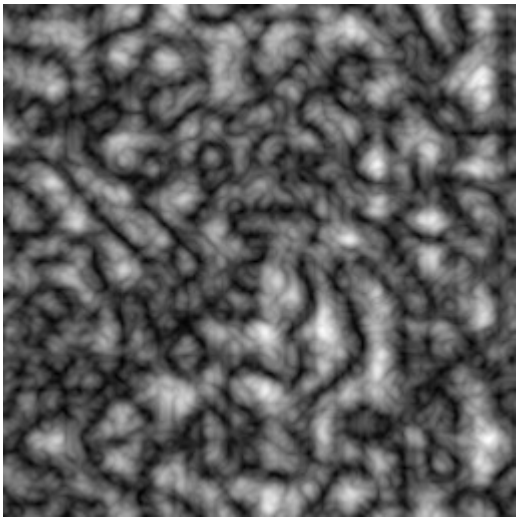
```
for (int i = 0; i < data.octaves; i++)
{
float noiseValue = noiseObject.GetValue((new Vector2(sampleX, sampleY)) *
                  frequency + data.offset);

if (data.turbulence)
{
      noiseValue = Math.Abs(noiseValue);
}

value += noiseValue * amplitude;
amplitude *= data.persistance;
frequency *= data.lacunarity;

range += amplitude;
}
```

Listing 3 przedstawia fragment kodu generującego bitmapę, który jest odpowiedzialny za fBm. Dla każdego piksela bitmapy próbkujemy szum w pętli zależnej od ilości oktaw, w każdym przebiegu zmienia się wartość amplitudy i częstotliwości na podstawie przekazanych parametrów.

W powyższym listingu widać także implementacje efektu turbulencji, jest to po prostu wartość bezwzględna otrzymanej wartości. Rysunki 4 i 5 przedstawiają efekt działania algorytmu fBm na przykładzie szumu Perlina odpowiednio z wyłączonym i włączonym efektem turbulencji. Parametry zostały ustalone na odpowiednio: ilość oktaw = 3, lakunarność = 1.9, oraz trwałość = 0.5.



Rysunek 4. Szum Perlina z fBm



Rysunek 5. Szum Perlina z fBm i turbulencjami

5. Podsumowanie

Podczas projektu stworzono aplikacje umożliwiającą tworzenie i zapisywanie tekstur szumów, które są bardzo przydatne w dziedzinie grafiki komputerowej oraz proceduralnej generacji. Aplikacja korzysta z nowoczesnych technologii programistycznych i wzorców projektowych, które pomogą w dalszym jej rozwoju.

Program obsługuje dwa najczęściej stosowane warianty szumów: szum wartościowy i szum Perlina oraz fraktalny ruch Browna. Wygenerowany obraz można zapisać na dysk w postaci pliku .png.