Grupo 4 - Tiago Koji Castro Shibata (8988730), Adilson Torres Gregório de Souza (8988254)

Sistemas de Computação de Alto Desempenho
Projeto Final

Relatório Final

Implementação Paralela de Ruído Simplex

- **Introdução**

Para o trabalho final, escolhemos implementar o algoritmo de ruído Simplex. O ruído Simplex é usado no processamento de texturas e geração procedural de texturas e mapas.

Como dito no documento de especificação do trabalho, implementamos versão serial e paralela do algoritmo e realizamos benchmark por captura do tempo consumido para geração do ruído. Todos os programas foram implementados em C. A versão paralela usa OpenMP.

O programa é capaz de gerar imagens em escala de cinza de ruído de gradiente e ruído fractal e salvá-las em disco. As imagens podem ser carregadas e ter suas escalas de cores modificadas para criar texturas convincentes (como, por exemplo, padrões em verde e azul representando relevos e oceanos, ou padrões em azul/branco para criar textura de água, entre outros).

- **Descrição do algoritmo**

O algoritmo escolhido foi o **ruído Simplex**. Esse é um algoritmo de geração de ruído com aparência aleatória.

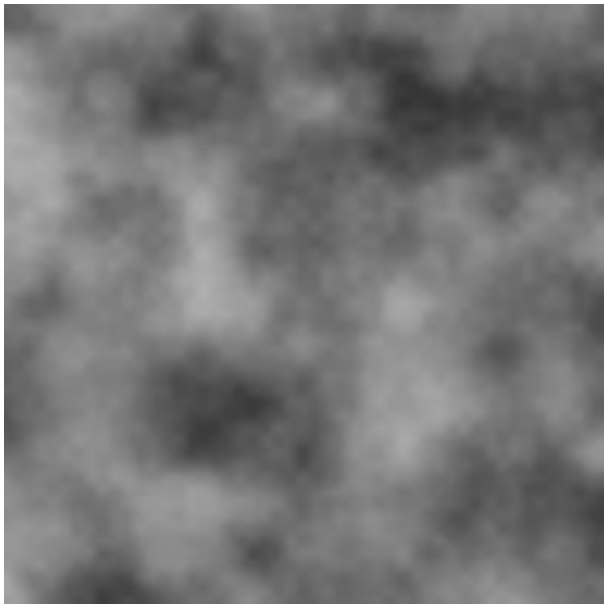
O algoritmo pertence aos geradores de ruído de gradiente: algoritmos que criam um *grid* de gradiente pseudo-aleatório, e, então, para qualquer ponto (x, y), calcula produtos escalares da posição (x, y) com os gradientes de pontos vizinhos do grid e realiza interpolação linear dos resultados. O resultado é um ruído suave e contínuo:

Exemplo de ruído de gradiente bidimensional

Uma aplicação principal do algoritmo é no tratamento de texturas em processamento gráfico. Como visto na amostra acima, seu ruído possui aparência aleatória, mas as “features” distinguíveis possuem tamanho similar. Esse ruído é frequentemente

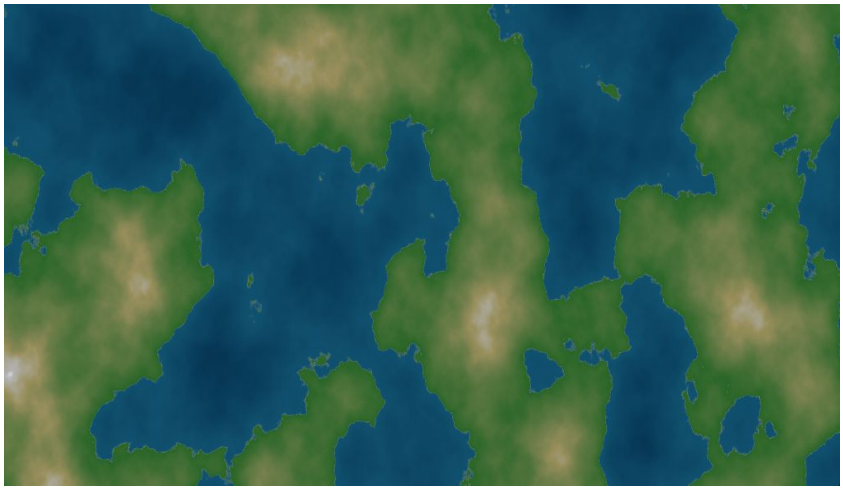
somado a texturas existentes (como paredes, fogo ou fumaça) para torná-los aparentemente mais realistas e menos “repetitivos” de maneira procedural.

Além de servir para pós-processamento de texturas existentes, o ruído pode servir para **geração** procedural de texturas totalmente novas. Somar o próprio ruído nele mesmo, após mudar sua escala de valores absolutos (escala de cada pixel/amplitude da imagem) e realizar *zoom* da imagem (mudança de frequência da imagem), gera efeitos visualmente agradáveis e convincentes, com regiões mais escuras/mais claras acompanhando o ruído de menor frequência, e detalhes adicionados pelos de menor frequência. Por exemplo, criando ruído fractal (onde são geradas oitavas contendo a mesma energia, ou seja, múltiplas imagens de ruído dividindo-se a frequência por 2 e reduzindo a amplitude a cada divisão), adquire-se texturas que lembram fumaça ou nuvens:



Exemplo de ruído fractal

Se colorirmos o ruído de maneira conveniente, temos a formação de imagens que lembram fumaça, nuvens, relevos ou mapas. Por exemplo, gerando ruído com média próxima de 0 e colorindo valores positivos em um gradiente de verde e negativos em um gradiente de azul, podemos obter a seguinte imagem:



Ruído fractal em escala de cores conveniente
gerado utilizando SimplexNoiseCImg

- **Funcionamento e implementação do algoritmo**

A implementação se baseou em fontes livremente disponíveis na Internet, como o paper *Simplex noise demystified* [1] e implementações em código aberto. O algoritmo é bastante parecido com o *Perlin Noise*, sendo que ambos foram desenvolvidos por Ken Perlin, mas possui complexidade menor em espaços multidimensionais e código um pouco mais convoluto. Por não ser um algoritmo trivial, dedicamos essa seção a explicar seu funcionamento.

Primeiramente, realizamos a implementação em uma dimensão, que é mais simples. O algoritmo funciona da seguinte forma:

1. Nas coordenadas inteiras (..., -2, -1, 0, 1, 2, …), o ruído de gradiente vale 0. Coordenadas inteiras possuem um valor de gradiente (uma derivada naquele ponto).
2. Nas coordenadas fracionais, é calculada sua distância para seus vizinhos inteiros. A distância é usada como peso para a contribuição do gradiente de cada vizinho.

O gradiente em cada ponto pode ser gerado por uma função de hash, que sempre retornará o mesmo valor se dadas as mesmas entradas. Dessa maneira, cada ponto pode ser calculado independentemente e o algoritmo é trivialmente paralelizável.

A implementação em **uma dimensão** é:

```
float simplex_noise_1d(float x) {
    int32_t i0 = (int32_t)floor(x);
    int32_t i1 = i0 + 1;
    float x0 = x - i0;
    float x1 = x0 - 1.f;
```

Primeiramente, para uma coordenada X, calculamos seus vizinhos inteiros (*i0* e *i1*) e sua distância para eles.

```
    // Contribution from the first corner
    float t0 = 1.f - x0 * x0;
    t0 *= t0;
    float n0 = t0 * t0 * grad_dot_residual_1d(hash(i0), x0);

    // Contribution from the second corner
    float t1 = 1.f - x1 * x1;
    t1 *= t1;
    float n1 = t1 * t1 * grad_dot_residual_1d(hash(i1), x1);

    // Scale to fit within [-1,1]
    return .395f * (n0 + n1);
}
```

Em seguida, realizamos interpolação entre os pontos. O peso de cada vizinho é inicializado com *1 - distância ^ 2*, ou seja, o ponto é mais afetado quanto mais próximo o vizinho estiver. É realizada integração dos gradientes vizinhos e soma das duas contribuições.

A função *grad_dot_residual_1d* gera o gradiente de um ponto e multiplica por *x*:

```
static float grad_dot_residual_1d(int32_t hash, float x) {
    float grad = 1.f + (hash & 7);
    if (hash & 8) grad = -grad;
    return grad * x;
}
```

Vemos que o gradiente será um valor entre 1 e 8 (soma de 1 com os 3 bits mais baixos do *hash*), e é negado se o quarto bit do hash estiver ativo.

A função de *hash* escolhida é bastante simples, já que precisa ser rápida e não tem requisitos de segurança:

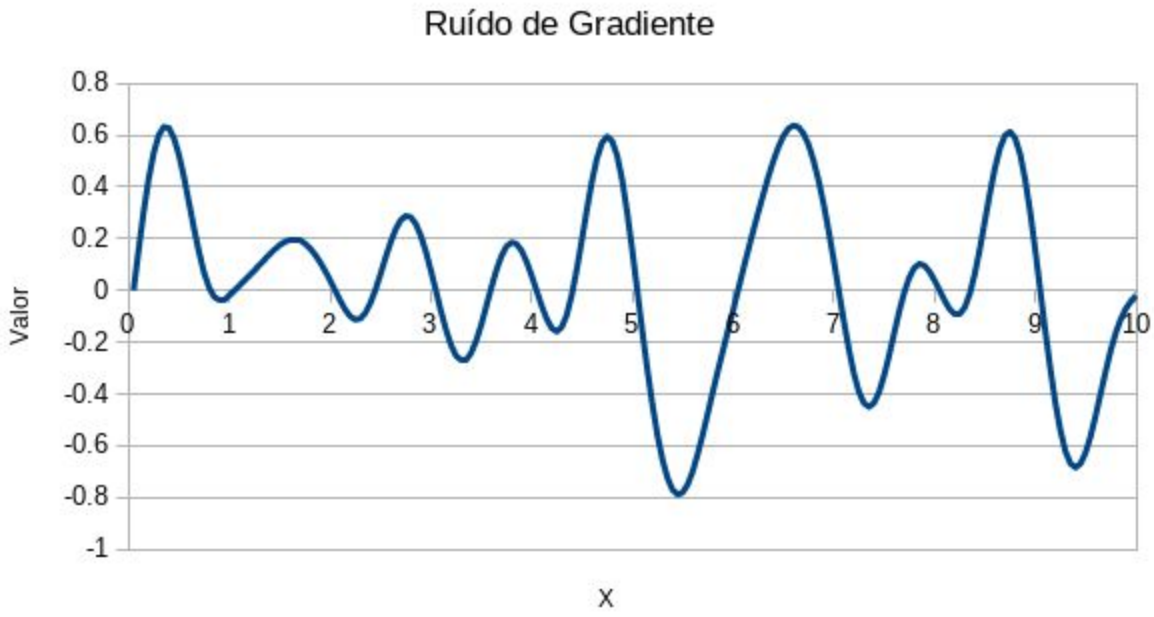
```c
static inline uint8_t hash(int32_t i) {
    uint8_t i8 = (uint8_t)i ^ 0xde;
    return (((i >> 4) | i8 << 4) * 199 + i8) & 0xff;
}
```

Realizamos algumas transformações simples com os bits de x, multiplicamos por um primo, realizamos uma soma e pegamos os 8 bits mais baixos.

Testando com o seguinte programa:

```c
#include <stdio.h>

#include "simplex_noise.h"

int main() {
    for (float x = 0; x < 10; x += 0.05) {
        printf("%f\n", simplex_noise_1d(x));
    }
    return 0;
}
```

Obtivemos a curva:



Observamos que, de fato, o valor da curva é 0 em todos os valores inteiros, entre os inteiros há interpolação de acordo com os gradientes vizinhos, e um “ruído” contínuo e de média 0 foi gerado.

Depois de obter ruído de gradiente com sucesso, tentamos obter ruído fractal. O ruído fractal, como explicado na especificação preliminar, é gerado somando oitavas (múltiplos de 2 da frequência) do ruído de gradiente. Às oitavas de maior frequência é aplicado um fator para possuirem menor amplitude; dessa maneira, as oitavas de menor frequência sobrepõe detalhes sobre as de maior frequência.

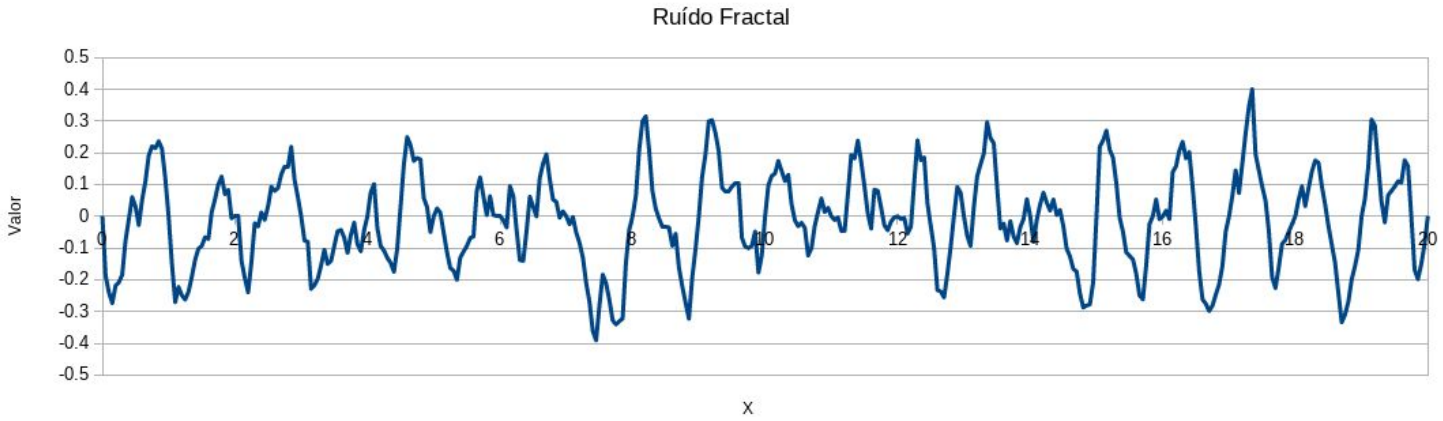
Escrevemos uma função para calcular o ruído fractal:

```
float simplex_noise_fractal_1d(size_t octaves, float x) {
    float output    = 0.f;
    float denom     = 0.f;
    float frequency = simplex_noise_frequency;
    float amplitude = simplex_noise_amplitude;

    for (size_t i = 0; i < octaves; i++) {
        output += (amplitude * simplex_noise_1d(x * frequency));
        denom += amplitude;

        frequency *= simplex_noise_lacunarity;
        amplitude *= simplex_noise_persistence;
    }

    return (output / denom);
}
```

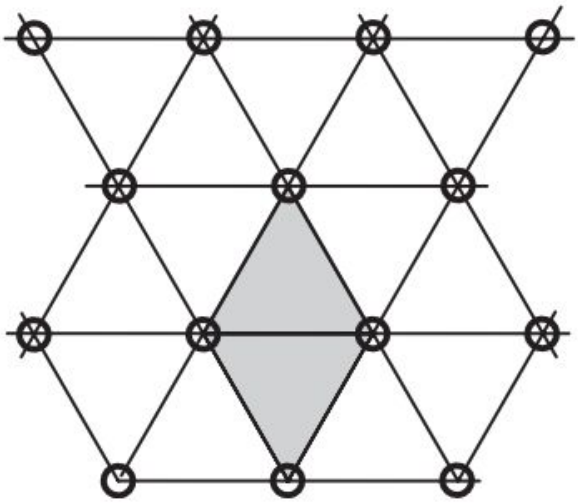
Ao executar o seguinte programa:

```
#include <stdio.h>

#include "simplex_noise.h"

int main() {
    for (float x = 0; x < 20; x += 0.05) {
        printf("%f\n", simplex_noise_fractal_1d(12, x));
    }
    return 0;
}
```

Obtivemos o seguinte gráfico:

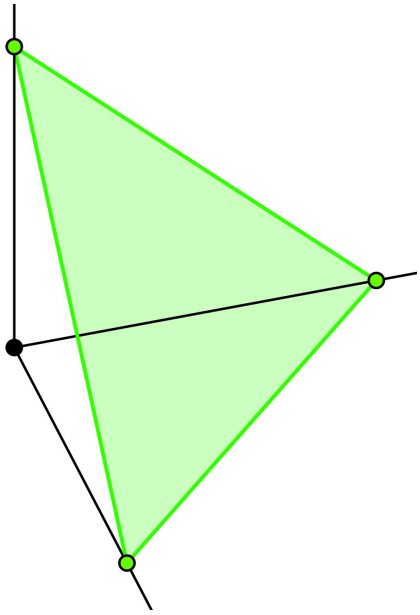


A implementação do algoritmo para mais que uma dimensão é bastante mais complexa. Para 2 ou mais dimensões, é usado um *grid simplex*, ou tesselação simplical. O grid tenta subdividir o espaço com a forma n-dimensional mais compacta possível que possa ser repetida para preencher todo o espaço. No caso 1D, são usados pontos. No caso 2D, o *simplex* é um triângulo:



Fonte: Simplex noise demystified [1]

No caso tridimensional, são usados “tetraedros” alongados:



Fonte: Simplex noise demystified [1]

Vamos trabalhar apenas com o caso 2D. A algoritmo em si é similar ao caso 1D: gerar um gradiente pseudo-aleatório para cada ponto e interpolar com os 3 vizinhos mais próximos. No entanto, há uma transformação de coordenadas para passar da posição 2D para a posição dentro do *simplex* que contém o ponto, e há tratamento diferente se for um triângulo apontando para “cima” ou para “baixo”.

Realizamos a implementação do caso 2D:

```
// Skewing/Unskewing factors for 2D
static const float F2 = 0.366025403f;  // F2 = (sqrt(3) - 1) / 2
static const float G2 = 0.211324865f;  // G2 = (3 - sqrt(3)) / 6

float simplex_noise_2d(float x, float y) {
    float n0, n1, n2;

    // Skew the input space to determine which simplex cell we're in
    const float s = (x + y) * F2;  // Hairy factor for 2D
    const float xs = x + s;
    const float ys = y + s;
    const int32_t i = (int32_t)floor(xs);
    const int32_t j = (int32_t)floor(ys);

    // Unskew the cell origin back to (x,y) space
    const float t = (float)(i + j) * G2;
    const float X0 = i - t;
    const float Y0 = j - t;
    const float x0 = x - X0;  // The x,y distances from the cell origin
    const float y0 = y - Y0;

    int32_t i1, j1;  // Offsets for second (middle) corner of simplex in (i,j) coords
```

```
    if (x0 > y0) {   // lower triangle, XY order: (0,0)->(1,0)->(1,1)
        i1 = 1;
        j1 = 0;
    } else {   // upper triangle, YX order: (0,0)->(0,1)->(1,1)
        i1 = 0;
        j1 = 1;
    }

    const float x1 = x0 - i1 + G2;          // Offsets for middle corner in (x,y) unskewed
coords
    const float y1 = y0 - j1 + G2;
    const float x2 = x0 - 1.f + 2.f * G2;   // Offsets for last corner in (x,y) unskewed coords
    const float y2 = y0 - 1.f + 2.f * G2;

    // Gradient of the three corners
    const int gi0 = hash(i + hash(j));
    const int gi1 = hash(i + i1 + hash(j + j1));
    const int gi2 = hash(i + 1 + hash(j + 1));

    // Contribution from the first corner
    float t0 = .5f - x0*x0 - y0*y0;
    if (t0 < 0.f) {
        n0 = 0.f;
    } else {
        t0 *= t0;
        n0 = t0 * t0 * grad_dot_residual_2d(gi0, x0, y0);
    }

    // Contribution from the second corner
    float t1 = .5f - x1*x1 - y1*y1;
    if (t1 < 0.f) {
        n1 = 0.f;
    } else {
        t1 *= t1;
        n1 = t1 * t1 * grad_dot_residual_2d(gi1, x1, y1);
    }

    // Contribution from the third corner
    float t2 = .5f - x2*x2 - y2*y2;
    if (t2 < 0.f) {
        n2 = 0.f;
    } else {
        t2 *= t2;
        n2 = t2 * t2 * grad_dot_residual_2d(gi2, x2, y2);
    }

    // Scale to [-1,1]
    return 45.23065f * (n0 + n1 + n2);
}
```

O código para ruído fractal em 2D é igual ao caso 1D:

```
float simplex_noise_fractal_2d(size_t octaves, float x, float y) {
    float output = 0.f;
    float denom  = 0.f;
    float frequency = simplex_noise_frequency;
    float amplitude = simplex_noise_amplitude;

    for (size_t i = 0; i < octaves; i++) {
        output += (amplitude * simplex_noise_2d(x * frequency, y * frequency));
        denom += amplitude;
```

```
        frequency *= simplex_noise_lacunarity;
        amplitude *= simplex_noise_persistence;
    }

    return (output / denom);
}
```

Implementamos código integrando nosso projeto com a *libpng* para poder gerar imagens do ruído 2D. Criamos a função *write_png_file*, que recebe um nome de arquivo, um buffer e largura/altura da imagem e a escreve em disco:

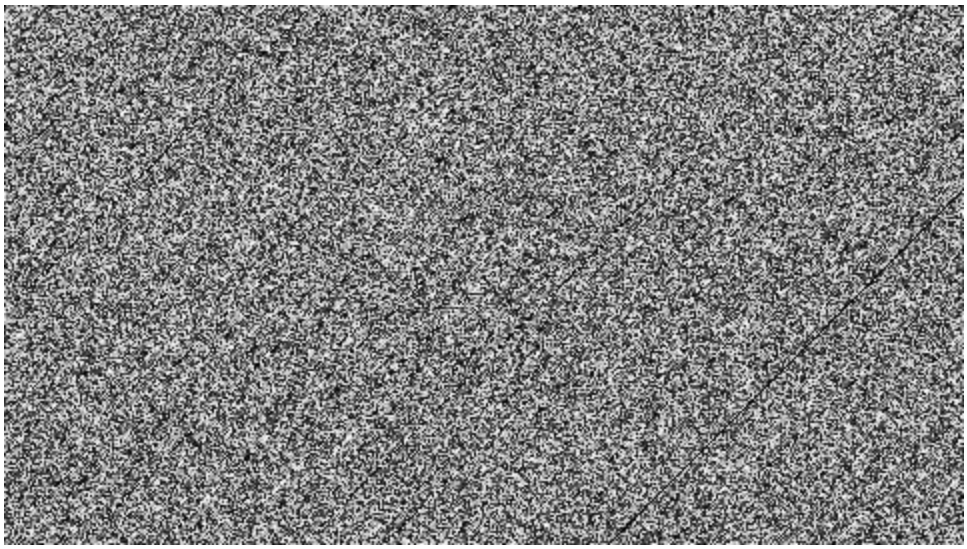
```
#include "png_writer.h"

#include <stdio.h>
#include <stdlib.h>

#include <png.h>

void write_png_file(char* file_name, uint8_t* data, int width, int height) {
    FILE *fp = fopen(file_name, "wb");
    if (!fp) {
        perror("fopen");
        exit(-1);
    }

    png_structp png_ptr = png_create_write_struct(PNG_LIBPNG_VER_STRING, NULL, NULL,
NULL);
    if (!png_ptr) {
        fprintf(stderr, "png_create_write_struct failed");
        exit(-1);
    }

    png_infop info_ptr = png_create_info_struct(png_ptr);
    if (!info_ptr) {
        fprintf(stderr, "png_create_info_struct failed");
        exit(-1);
    }

    if (setjmp(png_jmpbuf(png_ptr))) {
        fprintf(stderr, "libpng failed");
        exit(-1);
    }
    png_init_io(png_ptr, fp);

    const png_byte color_type = PNG_COLOR_TYPE_RGB;
    const png_byte bit_depth = 8;
    png_bytep *row_pointers = malloc(height * sizeof(data));
    for (int y = 0; y < height; y++) {
        row_pointers[y] = data + 3 * y * width;
    }
    png_set_IHDR(png_ptr, info_ptr, width, height, bit_depth, color_type,
PNG_INTERLACE_NONE, PNG_COMPRESSION_TYPE_BASE, PNG_FILTER_TYPE_BASE);
    png_write_info(png_ptr, info_ptr);
    png_write_image(png_ptr, row_pointers);
    png_write_end(png_ptr, NULL);
```

```
    free(row_pointers);
    fclose(fp);
}
```

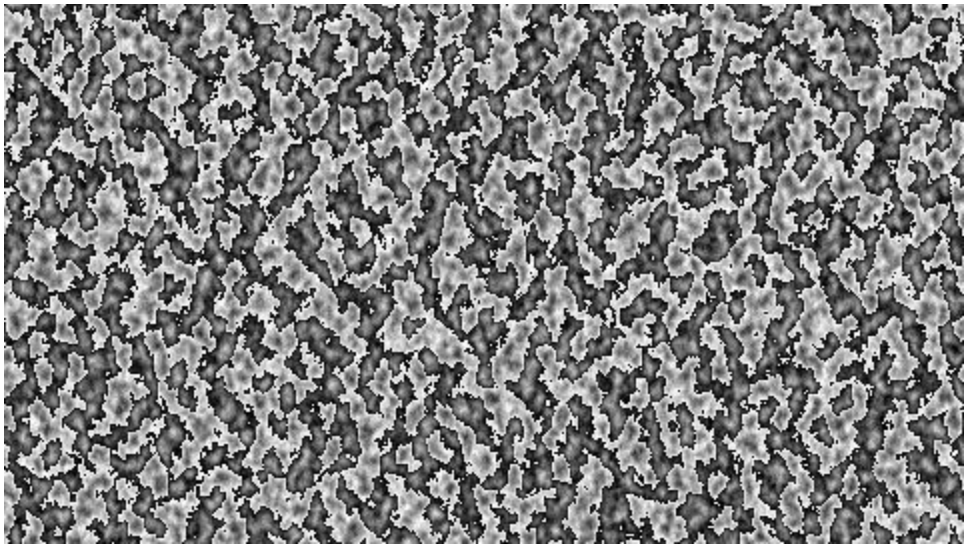
Executamos o seguinte código:

```
const int width = 1920 / 4, height = 1080 / 4;
uint8_t *buffer = malloc(width * height * 3);

for (int x = 0; x < width; x++) {
    for (int y = 0; y < height; y++) {
        uint8_t value = 255 * simplex_noise_2d(x, y);
        buffer[3 * (y * width + x)] = value;
        buffer[3 * (y * width + x) + 1] = value;
        buffer[3 * (y * width + x) + 2] = value;
     }
}
write_png_file("noise.png", buffer, width, height);
```

Gerando:

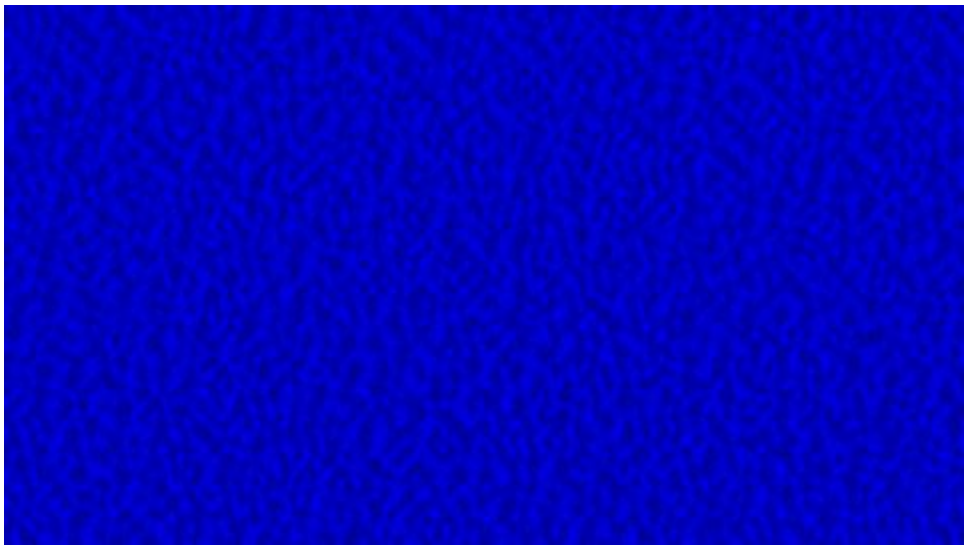


Por último, testamos a geração de ruído fractal em duas dimensões. O mesmo código com *simplex_noise_fractal_2d(16, 0.05 * x, 0.05 * y)* gera:



Desse ponto, podemos mudar a frequência e escala de cores do ruído para gerar várias texturas convincentes. Em testes simples, colocando valores empiricamente até atingir resultados satisfatórios, pudemos facilmente gerar algumas texturas interessantes:

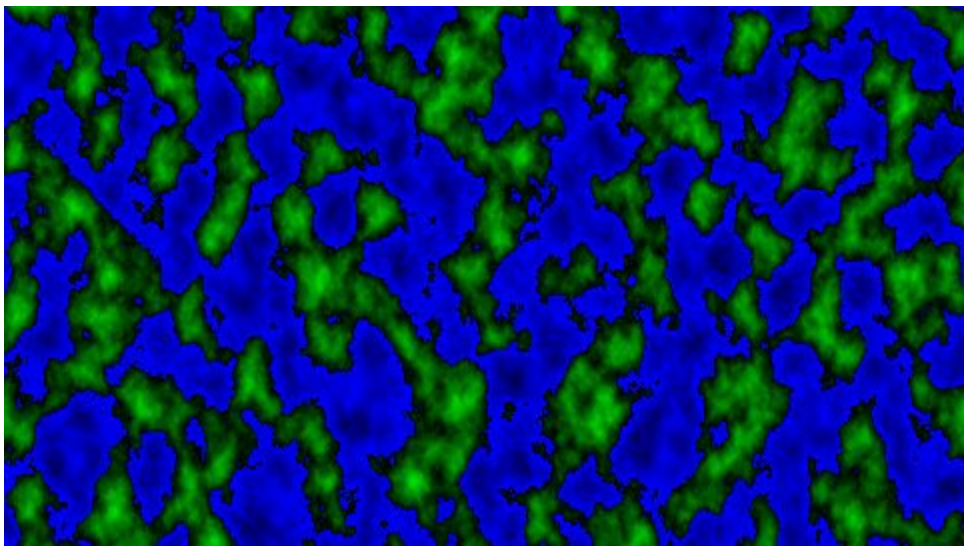
```
uint8_t value = 190 + 65 * simplex_noise_fractal_2d(16, 0.1 * x, 0.1 * y);
buffer[3 * (y * width + x)] = 0;
buffer[3 * (y * width + x) + 1] = 0;
buffer[3 * (y * width + x) + 2] = value;
```



```
uint8_t value = 210 + 25 * simplex_noise_fractal_2d(16, 0.2 * x, 0.2 * y);
buffer[3 * (y * width + x)] = 0;
buffer[3 * (y * width + x) + 1] = value;
```

```
buffer[3 * (y * width + x) + 2] = 0;
```

```
float fractal_value = simplex_noise_fractal_2d(16, 0.02 * x, 0.02 * y);
if (fractal_value > 0) {
    buffer[3 * (y * width + x)] = 0;
    buffer[3 * (y * width + x) + 1] = 255 * fractal_value;
    buffer[3 * (y * width + x) + 2] = 0;
} else {
    buffer[3 * (y * width + x)] = 0;
    buffer[3 * (y * width + x) + 1] = 0;
    buffer[3 * (y * width + x) + 2] = 255 + 200 * fractal_value;
}
```



Nos testes utilizamos apenas escalas simples (multiplicando e somando valores ao ruído para gerar os canais RGB). Com transformações de cores mais complexas, poderíamos atingir resultados ainda melhores para gerar diferentes materiais.

- **Paralelização**

Após testar e validar nosso programa, passamos a realizar *benchmarks*. O algoritmo é trivialmente paralelizável, já que a geração de cada pixel independe da computação dos vizinhos.

Primeiramente, paralelizamos o *loop* externo pelos *pixels* utilizando OpenMP utilizando *#pragma omp parallel for* no laço externo:

```
#pragma omp parallel for
for (int x = 0; x < width; x++) {
    for (int y = 0; y < height; y++) {
        simplex_noise_fractal_2d(16, 0.02 * x, 0.02 * y);
    }
}
```

Para validar a implementação paralela, rodamos os mesmos testes de geração de texturas. Obtivemos os mesmos resultados da implementação serial.

Então, para os *benchmarks*, desativamos funções não relacionadas ao algoritmo, como escrita do PNG.

- **Resultados**

Benchmarks foram realizados em processador *Intel(R) Core(TM) i5-8500 CPU @ 3.00GHz* de 6 núcleos, com uma *thread* por núcleo. Para reduzir variações no benchmark, outros processos pesados foram mortos e os processos do benchmark foram executados com *niceness* -9 (alta prioridade).

Benchmarks foram realizados na geração de imagens em resolução 1920x1080 com 16 oitavas. Após 50 execuções do programa serial, o tempo médio foi 1141.8ms. O programa paralelo (com OpenMP realizando paralelismo entre linhas) tomou 206.4ms, ganho de 1141.8 / 206.4 = 5.53 vezes. Vemos que o ganho de desempenho é bastante bom, já que esse é um problema fácil de paralelizar e com pouca dependência de recursos fora CPU (por exemplo, há muito poucos acessos à memória nesse algoritmo,

já que o único dado que varia dependendo do local é o resultado da computação da função de *hash*, que não requer acessos à memória).

Realizamos mais alguns testes para buscar melhoras no desempenho. Tentamos utilizar a keyword *collapse* no laço paralelo, fazendo com que o OpenMP paralelizasse não apenas o laço externo (em *x*) mas também o aninhado (em *y*). Essa mudança faz com que a computação de cada *pixel* seja distribuída pelo OpenMP ao invés de cada linha, reduzindo a granularidade do trabalho de cada thread.

Acreditamos que não haverá melhora, já que haverá maior overhead em escalonamento de cada *pixel* e o laço externo já propiciava bastante paralelismo (1080 linhas). O tempo médio adquirido foi 212.3ms, um pouco pior que sem o uso de *collapse*.

Testamos também mudar o *schedule*, mudando o algoritmo de divisão de trabalho entre threads. Primeiramente, forçamos escalonamento estático, dividindo o trabalho igualmente entre threads. O efeito visto foi que o tempo de execução não se alterou em algumas execuções, mas em outras cresceu muito:

```
time: 212.32215ms
time: 209.93843ms
time: 310.86036ms
time: 208.88124ms
time: 324.89933ms
time: 215.87119ms
time: 209.78599ms
```

Atribuímos esse efeito à pouca flexibilidade do escalonador: se outro processo da máquina estiver em execução e nosso algoritmo não conseguir os 6 processadores ao mesmo tempo, uma thread irá bloquear até alguma outra thread estar livre. No escalonamento estático, a divisão é feita inicialmente entre as threads em *chunks* iguais.

Testamos também o escalonador *dynamic*, que dá tarefas para cada thread dinamicamente, conforme elas acabam a tarefa anterior. Esse escalonador possui maior overhead do que os outros. O tempo médio adquirido foi de 224.8ms.

Com o escalonador *guided* (que começa com *chunks* grandes e passa a dividir *chunks* menores conforme o número de iterações restantes diminui) atingimos resultados parecidos com os iniciais.

Programa	Consumo de tempo médio (ms)	Melhora
Serial	1141.8	1x
Paralelo (escalonamento padrão)	206.4	5.53x
Paralelo (escalonamento padrão, laço colapsado)	212.3	5.38x
Paralelo (escalonamento estático)	233.0	4.90x
Paralelo (escalonamento dinâmico)	224.8	5.08x

- **Conclusão**

Pudemos implementar um algoritmo com bastante aplicação prática e otimizá-lo utilizando processamento paralelo. Observamos grandes ganhos de desempenho, já que o algoritmo escolhido possui nenhuma dependência de dados entre suas iterações e é bastante pesado em processamento, não sofrendo com outros gargalos (como memória). Atingimos ganhos de mais que 5.5x em um sistema de 6 núcleos, o que é bastante satisfatório.

- **Código fonte**

Disponível em :
https://github.com/tiagoshibata/pcs3868-high-performance-computing/tree/master/capstone

- **Referêcias**

[1]: Simplex noise demystified - Stefan Gustavson, Linköping University, Sweden (stegu@itn.liu.se) - Acesso em 03/12/2018
http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf