

Fractal Explorer – CUDA Ray Marcher

Jakub Janeczko

January 22, 2026

Cel projektu

- ▶ Interaktywna wizualizacja fraktali 3D w czasie rzeczywistym.
- ▶ Wykorzystanie ray marchingu na GPU z własnym kernelem CUDA.
- ▶ Płynne morfowanie między typami fraktali i eksploracja parametrów.

Kluczowe funkcje

- ▶ 60+ FPS przy 1280x720 na współczesnych kartach 3060 Ti.
- ▶ Obsługa fraktali: Mandelbox, Menger Sponge, Sierpinski, Tree Planet.
- ▶ Tryb auto-cycle: automatyczne przełączanie i morfowanie fraktali.
- ▶ Interaktywny ruch kamery (WASD + mysz) i zmiana rozdzielczosci.
- ▶ Dynamiczne oświetlenie: cieniowanie Phong'a, cienie, glow.

Architektura kodu

- ▶ **main.cpp**: petla aplikacji, zarzadzanie stanem fraktali i morfowaniem.
- ▶ **cuda/raymarcher.cu**: kernel ray marchingu z sphere tracingiem.
- ▶ **cuda/fractals.cu**: funkcje SDF i operacje "fold" dla kazdego fraktala.
- ▶ **window.cpp** + OpenGL/GLFW: bufor wyjsciowy i obsluga wejscia.
- ▶ **camera.cpp**: sterowanie kamera (WASD, mysz, delta time).

Potok renderowania

1. Pobranie parametrów kamery i aktywnego fraktala.
2. Uruchomienie kernela CUDA dla bloków 16×16 (grid zależny od rozdzielczości).
3. Sphere tracing: adaptacyjne kroki o długości SDF do trafienia lub przekroczenia zasięgu.
4. Obliczenie normalnej z gradientu SDF, cieniowanie Phong'a, efekt glow.
5. Kopiowanie bufora z GPU i prezentacja w oknie OpenGL.

Operacje fold (1/2)

- ▶ Box fold: $p = \text{clamp}(p, -r, r) \times 2 - p$; odbija współrzędne poza pudłem i zwiększa symetrie.
- ▶ Sphere fold: dla $|p| < r_{\min}$ skaluje do r_{\max}^2 / r_{\min}^2 ; dla $|p| < r_{\max}$ skaluje do $r_{\max}^2 / |p|^2$; stabilizuje detal blisko środka.
- ▶ Scale/translate: $p = p \times s + o$; prosty sposób na zoom i przesunięcie sceny.

Operacje fold (2/2)

- ▶ Menger fold: $|p|$ z abs, następnie permutacja i odejmowanie wzorca kostki (skala 3, przesuniecie -2) dla fraktala glebokosci 3D.
- ▶ Sierpinski fold: odbicia wzgledem plaszczyn wyznaczonych przez wierzcholki tetraedru, utrzymuje symetrie piramidy przy skalowaniu 2.
- ▶ Rotacja: obrot wektora p wokol osi (np. Y) o kat θ ; kluczowy w Tree Planet dla organicznych ksztaltow.

Kod: boxFold i sphereFold

```
__device__ inline void boxFold(float4& z, const float3& r)
{
    z.x = clamp(z.x, -r.x, r.x) * 2.0f - z.x;
    z.y = clamp(z.y, -r.y, r.y) * 2.0f - z.y;
    z.z = clamp(z.z, -r.z, r.z) * 2.0f - z.z;
}

__device__ inline void sphereFold(float4& z, float minR, float maxR)
{
    float r2 = z.x*z.x + z.y*z.y + z.z*z.z;
    float minR2 = minR*minR, maxR2 = maxR*maxR;
    if (r2 < minR2) { float f = maxR2 / minR2; z *= f; }
    else if (r2 < maxR2) { float f = maxR2 / r2; z *= f; }
}
```

Kod: absFold i planeFold

```
__device__ inline void absFold(float4& z) {  
    z.x = fabsf(z.x); z.y = fabsf(z.y); z.z = fabsf(z.z);  
}  
  
__device__ inline void planeFold(float4& z, const float3& n, float d) {  
    float dotVal = z.x*n.x + z.y*n.y + z.z*n.z - d;  
    float k = fminf(0.0f, dotVal);  
    z.x -= 2.0f * k * n.x;  
    z.y -= 2.0f * k * n.y;  
    z.z -= 2.0f * k * n.z;  
}
```

Kod: mengerFold i sierpinskiFold

```
__device__ inline void mengerFold(float4& z) {  
    float a = fminf(z.x - z.y, 0.0f);  
    z.x -= a; z.y += a;  
    a = fminf(z.x - z.z, 0.0f);  
    z.x -= a; z.z += a;  
    a = fminf(z.y - z.z, 0.0f);  
    z.y -= a; z.z += a;  
}
```

```
__device__ inline void sierpinskiFold(float4& z) {  
    if (z.x + z.y < 0.0f) { float t = z.x; z.x = -z.y; z.y =  
    if (z.x + z.z < 0.0f) { float t = z.x; z.x = -z.z; z.z =  
    if (z.y + z.z < 0.0f) { float t = z.y; z.y = -z.z; z.z =  
}
```

Kod: rotacje osiowe

```
__device__ inline void rotateY(float4& z, float angle) {  
    float s = sinf(angle), c = cosf(angle);  
    float tx = c * z.x - s * z.z;  
    float tz = c * z.z + s * z.x;  
    z.x = tx; z.z = tz;  
}
```

```
__device__ inline void rotateX(float4& z, float angle) {  
    float s = sinf(angle), c = cosf(angle);  
    float ty = c * z.y + s * z.z;  
    float tz = c * z.z - s * z.y;  
    z.y = ty; z.z = tz;  
}
```

Ray marching (sphere tracing)

- ▶ Iteracyjnie przesuwamy promień o wartość dystansu do najbliższej powierzchni.
- ▶ Zatrzymanie gdy $d < \varepsilon$ (trafienie) lub $t > t_{\max}$ (brak trafienia).

$$t_{k+1} = t_k + d(\mathbf{o} + t_k \mathbf{d})$$

Typy fraktali

- ▶ **Mandelbox:** box fold + sphere fold, 16 iteracji.
- ▶ **Menger Sponge:** fold abs + fold Menger, skala 3.0, 8 iteracji.
- ▶ **Sierpinski:** symetria tetraedru, skala 2.0, 9 iteracji.
- ▶ **Tree Planet:** hybryda rotacji i foldow, 30 iteracji.

Sterowanie

- ▶ Ruch: WASD, Space (gora), Shift (dol), mysz (obrot).
- ▶ Fraktale: klawisze 1–4; Tab włącza auto-cycle.
- ▶ Morfowanie: + / = przyspiesza, - spowalnia.
- ▶ Wyjście: Esc.

Wydajność i parametry

- ▶ Docelowo 60+ FPS przy 1280x720, do 1000 kroków promienia.
- ▶ LOD: mnożnik poziomu detalu (lodMultiplier) dla bliskich obiektów.
- ▶ Głowy i cienie zależne od gradientu SDF i kierunku światła.
- ▶ Przyrosty parametrów w auto-cycle bazują na funkcjach sinus/cosinus czasu.

Wyniki (GPU)

- ▶ ~ ponad 30 FPS na RTX 3060 Ti przy rozdzielczosci 2540x1440.

Budowanie i uruchomienie (Linux)

- ▶ Wymagane: CUDA 11+, CMake 3.18+, GLFW, GLEW, OpenGL, GPU CC ≥ 6.0 .
- ▶ Instalacja zaleznosci (Debian/Ubuntu):
`sudo apt-get install cmake nvidia-cuda-toolkit libglfw3`
- ▶ Budowanie w repozytorium:
`chmod +x build.sh`
`./build.sh`
- ▶ Uruchomienie: `./build/fractal-explorer 1920 1080`
(opcjonalna rozdzielczosc).

Mozliwe rozszerzenia

- ▶ Interpolacja parametrow z GUI i zapis presetow.
- ▶ Kolejne typy fraktali i edytor nodow fold/transform.
- ▶ Zrzuty ekranu/wideo, nagrywanie sciezek kamery.
- ▶ Kolizje i tryb eksploracji "gry" na bazie SDF.

Podsumowanie

- ▶ Fractal Explorer dostarcza szybki, interaktywny rendering fraktali 3D na GPU.
- ▶ Elastyczne morfowanie parametrow pozwala odkrywac bogate struktury SDF.
- ▶ Kod rozdziela logike aplikacji, obsluge wejscia i obliczenia CUDA dla czytelności.