CUDA Rybki

Zasady symulacji

Algorytm stada (boids) opiera się na trzech zasadach:

Dla każdej rybki i rozważamy wszystkie rybki j w odległości ograniczonej przez promień zdefiniowany dla danej zasady.

1. Rozdzielność (separation) - zapobiegamy lokalnym zbiorowiskoam/kolizjom.

$$v_i \leftarrow v_i + k_s \Delta t \sum_{j \neq i} \frac{x_i - x_j}{|x_i - x_j|^2}$$

2. Spójność (cohesion) - dążenie do lokalnego środka masy

$$v_i \leftarrow v_i + k_c \Delta t \left(\overline{x}_{j \neq i} - x_i \right)$$

3. Wyrównanie (alignment) - upodobnienie prędkości do lokalnej grupy

$$v_i \leftarrow v_i + k_a \Delta t \left(\overline{v}_{j \neq i} - v_i \right)$$

Współczynniki k_s, k_c, k_a pozwalają kontrolować siłę oddziaływania. Ciekawe efekty dają wartości ujemne – wtedy wzór realizuje odwrotność danej zasady, np. ustawienie ujemnej wartości współczynnika spójności daje efekt lokalnego rozproszenia (drapieżnik wpływający w ławicę).

Struktury

Używamy SoA.

```
 typedef struct {
    float *pos_x;
    float *pos_y;
    float *vel_x;
    float *vel_y;
    uint8_t *type;
    int count;
} Boids;
```

Oprócz pozycji i prędkości trzymamy jeszcze informację o rodzaju rybki, który decysuje o jej zachowaniu:

```
typedef struct {
    float cohesion_r;
    float cohesion_strength;
    float separation_r;
    float separation_strength;
    float alignment_r;
    float alignment_strength;

    float min_speed;
    float max_speed;
} BoidTypeParams;

typedef struct {
    BoidTypeParams type[MAX_TYPES];
    u8 type_count;
} BoidsParams;
```

Dodatkowo, reguły 2 i 3 (spójność, wyrównanie) działają jedynie między rybki tego samego rodzaju, co daje ławice z rybek głównie tego samego rodzaju.

Parametry możemy trzymać w stałej pamięci:

```
__constant__ BoidsParams d_params;
```

Optymalizacje

Naiwna implementacja ma złożoność $O(n^2)$.

Zakładając, że promień widzenia rybki jest znacznie mniejszy od rozmiarów planszy, możemy podzielić przestrzeń na kostki o krawędzi równej największemu promieniowi. Wtedy dla danej rybki uwzględniamy jedynie te, które są w jej kostce oraz w kostkach z nią graniczących.

W takiej implementacji, dobrym pomysłem może okazać się wstępne posortowanie rybek po indeksach przypisanym im kostek, aby lepiej wykorzystać pamięć podręczną.

Dodatkowe reguły

Dążenie do celu

$$v_i \leftarrow v_i + k\Delta t(x_c - x_i)$$

Jeśli ustawimy $k < 0$ oraz $k_c < 0$ (spójność), to możemy zasymulować rybki uciekające od drapieżnika.

Granice planszy

Można zadecydować o odbijaniu rybek od krawędzi, ale lepszą opcją wydaje się teleportacja w odpowiednie miejsce (przeciwna krawędź ekranu):

```
float wrap(float x) {
    if (x < -1.0f)
        return x + 2.0f;
    if (x > 1.0f)
        return x - 2.0f;
    return x;
}
```

Ograniczanie szybkości

Koniecznym okazuje się ograniczanie prędkości, zarówno od góry jak i od dołu.

$$v_i \leftarrow \begin{cases} v_{\min}\frac{v_i}{|v_i|} \text{ jeśli } v_i < v_{\min} \\ v_{\max}\frac{v_i}{|v_i|} \text{ jeśli } v_i > v_{\max} \\ v_i \text{ wpp} \end{cases}$$