TIPE: Simulation de fluide en temps réel via l'hydrodynamique des particules lissées.

Noam DERRUAU

Épreuve de TIPE

Session 2024

Clarification sur le sujet

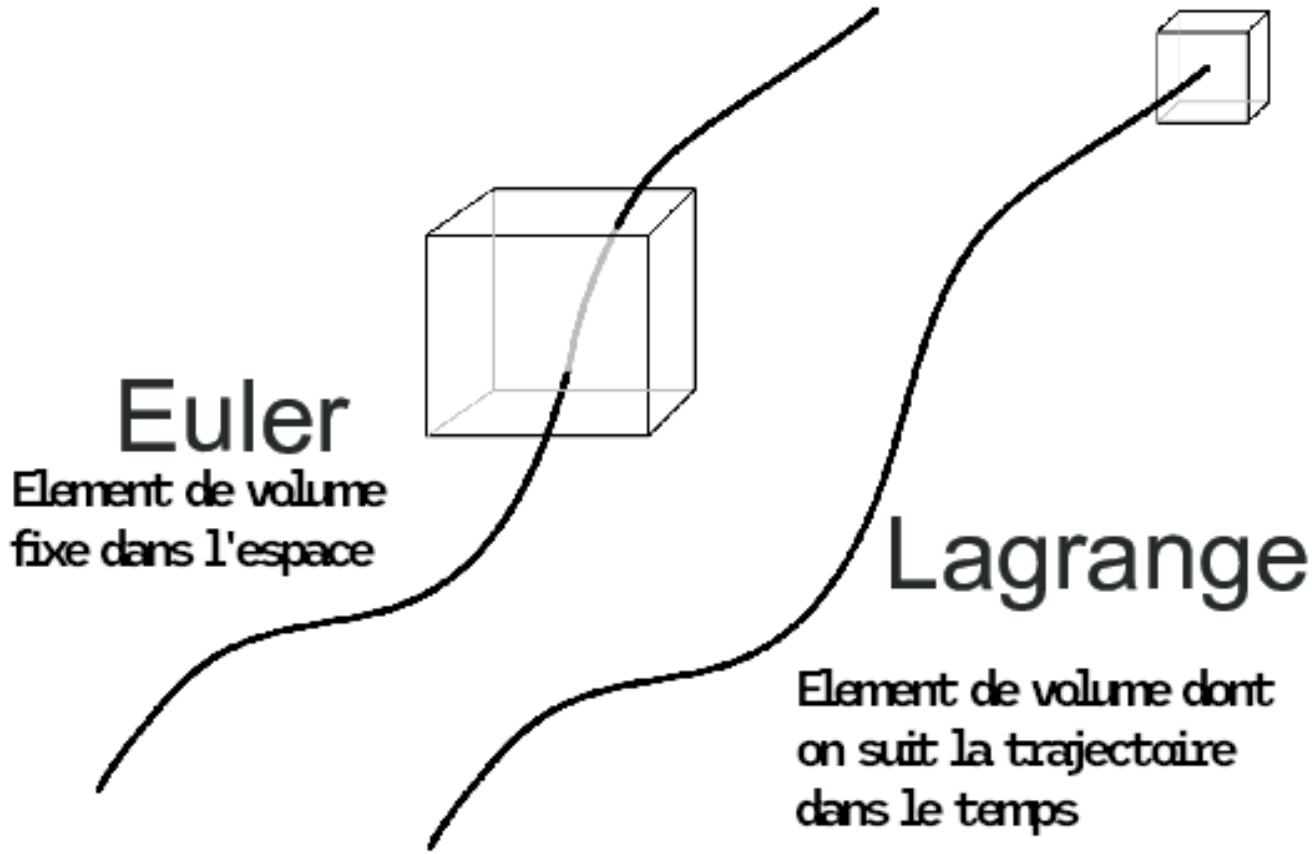
Figure – Assassin's Creed : Black Flag - Pas de simulation de fluide

Figure – Terrarria - Simulation de fluide

Plan de l'exposé

Notions de mécanique des fluides

Deux points de vue

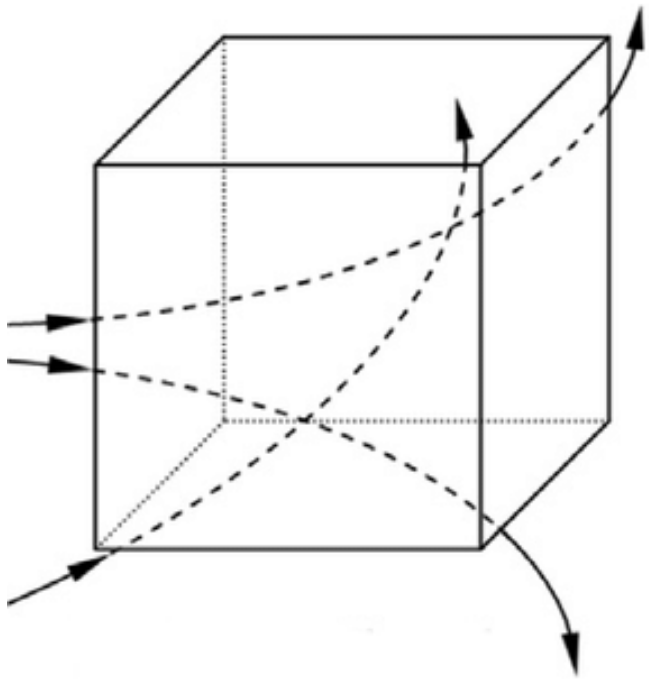


Notions de mécanique des fluides

La description Eulerienne

Caractéristiques :

- Élément de volume fixe dans l'espace
- Les variables d'intéret sont de la forme : $A(M, t)$

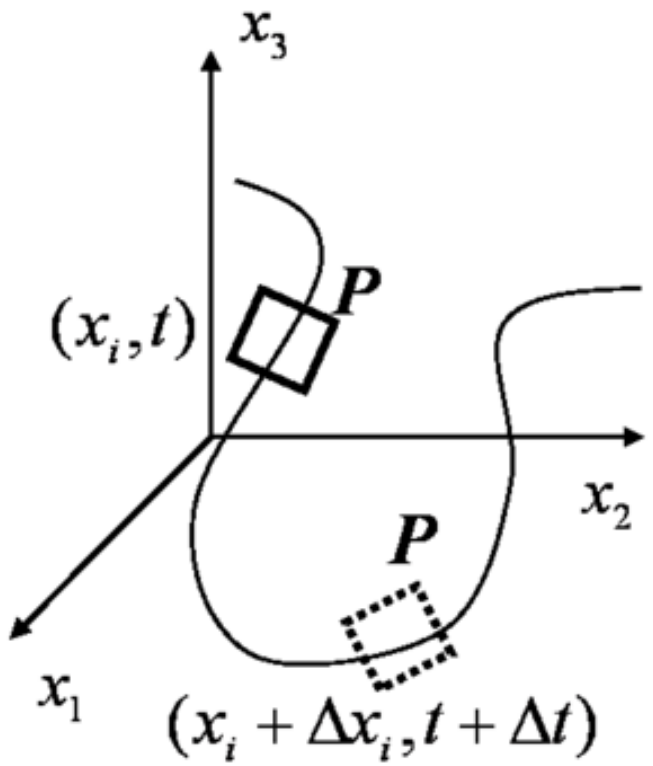


Notions de mécanique des fluides

La description Lagrangienne

Caractéristiques :

- Élément de volume bouge dans l'espace
- Les variables d'intéret sont de la forme : $A(M_0, t)$



Notions de mécanique des fluides

Choix pour mon projet

Eulerienne :

- Préférée pour la résolution des équations de Navier-Stokes
- Préférée pour modélisation de grands systèmes fluides

✗ Utilisation non pertinente

Lagrangienne :

- Utile pour des problèmes où les trajectoires des particules sont cruciales
- Par exemple la simulation des mouvements de particules de fluide

✓ Utilisation pertinente

Notions de mécanique des fluides

Les équations du mouvement d'une particule de fluide

Pour une particule de fluide de volume $d\tau$ de masse m et de masse volumique ρ, la 2nde loi de newton s'écrit :

$$m\frac{d\vec{v}}{dt} = \vec{F}_{contact} + \vec{F}_{distance}$$

Notions de mécanique des fluides

Les forces à distance

Forces à distances :

- Le poids : $\vec{P} = m\vec{g} = \rho d\tau \vec{g}$
- La force de Lorentz : $\vec{F_L} = q(\vec{E} + \vec{v} \wedge \vec{B}) = \vec{0}$

Expression :

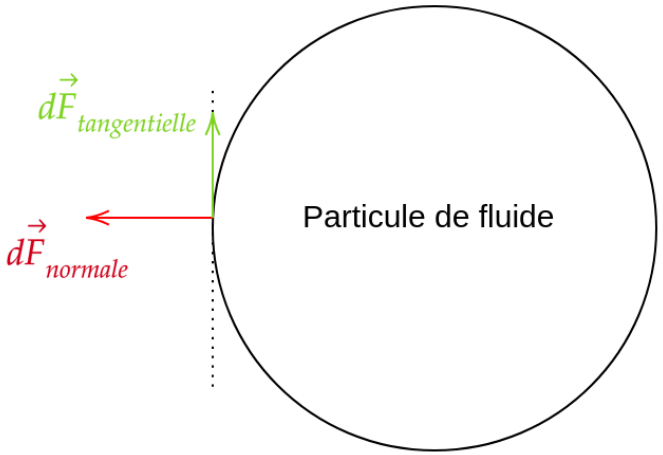
$$\boxed{\vec{F}_{distance} = \rho d\tau \vec{g}}$$

Notions de mécanique des fluides

Les forces de contact

Décomposition de la force de contact

$$d\vec{F}_{contact} = \underbrace{d\vec{F}_{tangentielle}}_{\text{Viscosite}} + \underbrace{d\vec{F}_{normale}}_{\text{Pression}}$$



Après calculs on obtient les expressions suivantes :

$$\boxed{d\vec{F}_{pression} = -\vec{\text{grad}}(P)d\tau} \qquad \boxed{d\vec{F}_{viscosite} = \eta\Delta\vec{v}d\tau}$$

Notions de mécanique des fluides

Les équations de Navier-Stokes

Équation de conservation de la quantité de mouvement :

$$\boxed{\rho\left[\frac{\partial \vec{v}}{\partial t} + \vec{v} \cdot \vec{\text{grad}}(v)\right] = \rho\vec{g} - g\vec{r}adP + \eta\Delta\vec{v} + \vec{F}}$$

Équation de conservation de la masse :

$$\boxed{\frac{\partial \rho}{\partial t} + \rho \text{div } \vec{v} = 0}$$

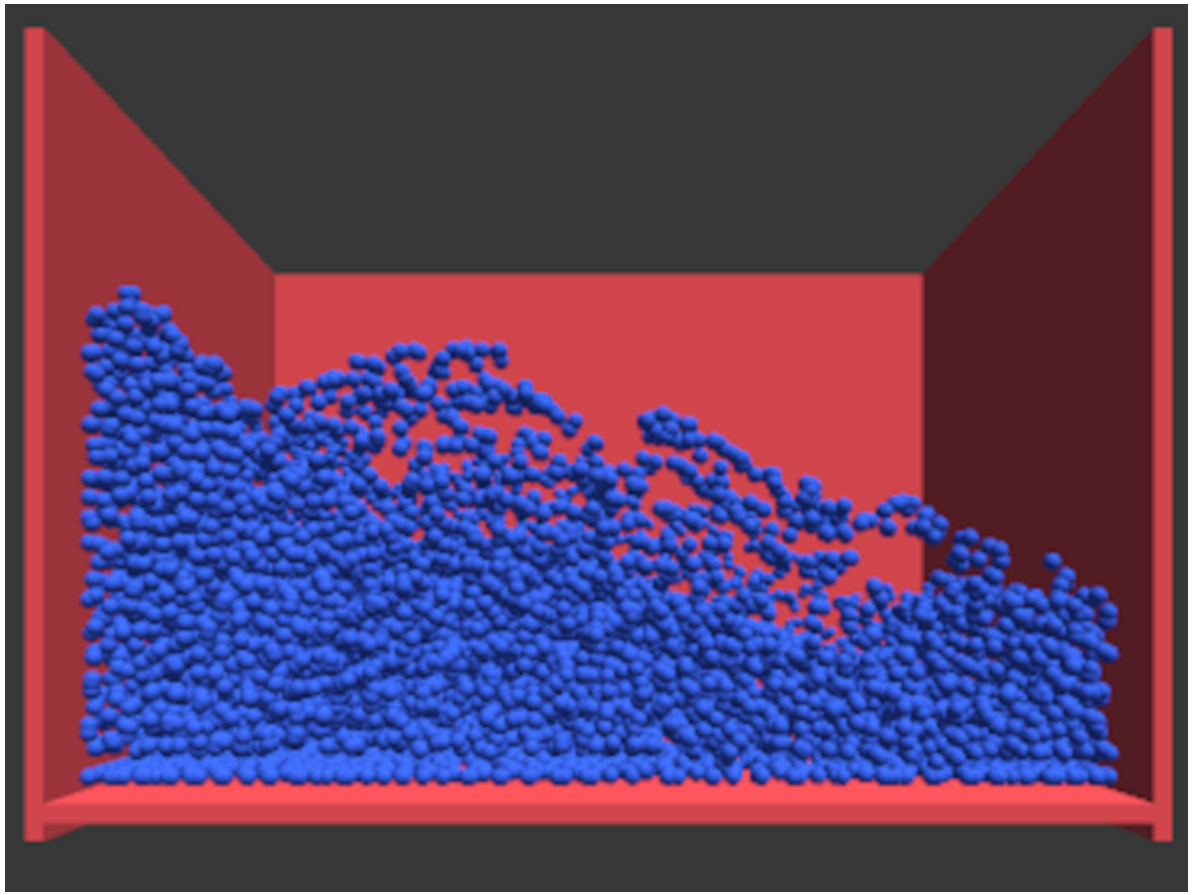
Obtention des formules de la simulation

Méthode SPH

Obtention des formules de la simulation

Obtention des formules de la simulation

Méthode SPH



Obtention des formules de la simulation

Avantages de cette technique

- Respect de l'équation de conservation de la masse
- le terme d'advection est nul

$$(\vec{v} \cdot \vec{\text{grad}})(\vec{v}) = 0$$

Obtention des formules de la simulation

Les équations que nous font résoudre cette méthode

On doit donc juste résoudre :

$$\rho\frac{\partial\vec{v}}{\partial t} = \rho\vec{g} - g\vec{rad}P + \eta\Delta\vec{v} + \vec{F}$$

Obtention des formules de la simulation

Obtiention des approximations : prérequis - pseudo-fonction de Dirac

La pseudo-fonction de Dirac :

$$\delta(x) = \begin{cases} +\infty & \text{si } x = 0 \\ 0 & \text{sinon} \end{cases}$$

$$\int_{-\infty}^{+\infty} \delta(x)dx = 1$$

Obtention des formules de la simulation

Obtiention des approximations : prérequis - l'identité du Dirac

L'identité du Dirac :
$\forall g$ continue et intégrable

$$\int_{\mathbb{R}} g(x)\delta(x-t)dx = g(t)$$

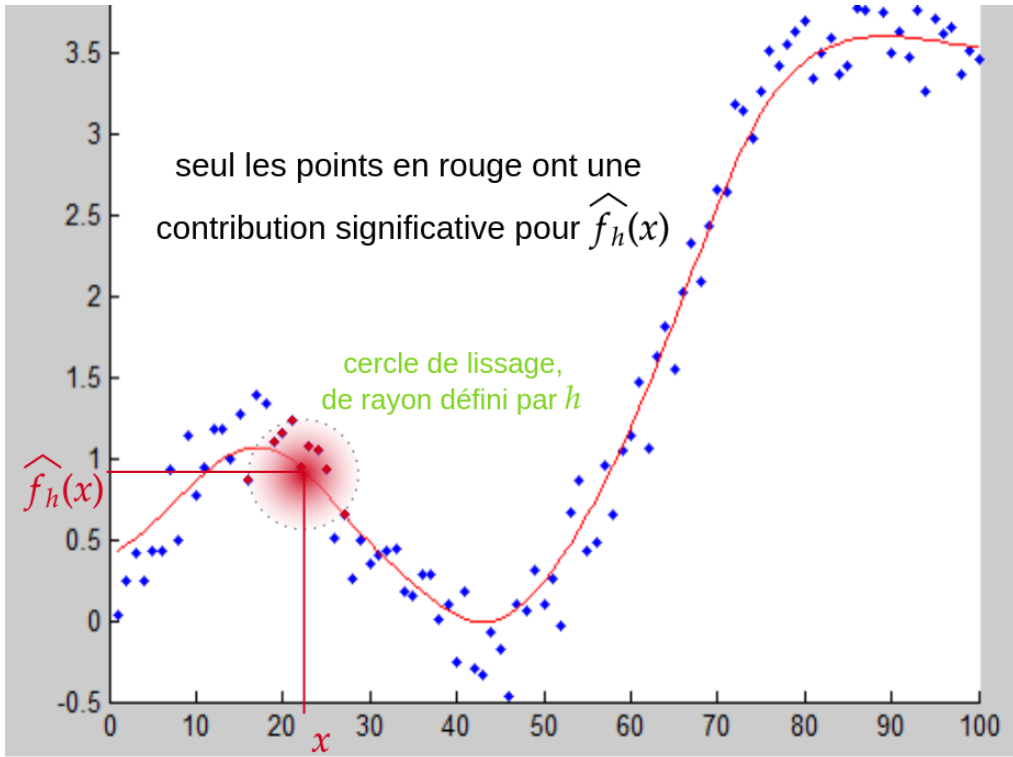
Obtention des formules de la simulation

Obtiention des approximations : prérequis - estimation par noyau

Estimation par noyau :

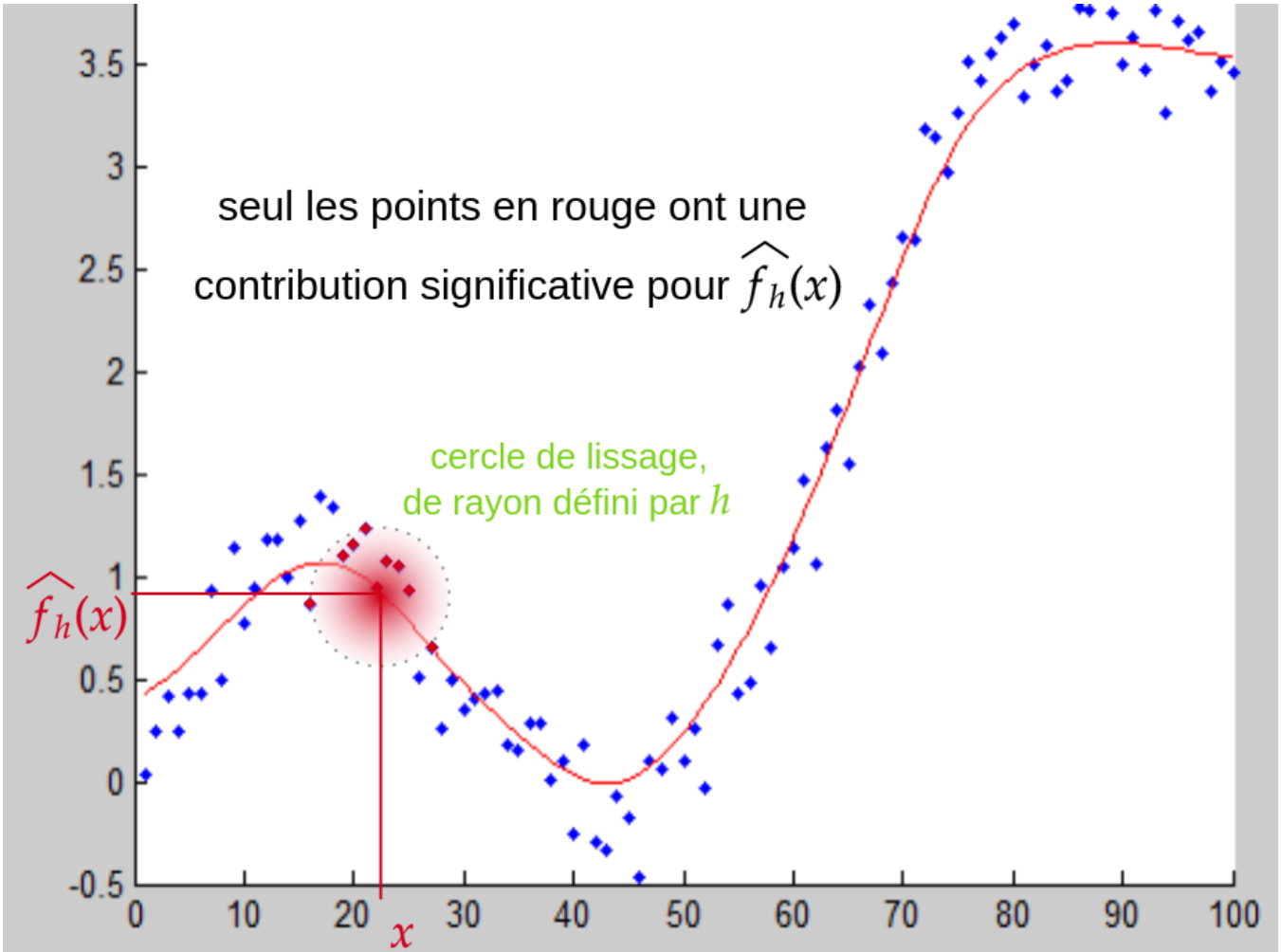
$$\hat{f}_h(x) = \frac{1}{nh} \sum_{i=1}^{n} K(\frac{x - x_i}{h})$$

- h : paramètre de lissage
- K : fonction noyau de lissage
- n : nombre de points



Obtention des formules de la simulation

Obtiention des approximations : prérequis - estimation par noyau



Obtention des formules de la simulation

Obtiention des approximations : prérequis - estimation par noyau

On pose $\mathbf{q} = \mathbf{r}/\mathbf{h}$. La fonction K doit répondre à certains critères :

- K positive, définie et décroissante
- $\lim\limits_{h \to 0} K(r, h) = \delta(r)$
- $\sigma \int_{\mathbb{R}^3} f(q) dV = 1$

Obtention des formules de la simulation

Expressions dont on doit faire l'approximation

- Approximation d'un champs scalaire
- Approximation d'un gradient

$$\vec{\text{grad}}(A)(M) \approx \sum_i \frac{m_i}{\rho_i} A(M_i) \vec{\text{grad}}(K(r_i, h))$$

- Approximation d'un laplacien scalaire

$$\Delta A(M) \approx \sum_i \frac{m_i}{\rho_i} A(M_i) \Delta K(r_i, h))$$

Obtention des formules de la simulation

Expressions dont on doit faire l'approximation

- Approximation d'un champs scalaire
  En posant :

$$A(M) = \int_{\mathbb{R}^3} A(M')\delta(\|M' - M\|)d^3M'$$

$$< A > (M) = \int_{\mathbb{R}^3} A(M')K(\|M' - M\|, h)d^3M'$$

- Approximation d'un gradient

$$\vec{\text{grad}}(A)(M) \approx \sum_i \frac{m_i}{\rho_i} A(M_i)\vec{\text{grad}}(K(r_i, h))$$

- Approximation d'un laplacien scalaire

$$\Delta A(M) \approx \sum_i \frac{m_i}{\rho_i} A(M_i)\Delta K(r_i, h))$$

Obtention des formules de la simulation

Expressions dont on doit faire l'approximation

- Approximation d'un champs scalaire
  On se rend compte que :

$$A(M) = < A > (M) + O(h^2)$$

- Approximation d'un gradient

$$\vec{\text{grad}}(A)(M) \approx \sum_i \frac{m_i}{\rho_i} A(M_i) \vec{\text{grad}}(K(r_i, h))$$

- Approximation d'un laplacien scalaire

$$\Delta A(M) \approx \sum_i \frac{m_i}{\rho_i} A(M_i) \Delta K(r_i, h))$$

Obtention des formules de la simulation

Expressions dont on doit faire l'approximation

- Approximation d'un champs scalaire

$$A(M) \approx \sum_i \frac{m_i}{\rho_i} A(M_i) K(r_i, h)$$

- Approximation d'un gradient

$$\vec{\text{grad}}(A)(M) \approx \sum_i \frac{m_i}{\rho_i} A(M_i) \vec{\text{grad}}(K(r_i, h))$$

- Approximation d'un laplacien scalaire

$$\Delta A(M) \approx \sum_i \frac{m_i}{\rho_i} A(M_i) \Delta K(r_i, h))$$

Obtention des formules de la simulation

Expressions dont on doit faire l'approximation

- Approximation d'un champs scalaire

$$A(M) \approx \sum_i \frac{m_i}{\rho_i} A(M_i) K(r_i, h)$$

- Approximation d'un gradient

$$\vec{\text{grad}}(A)(M) \approx \sum_i \frac{m_i}{\rho_i} A(M_i) \vec{\text{grad}}(K(r_i, h))$$

- Approximation d'un laplacien scalaire

$$\Delta A(M) \approx \sum_i \frac{m_i}{\rho_i} A(M_i) \Delta K(r_i, h))$$

Obtention des formules de la simulation

Expressions dont on doit faire l'approximation

- Approximation d'un champs scalaire

$$A(M) \approx \sum_i \frac{m_i}{\rho_i} A(M_i) K(r_i, h)$$

- Approximation d'un gradient

$$\vec{\text{grad}}(A)(M) \approx \sum_i \frac{m_i}{\rho_i} A(M_i) \vec{\text{grad}}(K(r_i, h))$$

- Approximation d'un laplacien scalaire

$$\Delta A(M) \approx \sum_i \frac{m_i}{\rho_i} A(M_i) \Delta K(r_i, h))$$

Obtention des formules de la simulation

Expression discrètes des forces

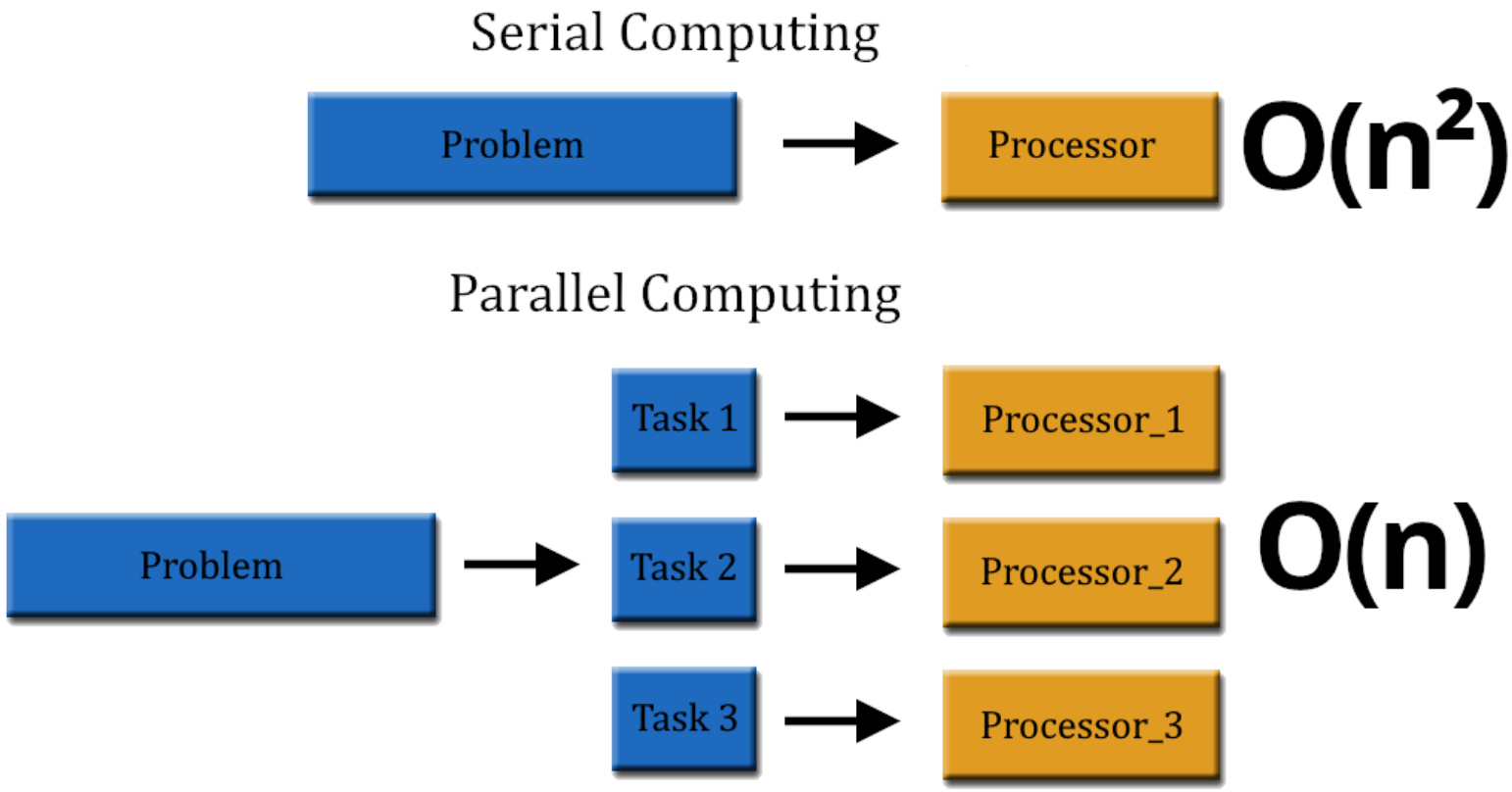
1. Densité :
$$\rho(M,t) = \sum_i m_i K(r_i, h)$$
2. Poids :
$$\vec{P}(M,t) = \rho(M,t)\vec{g}$$
3. Pression :
$$\vec{F_p}(M,t) = -\sum_i \frac{m_i}{\rho(M_i,t)} p(M_i,t) \vec{\text{grad}}(K(r_i,h))$$
4. Viscositée :
$$\vec{F_v}(M,t) = \eta \sum_{j=1}^{3} \sum_i m_i \frac{v_i}{\rho(M_i,t)} \Delta K(r_i,h)\vec{u_j}$$

Optimisation de l'algorithme

Optimisation de l'algorithme

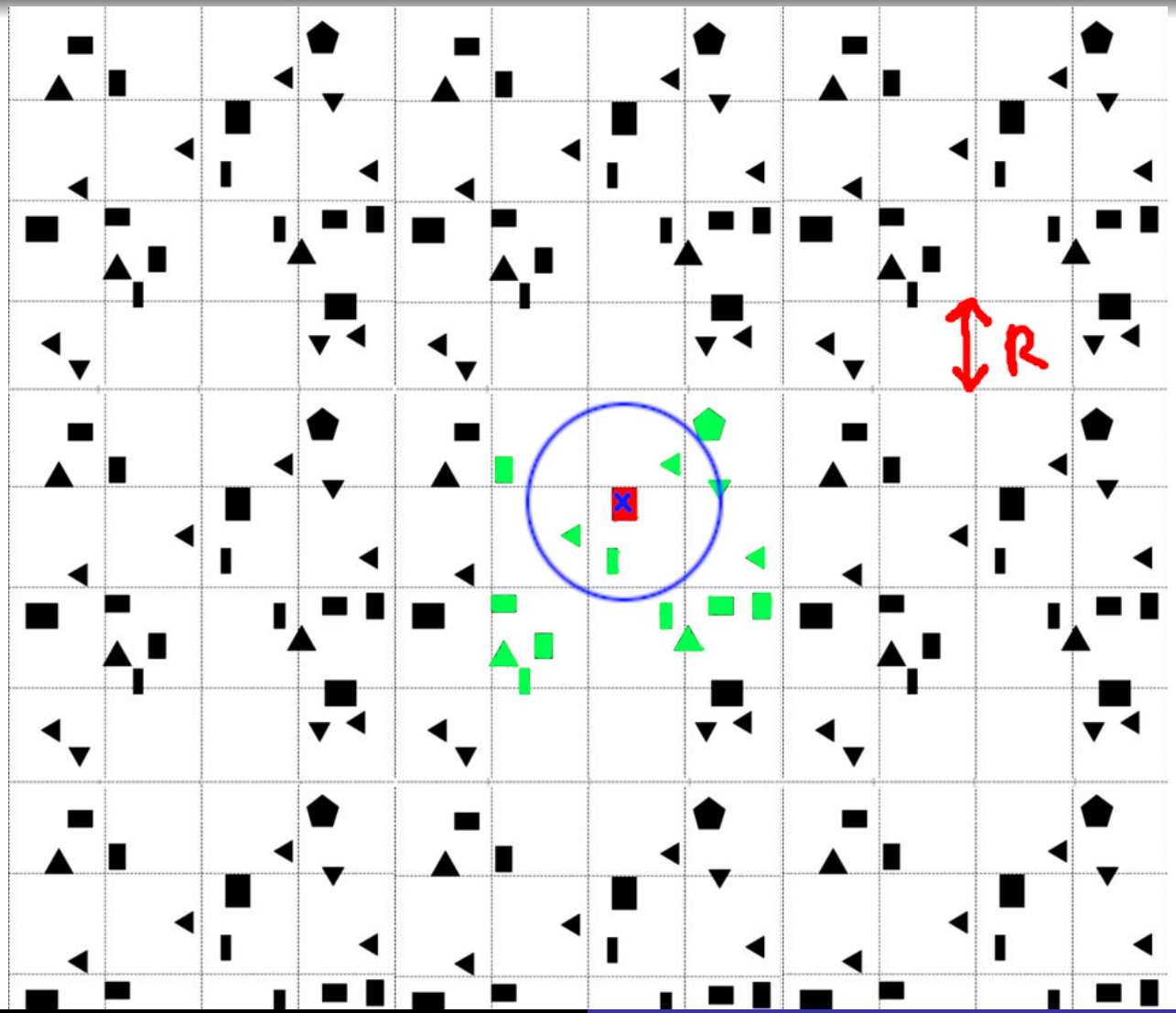
Optimisation de l'algorithme

Parrallélisme : La carte graphique



Optimisation de l'algorithme

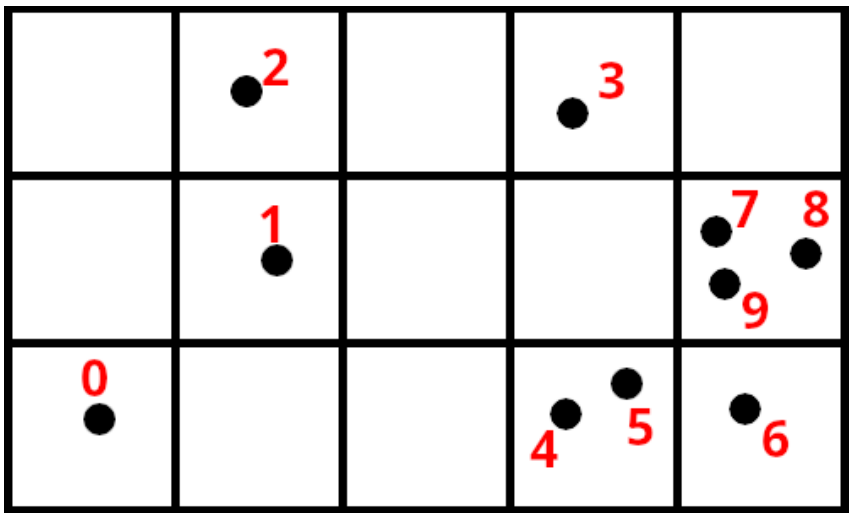
Partitionnement de l'espace



Optimisation de l'algorithme

Explication de l'algorithme de partitionnement

$$[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$$



Optimisation de l'algorithme

Explication de l'algorithme de partitionnement

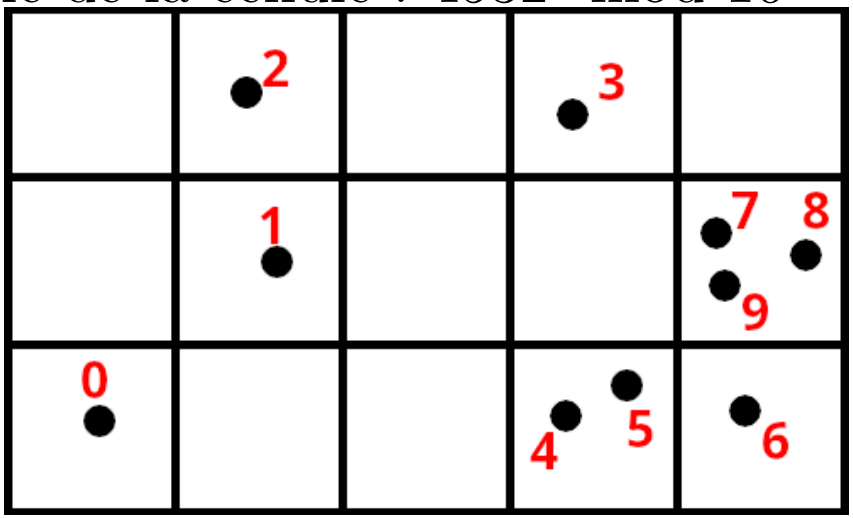
$$[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$$

Indice : 7

Coordonnées de la case : $(4, 1)$

Hachage des coordonnées : 4532

Clé de la cellule : $4532 \mod 10 = 2$



Optimisation de l'algorithme

Explication de l'algorithme de partitionnement

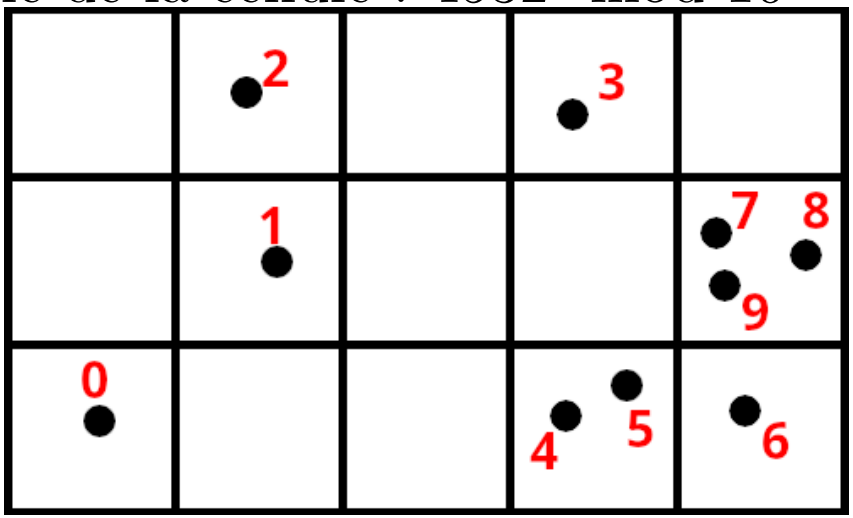
$$[0, 1, 2, 3, 4, 5, 6, \overset{7}{2}, 8, 9]$$

Indice : 7

Coordonnées de la case : $(4, 1)$

Hachage des coordonnées : 4532

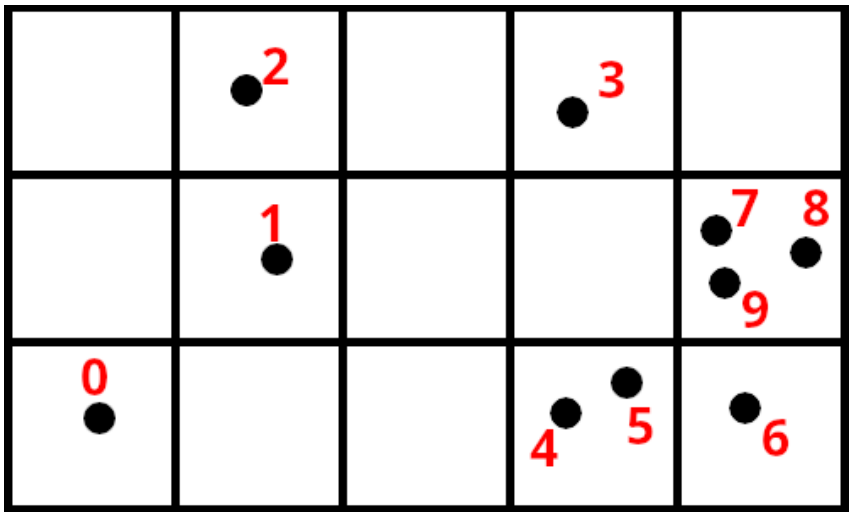
Clé de la cellule : $4532 \mod 10 = 2$



Optimisation de l'algorithme

Explication de l'algorithme de partitionnement

$$\begin{array}{cccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ [& 9, & 0, & 6, & 7, & 1, & 1, & 4, & 2, & 2, & 2 &] \end{array}$$

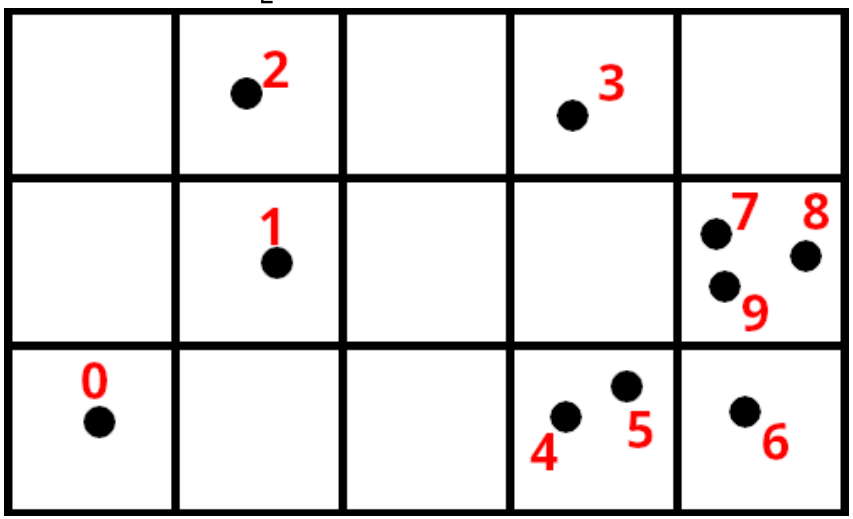


Optimisation de l'algorithme

Explication de l'algorithme de partitionnement

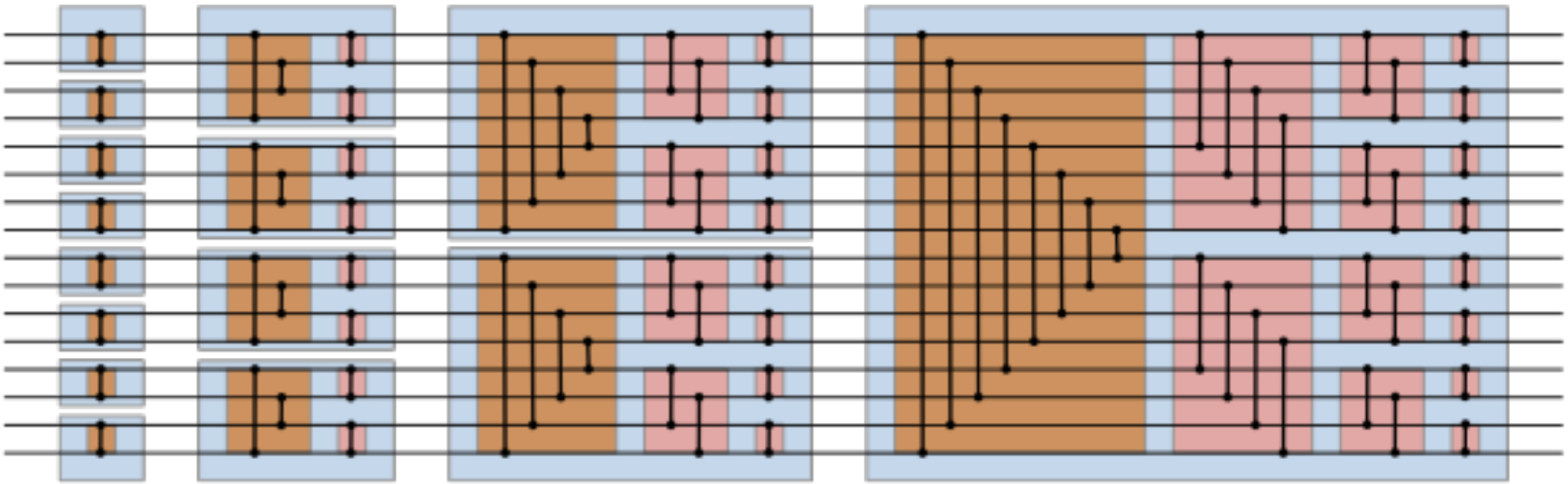
$$\text{Liste spatiale : } \left[\begin{matrix} 1 & 4 & 5 & 7 & 8 & 9 & 6 & 2 & 3 & 0 \\ 0, & 1, & 1, & 2, & 2, & 2, & 4, & 6, & 7, & 9 \end{matrix} \right]$$

$$\text{Liste des départs : } [\,0, 1, 3, +\infty, 6, +\infty, 8, +\infty, 9]$$



Optimisation de l'algorithme

Tri bitonique

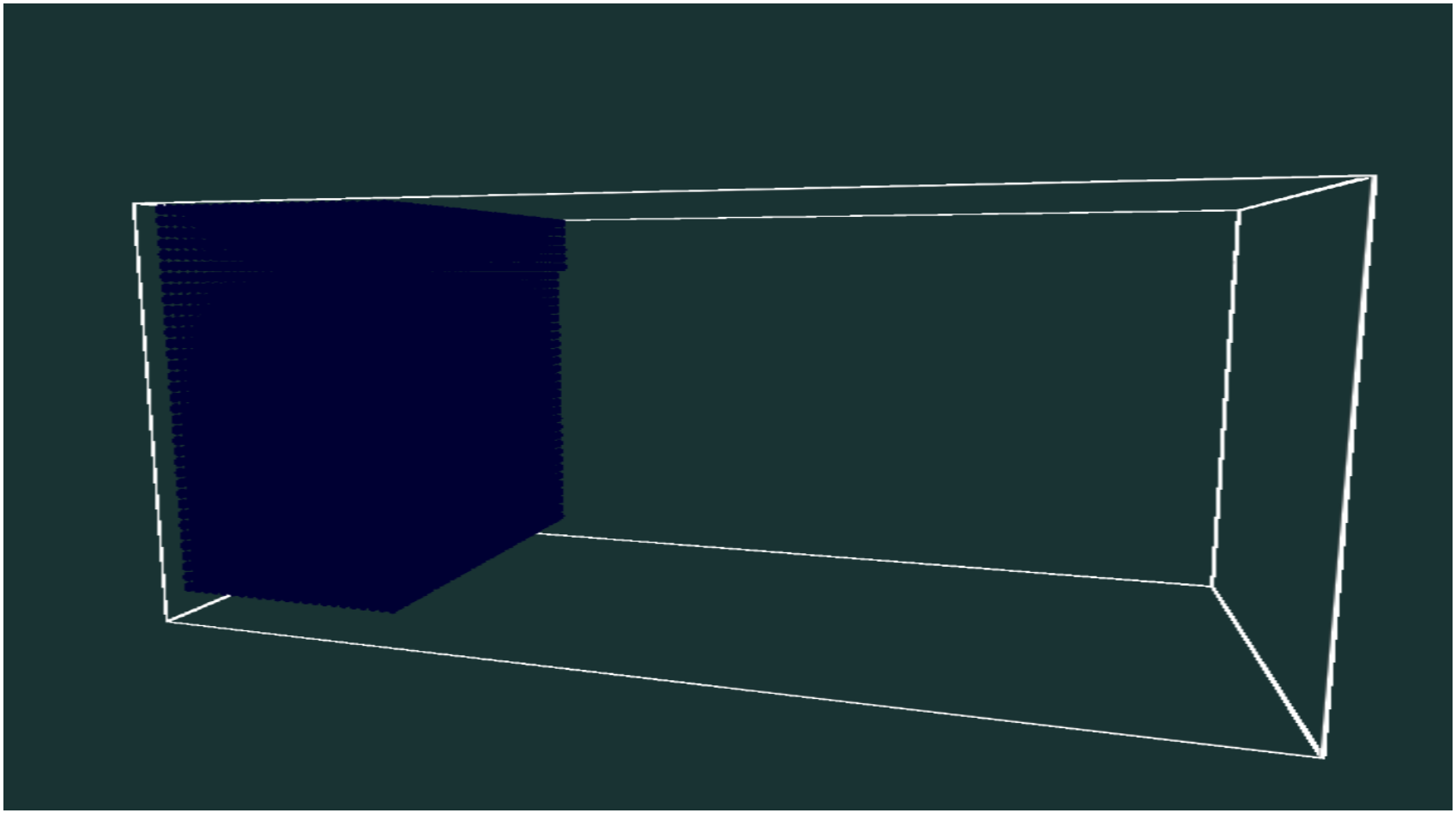


Conclusion

Conclusion

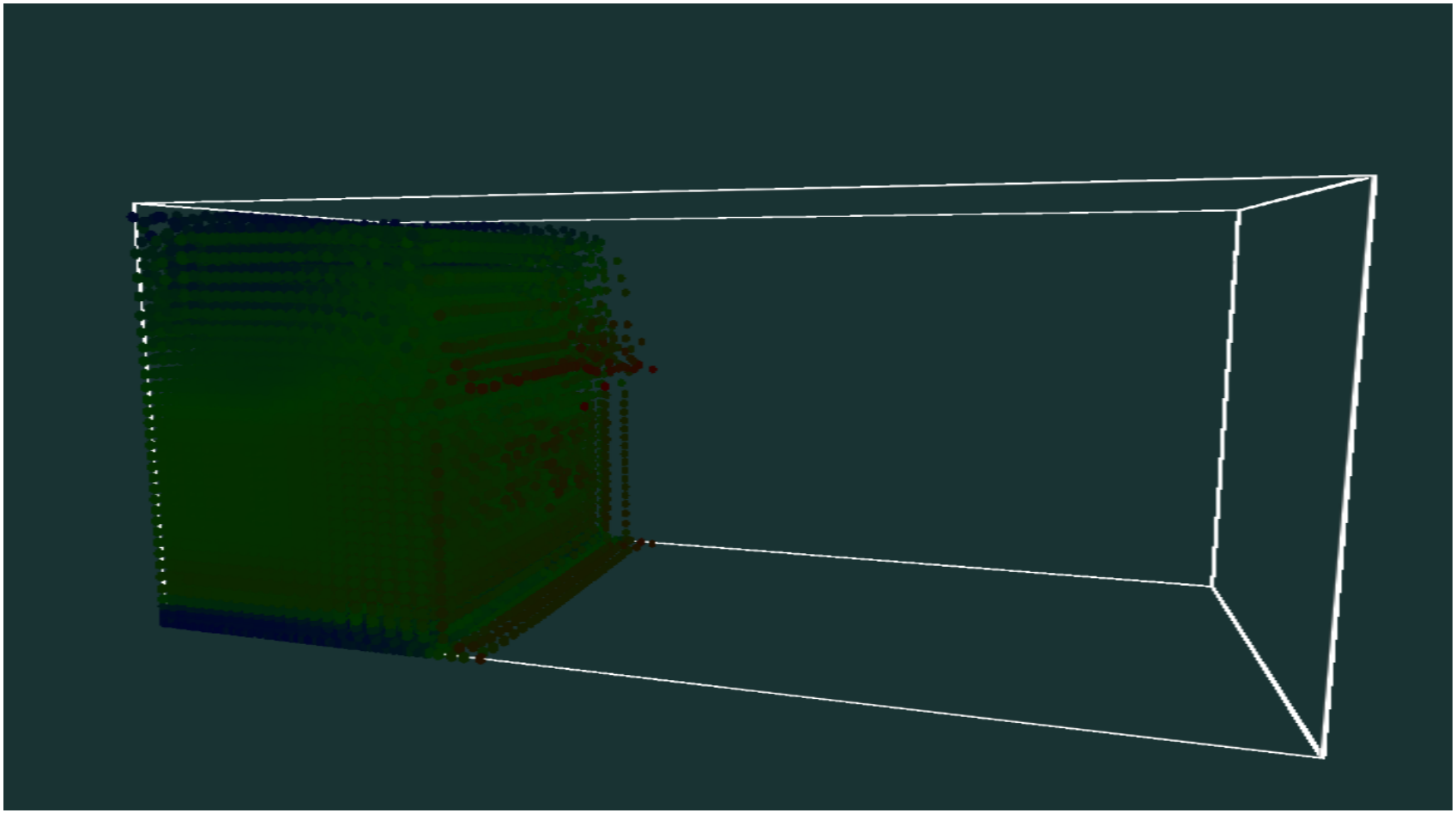
Conclusion

La simulation



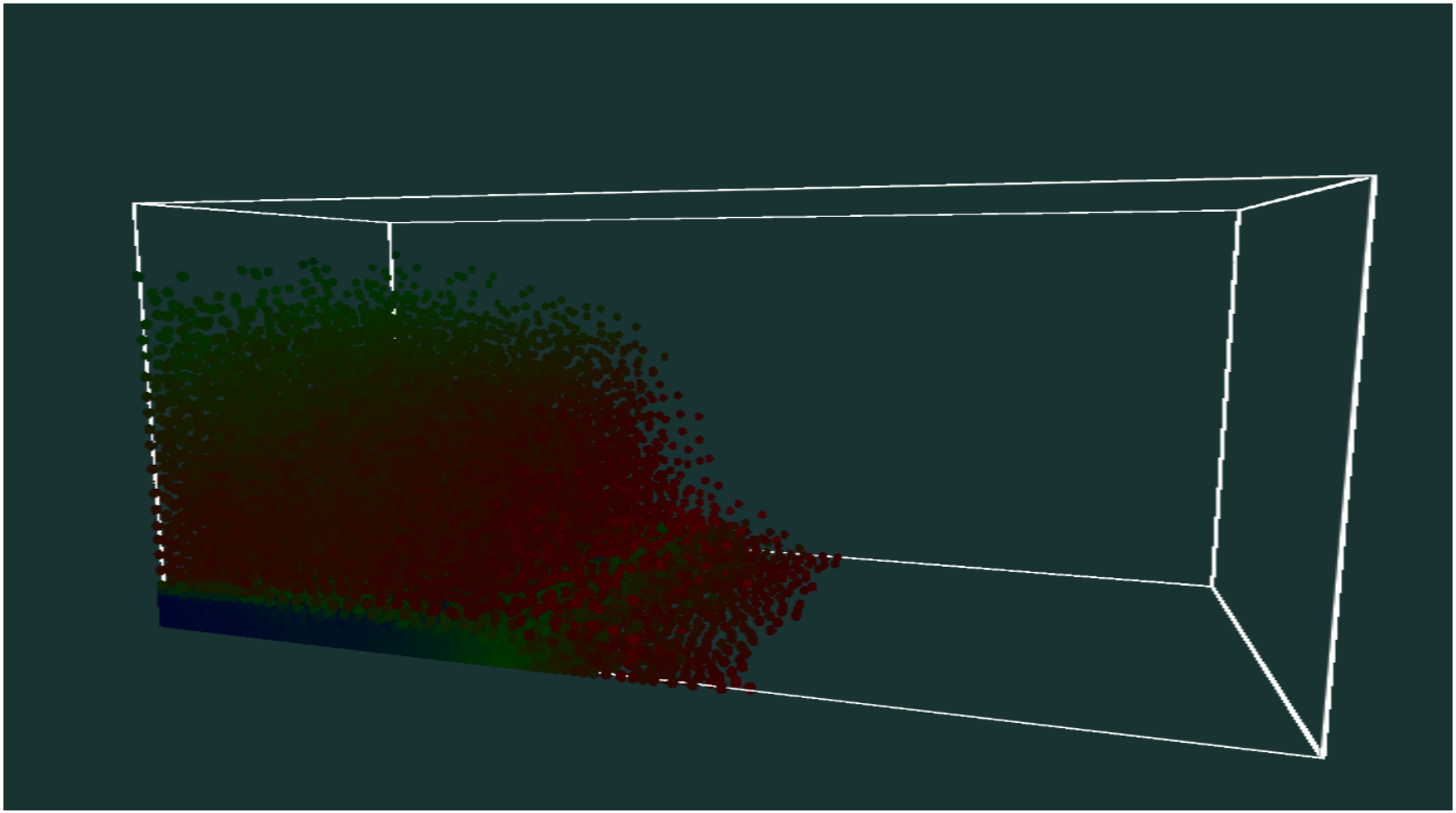
Conclusion

La simulation



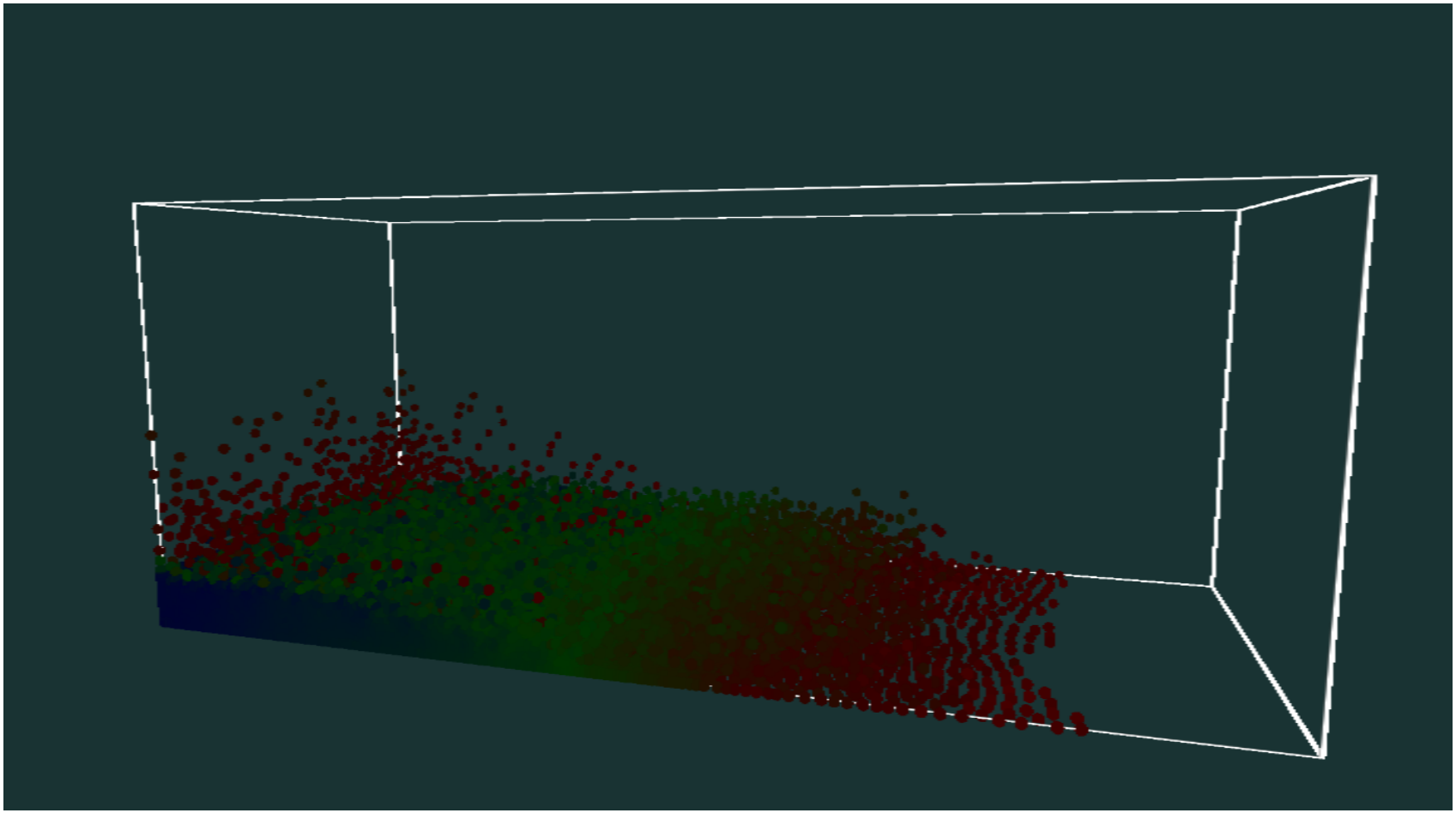
Conclusion

La simulation



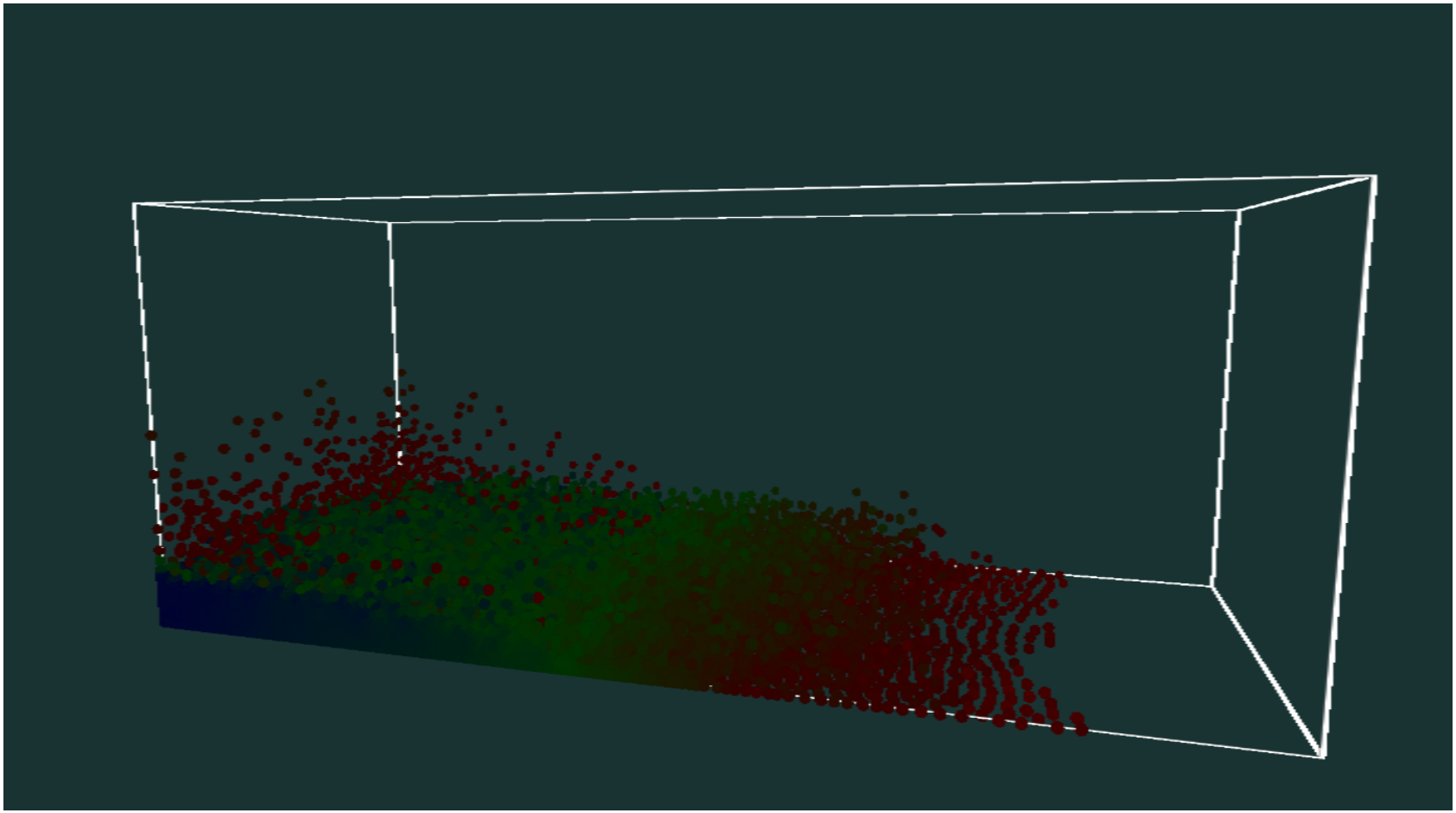
Conclusion

La simulation



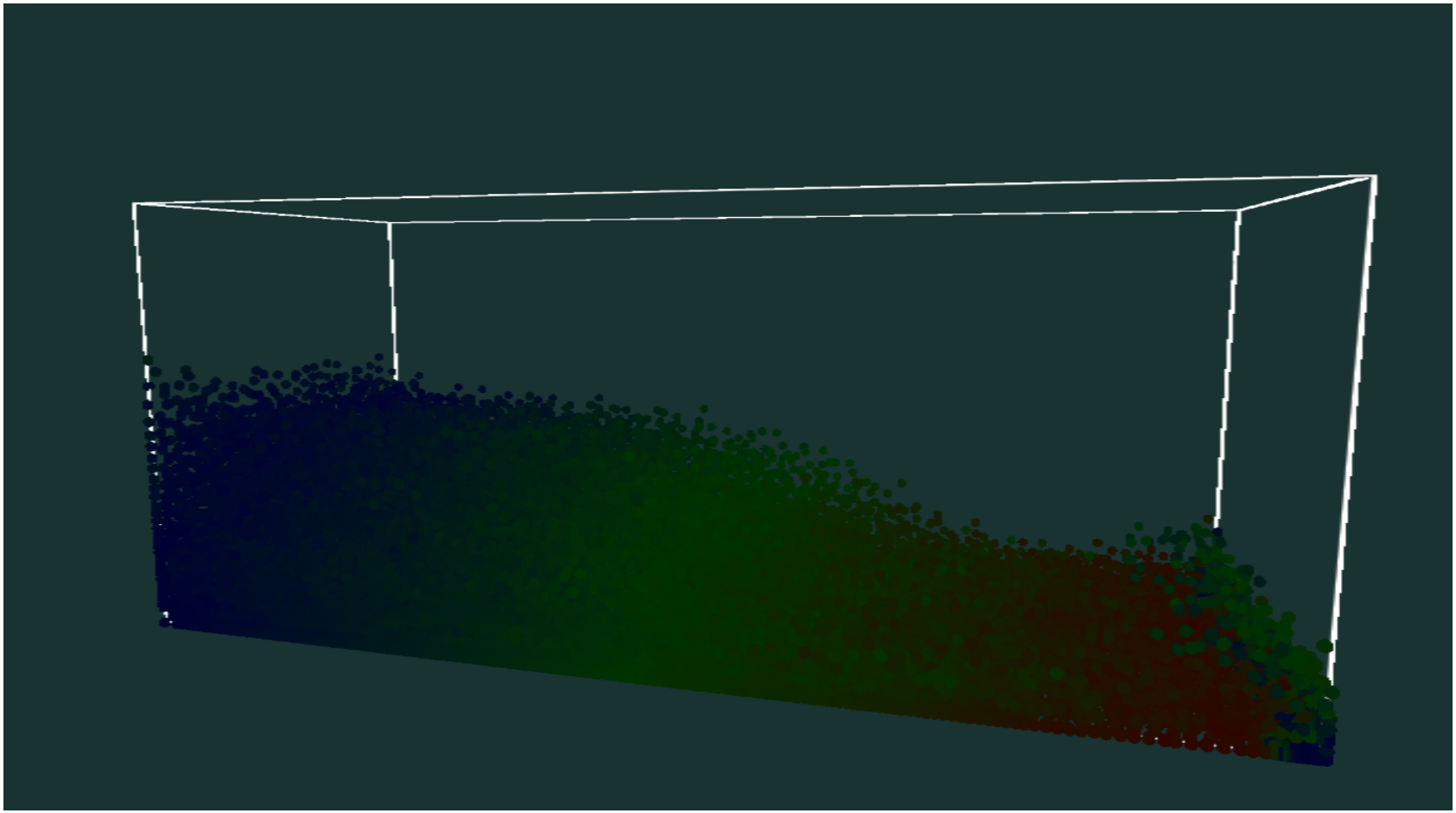
Conclusion

La simulation



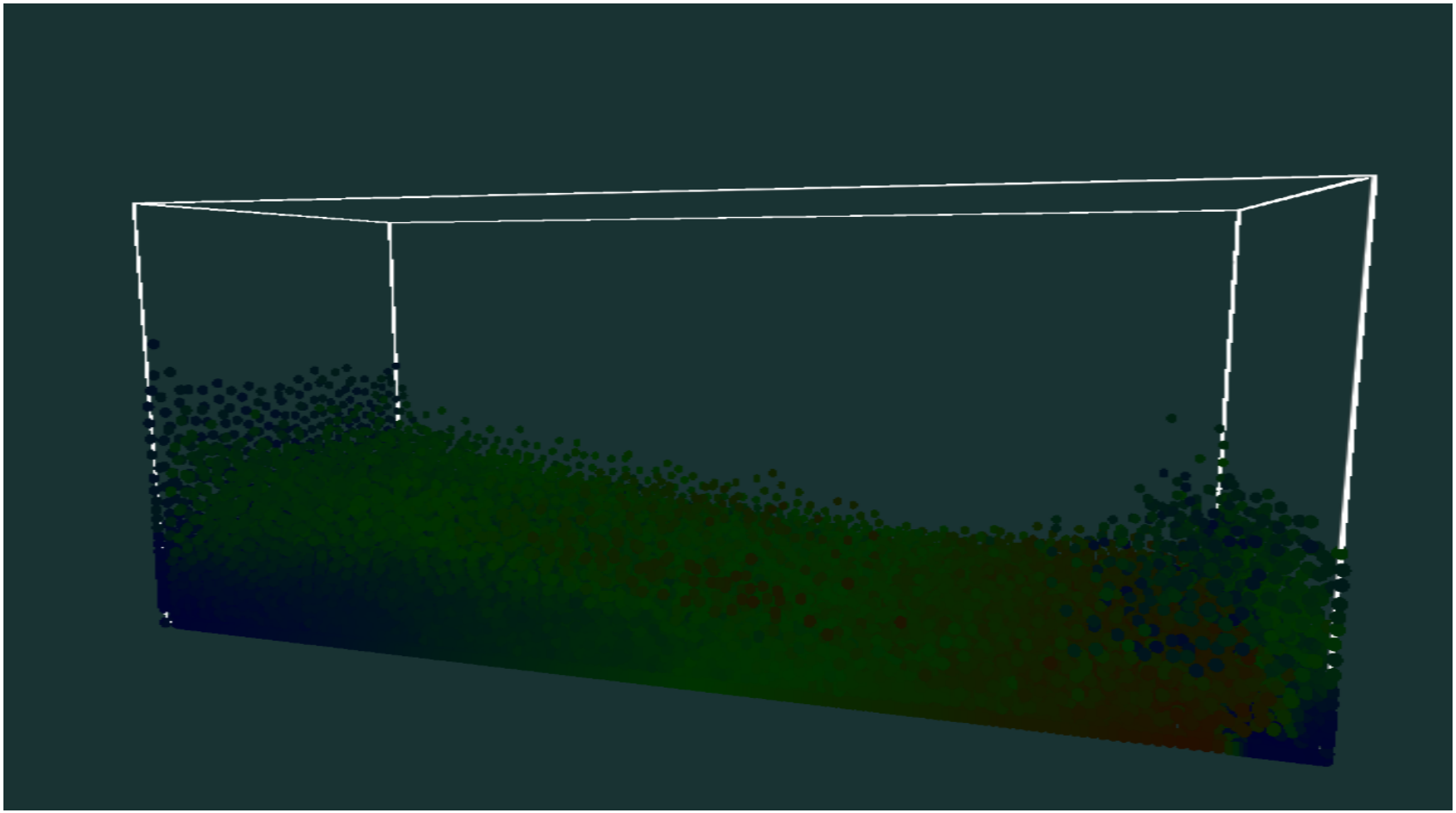
Conclusion

La simulation



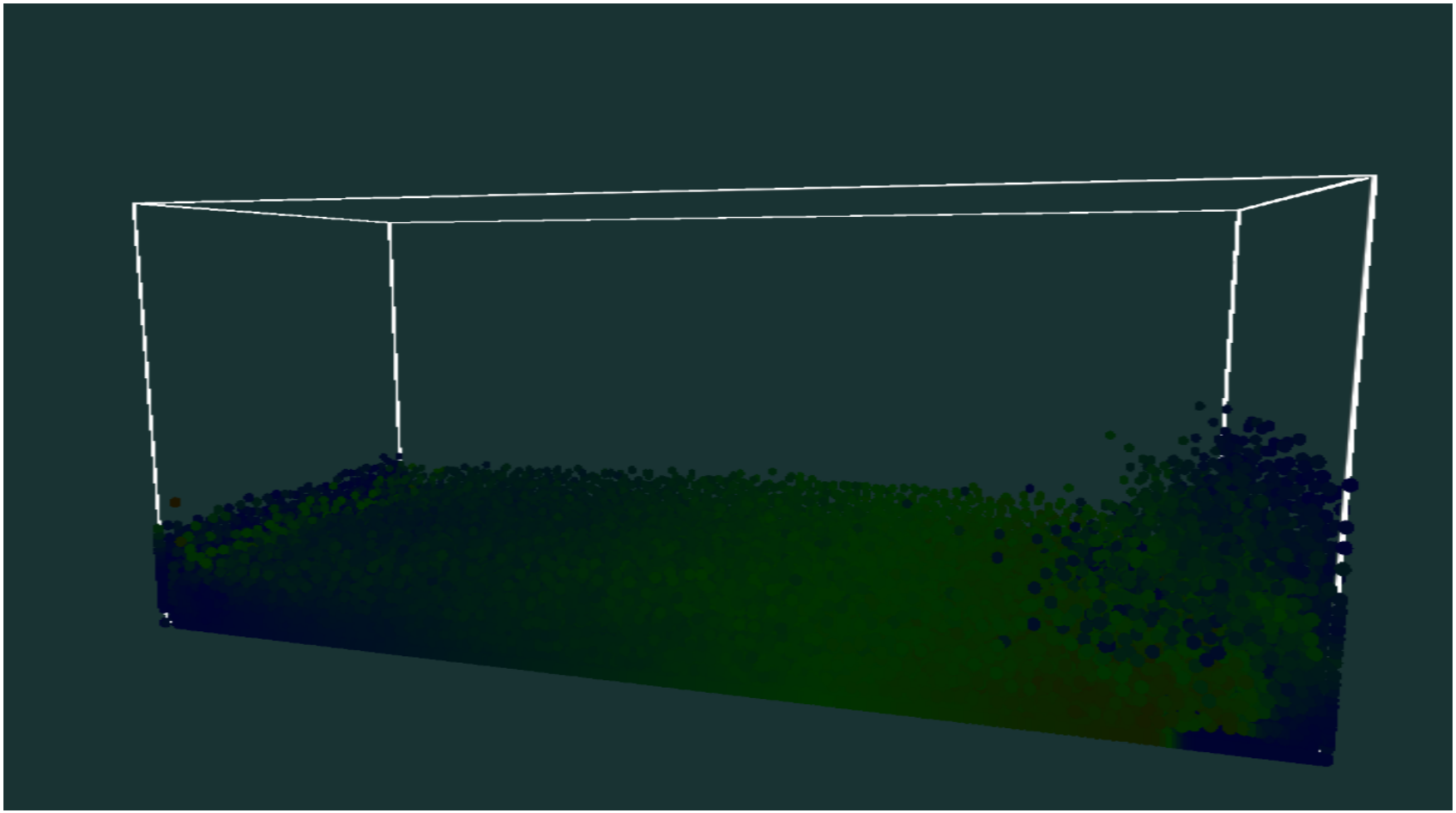
Conclusion

La simulation



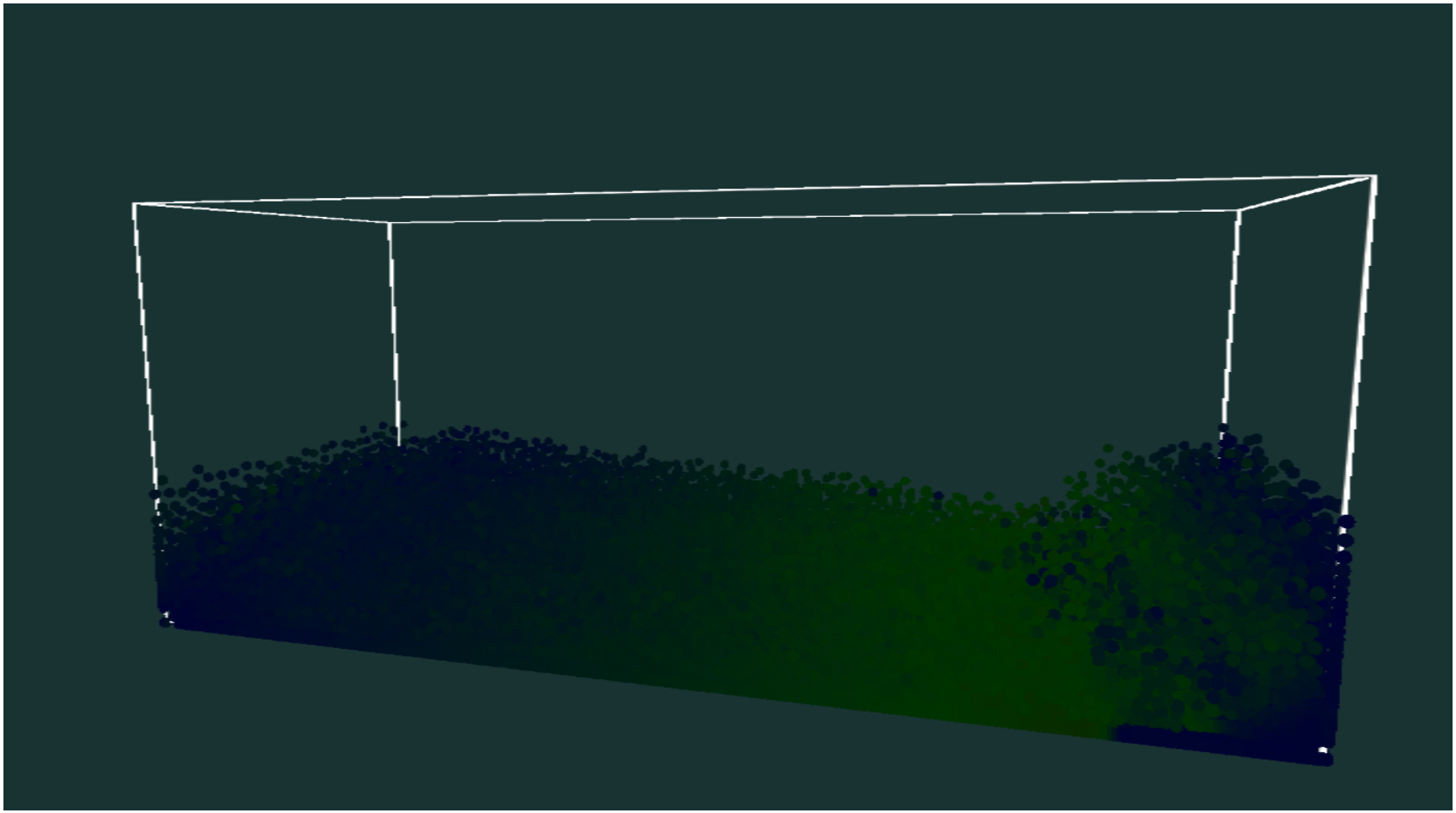
Conclusion

La simulation



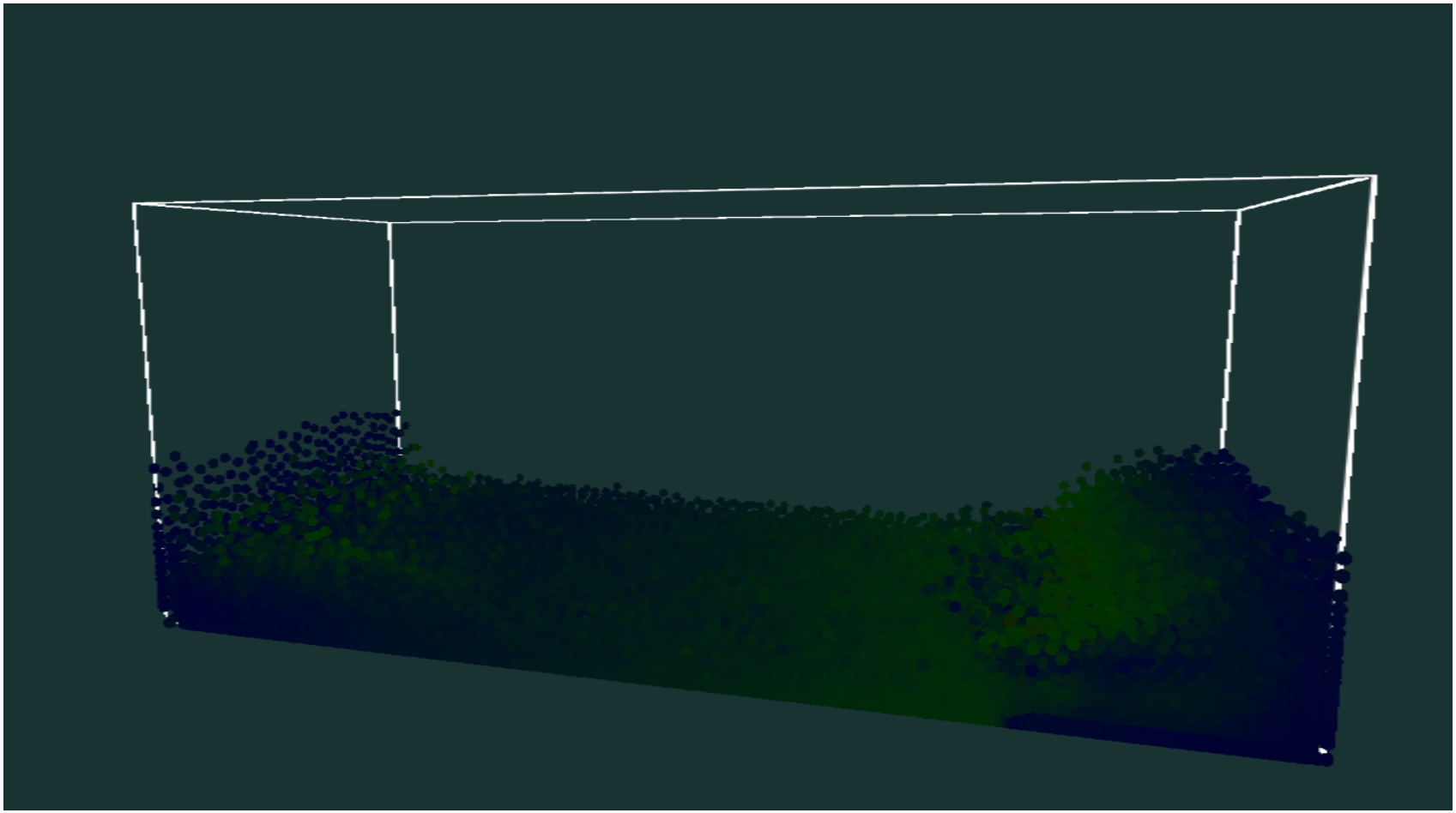
Conclusion

La simulation



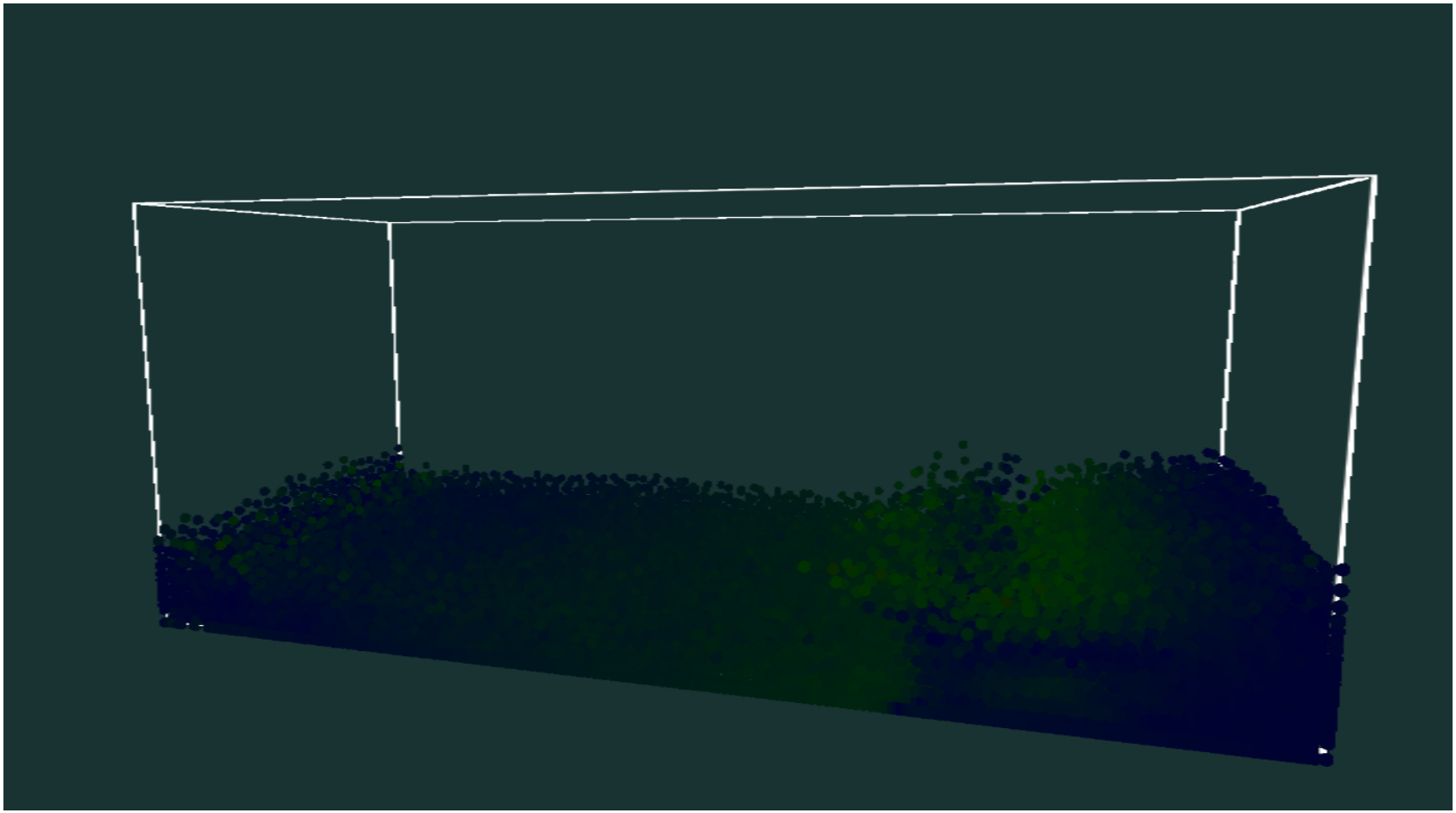
Conclusion

La simulation



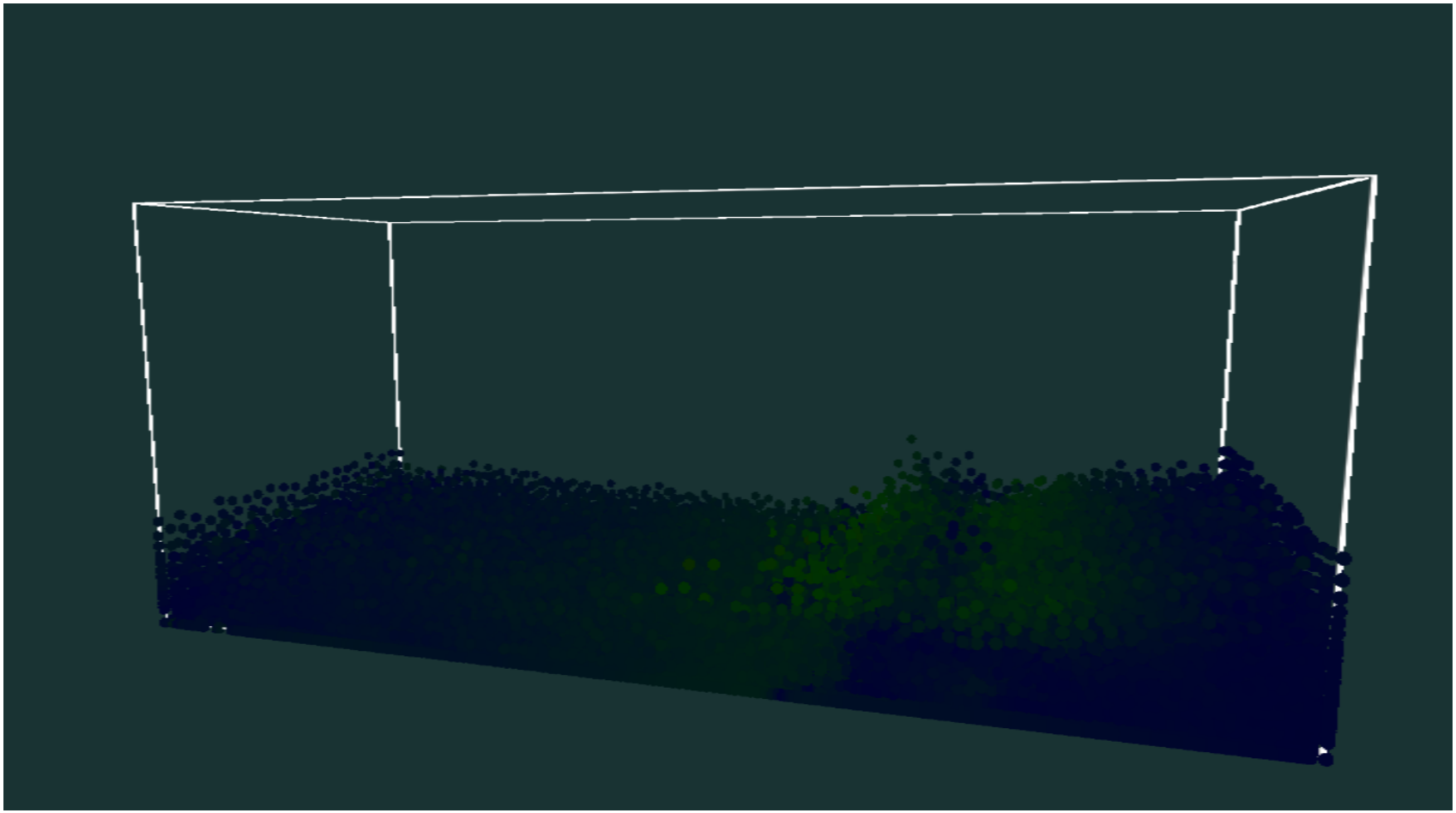
Conclusion

La simulation



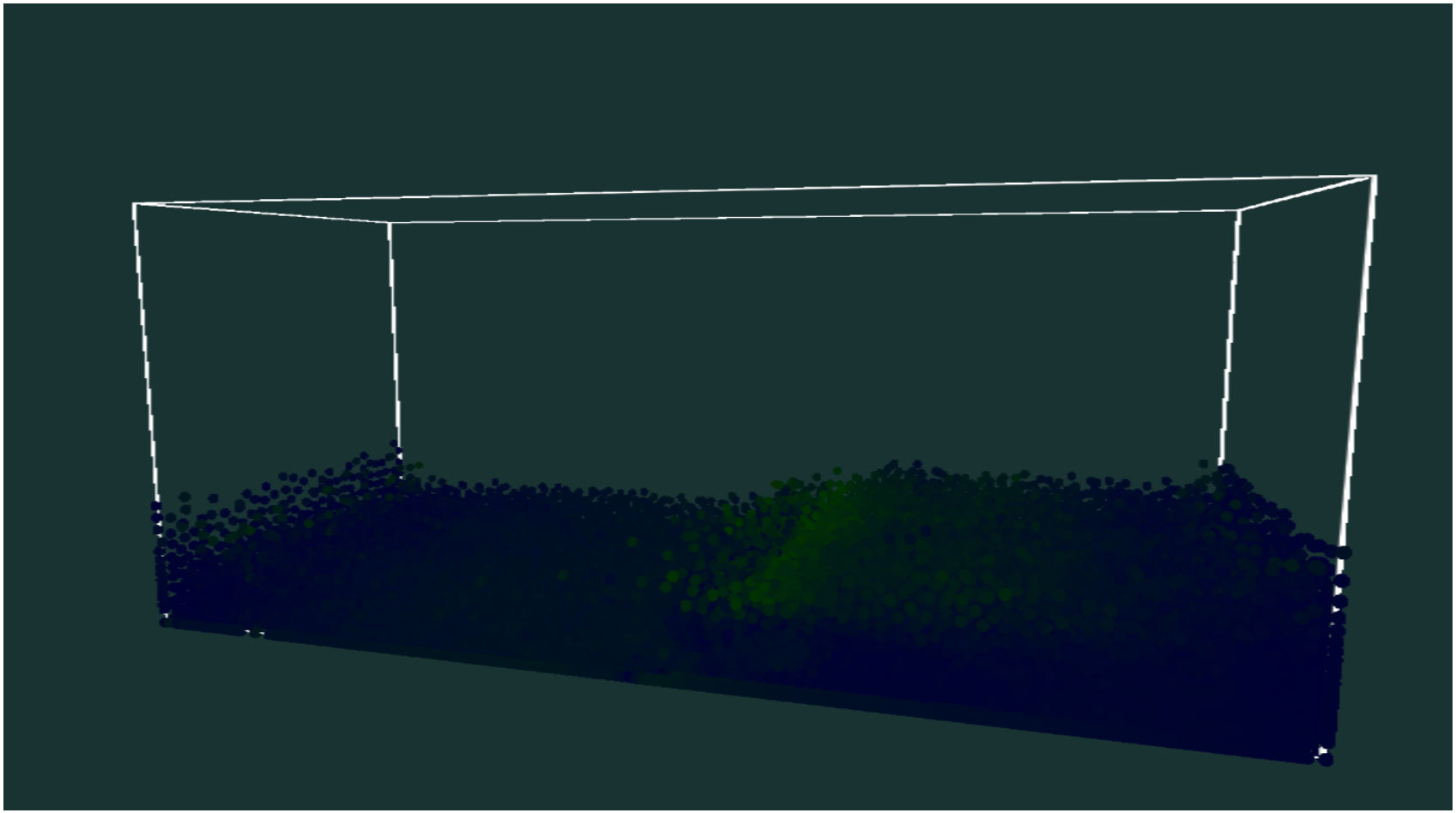
Conclusion

La simulation



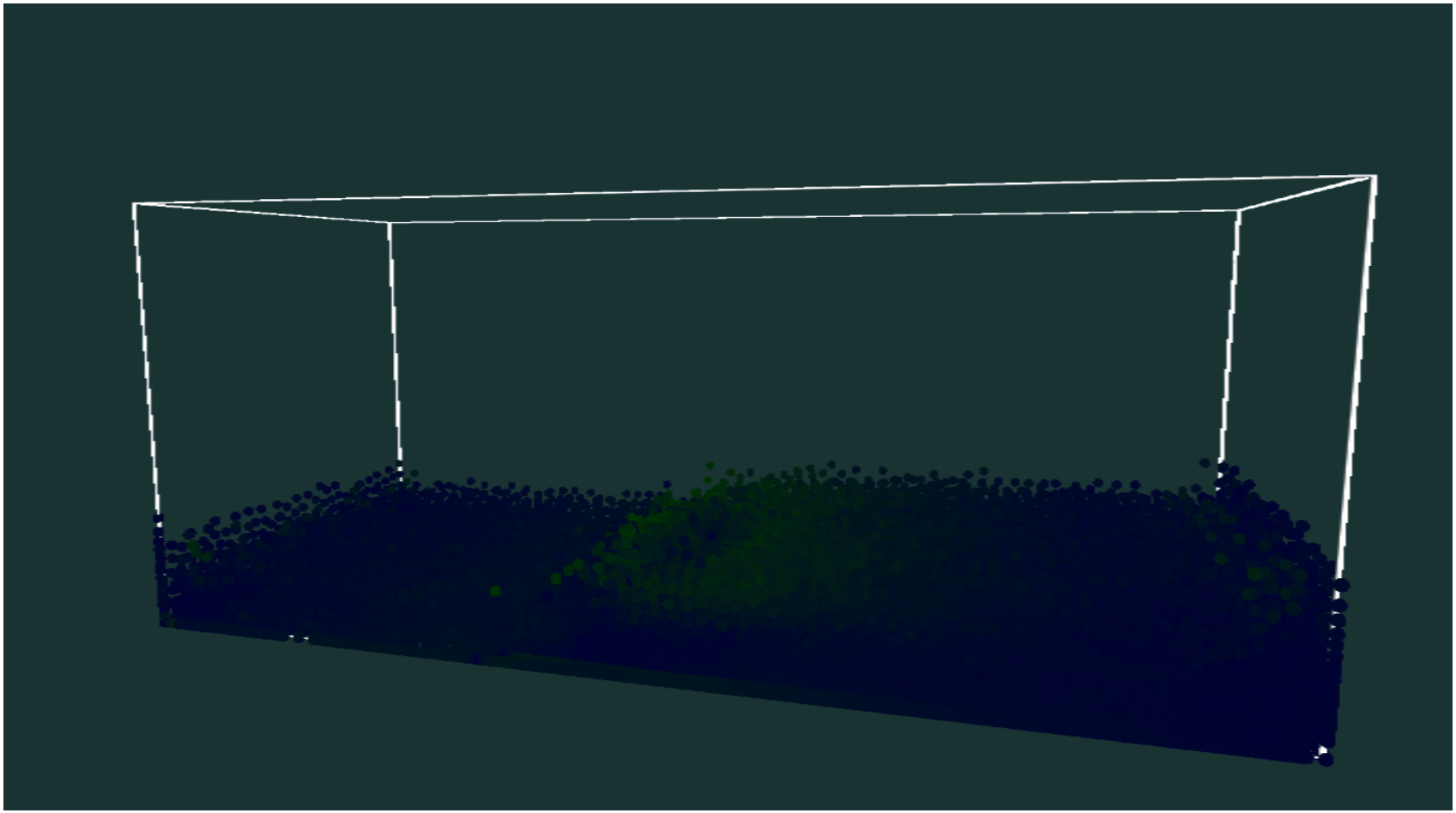
Conclusion

La simulation



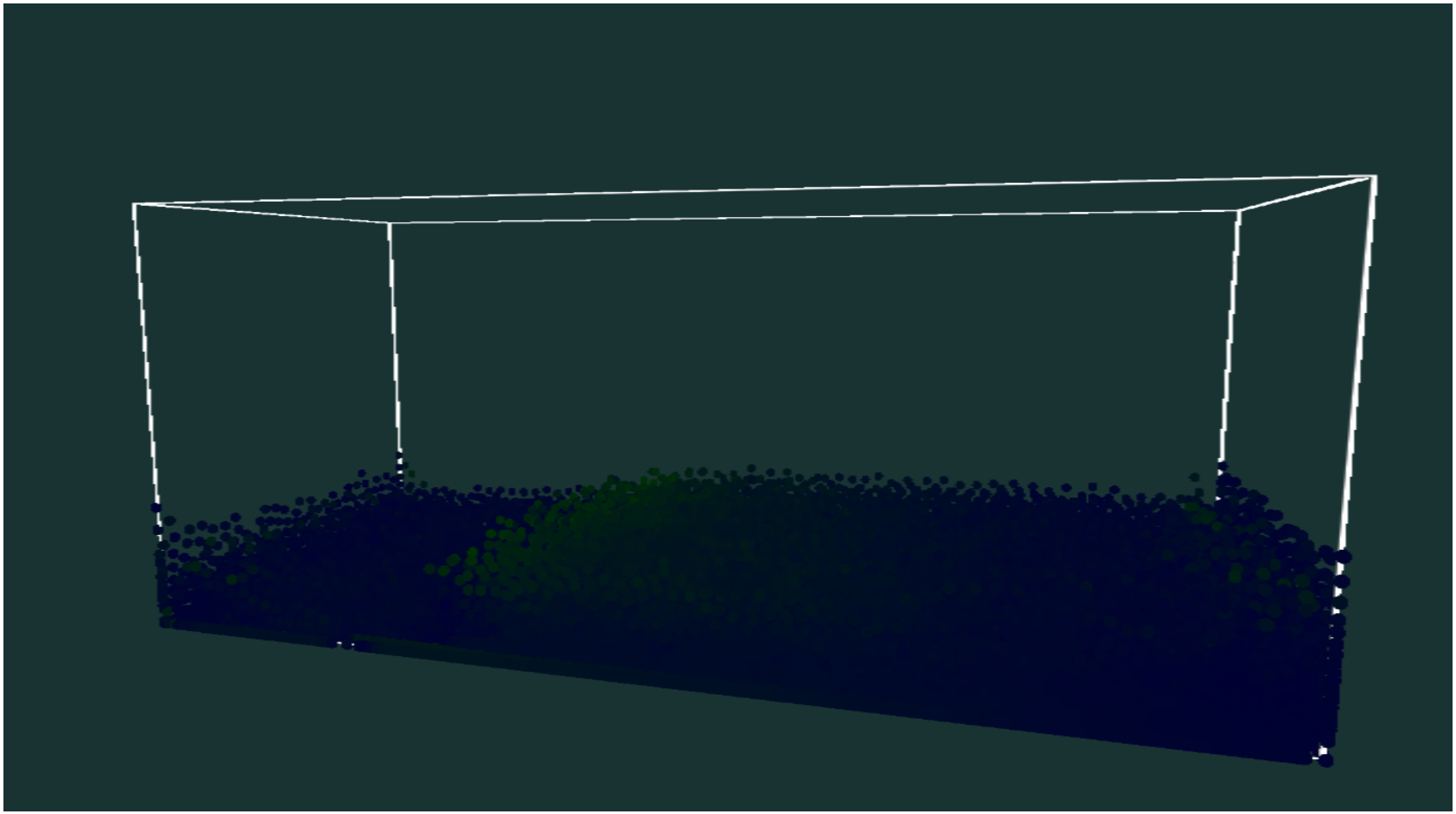
Conclusion

La simulation



Conclusion

La simulation



Conclusion

Impact de l'optimisation

Nombre de particules maximum pour que la simulation tourne à 60 images par secondes :

Ajout	Avant	Après
Passage sur la carte graphique	350	5000
Ajout du partitionnement de l'espace	5000	20.000
Ajout du tri bitonique	20.000	60.000

Matériel :

- Processeur : Intel i7-6700 (8) @ 4.0GHz
- Carte Graphique : NVIDIA GeForce GTX 1060 6GB
- Mémoire Vive : 16Go

Annexe

Annexe

Annexe

Code - fluid.py

```
class Fluid(Entity):
    def __init__(self, particle_count: int, particle_size: float, simulation_corner_1: list[float], simula
        ...
    def init_fluid_shaders(self) -> None:
        ...
    def create_bounding_box(self, scene: Scene) -> tuple[np.ndarray, np.ndarray]:
        ...
    def create_initial_particle_positions(self, position: np.ndarray, scale: np.ndarray) -> np.ndarray:
        ...
    def set_simulation_param(self, name: SimParams, value: any) -> None:
        ...
    def get_simulation_param(self, name: SimParams, from_gpu: bool = False) -> any:
        ...
    def get_buffers(self, binding_point: int, view_as: str = "<f4",start_point: int = 0, size: int = None)
        ...
    def mounted(self, scene: Scene) -> None:
        ...
    def draw(self, scene: Scene)-> None:
        PARTICLEAREA_MATERIAL.use()
        PARTICLEAREA_SHADERS.set_mat4x4("model", self.particle_area.get_model_matrix())
        PARTICLEAREA_MESH.draw()

        self.particle_shaders.set_vec3("camPos", scene.get_camera().get_position())
        self.particle_mesh.prepare_to_draw()
        # On dessine des instances de la particule et non n entités disctinctes pour aller beaucoup plus
        glDrawArraysInstanced(GL_TRIANGLES, 0, self.particle_mesh.vertex_count ,self.particle_count)
```

Annexe

Code - fluid.py

```
def update(self, delta:float, force_update = False) -> None:
    if self.disable_simulation and not(force_update):
        return

    self.set_simulation_param(SimParams.DELTA, delta)

    self.compute_external.dispatch(self.particle_count)
    self.compute_spatial_hash.dispatch(self.particle_count)
    self.compute_sort.sort_and_calculate_offsets()
    self.compute_density.dispatch(self.particle_count)
    self.compute_pressure.dispatch(self.particle_count)
    self.compute_viscosity.dispatch(self.particle_count)
    self.compute_update_pos.dispatch(self.particle_count)

def reset_simulation(self, particle_count: int) -> None:
    ...
def destroy(self) -> None:
    ...
```

Annexe

Code - fluid.py

```
class GPUSort:
    def __init__(self, index_buffer_size: int) -> None:
        ...
    def next_power_of_2(self, x :int):
        ...
    def sort(self):
        self.sort_shader.set_int("numEntries", self.buffer_size)

        num_stages = int(log2(self.next_power_of_2(self.buffer_size)))

        for stage_index in range(num_stages):
            for step_index in range(stage_index + 1):
                # Même chose que 2**(stage_index - step_index) mais beaucoup plus rapide
                groupWidth = 1 << (stage_index - step_index)
                groupHeigth = 2* groupWidth - 1
                self.sort_shader.set_int("groupWidth", groupWidth)
                self.sort_shader.set_int("groupHeight", groupHeigth)
                self.sort_shader.set_int("stepIndex", step_index)

                instances_to_dispatch = self.next_power_of_2(self.buffer_size) // 2
                self.sort_shader.dispatch(instances_to_dispatch)

    def sort_and_calculate_offsets(self):
        self.sort()

        self.offset_shader.set_int("numEntries", self.buffer_size)
        self.offset_shader.dispatch(self.buffer_size)
```

Annexe

Code - density.comp

```
void calculateDensities(uint id) {
    if (id >= numParticles) return;
    vec3 pos = predictedPositions[id];
    ivec3 originCell = GetCell3D(pos, smoothingRadius);
    float sqrRadius = smoothingRadius * smoothingRadius;
    float density = 0;
    for (int i = 0; i < 27; i ++) {
        int hash = HashCell3D(originCell + offsets3D[i]);
        int key = KeyFromHash(hash, numParticles);
        uint currIndex = spatialOffsets[key];
        while (currIndex < numParticles) {
            uvec3 indexData = spatialIndices[currIndex];
            currIndex ++;
            if (indexData[2] != key) break;
            if (indexData[1] != hash) continue;
            uint neighbourIndex = indexData[0];
            vec3 neighbourPos = predictedPositions[neighbourIndex];
            vec3 offsetToNeighbour = neighbourPos - pos;
            float sqrDstToNeighbour = dot(offsetToNeighbour, offsetToNeighbour);
            if (sqrDstToNeighbour > sqrRadius) continue;
            float dst = sqrt(sqrDstToNeighbour);
            density += DensityKernel(dst, smoothingRadius);
        }
    }
    densities[id] = density;
}
```

Annexe

Code - externalForces.comp

```
void calculateGravity(uint index) {
    if (index >= numParticles) return;

    //Force de gravité
    velocities[index] += vec3(0, gravity, 0) * delta;

    predictedPositions[index] = positions[index] + velocities[index] * delta;
}
```

Annexe

Code - gpuSortOffset.comp

```
#version 460 core

layout(binding=5, std430) buffer spatialIndicesBuffer { uvec3 spatialIndices[]; };
layout(binding=6, std430) buffer spatialOffsetsBuffer { uint spatialOffsets[]; };

uniform int numEntries;

layout(local_size_x = 64, local_size_y = 1, local_size_z = 1) in;
void main() {
    uint i = gl_GlobalInvocationID.x;

    if (i >= numEntries) { return; }

    uint null = numEntries;

    uint key = spatialIndices[i].z;
    uint keyPrev = i == 0 ? null : spatialIndices[i - 1].z;

    if (key != keyPrev)
    {
        spatialOffsets[key] = i;
    }
}
```

Annexe

Code - gpuSort.comp

```
#version 460 core
// le veteur est de la forme (indexe d'origine, hash, clé)
layout(binding=5, std430) buffer spatialIndicesBuffer { uvec3 spatialIndices[]; };
uniform int numEntries;
uniform int groupWidth;
uniform int groupHeight;
uniform int stepIndex;
layout(local_size_x = 64, local_size_y = 1, local_size_z = 1) in;
void main() {
    uint i = gl_GlobalInvocationID.x;
    uint hIndex = i & (groupWidth - 1);
    uint indexLeft = hIndex + (groupHeight + 1) * (i / groupWidth);
    uint rightStepSize = stepIndex == 0 ? groupHeight - 2 * hIndex : (groupHeight + 1) / 2;
    uint indexRight = indexLeft + rightStepSize;
    // S'arrete si on dépasse le nombre de particules à tier (lorsque le nombre de particules n'est pas u
    if (indexRight >= numEntries) return;
    uint valueLeft = spatialIndices[indexLeft].z;
    uint valueRight = spatialIndices[indexRight].z;
    // On échange les valeurs si elles sont décroissantes
    if (valueLeft > valueRight)
    {
        uvec3 temp = spatialIndices[indexLeft];
        spatialIndices[indexLeft] = spatialIndices[indexRight];
        spatialIndices[indexRight] = temp;
    }
};
```

Annexe

Code - pressure.comp

```
void main() {
    uint index = gl_GlobalInvocationID.x;
    if (index >= numParticles) return;
    float density = densities[index];
    float pressure = pressureFromDensity(density);
    vec3 pressureForce = vec3(0);
    vec3 pos = predictedPositions[index];
    ivec3 originCell = GetCell3D(pos, smoothingRadius);
    float sqrRadius = smoothingRadius * smoothingRadius;
    for (int i = 0; i < 27; i ++) {
        uint hash = HashCell3D(originCell + offsets3D[i]);
        uint key = KeyFromHash(hash, numParticles);
        uint currIndex = spatialOffsets[key];
        while (currIndex < numParticles) {
            uvec3 indexData = spatialIndices[currIndex];
            currIndex ++;
            if (indexData[2] != key) break;
            if (indexData[1] != hash) continue;
            uint neighbourIndex = indexData[0];
            if (neighbourIndex == index) continue;
            vec3 neighbourPos = predictedPositions[neighbourIndex];
            vec3 offsetToNeighbour = neighbourPos - pos;
            float sqrDstToNeighbour = dot(offsetToNeighbour, offsetToNeighbour);
            if (sqrDstToNeighbour > sqrRadius) continue;
            float densityNeighbour = densities[neighbourIndex];
            float neighbourPressure = pressureFromDensity(densityNeighbour);
            float sharedPressure = (pressure + neighbourPressure) / 2;
            float dst = sqrt(sqrDstToNeighbour);
```

Annexe

Code - spatialHash.comp

```
const int hashK1 = 15823;
const int hashK2 = 9737333;
const int hashK3 = 440817757;

ivec3 GetCell3D(vec3 position, float radius) {
    return ivec3(position.x / radius, position.y / radius, position.z / radius);
}

int HashCell3D(ivec3 cell) {
    return cell.x * hashK1 + cell.y * hashK2 + cell.z * hashK3;
}

uint KeyFromHash(uint hash, int tableSize) {
    return hash % tableSize;
}

layout(local_size_x = 64, local_size_y = 1, local_size_z = 1) in;
void main() {
    uint index = gl_GlobalInvocationID.x;
    if (index >= numParticles) return;
    spatialOffsets[index] = numParticles;
    ivec3 cell = GetCell3D(predictedPositions[index], smoothingRadius);
    uint hash = HashCell3D(cell);
    uint key = KeyFromHash(hash, numParticles);
    spatialIndices[index] = uvec3(index, hash, key);
}
```

Annexe

Code - updatePos.comp

```
void resolveCollisions(uint index) {
    vec3 center = (sim_corner_1 + sim_corner_2) * 0.5;
    vec3 size = vec3(abs(sim_corner_2.x - sim_corner_1.x),abs(sim_corner_2.y - sim_corner_1.y),abs(sim_co

    if (abs(center.x - positions[index].x) > size.x) {
        positions[index].x = center.x + sign(velocities[index].x) *( size.x - 0.001);
        velocities[index].x *= -1 * collisionDampingFactor;
    };
    if (abs(center.y - positions[index].y) > size.y) {
        positions[index].y = center.y + sign(velocities[index].y) *(size.y - 0.001) ;
        velocities[index].y *= -1 * collisionDampingFactor;
    };
    if (abs(center.z - positions[index].z) > size.z) {
        positions[index].z = center.z + sign(velocities[index].z) *(size.z - 0.001);
        velocities[index].z *= -1 * collisionDampingFactor;
    }
}


void UpdatePositions(uint id)
{
    if (id >= numParticles) return;

    positions[id] += velocities[id]* 1/60;
    resolveCollisions(id);
}
```

Annexe

Code - updatePos.comp

```
void main() {
    uint id = gl_GlobalInvocationID.x;
    if (id >= numParticles) return;
    vec3 pos = predictedPositions[id];
    ivec3 originCell = GetCell3D(pos, smoothingRadius);
    float sqrRadius = smoothingRadius * smoothingRadius;
    vec3 viscosityForce = vec3(0);
    vec3 velocity = velocities[id];
    for (int i = 0; i < 27; i ++) {
        uint hash = HashCell3D(originCell + offsets3D[i]);
        uint key = KeyFromHash(hash, numParticles);
        uint currIndex = spatialOffsets[key];
        while (currIndex < numParticles) {
            uvec3 indexData = spatialIndices[currIndex];
            currIndex ++;
            if (indexData[2] != key) break;
            if (indexData[1] != hash) continue;
            uint neighbourIndex = indexData[0];
            if (neighbourIndex == id) continue;
            vec3 neighbourPos = predictedPositions[neighbourIndex];
            vec3 offsetToNeighbour = neighbourPos - pos;
            float sqrDstToNeighbour = dot(offsetToNeighbour, offsetToNeighbour);
            if (sqrDstToNeighbour > sqrRadius) continue;
            float dst = sqrt(sqrDstToNeighbour);
            vec3 neighbourVelocity = velocities[neighbourIndex];
            viscosityForce += (neighbourVelocity - velocity) * SmoothingKernelPoly6(dst, smoothingRadius)
        }}
    velocities[id] += viscosityForce * viscosityStrength * delta;
}
```