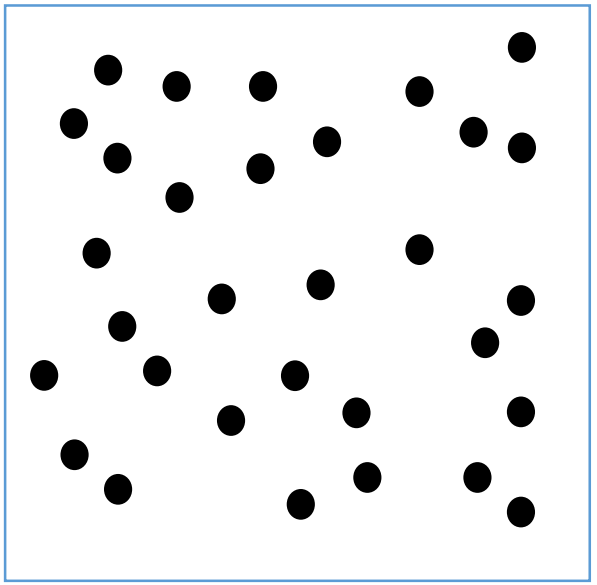
# Problemstellung

- N-Körper Problem
  - N Sterne mit bestimmter Masse
- Ziel: Berechnung der Gravitation zwischen jedem dieser Sterne
- Newtonsches Gravitationsgesetz:

$$\overrightarrow{F_1} = Gm_1m_2 \frac{\overrightarrow{r_2} - \overrightarrow{r_1}}{|\overrightarrow{r_2} - \overrightarrow{r_1}|^3}$$



# Problemstellung

- Kräfte $u_i$ müssen für jeden Stern $x_i$ berechnet werden:

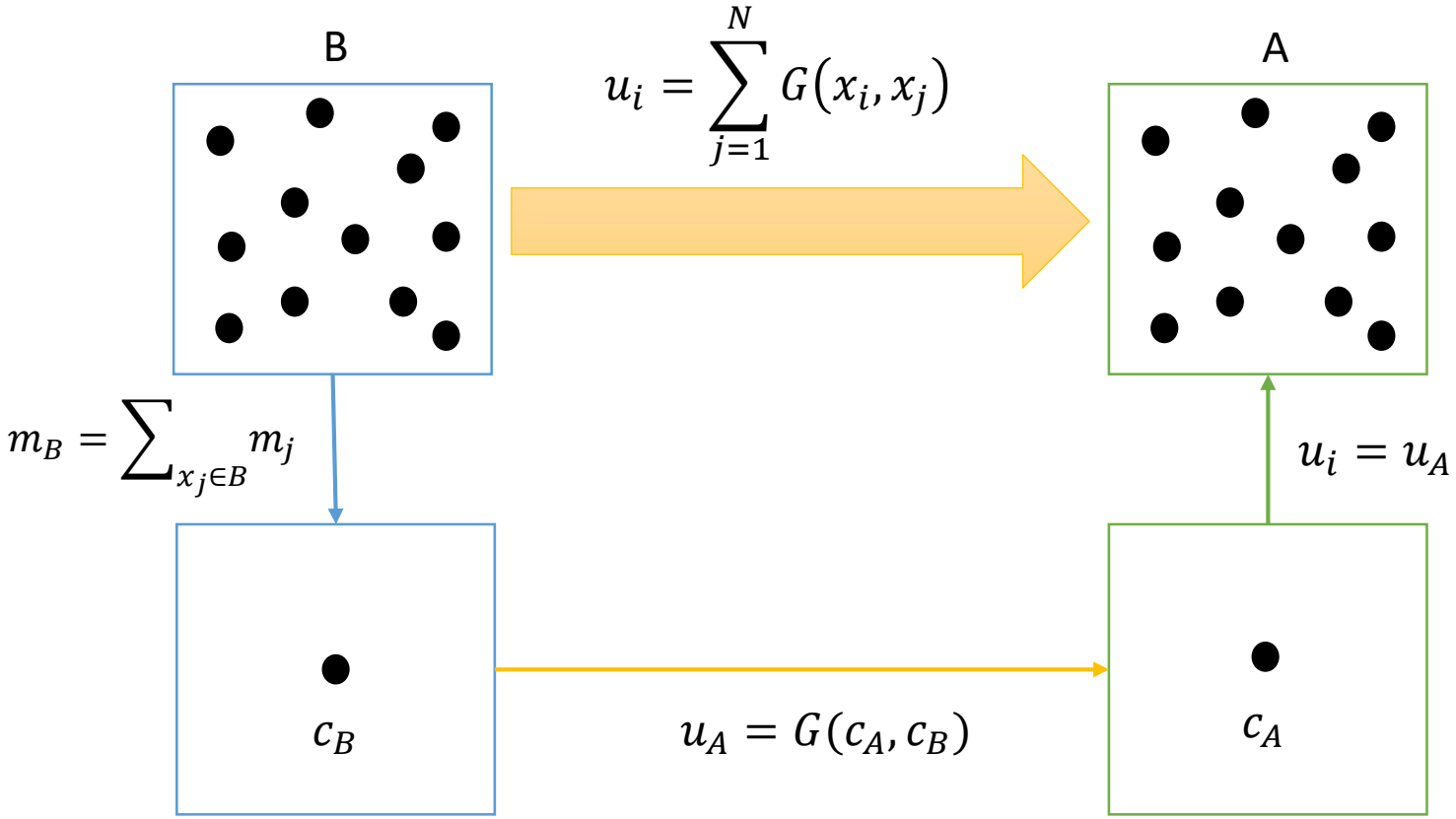
$$u_i = \sum_{j=1}^{N} G(x_i, x_j)$$

- Matrix-Vektor Multiplikation
- Komplexität der naïven Berechnung: $O(n^2)$

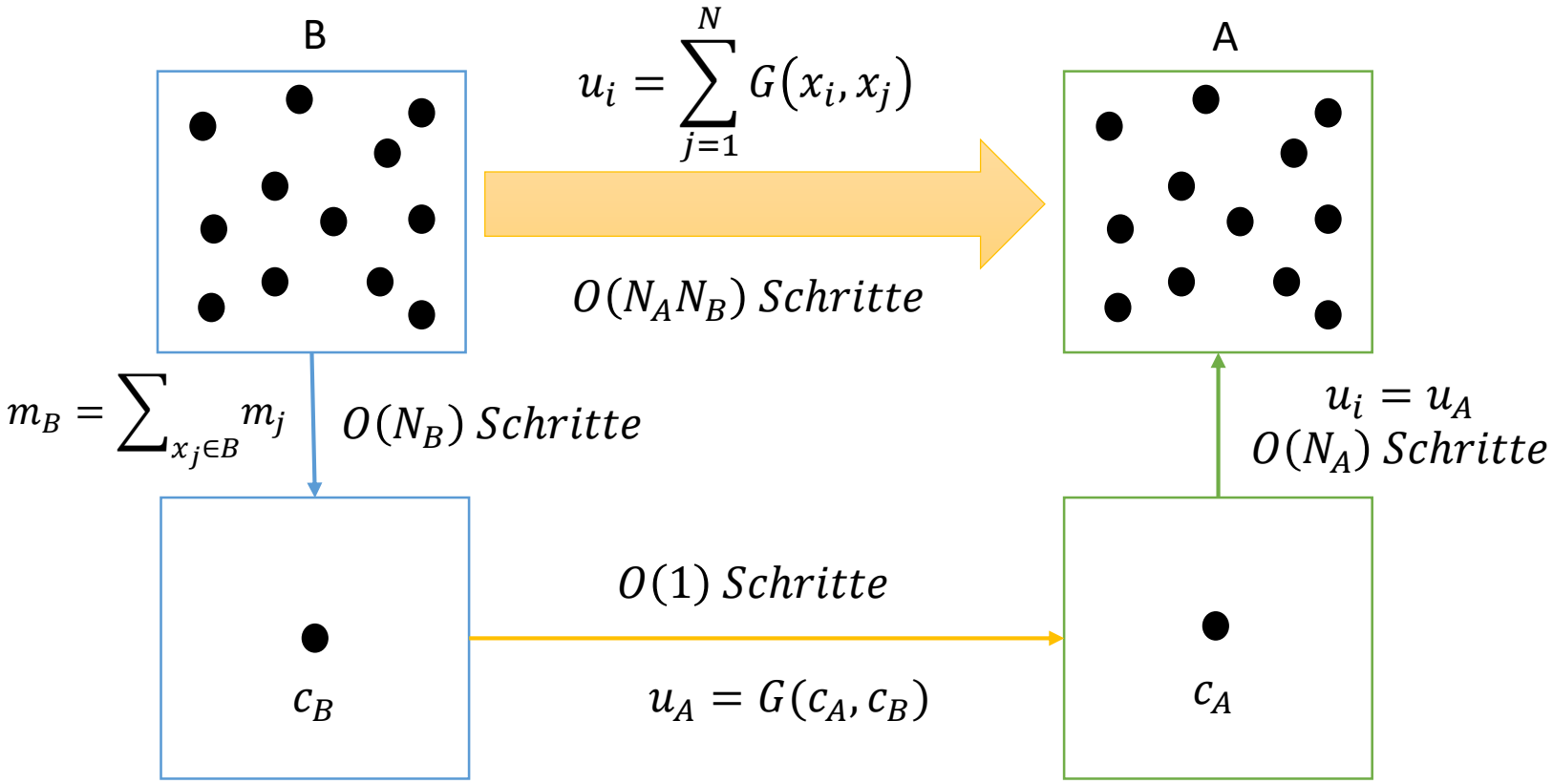
- Ziel: niedrigere Komplexität

→Fast Multipole Method

# Idee 1: well-separated regions



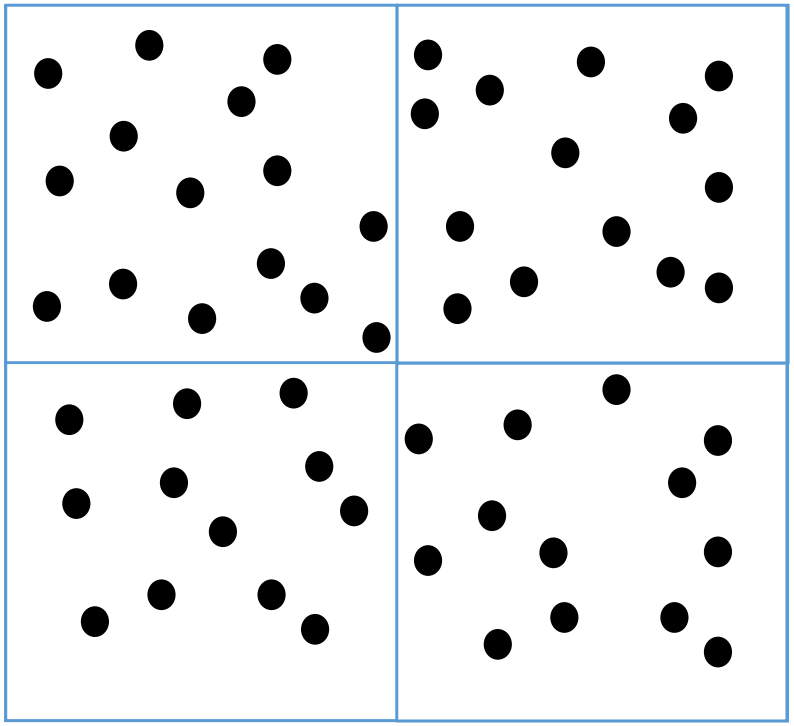
# Idee 1: well-separated regions




→ Von $O(N_A N_B)$ auf $O(N_A + N_B)$ reduziert

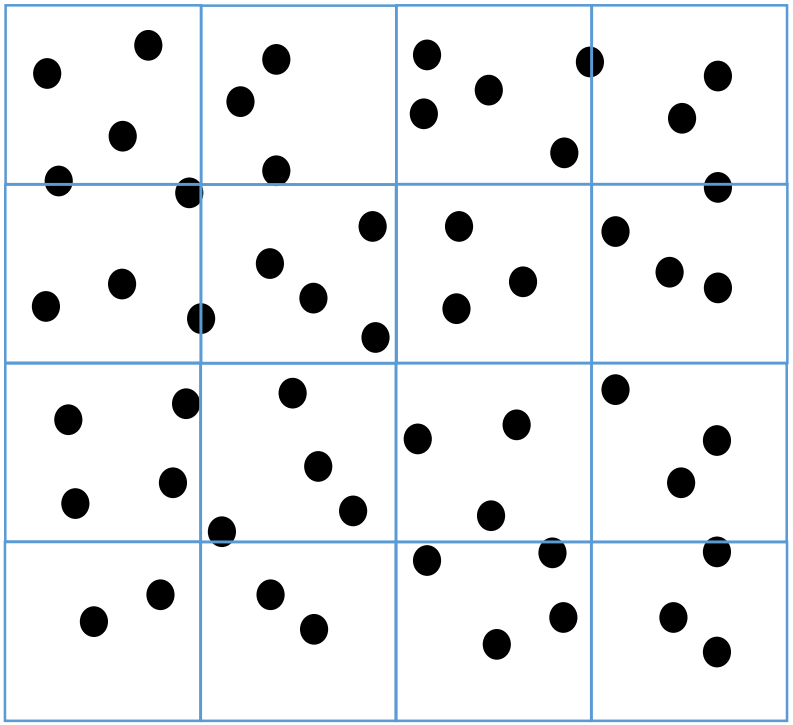
# Idee 2: Hierarchical Decomposition

- Der Wertebereich wird hierarchisch unterteilt
- Bis jede Leaf Box eine konstante Anzahl von Punkten enthält

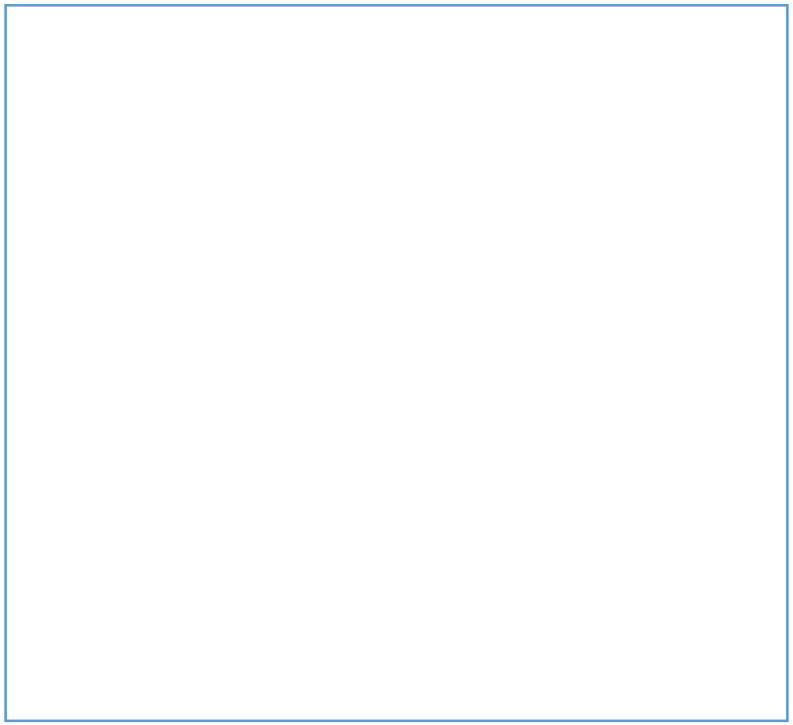


# Idee 2: Hierarchical Decomposition

- Der Wertebereich wird hierarchisch unterteilt
- Bis jede Leaf Box eine konstante Anzahl von Punkten enthält

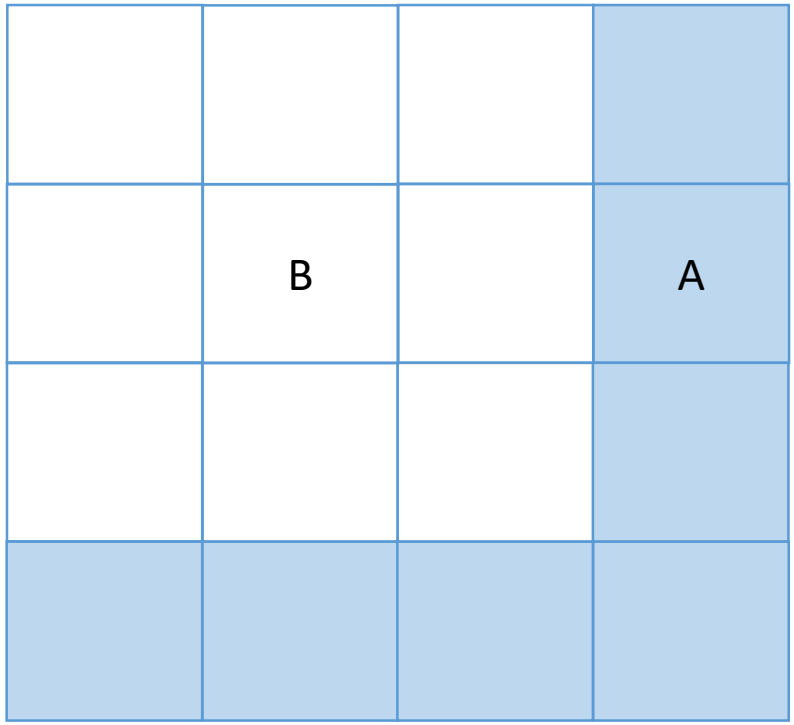


- Level 0 und 1



- Alle Boxen sind benachbart
- 3-Schritt Approximation kann nicht angewendet werden

# Level 2

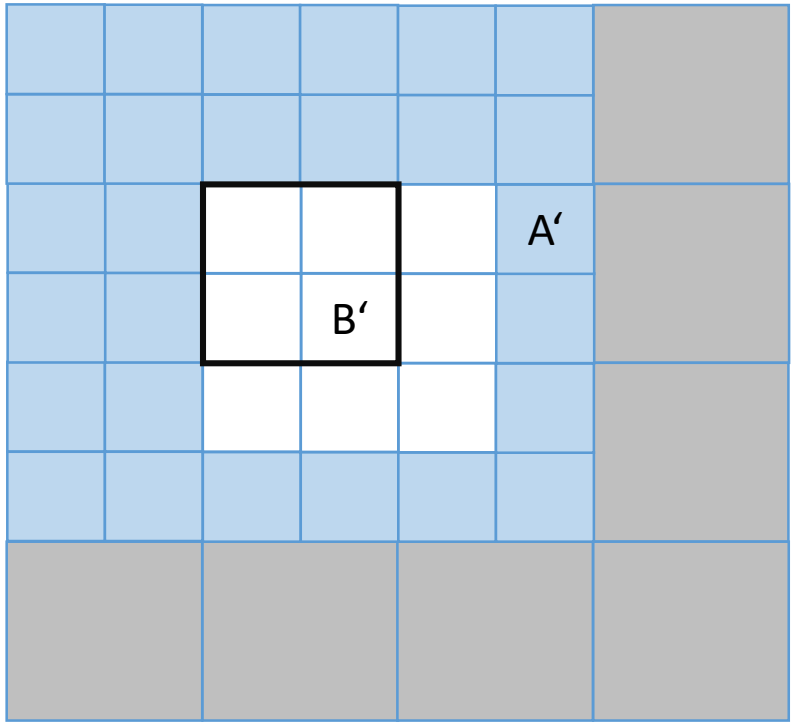



- well-separated

- near

- B und A sind well-separated
- 3-Schritt Approximation für den Einfluss von B auf A

# Level 3



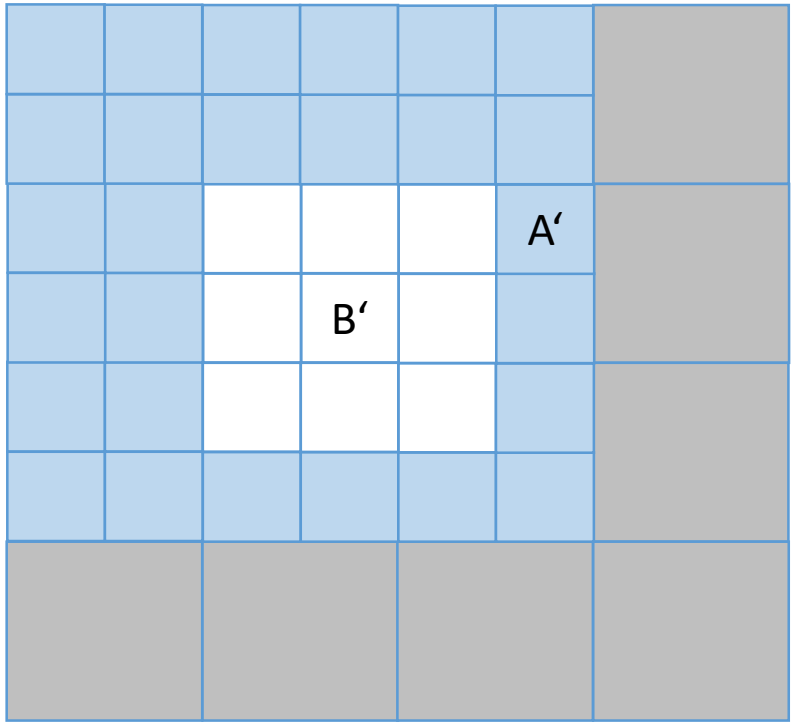

- well-separated

- near

- done

- B‘ und A‘ sind well-separated
- 3-Schritt Approximation für den Einfluss von B‘ auf A‘
- Interaktionsliste von B‘ = alle Boxen wie A‘ = 27 Boxen

# Leaf Level




- well-separated

- near

- done

- Level 3 sei das Leaf Level
- Level kann nicht erhöht werden

Berechne Einfluss der Nachbarn direkt

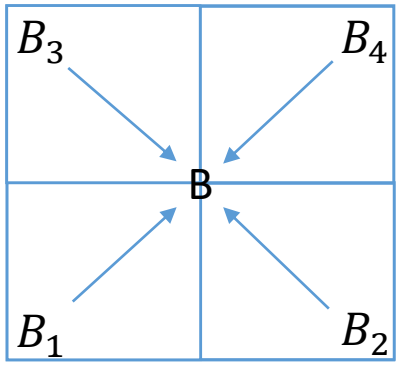
# Algorithmus 1

**for** *each box B in the tree* **do**
  $f_b \Leftarrow \sum_{x_j \in B} f_j$
**end**
**for** *L==2 to last level* **do**
  **for** *each B on level L* **do**
    **for** *each A in B's interaction list* **do**
      $u_A \Leftarrow u_A + G(c_A, c_B)$
    **end**
  **end**
**end**
**for** *each box A in the tree* **do**
  $u_i \Leftarrow u_i + u_{A'}$ for $x_i \in A$
**end**
**for** *each box B in the last level* **do**
  $u_i \Leftarrow u_i + \sum_{x_j \in Nbhd(B)} G(x_i, x_j)$, for $x_i \in B$
**end**

# Algorithmus 1: Kosten

**for** *each box B in the tree* **do**
  $f_b \Leftarrow \sum_{x_j \in B} f_j$ — O(N logN)
**end**
**for** *L==2 to last level* **do**
  **for** *each B on level L* **do**
    **for** *each A in B's interaction list* **do**
      $u_A \Leftarrow u_A + G(c_A, c_B)$ — O(N)
    **end**
  **end**
**end**
**for** *each box A in the tree* **do**
  $u_i \Leftarrow u_i + u_{A'}$ for $x_i \in A$ — O(N logN)
**end**
**for** *each box B in the last level* **do**
  $u_i \Leftarrow u_i + \sum_{x_j \in Nbhd(B)} G(x_i, x_j)$, for $x_i \in B$ — O(N)
**end**

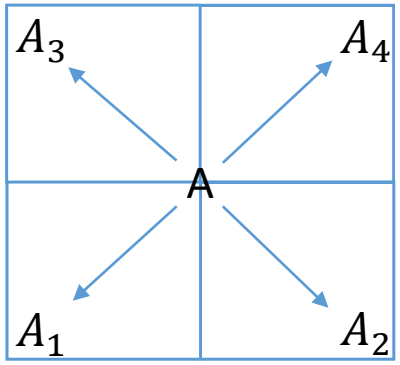
# Idee 3: Berechnungen wiederverwenden




$$f_B = f_{B_1} + f_{B_2} + f_{B_3} + f_{B_4}$$

- $f_B$ wird aus seinen Kindern berechnet

- Kosten $O(1)$
- Quadtree wird bottom-up traversiert




$$u_i := u_i + u_{A'} \quad \text{für } x_i \in A$$

- Analog wird nur den entsprechenden Kindern hinzugefügt

- Kosten $O(1)$
- Quadtree wird top-down traversiert

# Algorithmus 2

go up the tree, **for** *each box* $B$ **do**
  **if** $B$ *is a leaf box* **then**
    $f_b \Leftarrow \sum_{x_j \in B} f_j$
  **else**
    $f_b \Leftarrow f_{B_1} + f_{B_2} + f_{B_3} + f_{B_4}$
**end**
**for** $L$==*2 to last level* **do**
  **for** *each* $B$ *on level* $L$ **do**
    **for** *each* $A$ *in* $B$*'s interaction list* **do**
      $u_A \Leftarrow u_A + G(c_A, c_B)$
    **end**
  **end**
**end**
go down the tree, **for** *each box* $A$ **do**
  **if** $A$ *is a leaf box* **then**
    $u_i \Leftarrow u_i + u_{A'}$ for $x_i \in A$
  **else**
    $u_{A_1} \Leftarrow u_{A_1} + u_{A'}$ same for $A_2, A_3, A_4$
**end**
**for** *each box* $B$ *in the last level* **do**
  $u_i \Leftarrow u_i + \sum_{x_j \in Nbhd(B)} G(x_i, x_j)$, for $x_i \in B$
**end**

# Algorithmus 2: Kosten

go up the tree, **for** *each box* $B$ **do**
- **if** $B$ *is a leaf box* **then**
  - $f_b \Leftarrow \sum_{x_j \in B} f_j$ — $O(N)$
- **else**
  - $f_b \Leftarrow f_{B_1} + f_{B_2} + f_{B_3} + f_{B_4}$ — $O(N)$

**end**

**for** $L$==2 *to last level* **do**
- **for** *each* $B$ *on level* $L$ **do**
  - **for** *each* $A$ *in* $B$*'s interaction list* **do**
    - $u_A \Leftarrow u_A + G(c_A, c_B)$ — $O(N)$
  - **end**
- **end**

**end**

go down the tree, **for** *each box* $A$ **do**
- **if** $A$ *is a leaf box* **then**
  - $u_i \Leftarrow u_i + u_{A'}$ for $x_i \in A$ — $O(N)$
- **else**
  - $u_{A_1} \Leftarrow u_{A_1} + u_{A'}$ same for $A_2, A_3, A_4$ — $O(N)$

**end**

**for** *each box* $B$ *in the last level* **do**
- $u_i \Leftarrow u_i + \sum_{x_j \in Nbhd(B)} G(x_i, x_j)$, for $x_i \in B$ — $O(N)$

**end**

# Implementierung

- Grundlage: Quadtree
- Weniger „leere“ Blätter
- Effizienter als uniformes Raster
- Problem: Einige Operationen sind deutlich komplizierter zu implementieren
- Sterne und deren Eigenschaften werden zufällig generiert
- GUI für visuelle Repräsentation und Einstellungen

# Implementierung

Naive Berechnung:

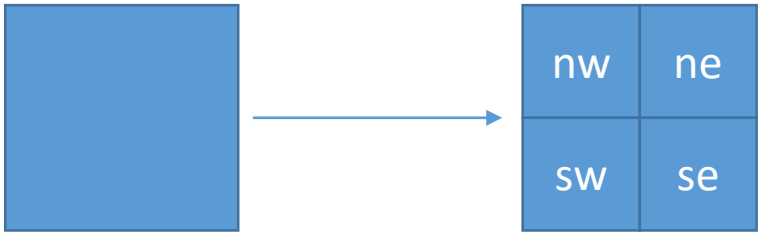
```
NaiveCalculation::NaiveCalculation(std::vector<Star*> &starList)
{
    for (Star* s0 : starList)
    {
        for (Star* s1 : starList)
        {
            if (s0 != s1)
                s0->u = s0->u + Force::G(s1, s0);
        }
    }
}
```

# Implementierung

1) Alle Sterne in den Quadtree einfügen

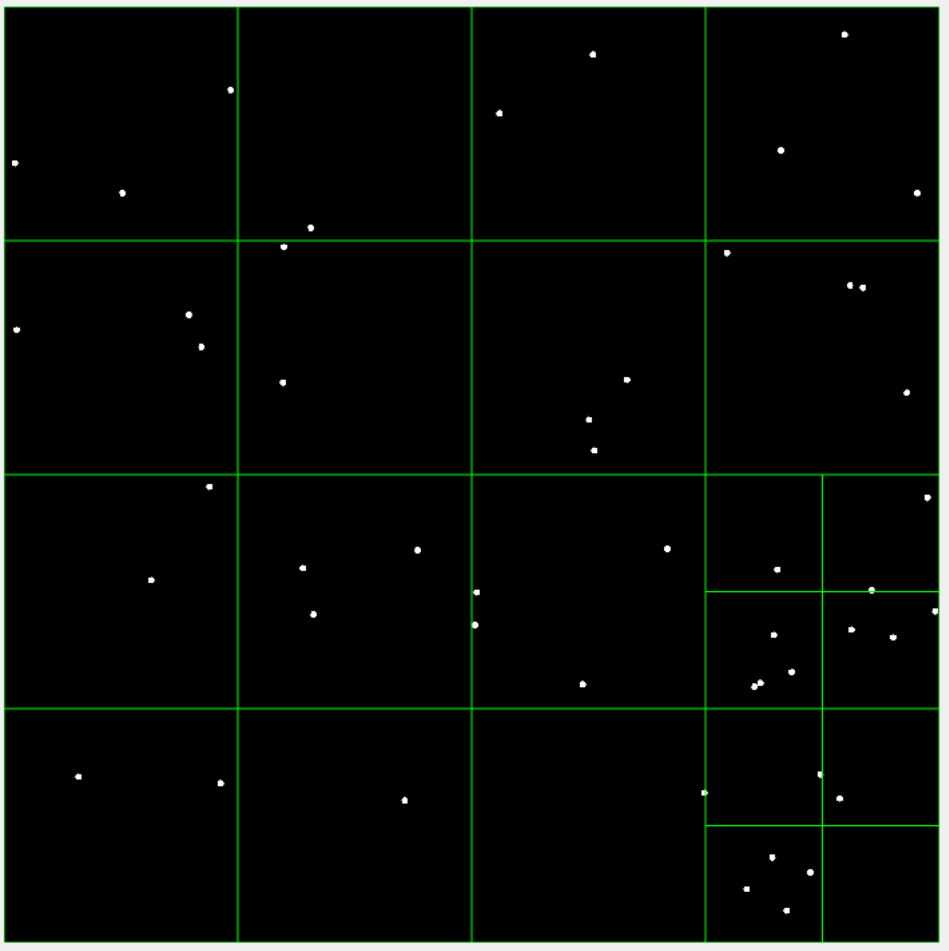
- Sterne werden in den Blättern abgelegt
- Falls diese voll sind (hier: 4 Objekte)

  → Subdivide:




- Sterne aus der oberen Ebene werden in die neuen Blätter einsortiert

# Implementierung



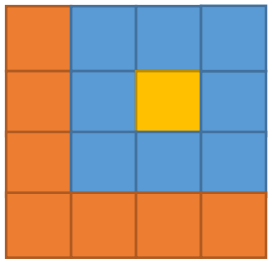
→ Weniger Blätter in dünn besiedelten Regionen

# Implementierung

2) Berechnung der „interaction list“

(Alle Kindknoten der Nachbarknoten der Elternknoten, die nicht angrenzen)

- Problem: Nachbarknoten könnte in ganz anderem Zweig liegen
  - Zur Berechnung müsste durch den ganzen Baum iteriert werden



→ Erste Iteration durch den Baum (1. Schleife des Algorithmus) benutzen um Nachbarn zu speichern

# Implementierung

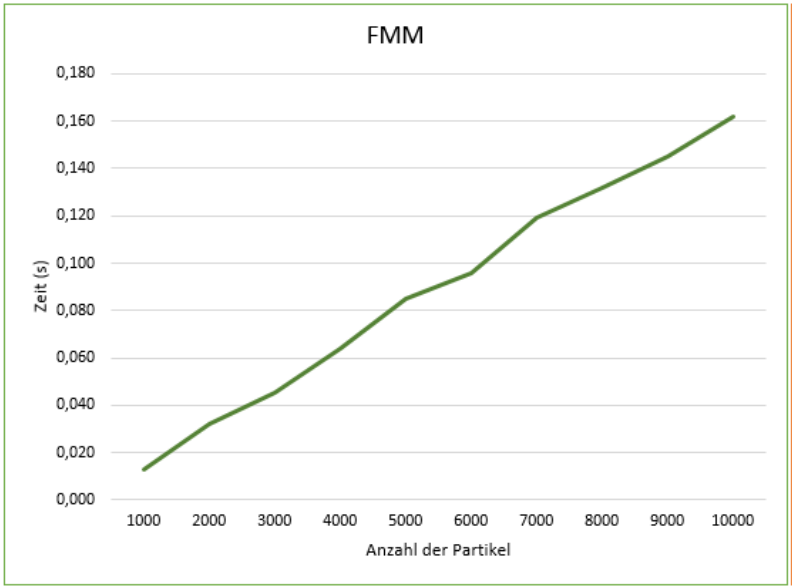
Nachbarn finden

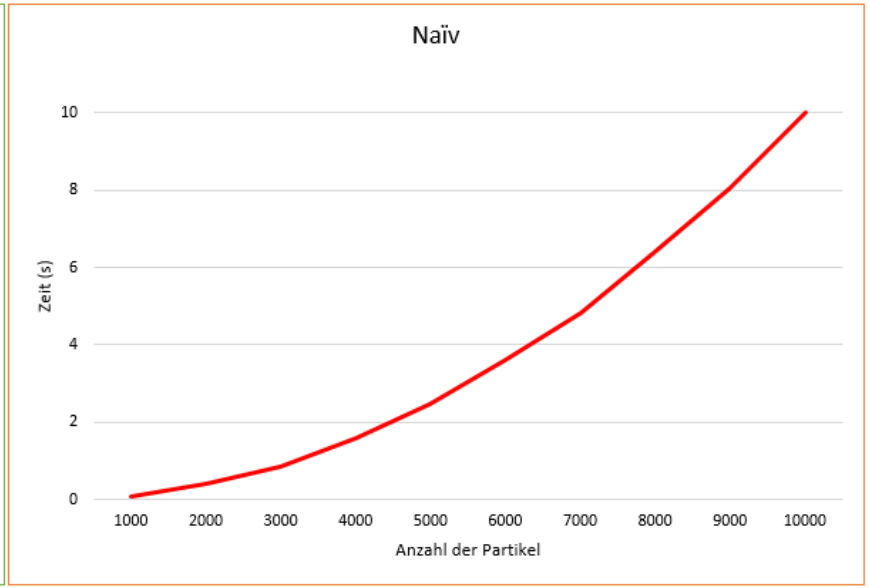
- Für jeden Knoten
  - Falls Knoten = Wurzelknoten: tue nichts (keine Nachbarn)
  - Falls Knoten auf 1. Ebene: Alle Geschwister sind Nachbarn
  - Sonst:

```
if (this->parent->getNeighbors().size() > 0)
   {
      for (Quadtree* n : this->parent->getNeighbors())
      {
           if (this->getBoundary()->adjacent(n->nw->getBoundary()))
              this->neighbors.push_back(n->nw);
           […]
```

# Live-Demo

# Performance




- Laufzeit reduziert um den Faktor N

# Fazit & Ausblick

- Berechnung noch fehlerbehaftet
- Laufzeit allerdings repräsentativ
- Einige C++11 Features genutzt

- Shared Pointer nutzen
- Compare-Funktion
- Animation