Ameisenfutter

Kevin Schier

29. November 2015

1 Lösungsidee

Die Lösungsidee lässt sich hier erstaunlich kurz fassen. Es ist die Simulation einer Ameisenkolonie in einem Raster gefragt, welche zu jedem Zeitschritt einige erlaubte Möglichkeiten hat. Genau das wird eins zu eins umgesetzt.

Die Ameisen werden nicht als individuelle Objekte angesehen, sondern in den Daten der Felder selbst gespeichert. In den Feldern werden daher die Informationen gespeichert, wie viele Ameisen ohne und mit Futter enthalten sind, wie viel Futter übrig ist, wie viele Pheromonmarken übrig sind und ob das Feld ein Nest ist.

Die Simulation für jeden Zeitschritt erzeugt ein komplett neues Raster an Feldern aus dem letzten. Die Ameisen werden gleichmäßig verteilt über die Optionen, sich in einer der vier Richtungen zu bewegen oder stehen zu bleiben. Diese zufällige Verteilung lässt sich mit Hilfe der Binomialverteilung modellieren. Sollten sich aber Pheromonmarken in der Nähe der Ameisen befinden, bewegen sie sich immer zu den Marken, aber immer weg vom Nest. Sollte es dennoch mehrere mögliche Wege weg vom Nest geben, werden die Ameisen gleichmäßig verteilt. Die Ameisen bewegen sich nicht immer zu den Feldern mit der höchsten Konzentration, da die Ameisen sonst konstant ihre Futterquellen wieder verlieren und sie neu suchen müssen.

Ameisen mit Futter bewegen sich immer Richtung Nest und hinterlassen eine große Menge an Pheromonmarken auf jedem Feld, das sie überqueren. Diese große Menge ist notwendig, da es oft hunderte bis tausende Schritte dauert bis die Ameise selbst, oder eine andere die Spur findet.

Normale Ameisen auf Feldern mit Futter werden in Ameisen mit Futter umgewandelt, ja nach dem wie viel Futter noch auf dem Feld übrig ist. Ameisen mit Futter auf einem Nestfeld werden wieder in normale Ameisen umgewandelt.

Jede einzelne Pheromonmarke hat jeden Zeitschritt eine gewisse Chance zu verschwinden. Die Anzahl an übrigen Pheromonmarken auf jedem Feld lässt sich somit wieder mit einer Binomialverteilung modellieren.

2 Umsetzung

Die Umsetzung der Simulation selbst ist exakt wie in der Lösungsidee beschrieben in der Methode void Ameisenfutter::Map::doUpdateStep() zu finden. Damit die Daten der Karte nicht verfälscht werden, hat die Klasse Map zwei Buffer: einen zum Lagern der momentanen Daten und den anderen um die neuen Daten zu speichern. Die Zeiger auf diese Buffer werden dann bei jedem Update ausgetauscht (eigentlich einfach nur double-buffering).

Die Darstellung findet (völlig unnötigerweise) mit OpenGL[1] statt. Die Vertexpositionen der Quadrate werden einmal am Anfang übertragen und bleiben dann konstant. Der Vertexshader nimmt aber noch einen Parameter an, welcher angibt wie ausgefüllt das Quadrat ist. Das sorgt dafür, dass bei näheren Zoomstufen kleine Lücken zwischen den Quadraten anzeigbar sind. Die Daten der Karte werden dabei komprimiert als Bitflags von Ganzzahlen in einem Shader Storage Buffer Object übertragen. Der Fragmentshader liest dann die Daten aus diesem Buffer aus, wobei der Index in diesem Buffer mit in den konstanten Vertexdaten enthalten ist. Alle Darstellungsrelevanten Methoden sind in der Klasse MapViewer.

In der Klasse Ameisenfutter findet die Kontrolle über die Simulation und das Eventlistening statt. Dort werden die Threads zum Updaten der Simulation selbst und zum Updaten des Bildschirms gestartet.

Die meisten Operationen werden asynchron erledigt um die Darstellung möglichst ruckelfrei zu halten.

3 Beispiele

Bei den Standardparametern, wie sie in der Aufgabe vorgegeben sind, verteilen sich die Ameisen langsam zufällig über die Karte, bis eine auf eine Futterquelle stößt. Sie nimmt ein Futterstück mit zum Nest und pendelt daraufhin zwischen Futter und Nest hin und her. Wenn eine andere Ameise auch auf die Spur stößt tut sie das selbe. Das geht dann so lange bis die Futterquelle erschöpft ist. Dann sammeln sich die Ameisen dort und warten bis die Pheromonmarken verschwinden. Die Wege zum Nest folgen immer dem gleichen Muster, es gibt aber auch nicht wirklich kürzere Wege in einem quadratische Raster.

Die Ergebnisse sind auch in den Abbildungen 1, 2 und 3 zu sehen.

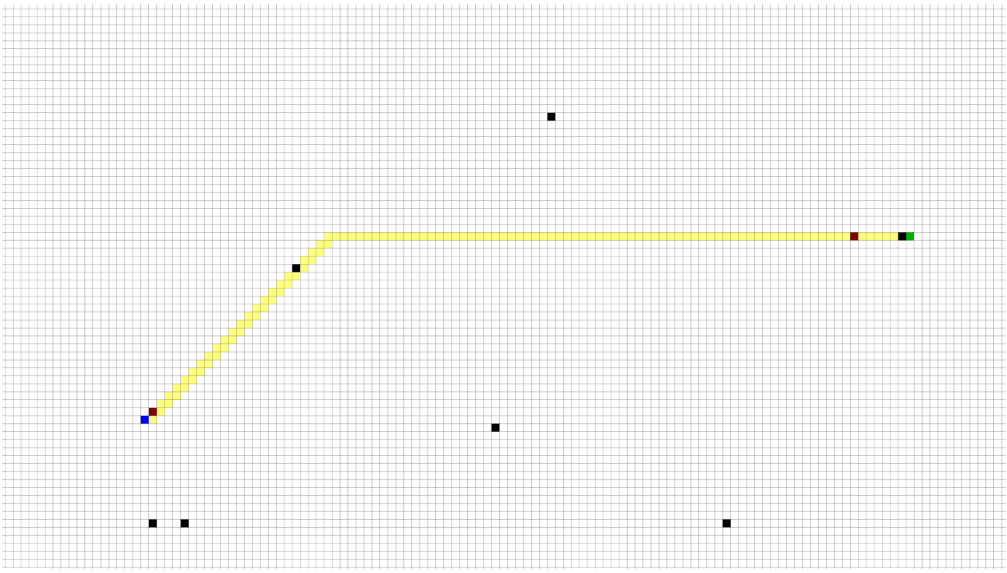

Abbildung 1: Die in der Aufgabe gegebenen Parameter: 100 Ameisen, 5 Futterquellen, 2% Verdunstungschance

[1]Die Implementierung in OpenGL war eher ein Experiment für mich selbst als alles andere, aber die Konsolendarstellung wäre vermutlich etwas groß gewesen.

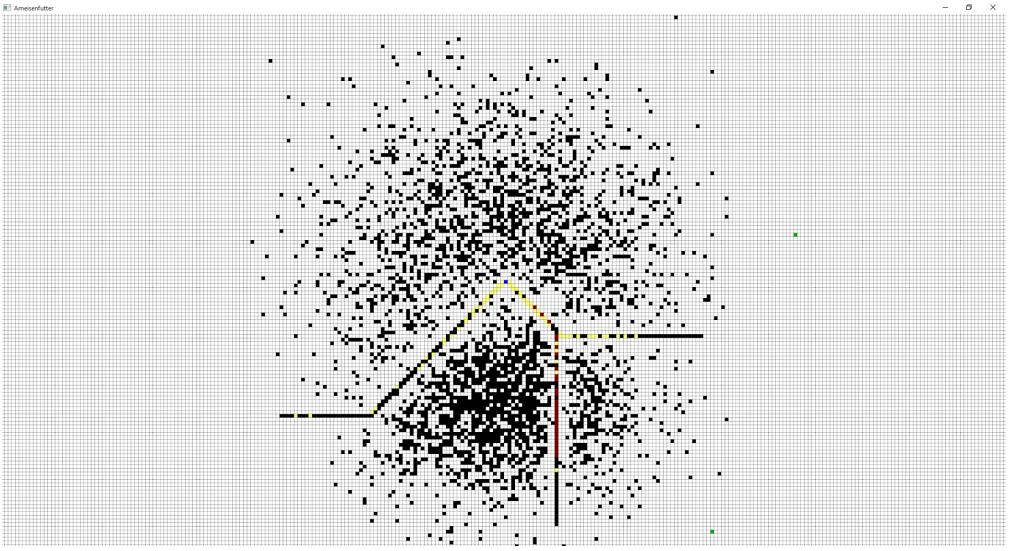

Abbildung 2: Etwas mehr: 5000 Ameisen, 50 Futterquellen, 4% Verdunstungschance

4 Quelltext

Der Code welcher die eigentliche Simulation erledigt.

```
void Ameisenfutter::Map::doUpdateStep()
{
  std::uniform_int_distribution<int> dist;
  std::binomial_distribution<int> p;
  std::vector<MapData>& newMap{ GetInactiveMapData() };
  std::binomial_distribution<int> pheromone_dist;
  ivec2 nest = { 0,0 };

  //this function makes sure there are no out of bounds errors
  MapData dummy;
  auto getNewPoint = [&](ivec2 p) -> MapData& {return in_area(p, ivec2{ 0,0 }, ivec2{ width - 1, height - 1 }) ?
      newMap[p.y * width + p.x] : dummy; };

  //This function returns the direction priorities for a given direction vector
  auto getDirectionOrder = [&](ivec2 dir)->std::array<int, 4>
  {
    std::array<int, 4> P{ NORTH, EAST, SOUTH, WEST };
    std::array<int, 2> nsP = (0 < dir.y) ? std::array<int, 2>{ SOUTH, NORTH } : std::array<int, 2>{ NORTH, SOUTH
        };
    std::array<int, 2> ewP = (0 < dir.x) ? std::array<int, 2>{ EAST, WEST } : std::array<int, 2>{ WEST, EAST };
    P = (std::abs(dir.y) < std::abs(dir.x)) ?
    std::array<int, 4>{ ewP[0], nsP[0], nsP[1], ewP[1] } :
    std::array<int, 4>{ nsP[0], ewP[0], ewP[1], nsP[1] };
    return P;
  };

  //This function calculates where ants on a tile should go
  auto getDistribution = [&](int max, std::array<int, 4> markers, ivec2 toNest)->std::array<int, 5>
  {
    std::array<int, 5> res = { 0,0,0,0,0 };
    if (max > 0)
    {
      bool markersNearby = false;
      for (bool b : markers)
      if (b) markersNearby = true;

      if (markersNearby)
      {
        //if there are markers nearby, send the ants to the tile farthest away from the nest according to the
```

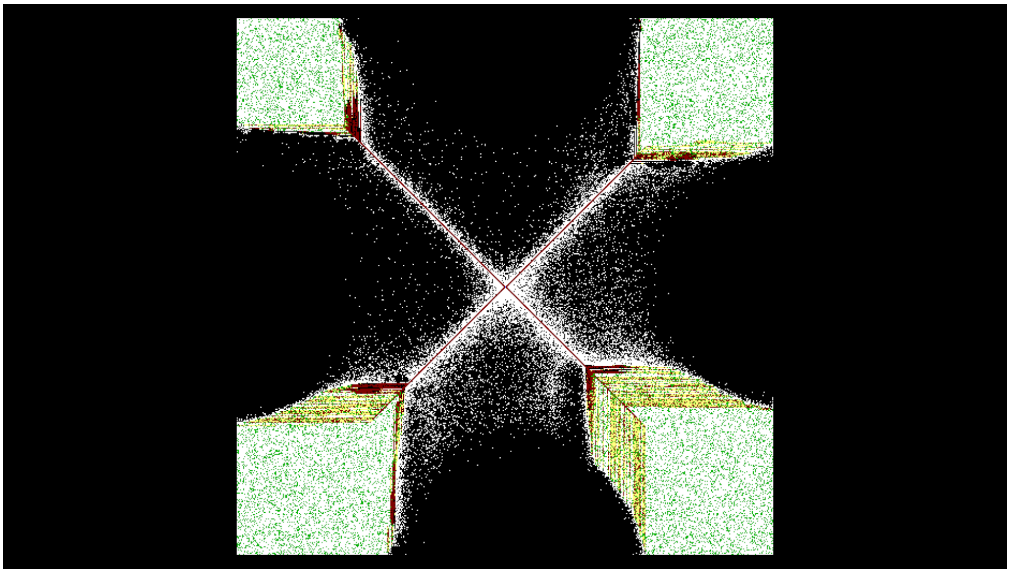

Abbildung 3: Völlig übertrieben: 10000000 Ameisen, 50000 Futterquellen, 50% Verdunstungschance (Es bilden sich allerdings interessante Muster)

```
        getDirectionOrder() function
    ivec2 dir = -toNest;
    std::array<int, 4> P = getDirectionOrder(dir);
    int mCount = std::count_if(P.begin(), P.end(), [&markers](int i) {return markers[i]; });
    if (mCount <= 2 && dir != ivec2{ 0, 0 })
    {
      for (auto i : P)
      if (markers[i])
      {
        res[i] = max;
        break;
      }
    }
    else
    {
      //if there are multiple choices, distribute them randomly
      for (auto i : P)
      if (markers[i])
      {
        if (mCount > 1)
        {
          p = std::binomial_distribution<int>{ max, mCount / (double) (mCount * 2 - 1) };
          float amount = clamp(p(rng), 0, max);
          res[i] = amount;
          max -= amount;
          --mCount;
        }
        else
        {
          res[i] = max;
          break;
        }
      }
    }
  }
  else //!markersNearby
  {
    if (max < 6) //avoid the binomial_distribution call if there only a few ants in the cell
    {
      const auto smalldist = std::uniform_int_distribution<int>{ 0,4 };
      while (max--) ++res[smalldist(rng)];
    }
    else
    {
```

```
            p = std::binomial_distribution<int>{ max, 0.2 };
            res[4] = p(rng);
            max -= res[4];
            dist = std::uniform_int_distribution<int>{ 0, max };
            int ns = dist(rng);
            int we = max - ns;
            dist = std::uniform_int_distribution<int>{ 0, ns };
            res[0] = dist(rng);
            res[1] = ns - res[0];
            dist = std::uniform_int_distribution<int>{ 0, we };
            res[2] = dist(rng);
            res[3] = we - res[2];
          }
        }
      }
      return res;
    };

    //Clear the map
    for (int i = 0; i < width*height; i++)
    {
      auto& d = GetInactiveMapData()[i];
      d.numAnts = 0;
      d.numAntsWithFood = 0;
      nest = d.isNest ? ivec2{ i % width, i / width } : nest;
      }

      //Main loop iterating over every source cell
      for (auto it : *this)
      {
        auto& data = it.second;
        auto& ndata = getNewPoint(it.first);

        ndata.isNest = data.isNest;
        ndata.numFood = data.numFood;

        //if there is food on this cell, convert ants on it
        if (data.numFood > 0)
        {
          if (data.numFood < data.numAnts)
          {
            data.numAnts -= data.numFood;
            ndata.numAntsWithFood += data.numFood;
            ndata.numFood = 0;
          }
          else
          {
            ndata.numFood -= data.numAnts;
            ndata.numAntsWithFood += data.numAnts;
            data.numAnts = 0;
          }
        }

        //update pheromone
        if (data.numPheromone > 0 || data.numAntsWithFood > 0)
        {
          if (data.numPheromone <= 1)
          data.numPheromone = 0;
          pheromone_dist = std::binomial_distribution<int>{ (int) data.numPheromone , 1.0 - evaporationChance };
          ndata.numPheromone = std::floor(pheromone_dist(rng)) + (data.numAntsWithFood << 16); //every ant leaves 2^16
              pheromones
        }

        //deliver food to nest
        if (data.isNest)
        {
          ndata.numAnts += data.numAntsWithFood;
          data.numAntsWithFood = 0;
        }

        //update positions of ants without food
        if (data.numAnts > 0)
```

```
    {
      std::array<int, 4> markers;
      for (int dir : Directions)
      {
        ivec2 p(it.first + DirectionVectorsWithNone[dir]);
        markers[dir] = (in_area(p, ivec2{ 0,0 }, ivec2{ width - 1, height - 1 })) ? (GetPoint(p).numPheromone) : 0;
      }

      auto antDistribution = getDistribution(data.numAnts, markers, nest - it.first);
      //all these calculations make sure that ants don't go out of bounds
      if (it.first.x == 0)
      antDistribution[EAST] += antDistribution[WEST];
      if (it.first.x == width - 1)
      antDistribution[WEST] += antDistribution[EAST];
      if (it.first.y == 0)
      antDistribution[SOUTH] += antDistribution[NORTH];
      if (it.first.y == height - 1)
      antDistribution[NORTH] += antDistribution[SOUTH];

      getNewPoint(it.first + DirectionVectors[NORTH]).numAnts += antDistribution[NORTH];
      getNewPoint(it.first + DirectionVectors[EAST]).numAnts += antDistribution[EAST];
      getNewPoint(it.first + DirectionVectors[SOUTH]).numAnts += antDistribution[SOUTH];
      getNewPoint(it.first + DirectionVectors[WEST]).numAnts += antDistribution[WEST];
      ndata.numAnts += antDistribution[NONE];
    }

    //update positions of ants with food
    if (data.numAntsWithFood > 0)
    {
      ivec2 toNest(nest - it.first);
      int nestDirection = std::abs(toNest.x) > std::abs(toNest.y) ? (toNest.x > 0 ? EAST : WEST) : (toNest.y > 0 ?
          SOUTH : NORTH);
      getNewPoint(it.first + DirectionVectors[nestDirection]).numAntsWithFood += data.numAntsWithFood;
    }
  }
  swapMapData();
}
```