Глава 1

Алгоритмы визуализации данных на клиенте

В данной главе описываются реализации двух алгоритмов визуализации данных: Fruchterman Reingold и t-SNE.

Несмотря на наличие множества реализаций этих алгоритмов на JavaScript, ни одна из них не использует GPU. При этом нет простого способа портировать существующие реализации этих алгоритмов на GPU с других языков программирования, так как WebGL API очень ограничен по сравнению с OpenGL и другими API для графических карт.

1.1 WebGL для обработки данных

1.2 Fruchterman Reingold

Это классический алгоритм рисования графа из категории force-directed drawing.

Он использует физическую аналогию, в которой ребрам соответствуют пружины, а вершинам — одинаково заряженные частицы. Таким образом, соединенные ребром вершины стремятся быть на фиксированном расстоянии друг от друга, равном оптимальной длине пружины, а не соединенные вершины отталкиваются друг от друга.

На практике, закон Гука для силы пружины не используется, так как он слишком сильно действует на соединенные вершины, находящиеся в разных частях графа.

Для алгоритма Fruchterman Reingold силы притяжения и отталкивания равны $f_a(d) = d^2/k$ и $f_r(d) = -k^2/d$, где k — оптимальная длина пружины, при которой эти силы сбалансированы.

Обычно, физическая симуляция осуществляется при помощи интегрирования Верле, но в этом алгоритме силы по сути являются скоростями, так как это ускоряет сходимость.

Алгоритм начинает со случайных позиций вершин и останавливается, когда система достигает равновесия. Для ускорения этого процесса вводится понятие *температуры*, которая ограничивает максимальное изменение

положения вершин и уменьшается по заданному закону.

1.3 t-SNE

Применение этого алгоритма считается одним из наиболее мощных методов уменьшения размерности до 1-3 измерений.

1.4 Схема вычислений.

Два приведенных алгоритма имеют общую структуру:

```
const edges: buffer[m] = serialize_edges()
positions: buffer[n, dim] = random_positions()
attractions, repulsions: buffer[n, dim]
grid: buffer[s]
finished: boolean = false
iteration: integer = 0

while not finished:
    iteration += 1
    attractions = calc_attractions(positions, edges)
    grid = calc_grid(positions)
    repulsions, norm_sum = calc_repulsions(positions, grid)
    positions, finished = calc_positions(
        positions,
        attractions,
        repulsions,
        iteration,
        norm_sum,
    )
```

Здесь `n` — число вершин, `m` — число ребер, `dim` — размерность пространства визуализации (от 1 до 3).

На вход алгоритму поступают два буффера: `edges` и `positions`. Позиции инициализируются случайными числами в заданных пределах. Ребра преобразуются в буффер при помощи одной из нескольких схем, перечисленных далее.

После этого следует некоторое число итераций. Для алгоритма Fruchterman Reingold номер итерации нужен для уменьшения температуры со временем, t-SNE останавливается, когда вершины пересают менять положение больше, чем на ε.

Силы притяжения действуют только между парами вершин, соединенных ребром. Таким образом, вычисление сумм этих сил для всех вершин не возможно без прохода по всем ребрам, т.е. за $O(m)$ времени и памяти. Улучшение скорости этой части возможно через уменьшение числа ребер.

Силы отталкивания действуют между всеми парами вершин, что позволяет делать некоторые оптимизации, при этом уменьшая точность этих вычислений на несколько процентов. Сложность этой части варьируется $O(n \log n)$ до $O(n^2)$ по времени и $O(n)$ по памяти.

Для алгоритма t-SNE требуется вычисление нормирующего множителя, равному сумме всех ненормированных сил притяжения между всеми парами вершин.

При использовании оптимизаций требуется дополнительная структура `grid`, которая, как правило, требует одного прохода по позициям вершин и $O(n)$ памяти.

Последняя функция `calc_positions` также проходит по всем вершинам, при этом она может содержать дополнительную логику, которая проверяет, нпример, выход вершин за границы ограничивающего прямоугольника.

Исходя из приведенных выше оценок, лучше всего для исполнения на GPU подходят функции `calc_attractions` и `calc_repulsions`.

1.5 Выбор ребер и их представление

Вычисление сил притягивания определяется выбором подмножества ребер и способом их представления в виде текстуры.

Рассмотрим три способа выбора ребер:

(a) Все ребра.

(b) Оставить не более k ребер наибольшего веса у каждой вершины.

(c) Оставить не более k ребер наибольшего веса у всего графа.

Также предлагается четыре способа представления ребер:

(p) Матрица смежности. Элементы — веса ребер.

(q) Матрица $n \times 2k$, в которой строки соответствуют вершинам, в каждой строке слева направо указан список ребер в виде чередующегося списка номеров вершин и весов ребер. k — наибольшая степень вершины.

(r) Вектор длины $n+2m$. Первые n чисел — «указатели» на индексы вектора, с которых начинается список ребер соответствующей вершины. Следующие $2m$ чисел являются объединением списков ребер, выходящих из каждой вершины.

(s) Матрица $n' \times (1+2s)$. Строки соответствуют вершинам, при этом одной вершине может соответствовать несколько подряд идущих строк. Номер вершины, соответствующей данной строке указан в первом столбце. В остальных столбцах указан список ребер и их весов. Если степень вершины больше s, она занимает несколько строк.

Способ (q) лучше всего подходит вместе со способом выбора ребер (b), иначе, если есть хотя бы одна вершина степени $n-1$, то этот способ будет хуже по времени и памяти, чем (p).

Способ (r) требует меньше всего памяти, но, в связи с особенностью работы GPU, время обработки всех вершин будет пропорционально максимальному времени обработки одной вершины, то есть максимальной степени k. Если есть хотя бы одна вершина степени $n-1$, то его время работы будет дольше, чем у способа (p).

Наконец, способ (s) требует не сильно больше памяти по сравнению с (r), при этом наиболее эффективно распараллеливается на GPU. Единственный

недостаток состоит в том, что после его применения необходимо аггрегировать результаты строк, соответствующих одной вершине.

1.6 Реализация вычисления сил притяжения

В данном разделе приводится псевдокод реализации функции `calc_attractions` для представления ребер (s). Его несложно упростить для представления (q).

Для хранения информации о ребрах используется RGBA-текстуры вместо floating point. Это позволяет хранить два 16-битных числа в одном пикселе. Первое число используется для номера вершины в списке ребер (которых будет значительно меньше 65536), а второе — для веса ребра. Такое представление уменьшит точность хранения весов ребер, но, с учетом того, что веса ребер входят в формулы линейно и не связаны уравнениями с другими переменными, это не должно значительно ухудшить качество визуализации.

Значение 0 для веса ребер означает отсутствие ребра. Это может потребоваться, чтобы заполнить оставшиеся ячейки в строках представления (s). Первая колонка матрицы, кодирующая номер вершины, соответствующей данной строке, использует только первое 16-битное число.

Результатом работы шейдера будут являться dim вещественных текстур $1 \times n$, содержащих вектора сил, действующих на вершины от ребер по строкам.

На вход шейдер получает текстуру `edges` размера $n' \times (1 + s)$ и текстуру `positions` размера $n \times dim$. Для каждой размерности `dim` код компилируется отдельно.

Псевдокод шейдера для $dim = 2$:

```
const edges_chunks, s: integer;
const max_weight: float;

int high(rgba pixel):
    return pixel.r * 256 + pixel.g

int low(rgba pixel):
    return pixel.b * 256 + pixel.a

calc_attractions(chunk_no):
    int source = high(edges[chunk_no, 0])
    float x = positions[source, 0]
    float y = positions[source, 1]
    float x_sum = 0
    float y_sum = 0
    for i = 1 ... s:
        rgba edge_pixel = edges[chunk_no, i]
        float weight = low(edge_pixel) / 256 / 256 * max_weight
        if (weight == 0) break
        int target = high(edge_pixel)
        float dx = positions[target, 0] - x
```

```
        float dy = positions[target, 1] - y
        float sqdist = dx ^ 2 + dy ^ 2
        float mul = calc_attr_mul(sqdist, weight)
        x_sum += mul * dx
        y_sum += mul * dy
    return x_sum, y_sum
```

Функция `calc_attr_mul` зависит от реализуемого алгоритма:

```
calc_attr_mul_fuchterman_reingold(float sqdist, float weight):
    return weight * sqdist / k

calc_attr_mul_tsne(float sqdist, float weight):
    return weight / (sqdist + 1)
```

1.7 Подходы к оптимизации вычисления сил отталкивания

В обоих алгоритмах «узким местом» по времени выполнения является вычисление сил отталкивания между всеми парами вершин. В данной работе рассматриваются три подхода к этому:

(k) Вычислять отталкивание между всеми парами вершин.

(l) Вычислять отталкивание только между близкими вершинами.

(m) Группировать вершины при вычислении отталкивания на большом расстоянии.

Подход (l) рекомендуется создателями Fruchterman Reingold, подход (m) испоьзуется во многих реализациях подобных алгоритмов визуализации данных.

В связи с ограничениями шейдеров WebGL, в данной работе в методе (m) может вычисляться только силы между двумя вершинами или между вершиной и группой вершин, а не между двумя группами вершин.

Чтобы применить оптимизации (l) и (m), необходимо иметь дополнительную структуру, позволяющую получать группы близких вершин, желательно на разных масштабах.

Существует множество подобных структур, мы рассмотрим только две из них:

(x) Прямоугольная 1,2,3-мерная сетка.

(y) Двоичное/квадро-/окто-дерево.

Сочетание (m)(y) называется Barnes-Hut simulation.

1.8 Представление иерархии клеток в виде текстуры

Предлагаемый способ обладает следущими характеристиками:

- Вычисление на CPU за линейное от числа вершин время и потребление памяти.
- Поддержка как прямогольних сеток, так и различных древовидных иерархий клеток.
- Клетки могут иметь любую форму.
- Поддержка 1, 2 и 3 измерений.
- Возможность пропуска дочерних клеток, в которых нет вершин.
- Возможность получить список вершин для клеток любого уровня.
- Центр клетки может совпадать с центром ограничивающего прямоугольника (отрезка, параллелепипеда), либо может совпадать с центром масс входящих в него точек.

Также для этой иерархии требуется дополнительная текстура, содержащая отсортированный список номеров вершин. Номера должны быть отсортированы так, чтобы множество номеров внутри клетки любого уровня шло подряд в отсортированном списке. Это можно получить при помощи обхода дерева.

В основной текстуре строки взаимно однозначно соответствуют клеткам. При этом строки отсортированы по такому же принципу, что и дополнительная структура, т.е. все потомки любой клетки должны идти подряд в основной текстуре сразу после строки, соответствующей родительской клетке. Эти свойства выполняются при обходе дерева.

Как основная, так и дополнительная текстуры имеют формат RGBA и каждый пиксель используется как пара 16-битных целых чисел. Эти два числа будут обозначаться суффиксами L и H.

Для указания координат и размеров используются 16-битные числа. Это не должно значительно повлиять на качество работы алгоритма т.к. такими числами кодируются лишь центры и размеры клеток, а координаты вершин и их силы по-прежнему представлены 32-битными вещественными числами. Если клетки делятся степенями двоек, как в Barnes-Hut, то 16-битные числа будут точно приближать центры и стороны клеток.

Вместе с текстурой должна передаваться константа, отражающая масштабный коэффициент для 16-битных чисел. Ее рекомендуется подбирать так, чтобы значение 65535 соответствовало максимуму из ширины и высоты ограничивающего прямоугольника.

В основной текстуре столбцы устроены следующим образом:

1H Размер клетки. Если клетка не квадратная, это значение должно равняться максимуму из ширины и высоты клетки.

1L Номер следущей клетки, которая не пересекается с текущей и имеет такой же уровень в иерархии клеток.

2, 3 Координаты центра клетки ($x \to 2H, y \to 2L, z \to 3H$) и количество вершин в ней (3L). В случае менее трех измерений неиспользуемые координаты равны нулю.

4H Индекс первой вершины данной клетки в дополнительной текстуре.

4L Индекс последней вершины данной клетки в дополнительной текстуре.

1.9 Вычисление сил отталкивания при помощи иерархии клеток