Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Южно-Уральский государственный университет
(национальный исследовательский университет)»
Институт естественных и точных наук
Факультет математики, механики и компьютерных технологий
Кафедра прикладной математики и программирования
Направление подготовки Прикладная математика и информатика

РАБОТА ПРОВЕРЕНА
Рецензент,
_____
«____»_____2018г.

ДОПУСТИТЬ К ЗАЩИТЕ
Заведующий кафедрой, д.ф.-м.н.,
доцент
_____А.А.Замышляева
«____»_____2018 г.

Разработка модуля «Построение сглаженных динамических поверхностей»
Математики, механики и компьютерных технологий.

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ
ЮУрГУ–01.03.02.2018.049.ПЗ ВКР

Руководитель работы, старший преподаватель
_____/М.Ю. Сартасова
«____»_____2018 г.

Автор работы
Студент группы ЕТ-412
_____/ А.Ю. Кислицын
«____»_____2018 г.

Нормоконтролер, доцент
_____/Д.А. Дроздин
«____»_____2018 г.

Челябинск 2018

АННОТАЦИЯ

Кислицын А.Ю. Построение сглаженных динамических поверхностей.– Челябинск: ЮУрГУ, ЕТ-412, 45 с., 29 ил., библиогр. список –14 наим..

Данная работа посвящена исследованию и построению гладких поверхностей. В работе выполнен обзор наиболее используемых вариантов построения сглаженных поверхностей в сфере компьютерной графике.

Разработана математическая модель и алгоритм работы приложения.

Программа реализована на языке программирования Java на базе Android Studio

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Область применения трехмерной графики необычайно широка:

Кинематография и мультипликация – создание необычного персонажа без грима и костюма, а также сделать объёмного нарисованного героя.

Реклама – очень часто встречаются у больших компаний свой талисман, например, у рекламы M&M, олицетворяющий саму компанию и их продукцию. Помимо этого, 3D графика помогает в самой рекламе сделать продукт интереснее, придав ему красивые цвета и оболочку.

Дизайн – очень часто применяются модели при создании макета мебели. Визуализировать будущий дом, квартиру, сад.

Игровая индустрия – самая популярная отрасль для трёхмерного моделирования. В этот список входят создание персонажей, предметов, ландшафт.

Сглаживание поверхности – одна из основных задач компьютерной графики, результат которой применяется для создание таких объектов.

1 МЕТОДЫ И СПОСОБЫ РЕШЕНИЯ ЗАДАЧИ СГЛАЖИВАНИЯ ПОВЕРХНОСТЕЙ

1.1 Полигональная сетка

Полигональная сетка представляет из себя совокупность вершин, ребер и многоугольников.

Рассмотрим три способа описания полигональных сеток.

1.1.1 Явное задание многоугольников

Каждый многоугольник возможно представить в виде списка координат его вершин:

$$P = ((x_1, y_1, z_1), (x_2, y_2, z_2), \dots , (x_n, y_n, z_n)) \quad (1)$$

Вершины находятся в том порядке, в котором они встречаются при обходе. При этом все вершины многоугольника соединяются в рёбра. Для каждого многоугольника данная условие имеет положительный результат, но при задании полигональной сетки происходит потеря памяти, из-за дубликатов соседних вершин

Изображение сетки средством очерчивания ребер каждой геометрической фигуры, но это может привести отрисовки общих рёбер фигур. Рендер будет по каждой фигуре дважды.

1.1.2 Задание многоугольников с помощью указателей в список вершин

Во время использовании этого метода узел полигональной сетки сохраняется всего один раз в списке вершин $V = ((x_1, y_1, z_1), \dots , (x_n, y_n, z_n))$. Многоугольник определяется списком индексов в массиве вершин. Многоугольник с вершинами 3, 5, 7 и 10 этого списка определяется как $P = (3, 5, 7, 10)$. [1]

Это представление имеет преимущество над обычным заданием многоугольника так как запоминается вершина один раз, то сохраняется большая часть памяти. Так же легко меняются координаты вершин. Но проблема сохранилась при отыскивании общих рёбер. При рендеринге рёбра всё ещё отрисовываются дважды. Эти два недостатка решаются, если описывать методом, который приведён ниже.

1.1.3 Явное задание ребер

Тот же список вершин, но теперь плоскость будет указывать на списки элементов рёбер, где как раз рёбра встречаются всего один раз. Каждое ребро в списке будет указывать на вершину, определяющие это ребро и на многоугольник, которому принадлежит эта вершина. Зададим многоугольник

как $P = (E_1, ..., E_2)$, а ребро — как $E = (V_1, V_2, P_1, P_2)$. то либо P_1 либо P_2 – пусто, когда ребро будет принадлежать только одном у многоугольнику.[2]

При явном задании ребер полигональная сетка рендериться путём вычерчивания рёбер. Поэтому получается избегать многократного отрисовывания рёбер. Простые многоугольники так же легко отрисовываются. [3]

Зачастую ребра полигональных сеток — общие для двух фигур и более. Можем рассмотреть случай в картографии, где подобные подразделения (округа, области, федерации) описываются многоугольниками. Представляющее часть границы между двумя областями ребро (или их последовательность) является также границей округа в каждой области, а иногда, и города. Получается, ребро может лежать сразу в шести фигурах. Также число будет соответственно возрастать при делении городов на районы, школьные участки и избирательные округа. Для подобных приложений описания этих ребер могут быть расширены, чтобы было возможно включить любое число многоугольников: $E = (V_1, V_2, P_1, P_2, \ldots, P_n)$.[1]

Задача не является простой ни в одном из этих представлений (инцидентных вершине): необходимо перебрать все ребра для её решения. Естественно, для определения этих отношений можно непосредственно использовать дополнительную информацию.

1.2 Алиасинг

Алиасинг (aliasing) — это, возможно, наиболее фундаментальный и самый широко обсуждаемый артефакт 3D-рендеринга всех времён. Однако в игровом сообществе его часто недопонимают. В этой статье я подробно расскажу о теме сглаживания (антиалиасинга, anti-aliasing, AA) в реальном времени, особенно о том, что касается игр, и в то же время буду излагать всё достаточно простым языком.

Алиасинг представляет одну из фундаментальных проблем компьютерной графики, и для борьбы с ним придумано множество разнообразных алгоритмов антиалиасинга. Появление MLAA привлекло интерес к алгоритмам, работающим на этапе постобработки. Одним из таких алгоритмов (с небольшой оговоркой) является *Geometry Buffer Anti-Aliasing* (GBAA). В этом материале описана попытка модификации оригинального алгоритма для улучшения качества антиалиасинга в некоторых случаях.

Рисунок 1.1-Geometric Post-process Anti-Aliasing(GPAA)

GBAA является усовершенствованной версией алгоритма Geometric Post-process Anti-Aliasing (GPAA). Лежащая в его основе идея заключается в том, что вместо поиска резких границ в исходном изображении для оценки расположения геометрических ребер (как это делает MLAA) можно использовать информацию о ребрах в «чистом виде», получив ее от рендерера. Алгоритм довольно прост:

1. Отрендерить сцену (основной проход);
2. Сделать копию backbuffer;
3. Отрендерить, при этом смешивая цвета соседних пикселей для получения необходимых сглаженных ребер.

Слияние цветов пикселей происходит следующим образом:

1. Для каждого из пикселей определяется направление (по вертикали или горизонтали) и расстояние до ближайшего ребра;
2. По направлению и расстоянию определяется покрытие пикселя соседним треугольником;
3. Покрытие используется для расчета коэффициентов блендинга, а направление — для выбора соседнего пикселя.

Данное изображение показывает логику работы алгоритма:

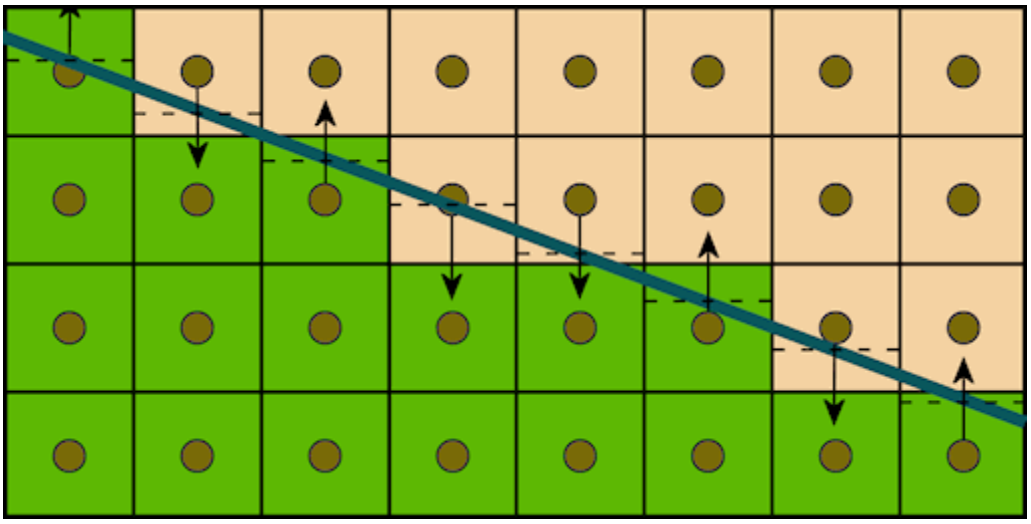
Рисунок 1.2 – Блендинг

Синей линией обозначено геометрическое ребро. Находим соседние пиксели. Чёрной пунктирной линией обозначены смещения пикселя относительно центра, использующиеся для вычисления коэффициентов блендинга. Блендинг осуществляется единственной выборкой из текстуры: к текстурным координатам определенного пикселя добавляется смещение, а линейный фильтр совершает всю остальную работу.

Координаты геометрического ребра в вершинном шейдере проецируются на экранную плоскость и служат для определения уравнения прямой линии, на которой и лежит ребро. Уравнение попадает в пиксельный шейдер в виде четырехмерного вектора, и там вычисляется цвет пикселя и его покрытие. [4]

1.2.1 Код шейдера (HLSL)

Главными достоинством этого алгоритма являются:

1) Качество, не зависимая от угла наклона ребра, что является основной проблемой постобработки
2) Производительность

.На первом изображении показаны результаты работы FXAA с различными предустановками качества, а на втором — результаты работы GPAA.

Рисунок 1.3 – FXAA3,FXAA5

Рисунок 1.4 - GPAA

Наиболее затратный метод — это копирование экранного буфера: отрисовка одного кадра (в исходной реализации) на стандартной видео карте в разрешении 1280x720 выполняется за 0.93 мс, из которых только копирование экранного хранилища занимает 0.08 мс, а само сглаживание ребер — 0.01 мс. Необходимость предварительной обработки геометрии для извлечения ребер и дополнительной памяти для их хранения является недостатком. Помимо этого, графический процессор обычно выполняет растеризацию линий довольно медленно. Эти проблемы в целом плохо влияют на масштабируемость GPAA при растущей геометрической сложности сцены. [5]

1.2.2 Geometry Buffer Anti-Aliasing (GBAA)

GBAA — это улучшенная версия GPAA. Собственно, улучшение заключается в том, что расстояние и направление до пределов треугольников вычисляются в геометрическом шейдере, что устраняет необходимость заранее обрабатывать и растериализовывать линии, уменьшая объем занимаемой памяти. Также исключается зависимость производительности от сложности геометрической фигуры.

Изображение ниже демонстрирует определения расстояний до границ: для каждого ребра геометрический шейдер сначала вычисляет высоту *d*, а потом и расстояние *d,* выровненное по осям. Все результаты фиксируются в вершинных атрибутах, интерполируются растеризатором и потом используются для получения коэффициентов блендинга в пиксельном шейдере.

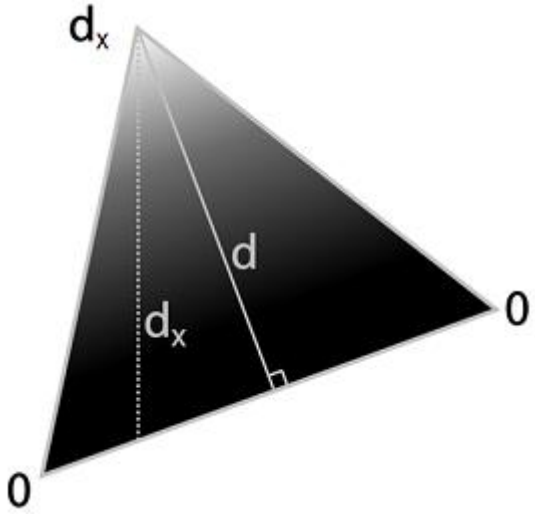

Рисунок 1.5 - GBAA

Ещё одно достоинство перед GPAA — это возможность выполнения Anti-Aliasing не только геометрических фигур, но и других границ, расстояние до которых можно оценить. К примеру, границ в альфа-прозрачных текстурах:

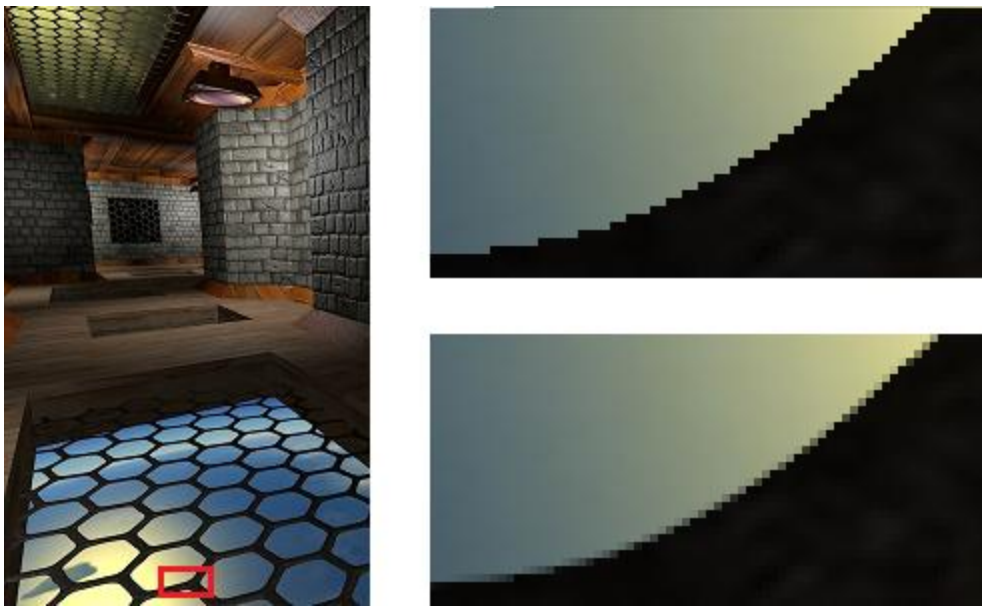

Рисунок 1.6 - Блендинг

Вычисления геометрического шейдера используются пиксельным шейдером для поиска ребер, пересекающих нужный пиксель. Если минимальное смещение до ребра меньше, чем половина этого пикселя, то выбирается смежный с ним пиксель, вычисляются коэффициенты блендинга и производится блендинг. В ином случае пиксель остается без изменений. Вся информация о смещениях силуэтных ребер доступна у тех пикселей, которые присутствуют на внутренней стороне силуэта, поэтому они требуют дополнительной обработки:

1) для начала выбирается смещение из 4 соседних пикселей;
2) в зависимости от этого смещения выбирается один из 4 соседей
   — правый: -1.0 <= offset.x <= -0.5
   — левый: 0.5 <= offset.x <= 1.0
   — верхний: 0.5 <= offset.y <= 1.0
   — нижний: -1.0 <= offset.y <= -0.5;
3) из данного смещения выходит другое, скорректированное смещение для текущего пикселя, и после необходимо лишь вычислить коэффициенты и выполнить блендинг.[6]

1.2.3 Модификация кода шейдера (HLSL)

У GBAA имеется неприятная особенность, выражающаяся в артефактах возле сходящихся ребер:

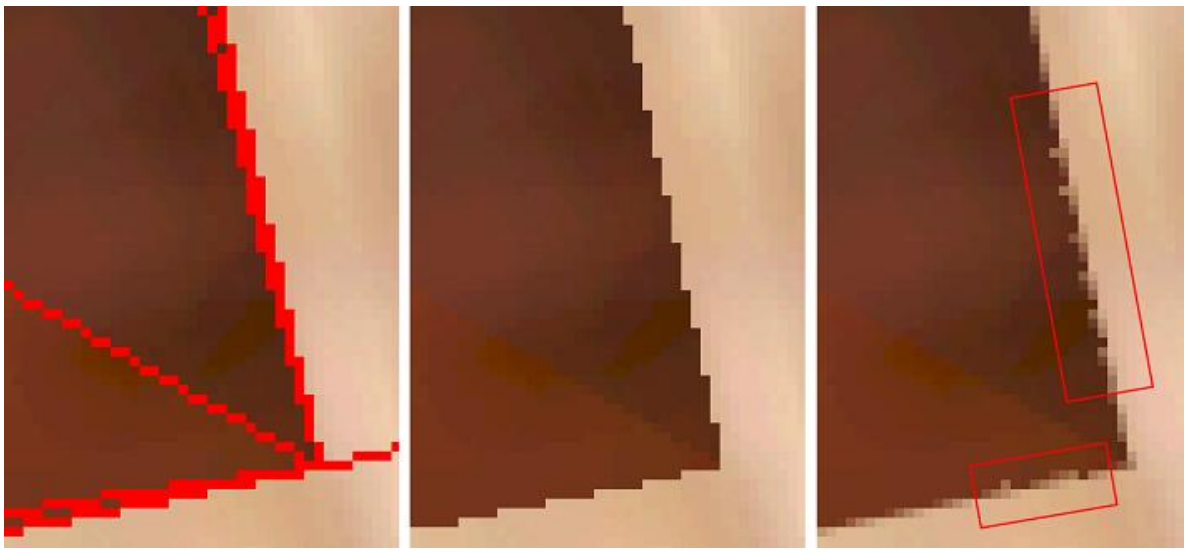

Рисунок 1.7-сходящиеся рёбра у GBAA

Тонкие sub-pixel треугольники являются основной проблемой для всех алгоритмов обработки, работающих с изображением в экранном разрешении, и, к сожалению, GBAA не является исключением. Рассмотрим первый случай:

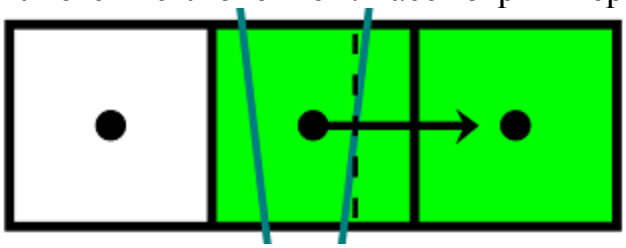

Рисунок 1.8-Пример 1.

В этом случае точка выборки текущего пикселя в центре попадает в тонкий треугольник, а точки выборки соседних пикселей – в большие треугольники, смежные с тонким. Если правое ребро тонкого треугольника оказалось ближе к центру среднего пикселя, как изображено на рисунке, то GBAA вычислит покрытие правого треугольника средним пикселем с помощью смещения правого ребра относительно его центра, а затем выдаст линейно интерполированный цвет между средним и правым пикселем. Однако средний пиксель покрывает фрагменты всех трех треугольников, и если цвет одного пикселя будет отличаться от остальных, то результирующий цвет не будет определен верно. Предположим, что a, b, c – исходные цвета трех пикселей, а α, β, γ – отношение площадей покрытых средним пикселем фрагментов треугольников к площади пикселя. Скорректированный цвет среднего пикселя в таком случае может быть определен по формуле

$b_{out} = \alpha a + \beta b + \gamma c,$

в то время как оригинальный алгоритм вычислит его по формуле

$b_{out} = (\alpha + \beta)b + \gamma c$

Если, например, левый пиксель окажется белым, а средний и правый – черными, то в описанной ситуации оригинальный алгоритм всегда будет выдавать

черный цвет для среднего пикселя, оставляя фрагмент исходного изображения без изменений.

Второй случай возникает, когда тонкий треугольник располагается между центрами двух пикселей:

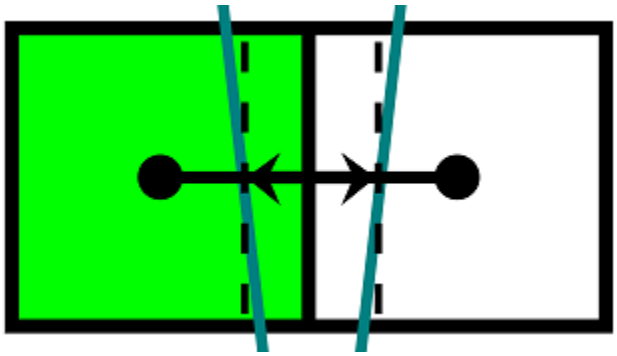

Рисунок 1.9-Пример 2.

Здесь, в отличие от первого случая, часть необходимой для расчета корректного цвета информации потеряна: нет точки выборки, которая попала бы внутрь тонкого треугольника. Чтобы понять, как такой случай может повлиять на конечное изображение, рассмотрим более крупный фрагмент:

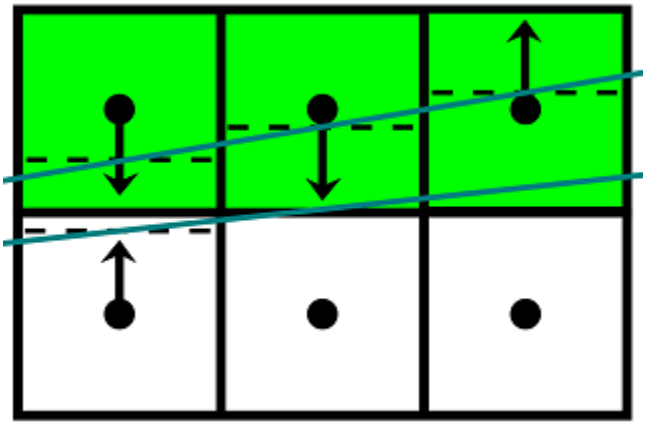

Рисунок 1.10-Пример 3.

Поскольку при смещении вправо более темные треугольники смещаются вверх, занимая все меньшую площадь, яркость пикселей в верхнем ряду должна возрастать. Так и происходит, пока очередь не доходит до последнего столбца. Случай, который возникает при его обработке, был рассмотрен ранее. Здесь же главным источником проблемы являются первые два столбца: верхние пиксели должны получить свой исходный цвет, но вместо этого оригинальный алгоритм смешивает их с цветами нижних пикселей. Слева показан фрагмент границы треугольника, полученный с использованием неправильных коэффициентов блендинга, справа — корректный результат:

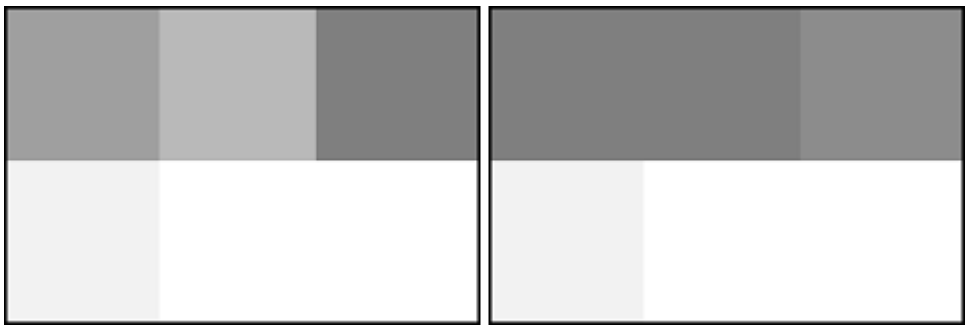

Рисунок 1.11-Сравнение результатов.

Поведение оригинального алгоритма в этом случае можно улучшить, сохраняя исходные цвета пикселей, между которыми расположен тонкий треугольник.

Для обработки этих двух случаев можно внести несколько изменений в оригинальный алгоритм.

1. Корректное вычисление цвета пикселя в случае первом случае требует наличия информации о втором смещении, в то время как оригинальный алгоритм хранит лишь одно. Для этого потребуется дополнительное место в геометрическом буфере. Если существует второе смещение по той же оси, что первое, но противоположное ему, то это смещение необходимо также сохранить в геометрическом буфере. На этапе постобработки для определения случая тройного покрытия следует проверить, пересекается ли пиксель двумя ребрами с разных сторон, и если пересекается, вычислить скорректированный цвет.
2. Обработка второго случая оказывает минимальное влияние на структуру алгоритма, требуя внесения дополнительной проверки в этап постобработки. Пиксель должен получить свой исходный цвет, если в направлении соответствующего ему смещения расположен соседний пиксель, которому соответствует противоположное направление смещения в той же оси. [6]

1.3 Сплайн интерполяция

Дан набор точек $(x_i, y_i)_{i=0}^{n-1}$ и набор положительных весов $(w_i)_{i=0}^{n-1}$. Предположим, что некоторые точки могут быть важнее (если нет, то все веса одинаковые). Необходимо, что между этими точками прошла кривая таким образом, чтобы обходила все данные.

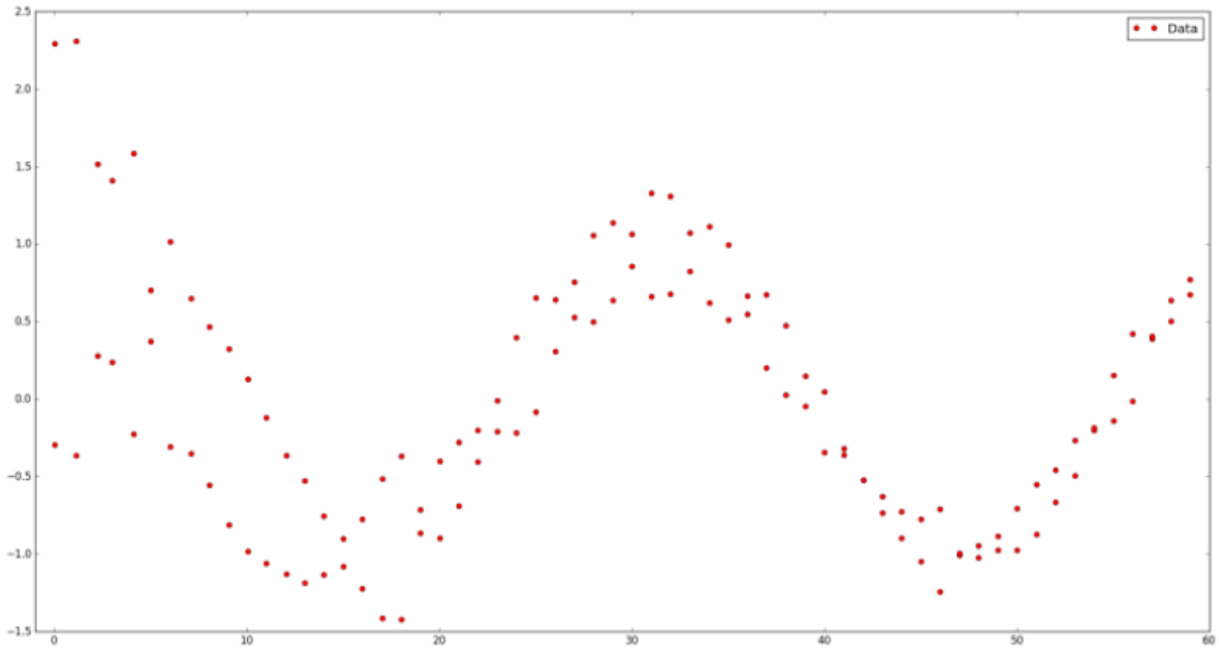

Рисунок 1.12-График случайных значений

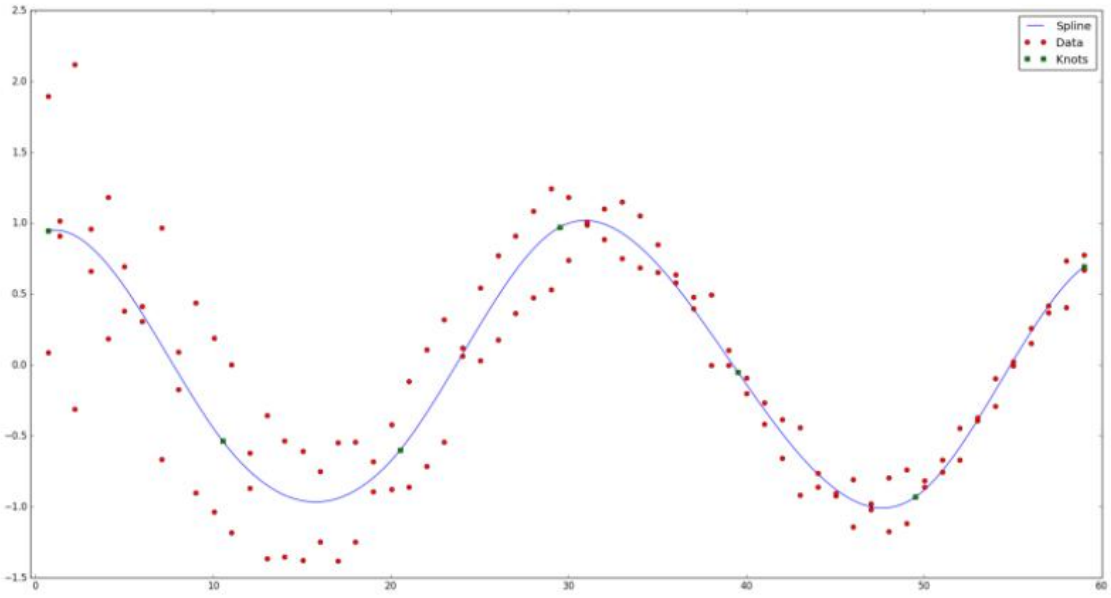

Рисунок 1.13- Получившийся сплайн

Основные определения:

Функция s(x) на интервале [a, b] — это ***сплайн степени k*** на сетке с горизонтальными узлами $(\lambda_i)_{i=0}^{g+2k+1}$, если выполняются следующие условия:

1. Функция s(x) является полиномом k-й степени на интервалах $[\lambda_i, \lambda_{i+1}]_{i=k}^{g+k}$.

2. n-ая производная функции s(x) непрерывна в любой точке [a, b] для любого n = 1,…, k-1.

Для построения сплайна необходимо задать сетку из горизонтальных узлов. Расположим их так, чтобы внутренний интервала (a, b) стояло g узлов, а по краям — k+1: $a = \lambda_0 = \ldots = \lambda_k$ и $b = \lambda_{g+k+1} = \ldots = \lambda_{g+2k+1}$.

Каждый сплайн в точке $x \in [\lambda_j, \lambda_{j+1}]$ может быть представлен в базисной форме:

$$s(x) = \sum_{i=j-k}^{j} c_i N_{i,k+1}(x) \tag{1}$$

где $N_{i,k+1}(x)$ — B-сплайн k+1-го порядка:

$$N_{i,k+1}(x) = \alpha_{i,k}(x) N_{i,k}(x) + (1 - \alpha_{i+1,k}(x)) N_{i+1,k}(x) \tag{2}$$

$$N_{i,1}(x) = \begin{cases} 1, & x \in [\lambda_i, \lambda_{i+1}), \\ 0, & x \notin [\lambda_i, \lambda_{i+1}), \end{cases} \tag{3}$$

$$\alpha_{i,k}(x) = \begin{cases} \frac{x - \lambda_i}{\lambda_{i+k} - \lambda_i}, & \lambda_i \neq \lambda_{i+k}, \\ 0, & \lambda_i = \lambda_{i+k}. \end{cases} \tag{4}$$

Например, так выглядит базис на сетке из g = 9 узлов, равномерно

распределенных на интервале [0, 1]:

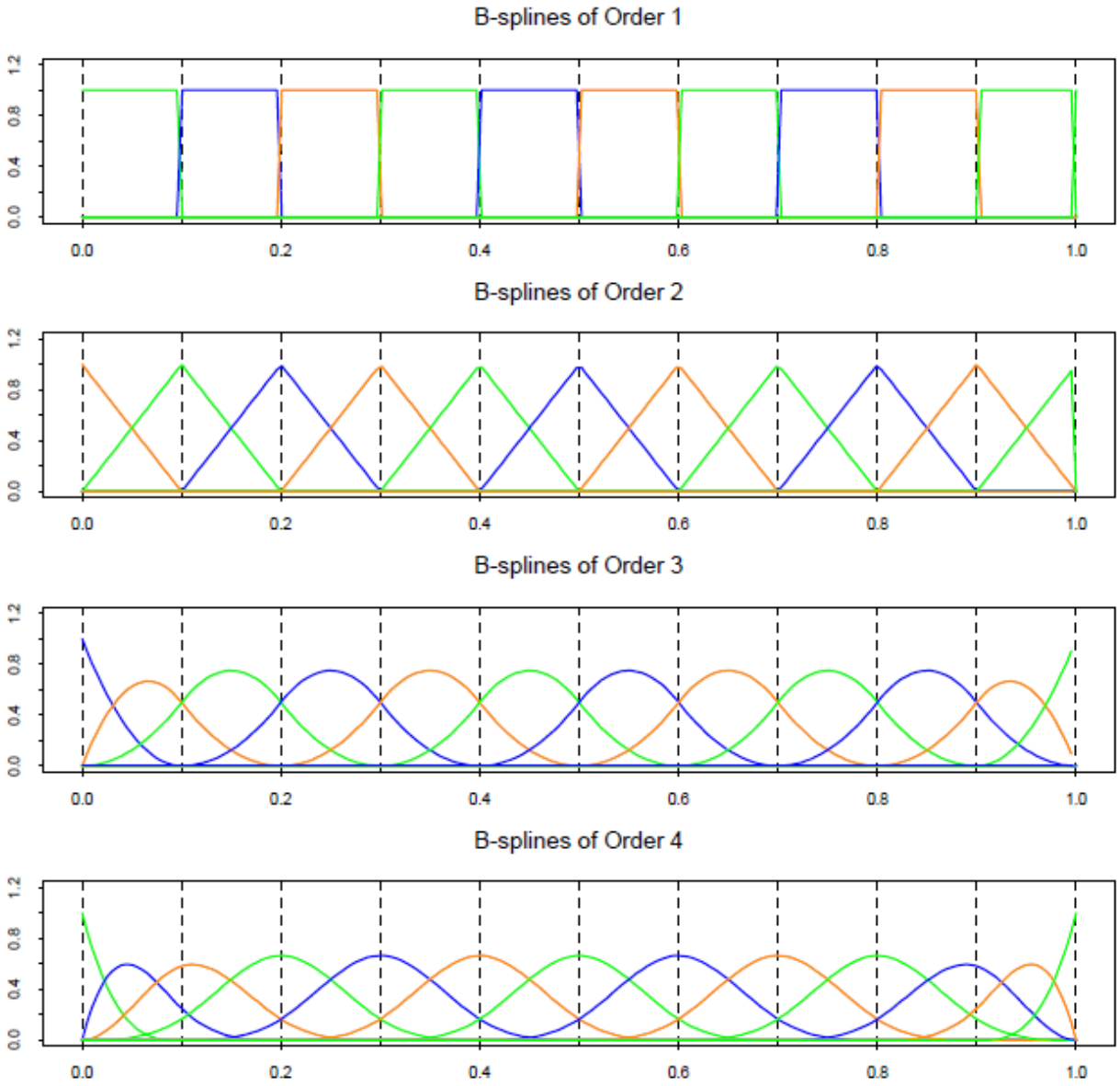

Рисунок 1.14-B-сплайн распределение

[7]

1.3.1 Аппроксимация с заданными горизонтальными узлами

Сплайн однозначно определяется узлами и коэффициентами. Предположим, что узлы $(\lambda_i)_{i=0}^{g+2k+1}$ нам известны. Также на вход подается набор данных $(x_i, y_i)_{i=0}^{n-1}$ с соответствующими весами $(w_i)_{i=0}^{n-1}$. Необходимо найти коэффициенты $(c_i)_{i=0}^{g+k}$, по максимуму приближающие кривую сплайна к этим данным. Строго говоря, они

обязаны доставлять минимум функции

$$\delta(c) = \sum_{i=0}^{n-1} \left[w_i \Big(y_i - \sum_{i=0}^{g+k} c_j N_{j,k+1}(x_i) \Big) \right]^2 \tag{5}$$

Запишем в матричном виде для удобства:

$$\delta(c) = \|y - Ec\|_2^2,$$

где

$$E = \begin{pmatrix} w_0 N_{0,k+1}(x_0) & \dots & w_0 N_{g+k,k+1}(x_0) \\ \vdots & \vdots & \vdots \\ w_{n-1} N_{0,k+1}(x_{n-1}) & \dots & w_{n-1} N_{g+k,k+1}(x_{n-1}) \end{pmatrix}, \tag{6}$$

$$y = \begin{pmatrix} w_0 y_0 \\ \vdots \\ w_{n-1} y_{n-1} \end{pmatrix}, c = \begin{pmatrix} c_0 \\ \vdots \\ c_{g+k} \end{pmatrix}. \tag{7}$$

Видно, что матрица E — блочно-диагональная. Минимум достигается, когда градиент ошибки по коэффициентам будет равен нулю:

$$\nabla \delta = 0$$

Зададим оператор $\langle \cdot, \cdot \rangle$, обозначающий взвешенное скалярное произведение:

$$\langle N_i, N_j \rangle = \sum_{r=0}^{n-1} w_r^2 N_{j,k+1}(x_r) N_{i,k+1}(x_r) \tag{8}$$

$$\langle N_i, y \rangle = \sum_{r=0}^{n-1} w_r^2 y_r N_{i,k+1}(x_r)$$

Пусть также

$$A = E^T E = \begin{pmatrix} \langle N_0, N_0 \rangle & \dots & \langle N_0, N_{g+k} \rangle \\ \vdots & \vdots & \vdots \\ \langle N_{g+k}, N_0 \rangle & \dots & \langle N_{g+k}, N_{g+k} \rangle \end{pmatrix}, \tag{9}$$

$$r = E^T y = \begin{pmatrix} \langle N_0, y \rangle \\ \vdots \\ \langle N_{g+k} y \rangle \end{pmatrix}. \tag{10}$$

Тогда задача и все предыдущие формулы сводятся к решению одной простой системы линейных уравнений:

$$Ac = r,$$

где матрица A (2k+1) — диагональная, так как $\langle N_i, N_j \rangle = 0$, если |i — j| > k. К тому же, матрица A положительно-определенная и симметричная, получается, решение возможно найти с помощью разложения Холецкого (также существует алгоритм для разреженных матриц).
Решив систему, получаем нужный результат:

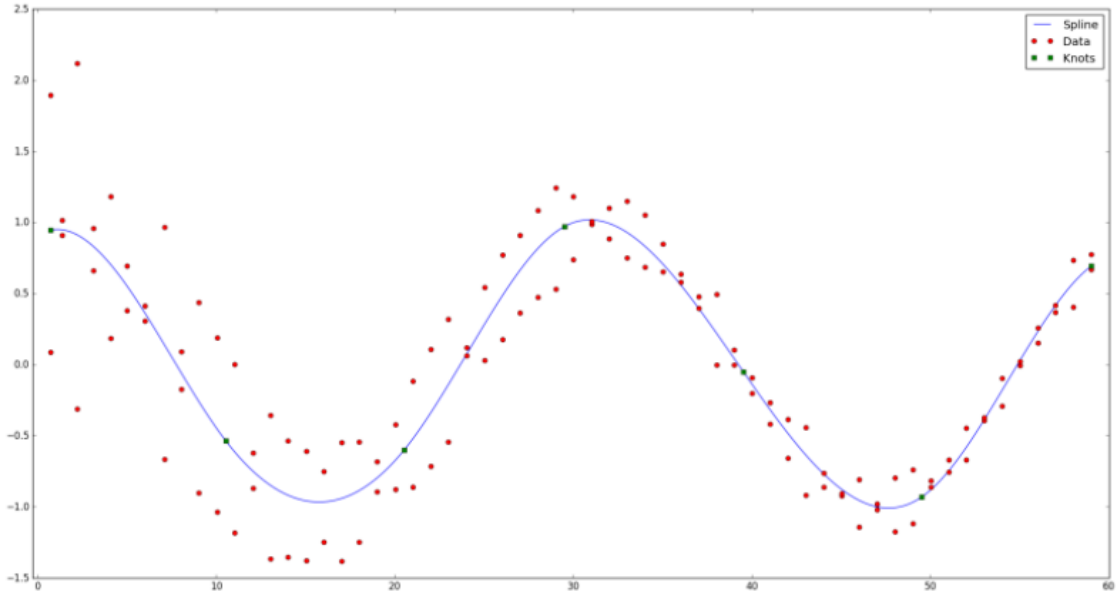

Рисунок 1.15 – Результат решения системы

Сглаживание

Но не всегда все так хорошо. При небольшом количестве данных по отношению к степени сплайна и количеству узлов может возникнуть проблема — сверхподгонки (overfitting). Далее изображен пример «плохого» кубического сплайна, при этом идеально проходящего сквозь данные: [8]

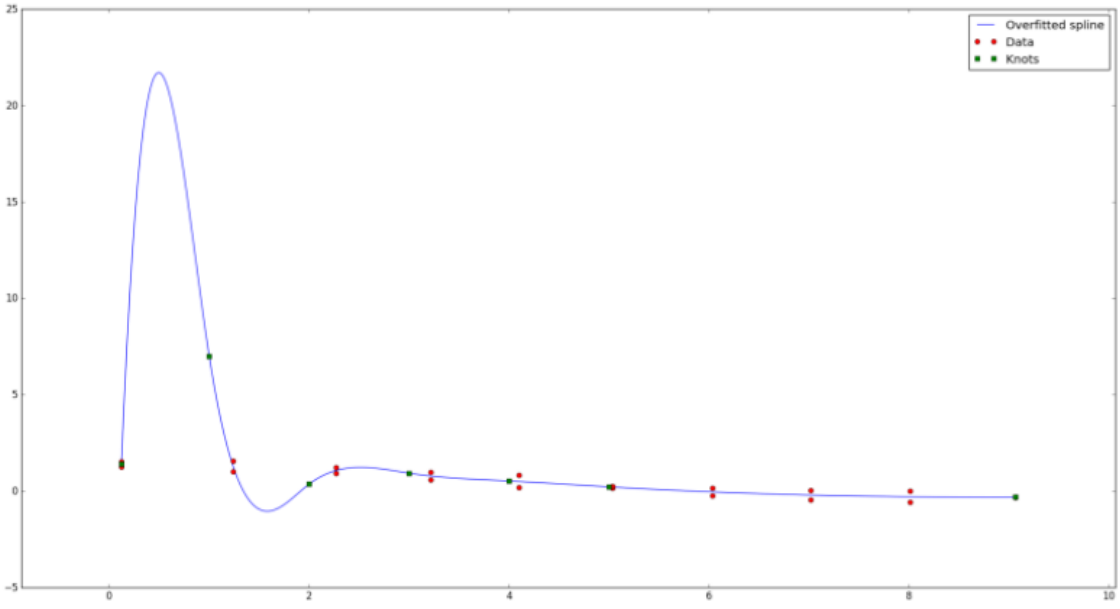

Рисунок 1.16 – Пример кубического сплайна

Кривая теперь не такая красивая. Попробуем уменьшить колебания сплайна. Для этого мы попытаемся «сгладить» его k-ю производную. Другими словами, мы сведем к минимуму разницу между производной слева и производной справа от каждого из узлов:

$$\eta = \sum_{q=k+1}^{g+k} \left(s^{(k)}(\lambda_q +) - s^{(k)}(\lambda_q -) \right)^2. \tag{11}$$

Разложив сплайн в базисную форму, получим:

$$\eta = \sum_{q=k+1}^{g+k} \left(\sum_{i=0}^{g+k} c_i \left[N_{i,k+1}^{(k)}(\lambda_q +) - N_{i,k+1}^{(k)}(\lambda_q -) \right] \right)^2 = \sum_{q=k+1}^{g+k} \left(\sum_{i=0}^{g+k} c_i \alpha_{i,q} \right)^2, \tag{12}$$

$$\alpha_{i,q} = \begin{cases} (-1)^{k+1} k! \frac{\lambda_{i+k+1} - \lambda_i}{\prod_{j=i, j \neq q}^{i+k+1} (\lambda_q - \lambda_j)} & q-k-1 \leq i \leq q, \\ 0, & \text{otherwise.} \end{cases} \tag{13}$$

Рассмотрим ошибку:

$$\delta(c) + q\eta(c) = \|\tilde{y} - \tilde{E}c\|_2^2 = \left\| \tilde{y} - \begin{pmatrix} E \\ \sqrt{q}H \end{pmatrix} c \right\|_2^2 \tag{14}$$

Здесь q — вес функции, влияющей на сглаживание, и

$$H = \begin{pmatrix} \alpha_{0,k+1} & \dots & \alpha_{g+k,k+1} \\ \vdots & \dots & \vdots \\ \alpha_{0,g+k} & \dots & \alpha_{g+k,g+k} \end{pmatrix}, \tag{15}$$

$$\tilde{y} = (w_0 y_0 \ \dots \ w_{n-1} y_{n-1} \ 0 \ \dots 0)^T.$$

Сводим к:

$$(A + qB)c = r \tag{16}$$

где

$$B = H^T H \tag{17}$$

Ранг матрицы B равен g. Она симметричная и будет положительно определенной, так как q > 0, A + qB. И поэтому разложение Холецкого по-прежнему применимо к новой системе уравнений. Однако матрица B вырожденная и при очень больших значениях q могут возникать численные ошибки.
При очень маленьком значении q = 1e-9 вид кривой изменяется очень слабо.[9]

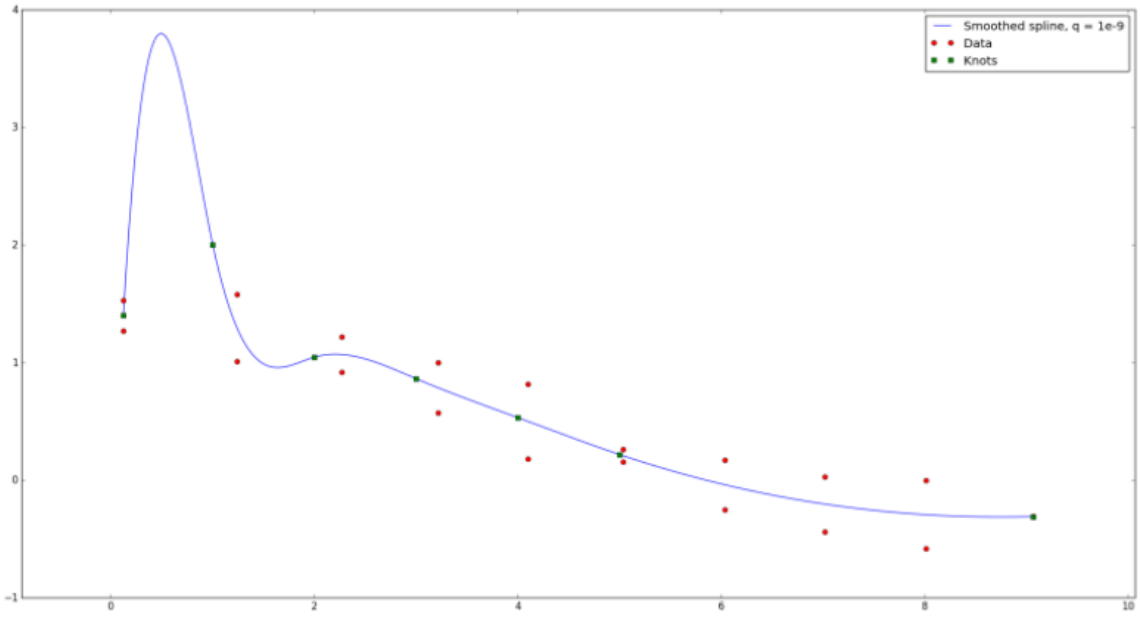

Рисунок 1.17 – Изменённый вид кривой

Но при q = 1e-7 в этом примере уже достигается достаточное сглаживание.

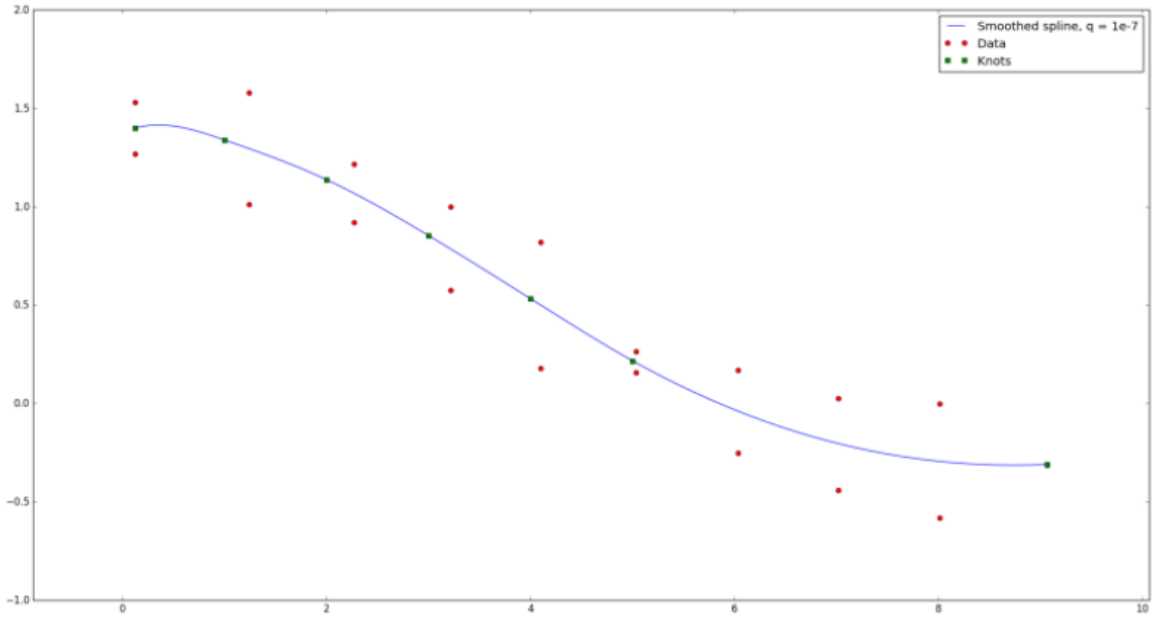

Рисунок 1.18 – Сглаженный сплайн

1.3.2 Аппроксимация с неизвестными горизонтальными узлами

На вход помимо данных подается только количество узлов g, интервал [a, b] и степень сплайна k. Предположим, что лучше всего расположить на интервале узлы равномерно:

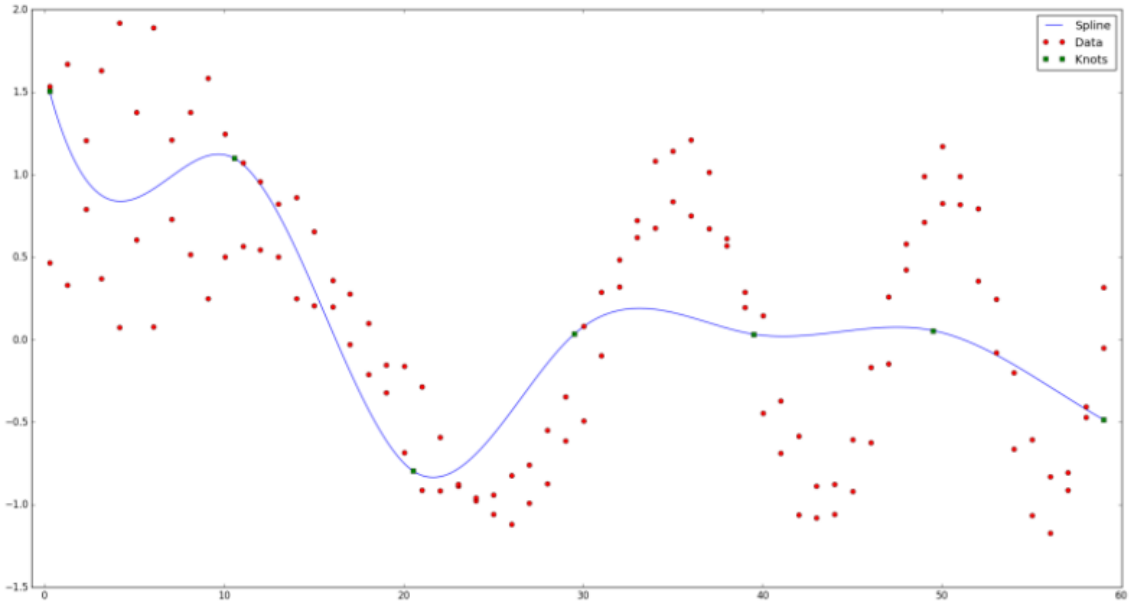

Рисунок 1.19 –Равномерный сплайн

Расположим их так, чтобы значение ошибки

$$\xi(\lambda) = \delta(\lambda) + q\eta(\lambda) + pP(\lambda) \tag{18}$$

было минимально. Последнее слагаемое имеет роль штрафной функции, чтобы узлы не так сильно приближались друг к другу:

$$P(\lambda) = \sum_{i=k}^{g+k-1} \frac{1}{\lambda_{i+1} - \lambda_i} \tag{19}$$

Положительный параметр p — вес штрафной функции. Чем больше его значение, тем быстрее узлы будут удаляться друг от друга и стремиться к равномерному расположению.[10]

Для решения этой задачи мы воспользуемся методом сопряженных градиентов. Его удобство в том, что для квадратичной функции он сходится за фиксированное (в данном случае g) количество шагов.

Инициализируем направление $d^0 = -\nabla\xi(\lambda^0)$.

Как необходимо рассчитать производную ошибки по узлам?

1. Для j = 0,…, g-1

$$\theta(\alpha) = \xi(\lambda^j + \alpha d^j) \tag{20}$$

возвращающую ошибку в зависимости от выбора шага вдоль заданного направления. На этом шаге мы находим оптимальное значение α*, доставляющее минимум этой функции. Для этого мы решаем задачу одномерной оптимизации. О том, каким образом, будет сказано позже.

Обновляем значения узлов:

$$\lambda^{j+1} = \lambda^j + \alpha^* d^j \tag{21}$$

Обновляем вектор направления:

$$d^{j+1} = -\nabla\xi(\lambda^{j+1}) + \frac{\|\nabla\xi(\lambda^{j+1})\|_2^2}{\|\nabla\xi(\lambda^j)\|_2^2} d^j \tag{22}$$

Если

$$\frac{|\xi(\lambda^{j+1}) - \xi(\lambda^j)|}{\xi(\lambda^j)} < \varepsilon_1 \tag{23}$$

и

$$\frac{\|\lambda^{j+1} - \lambda^j\|_2^2}{\|\lambda^j\|_2^2} < \varepsilon_2 \tag{24}$$

где ε_1 и ε_2 — заранее заданные величины, отвечающие за точность работы алгоритма, то выходим. Иначе, обнуляем счетчик и возвращаемся на первый шаг.[11]

Решение задачи одномерной минимизации

Для того, чтобы найти значение α, доставляющее минимум функции

$$\theta(\alpha) = \xi(\lambda^j + \alpha d^j)$$

мы используем алгоритм, позволяющий сократить количество обращений к оракулу, а именно количество операций аппроксимации с заданными узлами и подсчета функции ошибки. Мы будем использовать нотацию $\theta(\alpha_i) = \theta_i$.

Пусть первая и последняя компоненты вектора направления равны нулю: $d_0^j = d_{g+1}^g = 0$. Зададим также максимально возможный шаг вдоль этого направления:

$$\alpha_{\max} = \min\left(\frac{\lambda_{l+1}^j - \lambda_l^j}{d_l^j - d_{l+1}^j} \middle| l = 0, \ldots, g; d_l^j > d_{l+1}^j \right) \tag{25}$$

Такой выбор обусловлен тем, что узлы не должны пересекаться.

Инициируем k = 0 и начальные шаги: $\alpha_0 = 0$, $\alpha_1 = \frac{\alpha_{\max}}{2}\left(1 - \frac{\theta_0}{\alpha_{\max}\theta'_0}\right)^{-1}$, $\alpha_2 = 2\alpha_1$.

До тех пор, пока $\theta_1 \geq \theta_0$:

Задаём

$$\hat{\alpha}_1 = \frac{-\theta'_0\alpha_1^2}{2(\theta_1 - \theta_0 - \theta'_0\alpha_1)} \tag{26}$$

и уменьшаем шаг

$$\alpha_1 = \max\left(\frac{\alpha_1}{10}, \hat{\alpha}_1\right) \tag{27}$$

k = k + 1

Если k > 0, то возвращаем α* = α_1. Иначе:

- До тех пор, пока $\theta_2 < \theta_1$: $\alpha_0 = \alpha_1$, $\alpha_1 = \alpha_2$ и

$$\alpha_2 = \min\left(2\alpha_1, \frac{\alpha_{max} + \alpha_1}{2}\right) \tag{28}$$

Возвращаем $\alpha^* = \tilde{\alpha}$, где $\tilde{\alpha}$ — корень уравнения I'(α) = 0 и I(α) — аппроксимация функции ошибки:

$$I(\alpha) = Q(\alpha) + pR(\alpha), \tag{29}$$

где

$$Q(\alpha) = a_0 + a_1(\alpha - \alpha_0) + a_2(\alpha - \alpha_0)^2, \tag{30}$$

$$R(\alpha) = b_0 + b_1(\alpha - \alpha_0) + b_2 \ln(\alpha_{\max} - \alpha_0). \quad (31)$$

Коэффициенты a_i и b_i могут быть найдены из уравнений[12]

$$Q(\alpha_i) = \delta(\lambda^j + \alpha_i d^j) \quad (32)$$

и

$$R(\alpha_i) = P(\lambda^j + \alpha_i d^j) \quad (33)$$

Объяснение алгоритма:

Идея заключается в том, чтобы расставить три точки $\alpha_0 < \alpha_1 < \alpha_2$ таким образом, чтобы по значениям ошибок, достигаемых в этих точках, можно было построить простую аппроксимирующую функцию и вернуть её минимум. Притом значение ошибки в α_1 должно быть меньше, чем значение ошибки в α_0 и α_2.[13]

Находим начальное приближение α_1 из условия $S'(\alpha_1)=0$, где $S(\alpha)$ — функция вида

$$S(\alpha) = \frac{c_0}{(\alpha + c_1)(\alpha_{\max} - \alpha)} \quad (34)$$

Константы c_0 и c_1 находятся из условий $S(0) = \theta_0$ и $S'(0) = \theta'_0$.

Если просчитались с начальным приближением, то уменьшаем шаг α_1 до тех пор, пока он доставляет большее значение ошибки, чем α_0. Выбор $\hat{\alpha}_1$ исходит из условия $Q'(\hat{\alpha}_1) = 0$, где $Q(\alpha)$ — парабола интерполирующая функцию ошибки $\theta(\alpha)$: $Q(\alpha_0 = 0) = \theta_0$, $Q'(0) = \theta'_0$ и $Q(\alpha_1) = \theta_1$.
Если $k > 0$, то нашли значение α_1, такое что при его выборе значение ошибки будет меньше, чем при выборе α_0 и α_2, и мы возвращаем его в качестве грубого приближения α^*.

Если же наше первоначальное приближение было верным, то пытаемся найти шаг α_2, такой что $\theta_1 < \theta_2$. Он будет найден между α_1 и α_{max}, так как α_{max} — точка сингулярности для штрафной функции.

Когда найдены все три значения α_0, α_1 и α_2, представляем функцию ошибки в виде суммы двух функций, приближающих разность квадратов и функцию штрафа. Функция $Q(\alpha)$ — парабола, чьи коэффициенты могут быть найдены, так как мы знаем её значения в трех точках. Функция $R(\alpha)$ уходит на бесконечность при α, стремящемся к α_{max}. Коэффициенты b_i также могут быть найдены из системы из трех уравнений. В результате, приходим к уравнению, которое может быть

приведено к квадратному и решено:[14]

$$I'(\tilde{\alpha}) = Q'(\tilde{\alpha}) + pR'(\tilde{\alpha}) = a_1 + 2a_2(\tilde{\alpha} - \alpha_0) + pb_1 - \frac{pb_2}{\alpha_{\max} - \tilde{\alpha}} = 0 \quad (35)$$

Для сравнения, результат оптимально построенного сплайна:

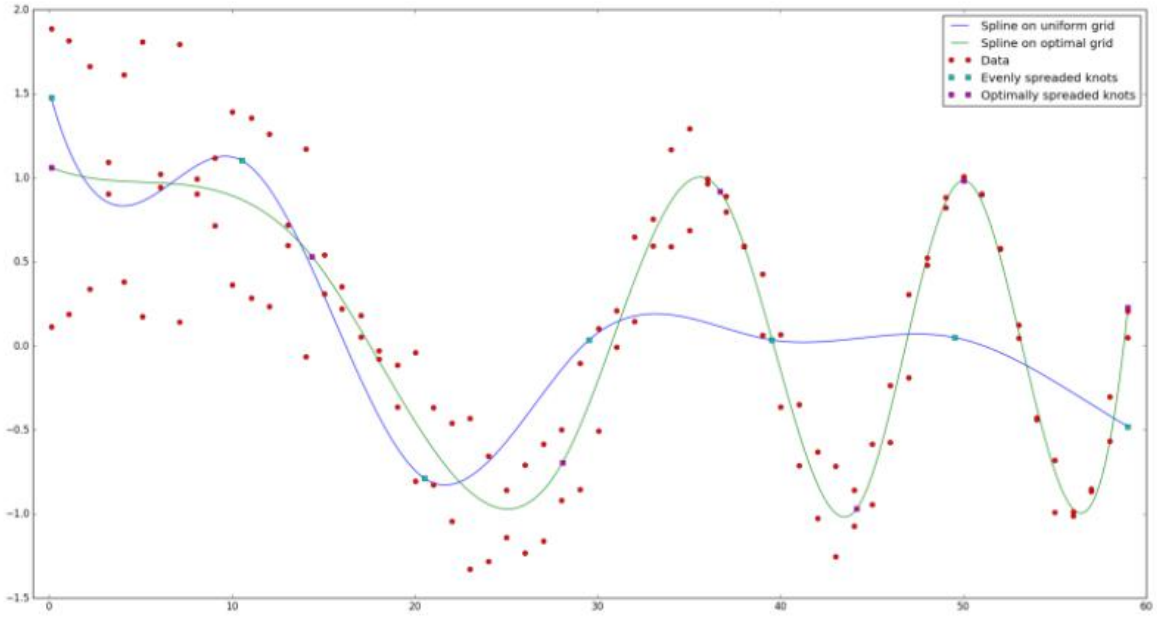

Рисунок 1.20 - Результат

1.4 Выбор языка программирования

1.4.1 Python

Python - интерпретируемый язык для скриптов различного назначения (хотя существуют и трансляторы данного языка).

Целью Python, как и Ruby, является приближение синтаксиса реальной программы к псевдокоду, который описывает задачу, что позволяет уменьшить объём программы.

Особенности дизайна и синтаксиса облегчают программистам совместную работу над кодом.

Python является мультипарадигмальным языком программирования, то есть позволяет совмещать процедурный подход с объектно-ориентированным и функциональным.

Интерпретатор языка Python можно использовать как для запуска скриптов, так и в режиме интерактивной оболочки.

Преимущества языка Python:

- открытость;
- простота в изучении;

- особенности синтаксиса позволяют писать легко читаемый код;
- язык предоставляет средства динамической семантики и быстрого прототипирования;
- обширное сообщество;
- множество полезных библиотек и расширений языка можно использовать в проектах благодаря унифицированному механизму импорта и программным интерфейсам;
- хорошо продуманы механизмы;
- всё является объектами в смысле объектно-ориентированного программирования, но собственно объектный подход применять необязательно.

Недостатки языка Python:

- неудачная поддержка многопоточности;
- довольно небольшое число качественных программных проектов в сравнении с другими языками программирования;
- отсутствует коммерческая поддержка средств;
- изначальная ограниченность средств для работы с базами данных;
- меньшая производительность по сравнению с основными Java VM.

1.4.2 Java

Java - сильно типизированный объектно-ориентированный язык программирования. Разработан компанией Sun Microsystems (в последующем приобретён компанией Oracle). Приложения Java обычно транслируются в специальный байт-код, поэтому они могут работать на любой компьютерной архитектуре, с помощью виртуальной Java-машины. Дата официального выпуска — 23 мая 1995 года.

Преимущества языка Java:

- Программы транслируют в байт-код, что приводит к полной независимости от операционной системы.
- Система безопасности во время исполнения программы, которая контролируется виртуальной машиной.
- Большое количество полезных стандартных библиотек.
- Автоматическое освобождение памяти, с помощью механизма “сбора мусора”.
- Полностью исключено множественное наследование.
- Все сущности языка являются объектами.

Недостатки языка Java:

- Жесткая политка Объектно-ориентированного подхода.
- Повышенное требование к оперативной памяти.
- Низкое быстродействие.

1.5 Среды разработки

1.5.1 Eclipse

Eclipse— свободная интегрированная среда разработки кроссплатформенных приложений. Первоначально среда была разработана для разработки приложений на языке Java, но в настоящее время существуют многочисленные расширения для поддержки множества языков, таких, как C ,C++, PHP, JavaScript, Perl, Python, Ruby и ряда других

1.5.2 Android Studio

Android Studio - это интегрированная среда разработки (IDE) для работы с платформой Android, которая является на сегодняшний день лучшей платформой для разработки мобильных приложений. Имеет удобный интерфейс, а также является бесплатной.

1.6 Выводы по первой главе

В результате просмотренных методов, можно сказать о том, что построение сглаженной плоскости через полигональную сетку – наиболее подходящий вариант для решения поставленных задач. Сетка широко используется в трёхмерной компьютерной графике, и она задаёт основу для любой плоскости. Трёхмерная графика зачастую используется в играх, поэтому Android Studio отлично подходит как среда разработки, так как она является доминирующий платформой для разработки игр. Опираясь на особенности языка Java можно выделить его на фоне остальных доступных в Android Studio своим удобством использования, наличием большой литературной базой, а также предоставленным пакетом OpenGL.

1.7 Постановка задачи

После анализа предметной области была уточнена цель работы: разработка математической модели поверхности в динамике, и ее программная реализация.

Для достижения поставленной цели необходимо решить следующие задачи:

1) разработать математическую модель;
2) разработать алгоритм

2 МАТЕМАТИЧЕСКАЯ МОДЕЛЬ СГЛАЖЕННОЙ ПОВЕРХНОСТИ

Сетка лежит в плоскости XZ, а значение Y будет вычисляться как функция от X и Z, т.е. Y=f(X,Z). В узлах сетки будут находится вершины.

Обозначим порядковый номер в узлах сетки вдоль оси X как i, а вдоль оси Z как j. Номера узлов будут меняться от нуля до imax или jmax соответственно.

Шаг сетки X обозначим как dx, а Z как dz.

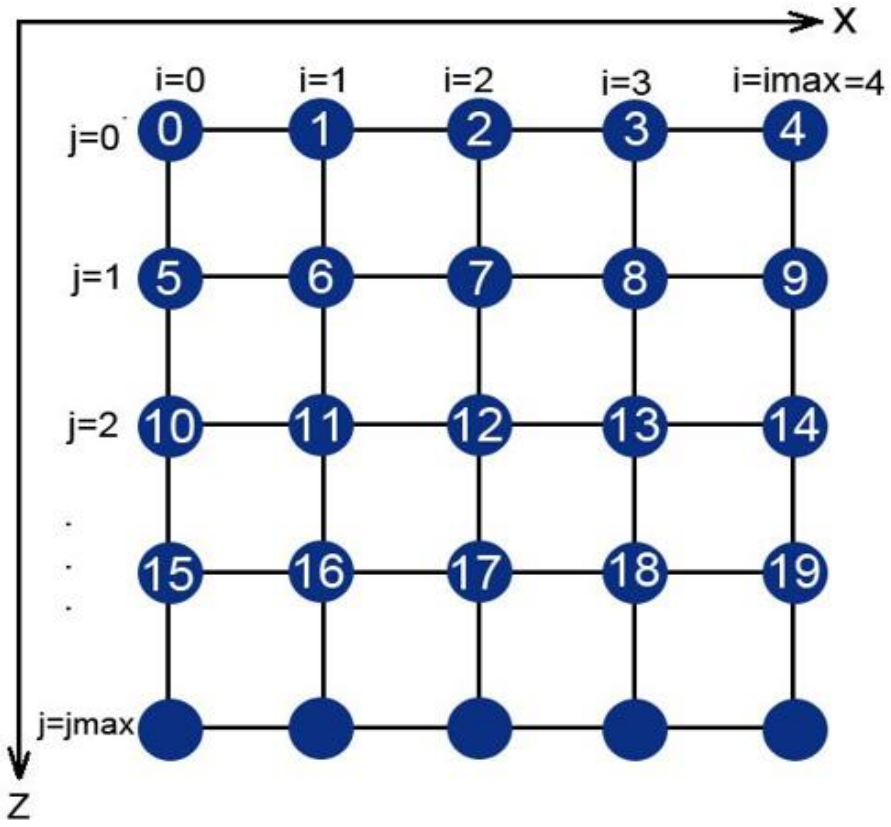

Рисунок 2.1 – Полигональная сетка

Нормаль — прямая, ортогональная касательному пространству. Чтобы нарисовать поверхность, нам потребуется её разбить на множество треугольников. Лицевой стороной треугольника является сторона, которая при рисовании обходится по вершинам против часовой стрелки.

Нормаль может быть вычислена как произведение двух векторов,

лежащих в одной плоскости.У каждого вектора есть свои координаты на плоскости:A(ax,ay,az);B(bx,by,bz).

Расчёт нормалей:

Xnorm = ay*bz-by*az;

Ynorm = bx*az-ax*bz;

Znorm = ax*by-bx*ay.

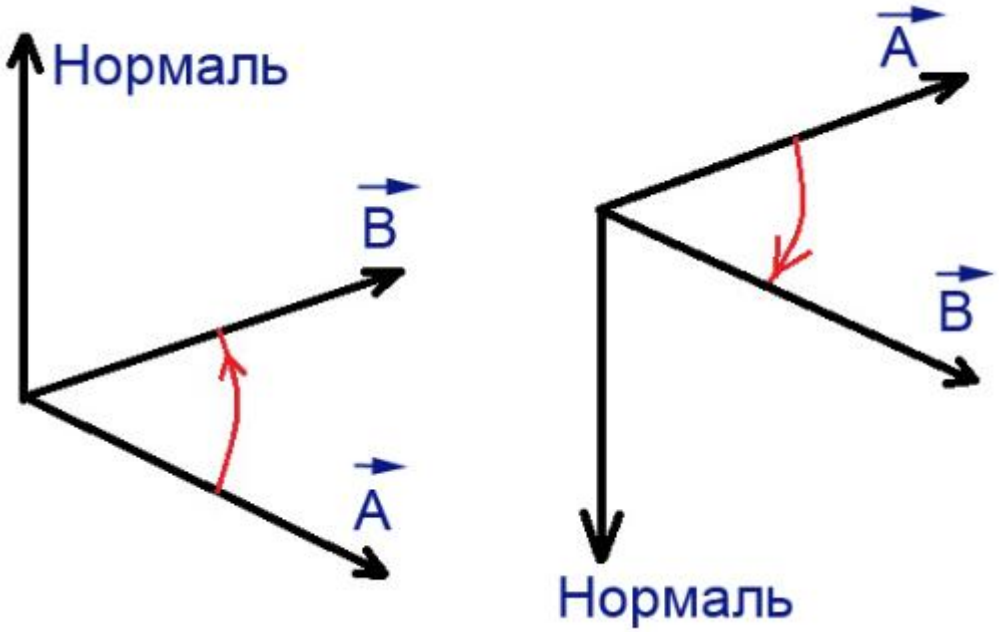

Рисунок 2.2 – Направление нормали

Разобьём сетку на ленту из треугольников и дадим каждой вершине номер. Обходим будем совершать против часовой стрелки:

0-5-1, 1-5-6, 1-6-2, 2-6-7… .

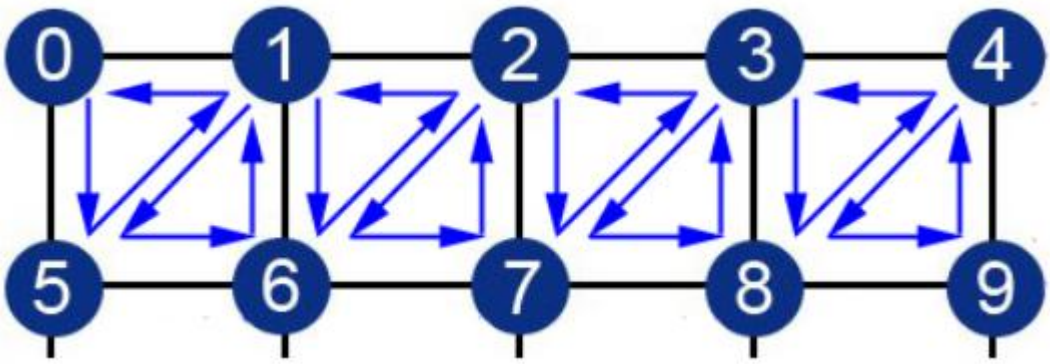

Рисунок 2.3 - Триангуляция

Дойдя до 9 вершины, переходим к обходу по часовой стрелке, однако для подсчёта удобно использовать первоначальный обход. Чтобы избежать этого, необходимо представить сетку, как бесконечный линейный ряд.

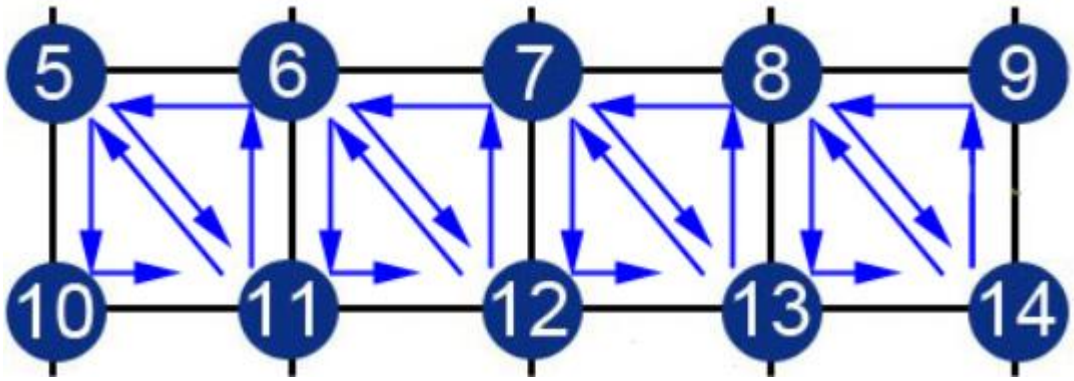

Рисунок 2.4 – Топология триангуляции

Теперь порядок перечисления будет сохранён против часовой стрелки. Топология подсчёта вершин решена.

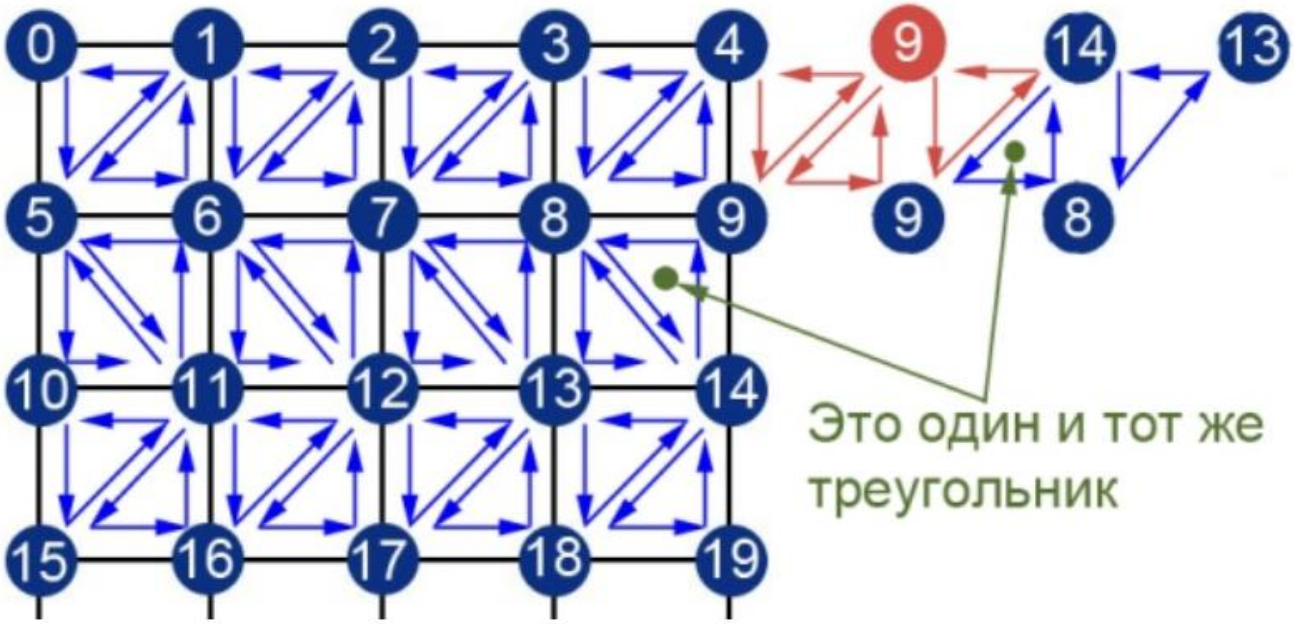

Рисунок 2.5 – Решение топологии

Функция Y отвечает за изменение плоскости в пространстве. Чтобы придать плоскости движение, зададим функцию Y, как:

$$y = \cos(k \times \Delta t + m \times (\partial x + \partial z)), \tag{36}$$

Где:

Δt - изменение времени;

∂x - шаг сетки по оси X;

∂z - шаг сетки по оси Z;

k,m –подобраны экспериментальным путём.

Алгоритмы работы программы

2.1 Алгоритм инициализации приложения

После инициализации приложения, создается окно и запускается цикл обработки сообщений.

При создании окна выполняется инициализации параметров поверхности и для каждого сообщения от таймера выполняется пересчет матрицы высот и отрисовка поверхности, как показано на рис(2.1) и рис (2.2).

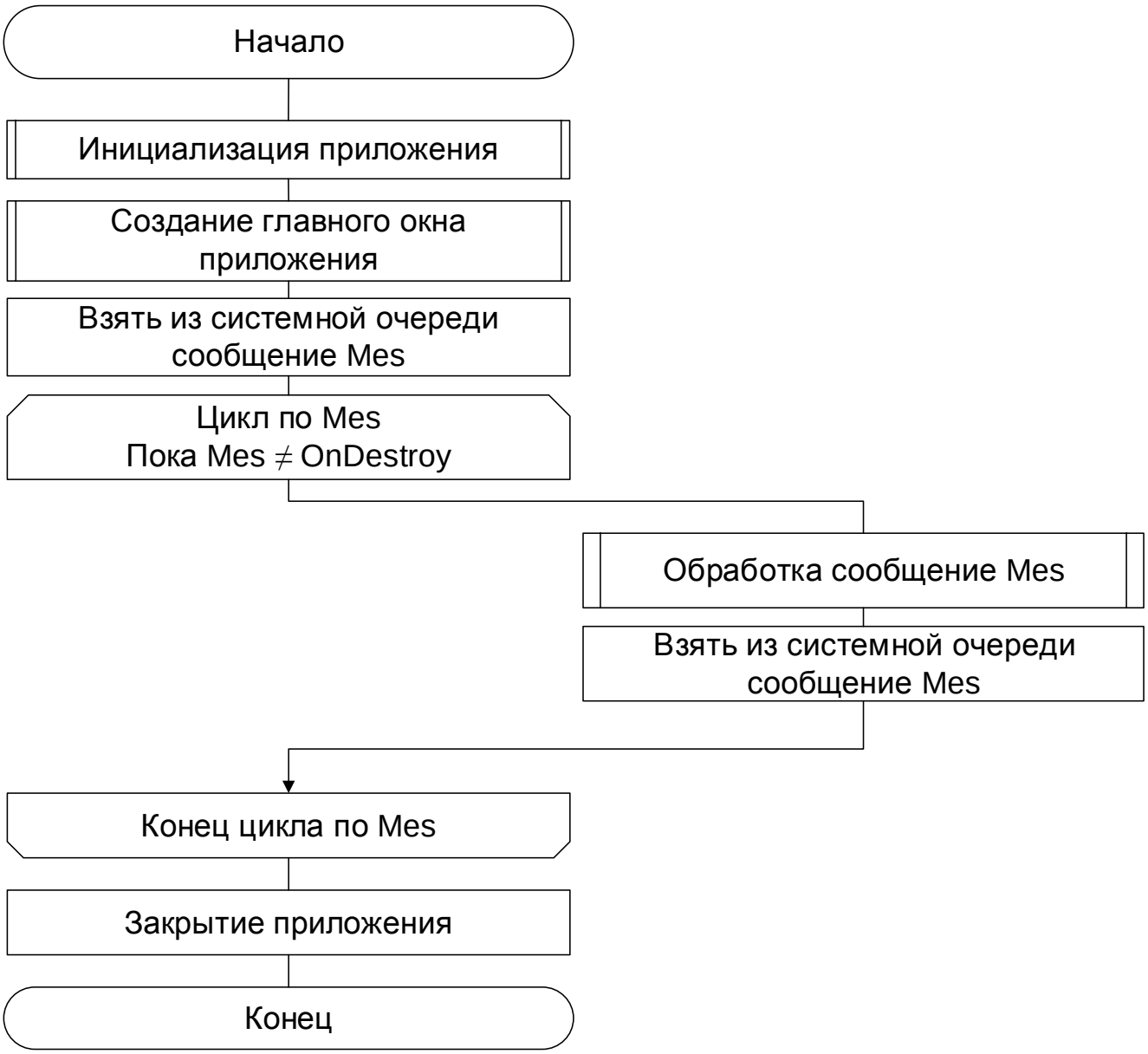

Рисунок 2.6 – Основной алгоритм

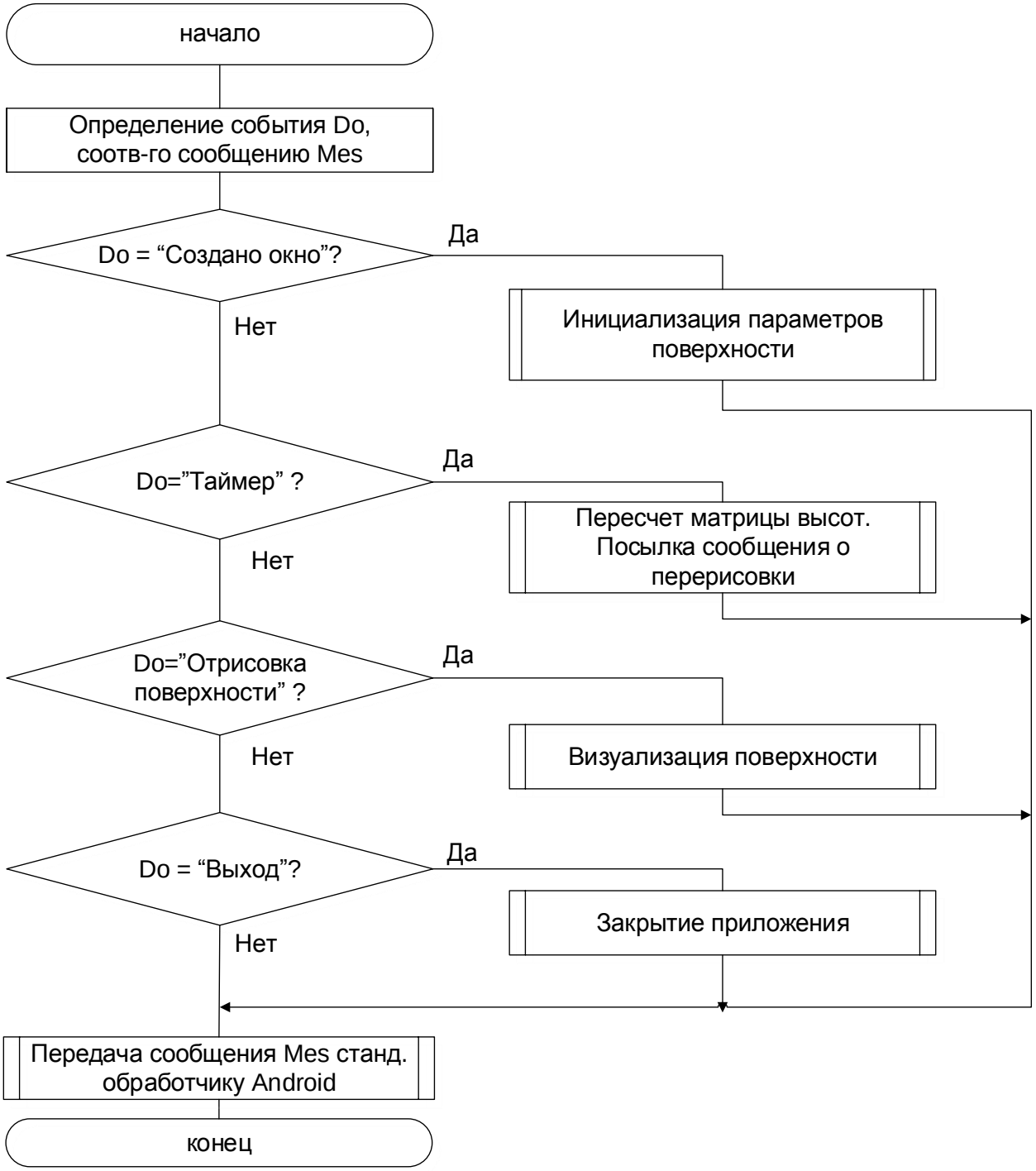

Рисунок 2.7 – Вспомогательный алгоритм

2.2 Алгоритм построения сетки

Создаём список вершин. За один проход по циклу вычисляем нормаль для каждой вершины и строим плоскость.

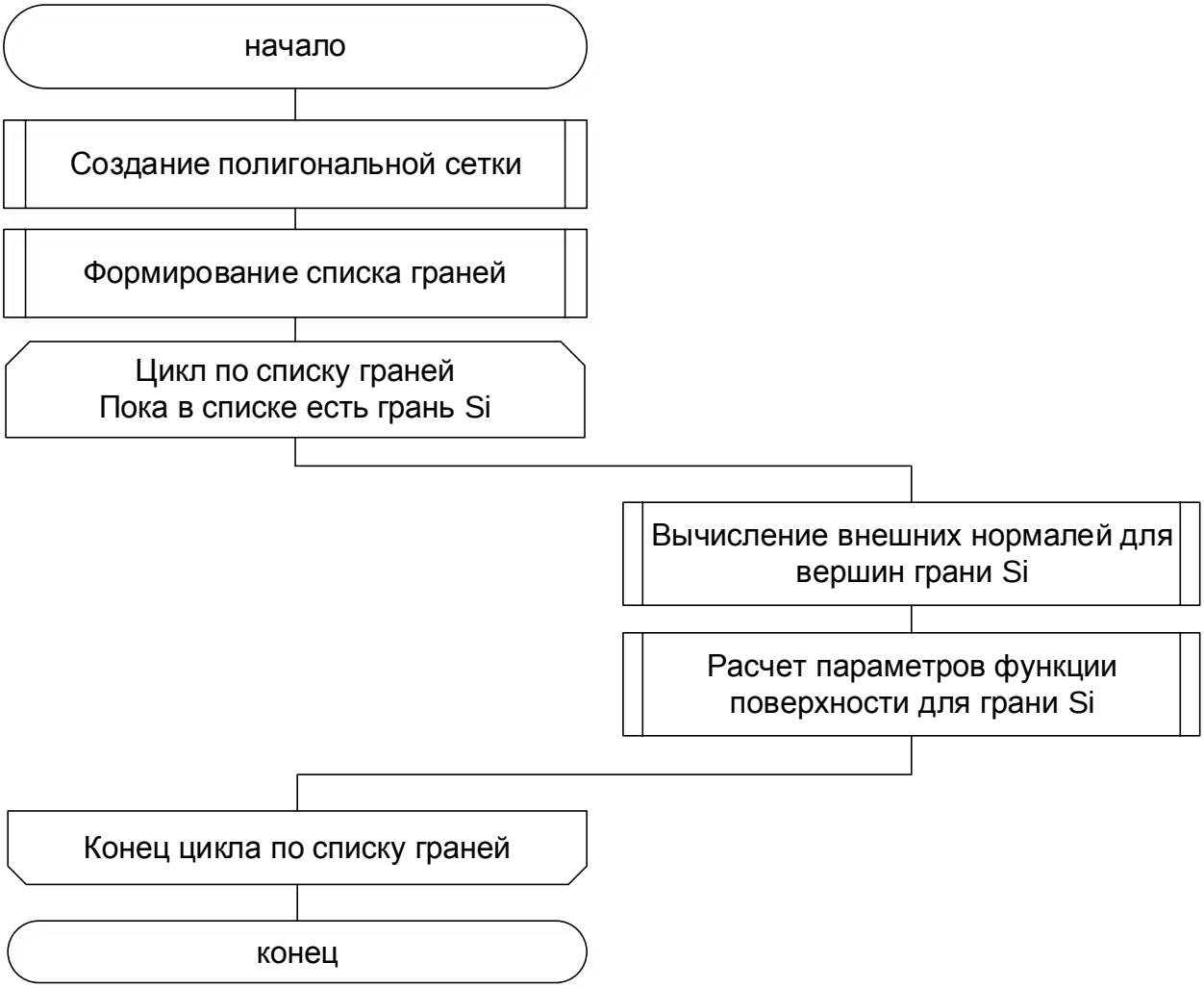

Рисунок 2.8 – Алгоритм построения сетки

3 ПРИМЕР РАБОТЫ ПРОГРАММЫ

Рисунок выполнен в .gif формате, чтобы показать динамику сглаженной поверхности.

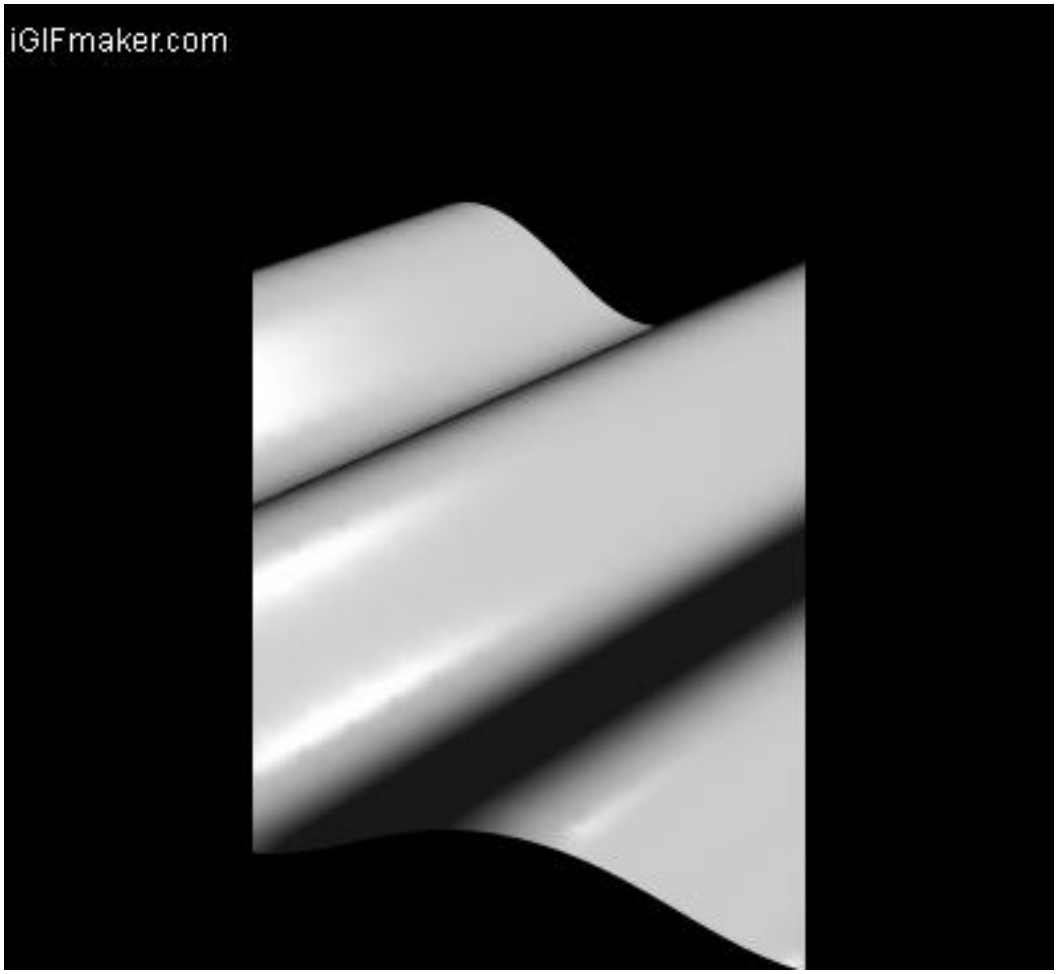

Рисунок 3.1 – пример работы программы

4 ЗАКЛЮЧЕНИЕ

Данная работа посвящена разработке сглаженной динамической поверхности, применяющейся для создания компьютерной программы. За главный метод была выбрана триангуляция полигональной сетки. Разработанная программа позволяет пользователю увидеть преимущества сглаженной поверхности, построенной с помощью полигональной сетки.

В ходе работы были решены следующие задачи:

1) выполнен обзор методов построения гладких поверхностей;
2) разработана математическая модель;
3) разработан алгоритм;
4) выполнена программная реализация.

Разработанное приложение имеет следующие преимущества:

1) достигается высокая скорость работы;
2) требует меньше памяти;
3) можно добиться построения трёхмерной фигуры, изменив одну функцию.

К недостаткам можно отнести то, что программа не сможет построить трёхмерную фигуру с полярными и сферическими координатами.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. В.И Корнеев Интерактивный графические системы // Москва БИНОМ. Лаборатория знаний, 2012. №4 – С.138-145 [1]
2. Полигональная сетка – Дата обновления 27 июня 2005г.URL: https://gamedev.ru/code/terms/Polymesh (дата обращения 14.04.2018) [2]
3. D. Blythe, "The Direct3D 10 System," ACM Transactions on Graphics, vol. 25, no. 3, pp. 724–734, 2006. [3]
4. J. Vad'ura, "Parallel mesh decimation with GPU," 2011[4]
5. Advanced Topics in Computer Graphics John C. Hart Antialiasing [5]
6. Jacek Lebie , Simple algorithm for antialiased scan conversion of straight line segments, 21-35, 2007[6]
7. Аппроксимация сплайнами – Дата обновления 07.12.2016 URL: https://habr.com/post/314218/ (дата обращения 12.05.2018)[7]
8. C. DeCoro, N. Tatarchuk, "Real–time Mesh Simplification Using GPU," in Proceedings of the symposium on interactive 3D graphics and games, 2007.[8]
9. D. Blythe, "The Direct3D 10 System," ACM Transactions on Graphics, vol. 25, no. 3, pp. 724–734, 2006. [9]
10. Построение сплайнов – Дата обновления 21.08.09 URL: http://lectoriy.mipt.ru/file/synopsis/pdf/Maths-NumAnalysis-M10-Aristova-141105.02.pdf (дата обращения 24.05.2018)[10]
11. P. Lindstrom, "Out–of–core simplification of large polygonal models," in Proceedings of the 27th annual conference on Computer graphics and interactive techniques, 259– 262, 2007. [11]
12. J. Rossignac, P. Borrel, "Multi–resolution 3D approximations for rendering complex scenes.," Modeling in Computer Graphics: Methods and Applications, pp. 455–465, 2009. [12]
13. Аппроксимация кубическими сплайнами-Дата обновления 22.05.2017 URL: http://alglib.sources.ru/interpolation/leastsquares.php#splinefit (дата обращения 28.04.2018)[13]
14. Шалагинов А.В. Кубическая сплайн экстраполяция временных рядов / 2010 – 397с [14]

```
        import android.opengl.Matrix;
import android.content.Context;
import java.nio.ByteOrder;
import java.nio.FloatBuffer;

import javax.microedition.khronos.egl.EGLConfig;
import android.opengl.GLSurfaceView;
import javax.microedition.khronos.opengles.GL10;

import java.nio.ShortBuffer;
import android.opengl.GLES20;

import java.nio.ByteBuffer;


public class MyClassRenderer implements GLSurfaceView.Renderer{

    private Context context;

    private float[] M_Matrix, V_Matrix, MV_Matrix, P_Matrix, MVP_Matrix;
    private float start_x =-1f;
    private float start_z =-1f;
    private float LightPosition_X, LightPosition_Y, LightPosition_Z;
    private float XCAM, YCAM, ZCAM;
    private float [][]vec_normZ, vec_normX, vec_normY;
    private float [] vertex;
    private int cout_ind =0, imax=50, jmax=50;
    private float [][] Shell_y;
    private float StepShell_dx =0.05f, StepShell_dz =0.05f;
    private float [] Shell_x,Shell_z;
    private Shader Matr_shade;
    private FloatBuffer vertexBuffer, normalBuffer;
    private ShortBuffer indexBuffer;
    private float [] vec_normal;

    //-------------------------------------------------------------------------------
--------------
//конструктор
    public MyClassRenderer(Context context) {
        this.context=context;
        LightPosition_X=5f;
        LightPosition_Y=5f;
        LightPosition_Z=5f;
        //матрицы
        M_Matrix =new float[18];
        V_Matrix =new float[18];
        MV_Matrix =new float[18];
        P_Matrix =new float[18];
        MVP_Matrix =new float[18];
        Matrix.setIdentityM(M_Matrix, 0);
        XCAM=0.4f;
        YCAM=1.8f;
        ZCAM=1.6f;
        Shell_x =new float [imax+1];
        Shell_z =new float [jmax+1];
        Shell_y =new float [jmax+1][imax+1];
        vertex=new float[(jmax+1)*(imax+1)*3];
```

```
        vec_normX =new float[jmax+1][imax+1];
        vec_normY =new float[jmax+1][imax+1];
        vec_normZ =new float[jmax+1][imax+1];
        vec_normal =new float[(jmax+1)*(imax+1)*3];
        Matrix.setLookAtM(V_Matrix, 0, XCAM, YCAM, ZCAM, 0, 0, 0, 0, 1, 0);
        Matrix.multiplyMM(MV_Matrix, 0, V_Matrix, 0, M_Matrix, 0);
        for (int i=0; i<=imax; i++){
            Shell_x[i]= start_x +i* StepShell_dx;
        }
        for (int j=0; j<=jmax; j++){
            Shell_z[j]= start_z +j* StepShell_dz;
        }
        ByteBuffer variable = ByteBuffer.allocateDirect((jmax+1)*(imax+1)*3*4);
        variable.order(ByteOrder.nativeOrder());
        vertexBuffer = variable.asFloatBuffer();
        vertexBuffer.position(0);
        ByteBuffer nb = ByteBuffer.allocateDirect((jmax+1)*(imax+1)*3*4);
        nb.order(ByteOrder.nativeOrder());
        normalBuffer = nb.asFloatBuffer();
        normalBuffer.position(0);
        short[] index;
        cout_ind =2*(imax+1)*jmax + (jmax-1);
        index = new short[cout_ind];
        int count=0;
        int temp=0;
        while (temp < jmax) {
            for (int i = 0; i <= imax; i++) {
                index[count] = change(temp,i);count++;
                index[count] = change(temp+1,i);count++;
            }
            if (temp < jmax-1){
                index[count] = change(temp+1,imax);count++;
            }
            temp++;
            if (temp < jmax){
                for (int i = imax; i >= 0; i--) {
                    index[count] = change(temp,i);count++;
                    index[count] = change(temp+1,i);count++;
                }
                if (temp < jmax-1){
                    index[count] = change(temp+1,0);
                    count++;
                }
                temp++;
            }
        }
        ByteBuffer variable_2 = ByteBuffer.allocateDirect(cout_ind * 2);
        variable_2.order(ByteOrder.nativeOrder());
        indexBuffer = variable_2.asShortBuffer();
        indexBuffer.put(index);
        indexBuffer.position(0);
        getShellVertex();
        getVertexNormal();
    }//конец конструктора
    //-------------------------------------------------------------------------
-------------

    private short change(int j, int i){
        return (short) (i+j*(imax+1));
    }
    //-------------------------------------------------------------------------
-------------
```

```
    private void getShellVertex(){
        double Change_Y=System.currentTimeMillis();
        for (int j=0; j<=jmax; j++){
            for (int i=0; i<=imax; i++){
                Shell_y[j][i]=0.2f*(float)Math.cos(0.005*Change_Y+5*(Shell_z[j]+
Shell_x[i]));
            }
        }
        int count=0;
        for (int j=0; j<=jmax; j++){
            for (int i=0; i<=imax; i++){
                vertex[count]= Shell_x[i];count++;
                vertex[count]= Shell_y[j][i];count++;
                vertex[count]= Shell_z[j];count++;
            }
        }
        vertexBuffer.put(vertex);
        vertexBuffer.position(0);
    }//конец метода
    //-------------------------------------------------------------------------------
-------------
    private void getVertexNormal(){
        for (int Shell_dj=0; Shell_dj<jmax; Shell_dj++){
            for (int Shell_di=0; Shell_di<imax; Shell_di++){
                vec_normX[Shell_dj] [Shell_di] = - ( Shell_y[Shell_dj]
[Shell_di+1] - Shell_y[Shell_dj] [Shell_di] ) * StepShell_dz;
                vec_normY[Shell_dj] [Shell_di] = StepShell_dx * StepShell_dz;
                vec_normZ[Shell_dj] [Shell_di] = -StepShell_dx * (
Shell_y[Shell_dj+1] [Shell_di] - Shell_y[Shell_dj] [Shell_di] );
            }
        }
        for (int Shell_dj=0; Shell_dj<jmax; Shell_dj++){
            vec_normX[Shell_dj] [imax] = ( Shell_y[ Shell_dj ] [ imax -1] -
Shell_y[ Shell_dj ] [ imax] ) * StepShell_dz;
            vec_normY[Shell_dj] [imax] = StepShell_dx * StepShell_dz;
            vec_normZ[Shell_dj] [imax] = -StepShell_dx * ( Shell_y[ Shell_dj+1 ] [
imax] - Shell_y[ Shell_dj ] [ imax ] );
        }
        for (int Shell_di=0; Shell_di<imax; Shell_di++){
            vec_normX[jmax] [ Shell_di ] = - ( Shell_y[ jmax ] [ Shell_di+1 ] -
Shell_y[ jmax ] [ Shell_di ] ) * StepShell_dz;
            vec_normY[jmax] [ Shell_di ] = StepShell_dx * StepShell_dz;
            vec_normZ[jmax] [ Shell_di ] = StepShell_dx * ( Shell_y[ jmax-1 ] [
Shell_di ] - Shell_y[ jmax ] [ Shell_di ] );
        }
        vec_normX[jmax] [ imax ]= (Shell_y[ jmax] [ imax-1] - Shell_y[ jmax]
[imax]) * StepShell_dz;
        vec_normY[jmax] [ imax ] = StepShell_dx * StepShell_dz;
        vec_normZ[jmax] [ imax ] = StepShell_dx * (Shell_y[jmax-1] [imax] -
Shell_y[jmax ] [imax]);
        int count=0;
        for (int j=0; j<=jmax; j++){
            for (int i=0; i<=imax; i++){
                vec_normal[count]= vec_normX[j][i];count++;
                vec_normal[count]= vec_normY[j][i];count++;
                vec normal[count]= vec normZ[j][i];count++;
            }
        }
        normalBuffer.put(vec_normal);
        normalBuffer.position(0);
    } // конец метода
    //-------------------------------------------------------------------------------
```

```
------------
    public void onSurfaceChanged(GL10 unused, int width, int height) {
        GLES20.glViewport(0, 0, width, height);
        float smfg = (float) width / height;
        float tempp=0.07f;
        float space_lt = -tempp*smfg;
        float space_rt = tempp*smfg;
        float space_bm = -tempp;
        float space_tp = tempp;
        float near = 0.05f;
        float far = 9.0f;
        Matrix.frustumM(P_Matrix, 0, space_lt, space_rt, space_bm, space_tp, near,
far);
        Matrix.multiplyMM(MVP_Matrix, 0, P_Matrix, 0, MV_Matrix, 0);
    } //конец метода
    //-------------------------------------------------------------------------------
------------
    public void onSurfaceCreated(GL10 unused, EGLConfig config) {
        GLES20.glEnable(GLES20.GL_DEPTH_TEST);
        GLES20.glEnable(GLES20.GL_CULL_FACE);
        GLES20.glHint(GLES20.GL_GENERATE_MIPMAP_HINT, GLES20.GL_NICEST);
        Matr_shade.linkVertexBuffer(vertexBuffer);
        Matr_shade.linkNormalBuffer(normalBuffer);
    }
    public void onDrawFrame(GL10 unused) {
        Matr_shade.linkModelViewProjectionMatrix(MVP_Matrix);
        Matr_shade.linkCamera(XCAM, YCAM, ZCAM);
        Matr_shade.linkLightSource(LightPosition_X, LightPosition_Y,
LightPosition_Z);
        getShellVertex();
        getVertexNormal();
        GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT
                | GLES20.GL_DEPTH_BUFFER_BIT);
        GLES20.glDrawElements(GLES20.GL_TRIANGLE_STRIP, cout_ind,
                GLES20.GL_UNSIGNED_SHORT, indexBuffer);
    }//конец метода
//-------------------------------------------------------------------------------
----------
}//конец класса
//-------------------------------------------------------------------------------
----------
import android.app.Activity;
import android.os.Bundle;


public class MainActivity extends Activity {

    private MyClassSurfaceView mGLSurfaceView;
    @Override
    public void onCreate(Bundle savedInstanceState){
        super.onCreate(savedInstanceState);
        mGLSurfaceView = new MyClassSurfaceView(this);
        setContentView(mGLSurfaceView);
    }
    @Override
    protected void onPause() {
        super.onPause();
        mGLSurfaceView.onPause();
    }
    @Override
    protected void onResume() {
        super.onResume();
```

```
        mGLSurfaceView.onResume();
    }
}
```

```
import android.content.Context;
import android.opengl.GLSurfaceView;

public class MyClassSurfaceView extends GLSurfaceView{
    private MyClassRenderer renderer;
    public MyClassSurfaceView(Context context) {
        super(context);
        setEGLContextClientVersion(2);
        renderer = new MyClassRenderer(context);
        setRenderer(renderer);
        setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);

    }
}
```