# **Определение**

**Дизеринг (англ. *dither*), псевдотонирование** — при [обработке цифровых сигналов](https://ru.wikipedia.org/wiki/%D0%A6%D0%B8%D1%84%D1%80%D0%BE%D0%B2%D0%B0%D1%8F_%D0%BE%D0%B1%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D1%81%D0%B8%D0%B3%D0%BD%D0%B0%D0%BB%D0%BE%D0%B2) представляет собой подмешивание в первичный сигнал псевдослучайного шума со специально подобранным спектром. Применяется при обработке цифрового звука, видео и графической информации для уменьшения негативного эффекта от [квантования](https://ru.wikipedia.org/wiki/%D0%9A%D0%B2%D0%B0%D0%BD%D1%82%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_(%D0%BE%D0%B1%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D1%81%D0%B8%D0%B3%D0%BD%D0%B0%D0%BB%D0%BE%D0%B2)).

|  |  |  |
| --- | --- | --- |
| В компьютерной графике дизеринг используется для создания иллюзии [глубины цвета](https://ru.wikipedia.org/wiki/%D0%93%D0%BB%D1%83%D0%B1%D0%B8%D0%BD%D0%B0_%D1%86%D0%B2%D0%B5%D1%82%D0%B0) для изображений с относительно небольшим количеством цветов в палитре. Отсутствующие цвета составляются из имеющихся путем их «перемешивания».  Например, если необходимо получить отсутствующий в палитре фиолетовый цвет, его можно получить, разместив красные и синие пиксели в шахматном порядке; серый цвет может быть составлен из черных и белых точек. |  |  |

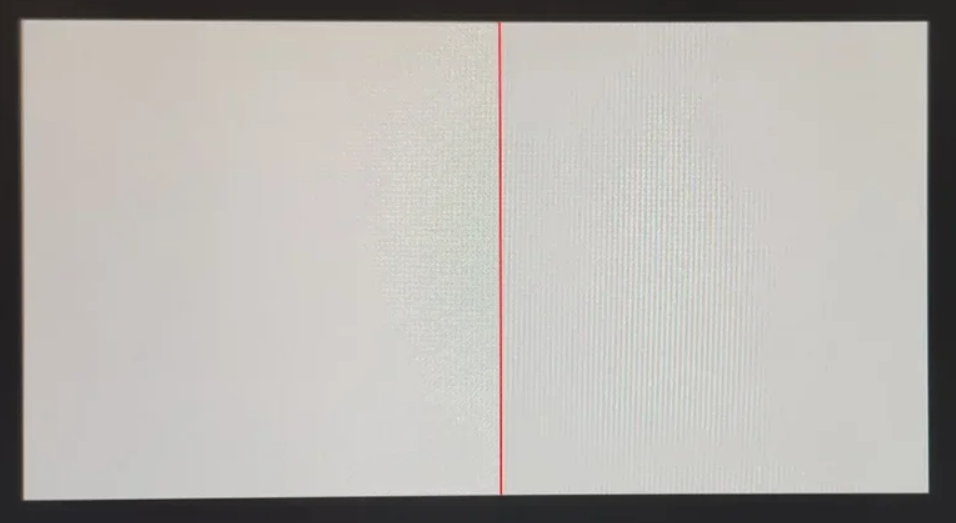
При оптимизации изображений путём уменьшения количества цветов, применение дизеринга приводит к визуальному улучшению изображения (однако для отдельных сжатых форматов (например, [PNG](https://ru.wikipedia.org/wiki/PNG)), увеличивает его размер). Дизеринг также находит применение в веб-дизайне, где этот полезный метод используется для сокращения числа цветов изображения, что уменьшает размер файла (и трафик) без ущерба для качества. Он также используется при уменьшении цифровых фотографий в формате RAW в 48 или 64 бита на пиксель до RGB в 24 бита на пиксель для редактирования.

Однако уменьшение количества цветов практически всегда приводит к появлению специфических эффектов. Обычные фотографии могут иметь тысячи и даже миллионы различающихся цветов и оттенков, и преобразование их в индексированный формат с фиксированной палитрой приводит к потере огромного количества информации о цвете.

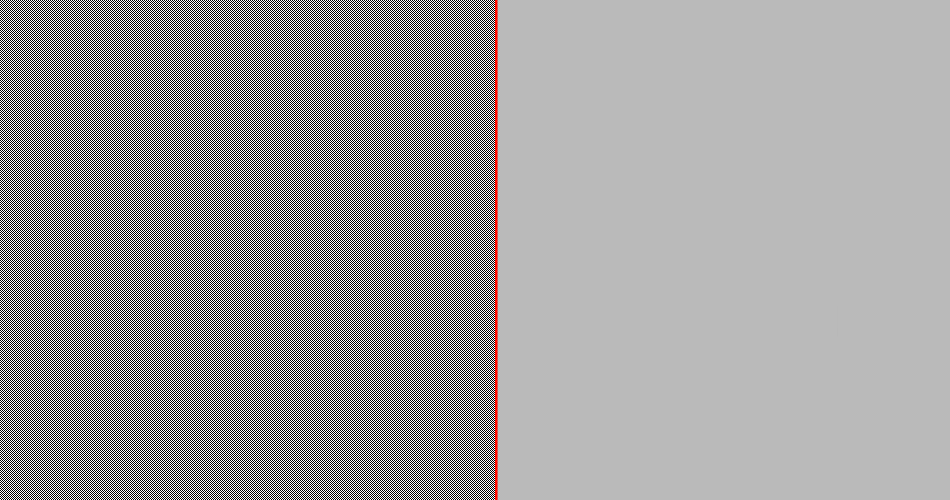
В большинстве недорогих настольных мониторов на основе TN- (и некоторых \*VA) матриц, а также во всех дисплеях ноутбуков используются матрицы с 18-битным цветом (6 бит на каждый RGB-канал), 24-битность эмулируется мерцанием цвета пикселов между их ближайшими цветами в 6 битной разрядности и/или незаметным глазу дизерингом. Стоит отметить, что очень часто производители мониторов не указывают в характеристиках “битность” мониторов. Как вариант, можно попробовать найти свои монитор на сайте [DisplaySpecifications - Specifications and features of desktop monitors and TVs](https://www.displayspecifications.com/) и там в пункте “Panel bit depth” посмотреть на битность монитора: 10 / 8 bits означает, что стоит честная матрица с 30 / 24-битным цветом соответственно, а вот 10 (8 + FRC) / 8 (6 +FRC) означает, что полная битность эмулируется.

Frame rate control (FRC) - это метод, который позволяет пикселям отображать больше оттенков цвета. С помощью быстрых циклических переключений между различными цветовыми оттенками создается иллюзия нового промежуточного цветового оттенка. Например, при помощи FRC 6-битная матрица экрана способна отображать характерные для 8-битной матрицы 16.7 миллионов цветов вместо стандартных для нее 262200 цветов. Быстро меняя яркость подсветки меняется оттенок изображения, добавляется количество оттенков. Поскольку особых затрат для этого не надо но позволяет позиционировать телевизор или монитор как устройство более высокого класса.

Рассмотрим пример ниже. Левая часть картинки представляет собой заливку по типу шахматной доски: чередование черного и белого пикселя. Правая часть - заливка яркостью 186 (что равняется фактической яркости 128 с учетом гамма-коррекции со степенью 2.2). Такое сгенерированное изображение будет выглядеть на экране следующим образом:



Если же рассматривать в масштабе не 100% или пытаться вставлять картинку во всякие документы, то будет видно, что левая часть закрашена неким паттерном, что связано с применением алгоритмов масштабирования, о которых будет в следующих лекциях.



Пример выше демонстрирует применение дизеринга на уровне отображения на мониторе.

Теперь поговорим о самих алгоритмах дизеринга.

Все алгоритмы дизеринга условно можно поделить на 2 вида: алгоритмы с рассеиванием ошибки (**Error diffusion**) и упорядоченные алгоритмы (**Ordered**).

## Определение пороговых цветов для битностей

Но перед описанием самих алгоритм определим, как именно будут определяться наши пороговые (threshold) цвета для наших битностей: для округления текущего значения цвета до ближайшего, который можно отобразить в задаваемой битности **B**, из целочисленного значения цвета берутся **B** старших бит и дублируются сдвигами по **B** бит в текущее значение цвета.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Порядок бит в значении цвета: | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| Битность **В** | Маска изменения порядка бит |
| 8 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 5 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 7 | 6 | 5 | 4 | 3 | 7 | 6 | 5 | |
| 2 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 7 | 6 | 7 | 6 | 7 | 6 | 7 | 6 | |
| 1 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | |

Рассмотрим пример для **В=2**

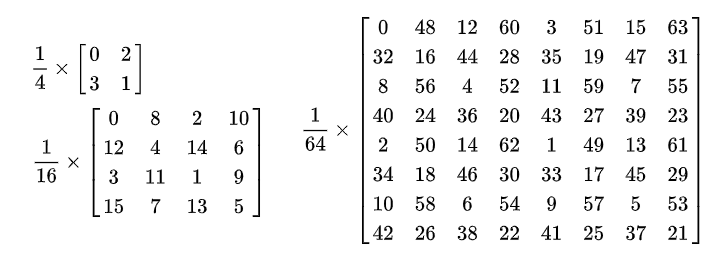
|  |  |  |  |
| --- | --- | --- | --- |
| Значение цвета в | Значение цвета в | Перестановка бит | Перестановка бит |
| 0 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | **0** | **0** | 0 | 0 | 0 | 0 | 0 | 0 | | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 |
| 50 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | **0** | **0** | 1 | 1 | 0 | 0 | 1 | 0 | | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 |
| 100 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | **0** | **1** | 1 | 0 | 0 | 1 | 0 | 0 | | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | | 85 |
| 150 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | **1** | **0** | 0 | 1 | 0 | 1 | 1 | 0 | | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | | 170 |
| 200 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | **1** | **1** | 0 | 0 | 1 | 0 | 0 | 0 | | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | 255 |
| 255 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | **1** | **1** | 1 | 1 | 1 | 1 | 1 | 1 | | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | 255 |

# 

# **Ordered dithering**

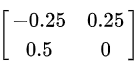
Алгоритм уменьшает количество цветов, применяя карту порогов **M** (другое обозначение: Bayer matrix) к отображаемым пикселям, в результате чего некоторые пиксели меняют цвет в зависимости от расстояния исходного цвета от доступных записей цветов в уменьшенной палитре.

Пороговые карты **M** бывают разных размеров:



Здесь представлено 3 карты: 2х2, 4х4, 8х8. Причем, все элементы матрицы должны быть представлены в диапазоне (0, 1]. Для удобства, нормирующий множитель был вынесен непосредственно перед матрицей для удобства.

А теперь как пользоваться данными матрицами на примере матрицы **2х2**:

1. Пусть Mint - матрица, заполненная целыми числами [0, 2^2-1=3]
2. Для уменьшения дальнейших вычислений матрицу можно хранить, сразу поделив на нормирующий множитель:
3. Дополнительно, для улучшения визуального эффекта . Тогда наша матрица примет следующий вид: 

После вычисления матрицы порогов (или же простого хардкода итоговой матрицы), можно переходить к алгоритму.

Алгоритм смещает для каждого пикселя его значение цвета на соответствующее значение из карты порогов **M** в соответствии с его местоположением, в результате чего значение пикселя квантуется на другой цвет, если оно превышает пороговое значение.

Для большинства случаев сглаживания достаточно просто добавить пороговое значение к каждому пикселю или *эквивалентно* сравнить значение этого пикселя с порогом: если значение пикселя меньше, чем число в соответствующей ячейке матрицы, записать в пиксель черный цвет, в противном случае, белый в случае битности 1.

Алгоритм выполняет следующее преобразование для каждого цвета c каждого пикселя:

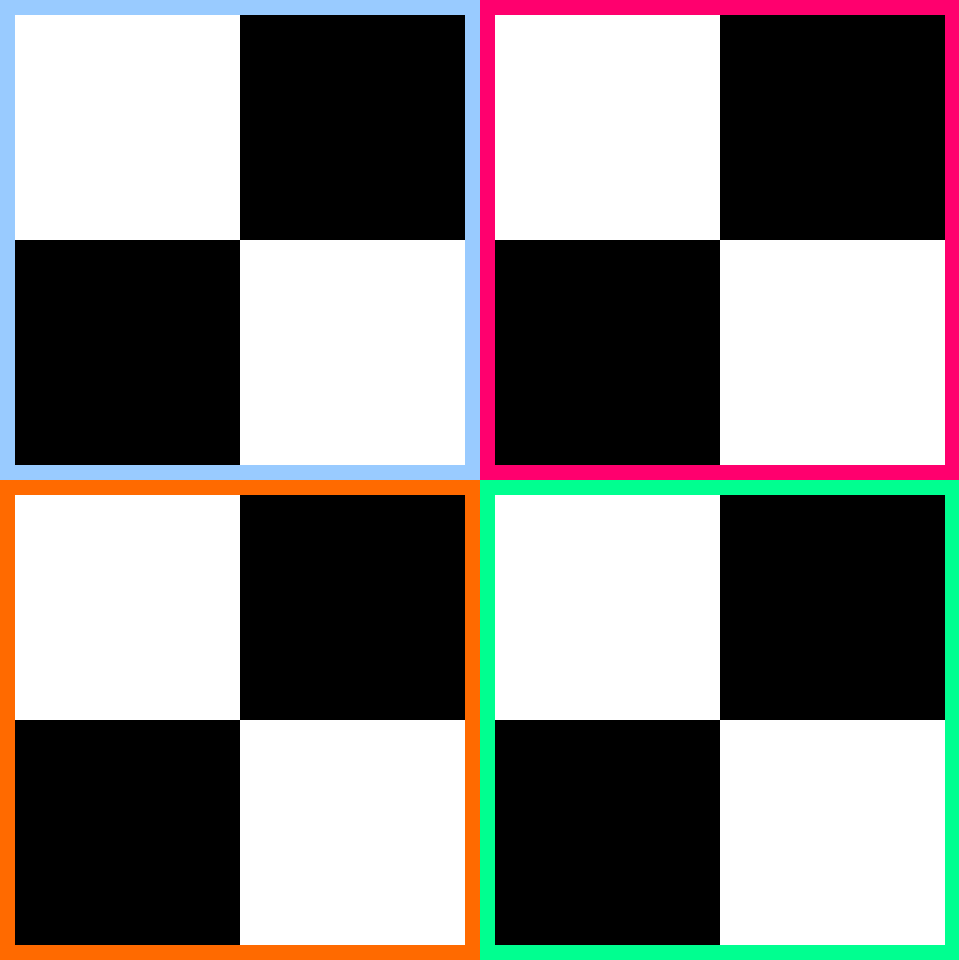
,

где:

* - старый цвет пикселя
* - элемент карты порогов
* - функция, возвращающая ближайший цвет к подаваемому, который можно [отобразить на текущей палитре](#_rj4j1g3h32y8)
* - новый цвет пикселя в текущей палитре

Значения, считанные из карты порогов, должны масштабироваться в том же диапазоне, что и . Для этого в формуле вводится .

По сути, наша квадратная матрица накладывается на изображение и для расчета значения цвета одного пикселя требуется взять соответствующий элемент матрицы. Так, на рисунке ниже, представляющем собой картинку 4х4 пикселей, продемонстрирован пример наложения матрицы 2х2 (для удобства все наложения выделены рамками разных цветов).



Пример применения дизеринга ordered 4x4 к горизонтальному градиенту:



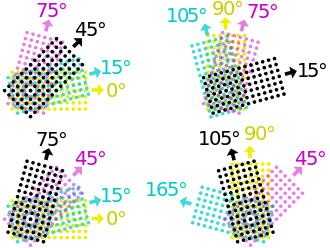
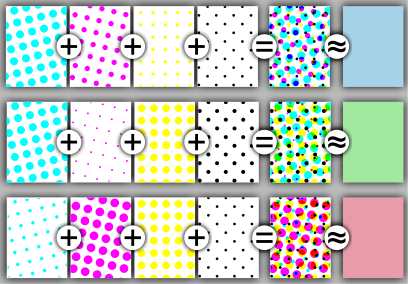
Поскольку алгоритм работает с одиночными пикселями и не имеет условных операторов, он очень быстрый и подходит для преобразований в реальном времени. Кроме того, расположение шаблонов сглаживания всегда остается одинаковым относительно кадра дисплея, что способствует улучшению сжатия изображения. Упорядоченное сглаживание больше подходит для линейной графики, так как приводит к более прямым линиям и меньшему количеству аномалий. Однако, результат, получаемый после работы данного метода получаются хуже, чем после применения алгоритмов с рассеянием ошибок, о которых будет изложено далее.

# **Halftone**

Полутонирование - создание изображения со многими уровнями серого или цвета (т.е. слитный тон) на аппарате с меньшим количеством тонов, обычно чёрно-белый принтер. В принципе, задача в том чтобы уменьшить разрешение, увеличивая видимую глубину тона (так называемое пространственное полутонирование).

Полутона широко используются для печати цветных изображений. Общая идея: изменяя плотность четырех вторичных цветов печати: голубого, пурпурного, желтого и черного (сокращение CMYK), можно воспроизвести любой конкретный оттенок.

В этом случае может возникнуть дополнительная проблема. В простом случае можно создать полутон, используя те же методы, что и для печати оттенков серого, но в этом случае разные цвета печати должны оставаться физически близко друг к другу, чтобы обмануть глаза и подумать, что они одного цвета. Чтобы сделать это, промышленность стандартизировала набор известных углов, которые приводят к образованию точек в виде маленьких кружков или розеток.



В случае обработки цифрового изображения halftone представляет собой матрицы порогов (для различных углов поворота), позволяющие воспроизводить это “точки” как при печать изображения.

# **Error diffusion algorithms**

Алгоритмы с распространением (рассеиванием) ошибок распределяют остаток квантования по соседним пикселям, которые еще не были обработаны.

В отличие от предыдущих алгоритмов, данные алгоритмы работают с некоторой окрестностью пикселя:то, что алгоритм делает в одном месте, влияет на то, что происходит в других местах. Это означает, что требуется буферизация и усложняет параллельную обработку.

Распространение ошибок имеет тенденцию усиливать края на изображении. Это может сделать текст на изображениях более читабельным, чем в других методах.

Рассмотрим пример: мы хотим уменьшить битность нашего изображения с 8 до 2х. То есть, наша новая палитра выглядит следующим образом:

Пусть у нас есть пиксель цвета 96. Используя ранее определенный [алгоритм](#_rj4j1g3h32y8) для округления к ближайшему цвету на доступной палитре, определим, что ближайший цвет к 96 будет 85.

На этом этапе стандартный подход - просто перемещаться к следующему пикселю и выполнять то же сравнение. Но проблема возникает, если у нас есть куча подобных серых пикселей со значением 96 - все они превращаются серые. У нас получается с огромный кусок серых пикселей, которые плохо представляют оригинальный серый цвет.

Рассеивание ошибок следует более умному подходу к проблеме. Как вы могли предположить, рассеивание ошибок работает путем «рассеивания» (распространения) ошибки каждого вычисления в соседние пиксели. Если алгоритм находит серый пиксель со значением 96, он также определяет, что 96 ближе к 85, и поэтому делает пиксель темно-серым. Но тогда алгоритм учитывает «ошибку» в его преобразовании. В частности, ошибку в том, что преобразованный пиксель на самом деле был на расстоянии в 11 шагов от темно-серого.

Когда мы перемещаемся к следующему пикселю, алгоритм рассеивания ошибок добавляет ошибку предыдущего пикселя к текущему пикселю. Если следующий пиксель имеет серый цвет 118, вместо того, чтобы сделать его темно-серым, алгоритм добавляет ошибку 11 из предыдущего пикселя. Это приводит к значению 129, которое на самом деле ближе к 170. Таким образом, алгоритм делает этот пиксель светло-серым и снова учитывает ошибку. В этом случае ошибка составляет −41, потому что 129 на 41 меньше, чем 170 — то значение, на которое этот пиксель поменяли.

Это куда лучше, чем окрашивать все пиксели подряд чёрным. Как правило, когда мы заканчиваем обработку строки изображения, мы отбрасываем значение ошибки, которое мы отслеживали, и начинаем заново с ошибкой «0» со следующей строки изображения.

Однако это был случай одномерного алгоритма, который хотелось бы расширить до двумерного для последующего применения к изображениям.

Для простоты расчетов все стандартные формулы дизеринга продвигают ошибку только вперед. Если обойти изображение попиксельно, начиная с верхнего левого угла и двигаясь вправо, учитывать ошибки назад (например, влево и/или вверх) необходимости не будет.

Таким образом, для стандартного поведения цикла (начиная с верхнего левого угла изображения и двигаясь вправо) мы хотим, чтобы движение пикселей шло только вправо и вниз.

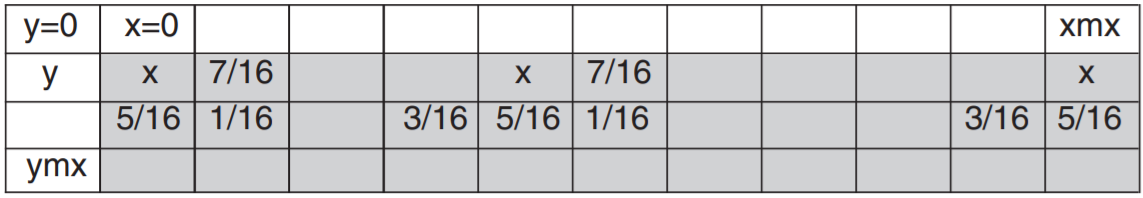
Таким образом, алгоритмы с рассеянием ошибки действуют следующим образом:

1. Обход по пикселям совершается слева направо, сверху вниз
2. При анализе текущего пикселя ():
   1. определяется его новый цвет через
   2. рассчитывается ошибка как
   3. затем ошибка распространяется согласно используемому паттерну/алгоритму (см. далее)

# **Floyd–Steinberg**

Первая и возможно самая известная формула рассеивания ошибок была опубликована Робертом Флойдом и Луисом Стейнбергом в 1976 году. Рассеивание ошибок происходит по следующей схеме:

* **х** - текущий пиксель, от которого распространяется ошибка
* **y,x** -строка/столбец изображения
* **ymx, xmx** - номер последние строки и столбца



Вернёмся к нашему оригинальному примеру преобразования пиксельного значения 96. Как отмечалось ранее, мы получаем ошибку 11 при окрашивании пикселя в темно-серый. Мы распространяем эту ошибку окружающим пикселям, поделив 11 на 16 ( = 0,6875), затем умножаем её на соответствующие значения, например:

|  |  |  |
| --- | --- | --- |
|  | Х | +7 \* 0,6875 |
| +3 \* 0,6875 | +5 \* 0,6875 | +1 \* 0,6875 |

Если наш пиксель округляется в большее значение (129 к 170 например), то ошибка будет отрицательной соответственно.

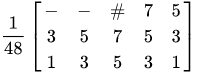
Путем распространения ошибки нескольким пикселям, каждому с различным значением, мы сводим к минимуму все отвлекающие полосы с точками, заметные в исходном примере алгоритма рассеивания ошибок. Вот изображение куба с применением алгоритма Флойда-Стейнберга:

Алгоритм даёт достаточно хорошее качество, а также требует только один передний массив (одномерный массив шириной в изображение, где хранятся значения ошибок, распространяемые к следующей строке). Кроме того, поскольку его делитель 16, вместо деления можно использовать битовые сдвиги. Так алгоритм достигает высокой скорости работы даже на старом оборудовании.

Что касается значений 1/3/5/7, используемых для распространения ошибки – они были выбраны специально, потому что они создают равномерный клетчатый узор для серого изображения.

# **Jarvis, Judice, Ninke**

В год, когда Флойд и Стейнберг опубликовали свой знаменитый алгоритм дизеринга, был издан менее известный, но гораздо более мощный алгоритм. Фильтр Джарвиса, Джудиса и Нинке значительно сложнее, чем Флойда-Стейнберга:



При таком алгоритме ошибка распределяется на в три раза больше пикселей, чем у Флойда-Стейнберга, что приводит к более гладкому и более тонкому результату.

# **Atkinson**

Формула Аткинсона немного отличается от других в этом списке, потому что она распространяет только часть ошибки, не всю целиком. В современных графических приложениях этот метод встречается под именем «уменьшить выцветание». Рассеивание только части ошибок помогает уменьшить зернистость, но непрерывные светлые и темные участки изображения могут потерять цвет.

# **Random dithering**

Интересное напоследок)

Данный алгоритм можно также отнести к типу ordered с тем условием, что для определения добавки к значению текущего пикселя берется не элемент матрицы, а случайное число из диапазона (0.1].

# **Источники и полезные ссылки:**

1. [Дизеринг](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D0%B7%D0%B5%D1%80%D0%B8%D0%BD%D0%B3)
2. <https://en.wikipedia.org/wiki/Dither>
3. <https://en.wikipedia.org/wiki/Frame_rate_control>
4. [» Глубина цвета в мониторах (8bit+A-FRC) 10bit, 10bit (8bit+FRC), 12 bit](https://tab-tv.com/?p=14227)
5. <http://www.imagemagick.org/Usage/quantize/>
6. <https://habr.com/ru/post/326936/>
7. [Ordered dithering](https://en.wikipedia.org/wiki/Ordered_dithering)
8. [Ordered Dithering - Computerphile](https://www.youtube.com/watch?v=IviNO7iICTM)
9. [Ordered Dither Upgrade -- IM v6 Examples](http://www.imagemagick.org/Usage/bugs/ordered-dither/)
10. [Gernot Hoffmann Color Rendering Printing Scanning](http://docs-hoffmann.de/colrend290800.pdf)
11. [Error diffusion](https://en.wikipedia.org/wiki/Error_diffusion)
12. [ImageSharp/DHALF.TXT at master · SixLabors/ImageSharp](https://github.com/SixLabors/ImageSharp/blob/master/src/ImageSharp/Processing/Processors/Dithering/DHALF.TXT)
13. [Error Diffusion Dithering - Computerphile](https://www.youtube.com/watch?v=ico4fJfohMQ)