## 1. Аддитивная цветовая модель.

Аддитивная цветовая модель — это метод смешивания цветов, при котором свет разных длин волн комбинируется для получения новых цветов. Основными цветами в этой модели являются красный (Red), зеленый (Green) и синий (Blue), образующие систему RGB. При наложении всех трех цветов с максимальной интенсивностью получается белый свет.

Используется в устройствах, которые излучают свет, таких как мониторы, телевизоры и проекторы. Основной принцип модели — добавление световых потоков для создания разнообразных оттенков.

## 2. Субтрактивная цветовая модель

Субтрактивная цветовая модель — это способ формирования цветов путем вычитания (поглощения) определенных длин волн из белого света. Основными цветами в этой модели являются циан (Cyan), маджента (Magenta) и желтый (Yellow), что формирует систему CMY. При наложении всех трех цветов в идеале получается черный цвет.

Эта модель используется в системах печати, где цвет формируется за счет поглощения света красками или чернилами, а отраженный свет воспринимается глазом. Основной принцип — вычитание световых потоков для создания разнообразных оттенков.

## 3. Перцепционная цветовая модель

Перцепционная цветовая модель описывает, как человек видит и воспринимает цвета. Она учитывает, как наши глаза и мозг работают вместе, чтобы различать оттенки.

Например, модель CIE LAB использует три параметра: яркость (насколько светлый или темный цвет) и две оси для оттенков (одна показывает переходы между красным и зелёным, другая — между синим и жёлтым). Эта модель помогает сделать цвета максимально похожими на то, как мы видим их в реальной жизни, и часто применяется для точной передачи цветов между разными устройствами, например, экраном и принтером.

## 4. Растровая графика и ее особенности

Растровая графика — это способ представления изображения с помощью сетки из маленьких точек (пикселей). Каждый пиксель имеет свой цвет, и вместе они формируют изображение.

**Особенности растровой графики:**

1. **Зависимость от разрешения**: качество изображения зависит от количества пикселей (разрешения). При увеличении размеры могут терять чёткость (пикселизация).
2. **Объем данных**: высокое разрешение требует большого объёма памяти.
3. **Фотографическое качество**: подходит для фотографий и сложных изображений с множеством цветов.
4. **Форматы файлов**: популярные форматы — JPEG, PNG, BMP, GIF.

Растровая графика широко используется в фотографии, веб-дизайне и цифровом искусстве.

## 5. Векторная графика и ее особенности

Векторная графика — это способ представления изображений с помощью математических объектов: точек, линий, кривых и фигур. Вместо пикселей векторное изображение строится по формулам.

**Особенности векторной графики:**

1. **Масштабируемость**: изображения можно увеличивать и уменьшать без потери качества.
2. **Компактность**: файлы обычно занимают меньше памяти, чем растровые, особенно для простых изображений.
3. **Простота редактирования**: легко изменять размеры, форму и цвета объектов.
4. **Ограничения в детализации**: не подходит для фотографий и сложных изображений, так как трудно воспроизвести плавные градиенты и текстуры.
5. **Форматы файлов**: популярные форматы — SVG, AI, EPS.

Векторная графика используется для логотипов, шрифтов, иллюстраций и другой графики, где важна чёткость и возможность масштабирования.

## 6. Фрактальная графика и ее особенности

Фрактальная графика — это тип графики, в основе которой лежат математические формулы и алгоритмы. Изображения создаются путём многократного повторения (рекурсии) простых геометрических форм, что позволяет создавать сложные и детализированные структуры.

**Особенности фрактальной графики:**

1. **Основа — математика**: изображения описываются формулами, а не пикселями или объектами.
2. **Бесконечная детализация**: при увеличении изображения его детали продолжают прорисовываться без потери качества.
3. **Эффект самоподобия**: части изображения похожи на целое (фракталы имеют одинаковый вид при увеличении или уменьшении масштаба).
4. **Сложность создания**: требует вычислений и мощных алгоритмов, часто используются специализированные программы.
5. **Применение**: используется для создания уникальных узоров, генерации природных объектов (деревья, облака) и в научных исследованиях.

Фрактальная графика широко применяется в компьютерной графике, цифровом искусстве и моделировании природных явлений.

## 7. Форматы файлов растровых изображений

1. **JPEG (JPG)**: для фотографий, сжатие с потерями, подходит для интернета.
2. **PNG**: Прозрачность, сжатие без потерь, используется для веб-графики.
3. **GIF**: Анимация, ограничение 256 цветов, для простых анимаций.
4. **BMP**: Несжатый, занимает много места, редко используется.
5. **TIFF**: Высокое качество, для печати и сканирования.
6. **PSD**: Слои и эффекты, редактируемые проекты.

**Итог**: JPEG — для фото, PNG и GIF — для веба, TIFF и PSD — для профессиональной работы.

## 8. Форматы файлов векторных изображений

1. **SVG**: Открытый формат, поддерживает интерактивность, для веб-дизайна.
2. **AI**: Формат Adobe Illustrator, для профессионального дизайна.
3. **EPS**: Универсальный, для печати и макетов.
4. **PDF**: содержит текст и графику, удобен для передачи и печати.
5. **CDR**: Формат CorelDRAW, для сложных проектов.
6. **WMF/EMF**: Форматы Windows, для офисных приложений.

**Итог**: SVG и AI — для веба и дизайна, EPS и PDF — для печати, CDR — для работы в CorelDRAW.

## 9. Векторное и аффинное пространства

Это математическая структура, состоящая из множества векторов, для которых определены операции сложения и умножения на скаляр. Основные свойства:

1. Коммутативность и ассоциативность сложения.
2. Наличие нулевого вектора.
3. Умножение на скаляр не изменяет свойства пространства. Пример: пространство всех 2D-векторов на плоскости.

**Аффинное пространство:**

Это обобщение векторного пространства, в котором отсутствует фиксированная точка начала координат. Вместо этого используются точки и связанные с ними векторы. Свойства:

1. Любая точка определяется как сумма другой точки и вектора.
2. Сохраняются понятия прямых, плоскостей и параллельности, но нет координатной системы. Пример: геометрическая плоскость, где точка произвольна, а смещения задаются векторами.

**Различие:** в векторном пространстве есть начало координат (ноль), а в аффинном пространство определяется через точки и их относительное положение.

## 10. Основные системы координат

1. **Декартовая система**: Ось X (горизонтальная) и Y (вертикальная), иногда Z (в 3D). Используется для описания точек в плоскости или пространстве.
2. **Полярная система**: определяет точку радиусом (r) и углом (θ) относительно фиксированного начала. Применяется для круговых и радиальных объектов.
3. **Цилиндрическая система**: Расширение полярной, добавляющее ось Z для высоты. Используется в 3D для объектов с осевой симметрией.
4. **Сферическая система**: Точка определяется радиусом (r), углом от вертикали (φ) и горизонтальной плоскости (θ). Подходит для описания сферических объектов.

Каждая система выбирается в зависимости от задачи: декартовая для прямолинейных объектов, полярная и сферическая для круговых и объемных.
Эти системы перекрывают все типы симметрии и формы: прямолинейные, радиальные, осевые и объемные, что делает их основными в математике и науке.

## 11. Системы координат в компьютерной графике и связь между ними

В компьютерной графике используются **локальная**, **мировая**, **камерная** и **экранная** системы координат.

- **Локальная**: координаты объекта относительно его центра.
- **Мировая**: координаты объекта в общей сцене.
- **Камерная**: преобразование сцены относительно точки зрения камеры.
- **Экранная**: конечное преобразование в пиксельные координаты дисплея.

Связь между системами координат в компьютерной графике основана на **математических преобразованиях**, которые изменяют положение, ориентацию и масштаб объектов на разных этапах обработки:

1. **Трансляция (перемещение)**: изменяет положение объекта, перемещая его из одной точки пространства в другую. Например, объект из локального пространства перемещается в мировую сцену.
2. **Поворот (ротация)**: изменяет ориентацию объекта вокруг оси (например, вращение вокруг оси X, Y или Z).
3. **Масштабирование**: изменяет размер объекта, увеличивая или уменьшая его относительно определённой точки.
4. **Проекция**: преобразует трёхмерные координаты (из камерной системы) в двумерные координаты экрана, учитывая перспективу.

Эти преобразования выполняются с помощью **матриц**, которые последовательно применяются для перехода между системами координат. Например:

- Локальные координаты преобразуются в мировые через матрицу трансформации.
- Мировые координаты преобразуются в камерные через матрицу вида.
- Камерные координаты преобразуются в экранные через матрицу проекции.

## 12. Полигонизация. Алгоритмы построения выпуклой оболочки на плоскости.

**Полигонизация** — это процесс представления сложных объектов с использованием полигонов, чаще всего треугольников. Она применяется для упрощения геометрии объектов в компьютерной графике и 3D-моделировании.

**Алгоритм Грэхема**

Идея: сортирует точки по углу относительно начальной, затем строит оболочку методом обхода.
Эффективность $O(n\log n)$.
Алгоритм:

- Выбрать точку с минимальным $y$ как начальную.
- Отсортировать остальные точки по углу к этой точке.
- Идти по отсортированным точкам, проверяя повороты:
- Если правый поворот — удалить последнюю точку.
- Если левый — добавить точку в стек.
- После обхода в стеке — точки выпуклой оболочки.

**Алгоритм Джарвиса (обход подарочной упаковкой)**

Идея: последовательно обходит все точки, выбирая внешние. Эффективность $O(nh)$, где $h$ — количество точек на оболочке.

- Начать с точки с минимальным $y$.
- Найти самую левую точку относительно текущей.
- Добавить её в оболочку.
- Повторять, пока не вернётесь к стартовой точке.

**Алгоритм Чана**:

Идея: комбинирует подходы Грэхема и Джарвиса, работает за $O(n \log h)$.

Алгоритм:

- Разделить точки на группы (по m точек в каждой).
- Для каждой группы построить выпуклую оболочку (Грэхем).
- Объединить частичные оболочки методом Джарвиса.
- Увеличивать m, пока не найдётся полная оболочка.

**Разделяй и властвуй**:
Идея: делит точки на группы, строит оболочки и объединяет их. Эффективность $O(n \log n)$.

Алгоритм:

- Разделить точки на две части по $x$-координате.
- Построить оболочки для каждой части рекурсивно.
- Найти верхнюю и нижнюю касательные для объединения.
- Объединить обе оболочки в одну.

**Итог:**

- **Грэхема**: хорошо для общего случая.
- **Джарвиса**: удобен для малых наборов точек.
- **Чана**: лучше всего для случаев с большим количеством точек на оболочке.
- **Разделяй и властвуй**: эффективен для больших данных и параллельных вычислений.

## 13. Триангуляция.

**Триангуляция** — это разбиение области на треугольники, так чтобы их объединение покрывало всю область, а соседние треугольники имели общие стороны или вершины.

### Применение:

- Упрощение сложных форм для компьютерной графики.
- Моделирование поверхностей в 3D.
- Решение задач интерполяции и анализа данных.

Основные алгоритмы:

1. **Ухо-сечение**:
   - Выбрать полигон с вершинами.
   - Найти "ухо" (треугольник с вершинами, образующими выпуклый угол, не содержащий другие точки внутри).
   - Удалить "ухо" и повторить для оставшейся части полигона.
   - Повторять, пока не останутся только треугольники.
2. **Алгоритм Делоне**:
   - Разбить множество точек на треугольники.
   - Проверить условие Делоне: окружность любого треугольника не должна содержать другие точки.
   - Если условие нарушено, выполнить перестройку соседних треугольников (флип ребра).
   - Повторять, пока все треугольники удовлетворяют условию.
3. **Разделяй и властвуй**:
   - Разделить множество точек на две группы по координате $x$.
   - Построить триангуляции для каждой группы рекурсивно.
   - Объединить локальные триангуляции, соединяя граничные точки.

- Убедиться, что объединение сохраняет корректную триангуляцию.

Каждый алгоритм подходит для разных задач: Делоне — для равномерных триангуляций, ухо-сечение — для простых полигонов, "разделяй и властвуй" — для больших наборов точек.

## 14. Преобразование координат на плоскости. Двумерные аффинные преобразования.

**Преобразование координат на плоскости** — это изменение положения, ориентации или масштаба точек через математические операции.

**Двумерные аффинные преобразования** сохраняют прямолинейность и параллельность. Основные виды:

1. **Смещение (трансляция)**: перемещение на вектор $(dx,dy)$.
   $(x',y')=(x+dx,y+dy)$.
2. **Масштабирование**: изменение размеров по осям $x$ и $y$.

$(x',y')=(sx\cdot x,sy\cdot y)$.

3. **Поворот**: вращение вокруг начала координат на угол $\theta$.

$(x',y')=(x\cdot\cos\theta-y\cdot\sin\theta,x\cdot\sin\theta+y\cdot\cos\theta)$.

4. **Сдвиг**: смещение вдоль одной оси пропорционально другой.

$(x',y')=(x+shx\cdot y,y+shy\cdot x)$.

- shx : Сдвиг вдоль оси $x$, зависящий от значения $y$.
- $shy$ : Сдвиг вдоль оси $y$, зависящий от значения $x$.

**Общая матричная форма**:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & tx \\ c & d & ty \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Эта форма объединяет все преобразования и позволяет эффективно их комбинировать. Аффинные преобразования широко используются в компьютерной графике и геометрии.

## 15. Однородные координаты

**Однородные координаты** — это расширение обычных координат для представления геометрических преобразований, включая проекции, векторные операции и аффинные преобразования, с использованием матриц.

**Особенности:**

1. Точка $(x,y)$ на плоскости представляется как $(x,y,w)$, где $w\neq0$. Для стандартных координат $w=1$.
2. Все преобразования, включая поворот, масштабирование, сдвиг и перспективу, можно выразить единой матричной формой.
3. Деление на $w$ позволяет вернуть обычные координаты: $(x,y)=(x/\omega, y/\omega)$.

**Преимущества:**

- Удобство работы с трансформациями (единая матрица для всех операций).
- Используется в графике, моделировании и компьютерном зрении.

**Пример:**

Матричное представление с использованием однородных координат:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & tx \\ c & d & ty \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

## 16. Пример использования комбинации аффинных преобразований на плоскости.

Задача: повернуть фигуру на 45°, увеличить её в 2 раза и переместить на вектор *(3,4)*.

**Шаги:**

1. **Масштабирование**: увеличим размеры фигуры.
   - Матрица:

$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

2. **Поворот на 45°**: изменим ориентацию.
   - Матрица:

$$\begin{bmatrix} \cos 45^\circ & -\sin 45^\circ & 0 \\ \sin 45^\circ & \cos 45^\circ & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

3. **Смещение**: переместим фигуру.
   - Матрица:

$$\begin{bmatrix} 1 & 0 & 3 \\ 0 & 1 & 4 \\ 0 & 0 & 1 \end{bmatrix}$$

## Комбинация:

Умножаем матрицы в порядке применения:
$M=T \cdot R \cdot S$
Где $T$ — трансляция, $R$ — поворот, $S$ — масштабирование.
Эта итоговая матрица позволяет за одно преобразование применить все изменения к объекту.
Пример: если начальная точка $(x,y)=(1,1)$, её новая позиция вычисляется как:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = M \cdot \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

## Объяснение:

1. **Что оно означает**:
   - $(x,y)$: начальные координаты точки.
   - $M$: матрица преобразования, которая объединяет все трансформации (например, масштабирование, поворот, трансляцию).
   - $(x',y')$: новые координаты точки после применения преобразований.
2. **Почему добавляется единица**:
   - Это однородные координаты. Третья координата ($1$) позволяет выразить аффинные преобразования (включая трансляцию) в матричной форме.
   - Например, трансляция $(x+dx,y+dy)$ невозможна с обычными *2×2 матрицами, но добавление 1* решает эту задачу.
3. **Как работает умножение**: Матрица $M$ представляет комбинацию преобразований. Например:

$$M = \begin{bmatrix} a & b & tx \\ c & d & ty \\ 0 & 0 & 1 \end{bmatrix}$$

Где:

- *a,b,c,d*: поворот, масштабирование.
- *tx,ty*: трансляция.

После умножения:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & tx \\ c & d & ty \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a \cdot x + b \cdot y + tx \\ c \cdot x + d \cdot y + ty \\ 1 \end{bmatrix}$$

- Это даёт новые координаты $(x', y')$:

$x' = a \cdot x + b \cdot y + tx,$
$y' = c \cdot x + d \cdot y + ty$

- **Итог**: Уравнение объединяет все преобразования в одно матричное умножение, обеспечивая удобство и универсальность для геометрических операций.

## 17. Аффинные преобразования в пространстве

**Аффинные преобразования в 3D-пространстве** — это преобразования, которые сохраняют прямолинейность и параллельность объектов. Они включают:

1. **Трансляция (перемещение)**:
   - Смещение точки на вектор $(dx, dy, dz)$.
   - Формула: $(x', y', z') = (x + dx, y + dy, z + dz)$.
2. **Масштабирование**:
   - Изменение размеров объекта по осям $x$, $y$, $z$.
   - Формула: $(x', y', z') = (sx \cdot x, sy \cdot y, sz \cdot z)$.
3. **Поворот**:
   - Вращение объекта вокруг одной из осей ($x$, $y$, $z$) на угол $\theta$.
   - Матрицы вращения:
     - Вокруг $x$-оси:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

     - Вокруг $y$-оси:

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

     - Вокруг $z$-оси:

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

4. **Сдвиг**:
   - Сдвиг точек вдоль одной оси пропорционально их положениям по другим осям.

**Общая матричная форма**:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c & tx \\ d & e & f & ty \\ g & h & i & tz \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

**Итог**: Аффинные преобразования в пространстве позволяют перемещать, масштабировать, вращать и деформировать объекты, сохраняя основные геометрические свойства.

## 18. Пример использования комбинации аффинных преобразований в трехмерном пространстве.

**Задача**: увеличить объект в 2 раза, повернуть его вокруг оси *y* на 90° и переместить на вектор *(3,4,5)*.

**Шаги преобразований:**

1. **Масштабирование**:
   - Увеличение объекта в 2 раза.
   - Матрица:

$$S = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2. **Поворот вокруг оси *y***:
   - На угол *90∘*.
   - Матрица:

$$R_y = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3. **Трансляция**:
   - Перемещение на вектор *(3,4,5)*.
   - Матрица:

$$T = \begin{bmatrix} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 4 \\ 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

**Итоговая матрица:**

Комбинация всех преобразований:

*M=T·Ry·SM*

**Пример применения:**

Для точки *(x,y,z)=(1,1,1)*:

1. Масштабирование увеличивает координаты: *(2,2,2)*).
2. Поворот изменяет ориентацию: *(2,2,−2)*.
3. Трансляция смещает позицию: *(5,6,3)*.

**Итог**: Комбинация аффинных преобразований позволяет выполнять сложные изменения объекта (размер, ориентация, положение) в одном процессе.

## 19. Ортогональные проекции в компьютерной графике

**Ортогональная проекция** — это способ отображения трёхмерных объектов на плоскость, при котором проекционные лучи параллельны друг другу и перпендикулярны к проекционной плоскости.

### Особенности:

1. **Отсутствие перспективы**: размеры объектов не зависят от их расстояния до камеры.
2. **Простота расчётов**: используются линейные преобразования без учёта перспективных искажений.
3. **Применение**: часто используется в инженерной графике, чертежах, изометрических проекциях и схемах.

### Пример матрицы ортогональной проекции:

Для проекции на плоскость $xy$ (обнуление $z$-координаты):

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Проекция точки $(x,y,z)$ даёт:

(x',y',z')=(x,y,0)

**Итог**: Ортогональная проекция сохраняет размеры и пропорции объектов, делая её удобной для точных представлений геометрии.

## 20. Аксонометрические проекции в компьютерной графике

**Аксонометрическая проекция** — это вид параллельной проекции, при которой объект отображается на плоскость под углом к осям координат, сохраняя пропорции.

**Виды:**

1. **Изометрическая**:
   - Углы между осями равны (120°).
   - Все оси масштабируются одинаково.
   - Применяется для равномерного отображения без искажений.
2. **Диметрическая**:
   - Две оси имеют одинаковый масштаб, третья — другой.
   - Используется для более реалистичного отображения.
3. **Триметрическая**:
   - Все три оси имеют разные масштабы.
   - Позволяет добиться наиболее индивидуального вида.

**Особенности:**

- Линии остаются параллельными, но размеры могут искажаться в зависимости от углов.
- Часто применяется для технических чертежей, игр (например, стратегий), архитектуры.

**Пример матрицы изометрической проекции:**

$$P = \begin{bmatrix} \frac{\sqrt{3}}{2} & 0 & -\frac{\sqrt{3}}{2} & 0 \\ \frac{1}{2} & 1 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

**Итог**: Аксонометрические проекции создают реалистичное изображение объектов с сохранением их структуры и пропорций, что делает их важными в технической графике и 3D-дизайне.

## 21. Косоугольные проекции в компьютерной графике

Косоугольные проекции — это метод проецирования трёхмерных объектов на двумерную плоскость, при котором проекционные линии не перпендикулярны плоскости проекции. Этот тип проекций часто используется для отображения объектов с сохранением большей части их пространственной структуры.

**Основные характеристики косоугольных проекций:**

1. **Углы между осями на плоскости проекции:**
   - В косоугольной проекции оси могут быть представлены под произвольным углом.
   - Наиболее распространённый случай — углы между осями X и Y составляют 90°, а ось Z наклонена.
2. **Искажение длин:**
   - Косоугольные проекции допускают масштабирование вдоль оси Z, из-за чего размеры по этой оси могут быть уменьшены.
3. **Типы косоугольных проекций:**
   - **Кавалерийская проекция:** Угол между осью Z и плоскостью проекции составляет 45°. Размеры вдоль оси Z сохраняются.
   - **Кабинетная проекция:** Угол между осью Z и плоскостью проекции составляет 63,4° (приблизительно). Длины вдоль оси Z уменьшаются в два раза.

---

**Матрица преобразования для косоугольной проекции**

Объект проецируется на плоскость XY путём преобразования его координат:

1. **Общая формула:**
   Пусть угол между осью Z и направлением проекции равен ϕ, а угол между осью X и направлением проекции — θ. Тогда координаты после проекции можно найти по следующим формулам:

$$x' = x + l \cdot z \cdot \cos(\phi), \quad y' = y + m \cdot z \cdot \sin(\phi)$$

где $l$ и $m$ — коэффициенты масштабирования (обычно $l = 1, m = 1$).

2. **Матрица преобразования в однородных координатах:**

$$M = \begin{bmatrix} 1 & 0 & l \cdot \cos(\phi) & 0 \\ 0 & 1 & m \cdot \sin(\phi) & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

---

**Алгоритм построения косоугольной проекции:**

1. **Задание исходной модели:**
   - Определите координаты точек объекта в трёхмерном пространстве.
2. **Применение матрицы проекции:**
   - Умножьте каждую точку объекта на матрицу косоугольной проекции.
3. **Отображение результата:**
   - Полученные 2D-координаты используются для отрисовки объекта на экране.

---

**Преимущества и недостатки**

**Преимущества:**

- Простота вычислений по сравнению с центральной перспективой.
- Позволяет легко контролировать вид объекта, выбирая параметры углов и коэффициентов масштабирования.

**Недостатки:**

- Искажения, которые могут выглядеть неестественно.
- Не подходит для реалистичного отображения трёхмерных объектов.

---

**Практическое применение:**

- Технические чертежи и схемы.
- Игры и приложения, где важна наглядность, но не требуется высокая реалистичность.
- Отображение изометрических и псевдо-изометрических объектов.

---

**Пример кода на Python (с использованием NumPy):**

```python
import numpy as np
import matplotlib.pyplot as plt

# Исходные координаты точки
points = np.array([
    [0, 0, 0],
    [1, 0, 0],
    [1, 1, 0],
    [0, 1, 0],
    [0, 0, 1],
    [1, 0, 1],
    [1, 1, 1],
    [0, 1, 1]
])

# Матрица косоугольной проекции
phi = np.radians(45)  # Угол наклона оси Z
l, m = 1, 1  # Коэффициенты масштабирования
projection_matrix = np.array([
    [1, 0, l * np.cos(phi)],
    [0, 1, m * np.sin(phi)],
    [0, 0, 0]
])

# Применяем проекцию
projected_points = points @ projection_matrix.T

# Рисуем результат
fig, ax = plt.subplots()
for point in projected_points:
    ax.scatter(point[0], point[1], color='red')
ax.set_aspect('equal')
plt.show()
```

## 22. Перспективное проектирование и его свойства

Перспективное проецирование — это способ отображения трёхмерных объектов на двумерную плоскость, при котором линии проекции сходятся в одной точке, называемой **центром проекции** (или центром перспективы). Этот метод позволяет передать реалистичное ощущение глубины, поскольку объекты, удалённые от наблюдателя, кажутся меньше.

---

**Основные характеристики перспективного проецирования**

1. **Центр проекции:**
   - Точка, в которой сходятся все проекционные линии. Она задаётся в пространстве относительно объекта.
2. **Плоскость проекции:**
   - Плоскость, на которую проецируется изображение. Чаще всего это z=0.
3. **Свойства перспективы:**
   - **Сжатие размеров:** Объекты удалённые от наблюдателя кажутся меньше.
   - **Линии схода:** Параллельные линии в пространстве кажутся сходящимися в одной точке (точка схода).
   - **Искажение формы:** Фигуры, находящиеся далеко от наблюдателя, могут казаться вытянутыми.

---

**Формулы перспективного проецирования**

1. **Проецирование на плоскость z=0:**
   Пусть центр проекции находится в точке (0,0,d), где d>0 — расстояние от плоскости проекции до центра. Тогда координаты точки (x,y,z) после проецирования вычисляются по формулам:

$$x' = \frac{x \cdot d}{d + z}, \quad y' = \frac{y \cdot d}{d + z}$$

2. **Матрица проецирования в однородных координатах:**
   Для однородных координат используется следующая матрица:

$$M = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

   После применения этой матрицы полученные координаты необходимо нормализовать:

$$x' = \frac{x}{w}, \quad y' = \frac{y}{w}, \quad z' = \frac{z}{w}$$

где $w = d + z$.

---

**Алгоритм построения перспективного изображения**

1. **Задание 3D-модели:**
   - Определите вершины объекта в трёхмерных координатах.
2. **Применение матрицы перспективного проецирования:**
   - Умножьте вершины объекта на матрицу перспективного преобразования.
3. **Нормализация координат:**
   - Разделите координаты на www, чтобы получить результат на плоскости проекции.
4. **Отображение объекта:**
   - Используйте нормализованные координаты для визуализации.

---

**Преимущества и недостатки**
**Преимущества:**

- Реалистичное отображение трёхмерных объектов.
- Позволяет передать ощущение глубины.

**Недостатки:**

- Требует больше вычислений, чем ортографическое или косоугольное проецирование.
- Может приводить к искажению форм, если объект находится слишком близко или далеко.

---

**Применение перспективного проецирования**

- Создание реалистичных сцен в 3D-графике.
- Визуализация архитектурных моделей.
- Используется в играх, фильмах и анимации.
- Оптические системы, такие как камеры или проекторы.

---

**Пример кода на Python (с использованием NumPy):**

```
import numpy as np
import matplotlib.pyplot as plt

# Исходные точки 3D-объекта
points = np.array([
    [0, 0, 0],
    [1, 0, 0],
    [1, 1, 0],
    [0, 1, 0],
    [0, 0, 1],
    [1, 0, 1],
    [1, 1, 1],
    [0, 1, 1]
])

# Параметры перспективы
```

```
d = 2  # Расстояние от центра проекции до плоскости проекции

# Матрица перспективного проецирования
projection_matrix = np.array([
    [d, 0, 0, 0],
    [0, d, 0, 0],
    [0, 0, d, 0],
    [0, 0, 1, 0]
])

# Преобразование точек в однородные координаты
homogeneous_points = np.hstack((points, np.ones((points.shape[0], 1))))
projected_points = homogeneous_points @ projection_matrix.T

# Нормализация координат
projected_points = projected_points[:, :2] / projected_points[:, 3].reshape(-1, 1)

# Рисование результата
fig, ax = plt.subplots()
for point in projected_points:
    ax.scatter(point[0], point[1], color='blue')
ax.set_aspect('equal')
plt.show()
```

---

**Основные выводы:**

- Перспективное проецирование передаёт реалистичную глубину, но требует нормализации координат.
- Оно широко используется в 3D-визуализации, моделировании и анимации.

## 23. Одноточечное перспективное проецирование на плоскость

Одноточечное перспективное проецирование — это частный случай перспективного проецирования, при котором все линии, параллельные одной из осей пространства, сходятся в единой точке схода. Обычно используется для упрощения восприятия глубины, когда объект находится строго перед наблюдателем.

---

**Основные характеристики одноточечного проецирования**

1. **Единственная точка схода:**
   - Все линии, параллельные одной из осей, сходятся в одной точке.
   - Наиболее часто используется ось Z, чтобы проецировать объекты на плоскость XY.
2. **Положение камеры:**
   - Камера или наблюдатель располагаются так, что линии зрения строго параллельны оси Z.
3. **Искажение перспективы:**
   - Объекты уменьшаются по мере удаления вдоль оси схода.

---

**Формулы одноточечного проецирования**

1. **Проецирование на плоскость z = 0:**
   Если центр проекции находится в точке (0, 0, d), то координаты точки (x, y, z) после проецирования определяются как:
   $$x' = \frac{x \cdot d}{d + z}, \quad y' = \frac{y \cdot d}{d + z}$$
   При этом z' игнорируется, так как проекция происходит на плоскость XY.
2. **Матрица преобразования в однородных координатах:**
   Для одноточечного проецирования матрица имеет вид:

$$M = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

После применения матрицы необходимо нормализовать координаты, деля на *w*:

$$x' = \frac{x}{w}, \quad y' = \frac{y}{w}$$

---

**Алгоритм построения одноточечного проецирования**

1. **Подготовка объекта:**
   - Определите координаты вершин трёхмерного объекта.
2. **Применение матрицы проецирования:**
   - Умножьте координаты на матрицу одноточечного проецирования.
3. **Нормализация координат:**
   - Разделите каждую координату на www, чтобы получить итоговые 2D-координаты.
4. **Визуализация:**
   - Постройте объект на плоскости XY, используя полученные координаты.

---

**Преимущества и недостатки**

**Преимущества:**

- Простота реализации, поскольку используется только одна точка схода.
- Хорошо подходит для изображений с прямым видом на объект.

**Недостатки:**

- Ограниченность по углу обзора (подходит только для ортогонального положения камеры).
- Слабое ощущение глубины для сложных сцен.

---

**Применение одноточечного проецирования**

- Архитектурные эскизы.
- Простые 3D-сцены, где важно упрощённое отображение глубины.
- Учебные примеры, иллюстрирующие перспективу.

---

**Пример кода на Python (с использованием NumPy)**

```
import numpy as np
import matplotlib.pyplot as plt

# Исходные точки объекта
points = np.array([
    [0, 0, 5],
    [1, 0, 5],
    [1, 1, 5],
    [0, 1, 5],
    [0, 0, 10],
    [1, 0, 10],
    [1, 1, 10],
    [0, 1, 10]
])

# Параметры перспективы
d = 2  # Расстояние от центра проекции до плоскости проекции

# Матрица одноточечного проецирования
projection_matrix = np.array([
    [d, 0, 0, 0],
    [0, d, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 1/d, 0]
```

```
])

# Преобразование точек в однородные координаты
homogeneous_points = np.hstack((points, np.ones((points.shape[0], 1))))
projected_points = homogeneous_points @ projection_matrix.T

# Нормализация координат
projected_points = projected_points[:, :2] / projected_points[:, 3].reshape(-1, 1)

# Рисование результата
fig, ax = plt.subplots()
for point in projected_points:
    ax.scatter(point[0], point[1], color='blue')
ax.set_aspect('equal')

# Соединяем точки для построения объекта
edges = [
    (0, 1), (1, 2), (2, 3), (3, 0),  # Нижняя грань
    (4, 5), (5, 6), (6, 7), (7, 4),  # Верхняя грань
    (0, 4), (1, 5), (2, 6), (3, 7)   # Вертикальные рёбра
]
for edge in edges:
    p1, p2 = projected_points[edge[0]], projected_points[edge[1]]
    ax.plot([p1[0], p2[0]], [p1[1], p2[1]], color='blue')

plt.show()
```

---

**Основные выводы:**

- Одноточечное проецирование подходит для простых сцен с прямым видом на объект.
- Реализуется через базовые преобразования с одной точкой схода, что делает его удобным для начинающих изучение компьютерной графики.

## 24. Одноточечное перспективное проецирование на линию

Одноточечное перспективное проецирование на линию — это метод проецирования, при котором все точки объекта проецируются на заданную прямую в пространстве. Линия выступает в качестве плоскости проекции, сведённой к одной размерности, а проекционные лучи сходятся в единой точке — центре проекции.

---

**Основные характеристики**

1. **Центр проекции:**
   - Точка, из которой выходят проекционные лучи. Обозначается $(x_c, y_c, z_c)$.
2. **Линия проекции:**
   - Линия, на которую происходит проецирование. Она задаётся векторным уравнением:

$$L(t) = \vec{p} + t \cdot \vec{d}$$

где $\vec{p}$ — точка на линии, $\vec{d}$ — направляющий вектор, $t \in \mathbb{R}$.

3. **Искажение глубины:**
   - По мере удаления точки от линии проекции её изображение сжимается вдоль направления линии.

---

**Формулы проецирования**

Пусть:

- Центр проекции: C$(x_c, y_c, z_c)$;
- Точка для проецирования: P(x, y, z);
- Линия проекции задана точкой $\vec{p}(x_0, y_0, z_0)$ и направляющим вектором $\vec{d}(d_x, d_y, d_z)$.

**Алгоритм:**

1. Найти вектор от центра проекции к точке:

$$\vec{v} = \vec{P} - \vec{C} = (x - x_c, y - y_c, z - z_c)$$

2. Подставить точку P в параметрическое уравнение линии:

$$\vec{P'} = \vec{p} + t \cdot \vec{d}$$

3. Определить параметр t, при котором точка P′ лежит на линии:

$$t = \frac{\vec{v} \cdot \vec{d}}{\vec{d} \cdot \vec{d}}$$

где $\vec{v} \cdot \vec{d}$ — скалярное произведение векторов.

4. Найти проекцию P′:

$$\vec{P'} = \vec{p} + t \cdot \vec{d}$$

Итоговые координаты точки после проецирования — P'(x', y', z').

---

**Свойства одноточечного проецирования на линию**

1. **Сходимость проекционных линий:**
   - Все лучи проекции сходятся в центре.
2. **Линейность:**
   - Отображение точки на линию сохраняет её относительное положение вдоль направления линии.
3. **Сжатие вдоль линии:**
   - Чем дальше объект от линии проекции, тем сильнее его отображение сжимается вдоль неё.

---

**Преимущества и недостатки**

**Преимущества:**

- Удобно для анализа поведения объекта вдоль одной оси или направления.
- Упрощает вычисления при задании проекционной линии.

**Недостатки:**

- Ограничивает визуализацию, так как изображение сжато до одной размерности.
- Сложно интерпретировать сложные формы объектов.

---

**Пример применения**

Используется в следующих случаях:

- Проекция объектов на прямую для анализа их движения вдоль линии.
- Упрощение трёхмерных данных для отображения в виде одномерных графиков.
- Моделирование теней в определённом направлении света.

---

**Пример кода на Python**

```python
import numpy as np
import matplotlib.pyplot as plt

# Исходные точки объекта
points = np.array([
    [1, 1, 3],
    [2, 2, 3],
    [3, 1, 3],
    [2, 0, 3]
])

# Параметры проекции
center = np.array([0, 0, 0])  # Центр проекции
line_point = np.array([0, 0, 1])  # Точка на линии
line_direction = np.array([0, 0, 1])  # Направляющий вектор линии

# Функция проецирования точки на линию
def project_point_on_line(point, center, line_point, line_direction):
    v = point - center
    t = np.dot(v, line_direction) / np.dot(line_direction, line_direction)
```

```
    projection = line_point + t * line_direction
    return projection

# Применяем проекцию к каждой точке
projected_points = np.array([project_point_on_line(p, center, line_point, line_direction) for p in
points])

# Рисуем исходные и проецированные точки
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Исходные точки
ax.scatter(points[:, 0], points[:, 1], points[:, 2], color='blue', label='Исходные точки')

# Проецированные точки
ax.scatter(projected_points[:, 0], projected_points[:, 1], projected_points[:, 2], color='red',
label='Проекции на линию')

# Рисуем линию
t = np.linspace(-5, 5, 100)
line = line_point + t[:, None] * line_direction
ax.plot(line[:, 0], line[:, 1], line[:, 2], color='green', label='Линия проекции')

ax.legend()
plt.show()
```

---

**Основные выводы:**

- Одноточечное проецирование на линию удобно для изучения объектов вдоль заданного направления.
- Для реализации необходимо лишь знать центр проекции и параметры линии.
- Метод активно используется в задачах анализа и визуализации движения.

## 25. Двухточечное перспективное проецирование

Двухточечное перспективное проецирование — это метод перспективной проекции, при котором линии, параллельные двум осям пространства, сходятся в двух точках на горизонте. Оно используется для создания более реалистичного трёхмерного изображения, особенно при отображении объектов, ориентированных под углом к наблюдателю.

---

**Основные характеристики**

1. **Две точки схода:**
   - Линии, параллельные двум координатным осям (например, X и Z), сходятся в двух различных точках схода.
   - Линии, параллельные третьей оси (например, Y), остаются вертикальными.
2. **Положение наблюдателя:**
   - Камера (или центр проекции) расположена так, что наблюдатель смотрит на объект под углом к его граням.
3. **Реалистичное отображение:**
   - Метод создаёт ощущение глубины и перспективы, особенно для зданий, мебели и других объектов с прямыми углами.

---

**Формулы двухточечного проецирования**

1. **Плоскость проекции z = 0:**
   Пусть центр проекции находится в точке (0, 0, d). Координаты точки (x, y, z) после проецирования на плоскость z = 0 вычисляются как:
   $$x' = \frac{x \cdot d}{d + z}, \quad y' = \frac{y \cdot d}{d + z}$$
   Здесь z′ игнорируется, так как проекция отображается в двухмерном пространстве.
2. **Матрица преобразования в однородных координатах:**
   Для двухточечного проецирования матрица имеет вид:

$$M = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

После применения матрицы необходимо нормализовать координаты, деля на $w$:

$$x' = \frac{x}{w}, \quad y' = \frac{y}{w}$$

3. **Точки схода:**
   - Линии, параллельные оси X, сходятся в точке схода $V_x$.
   - Линии, параллельные оси Z, сходятся в точке схода $V_z$.

---

**Алгоритм построения двухточечного проецирования**

1. **Определение центра проекции и плоскости проекции:**
   - Установите положение камеры и точки схода.
2. **Применение матрицы преобразования:**
   - Умножьте координаты вершин объекта на матрицу проецирования.
3. **Нормализация координат:**
   - Разделите x и y на w, чтобы получить итоговые координаты.
4. **Визуализация объекта:**
   - Постройте проекцию объекта на плоскость z = 0 с учётом двух точек схода.

---

**Преимущества и недостатки**

**Преимущества:**

- Более реалистичная перспектива, чем в одноточечном проецировании.
- Хорошо подходит для объектов с угловой ориентацией, таких как здания или комнаты.

**Недостатки:**

- Требует более сложных расчётов.
- Не подходит для случаев, где нужно сохранять симметрию без искажения.

---

**Применение двухточечного проецирования**

- Архитектурное проектирование и визуализация.
- Моделирование сцен с угловой перспективой.
- Создание реалистичных изображений зданий, мебели и других объектов с параллельными гранями.

---

**Пример кода на Python**

```
import numpy as np
import matplotlib.pyplot as plt

# Исходные точки объекта (куб)
points = np.array([
    [1, 1, 1],
    [1, -1, 1],
    [-1, -1, 1],
    [-1, 1, 1],
    [1, 1, -1],
    [1, -1, -1],
    [-1, -1, -1],
    [-1, 1, -1]
])

# Параметры проекции
d = 5  # Расстояние до плоскости проекции

# Матрица двухточечного проецирования
```

```
projection_matrix = np.array([
    [d, 0, 0, 0],
    [0, d, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 1/d, 0]
])

# Преобразование точек в однородные координаты
homogeneous_points = np.hstack((points, np.ones((points.shape[0], 1))))
projected_points = homogeneous_points @ projection_matrix.T

# Нормализация координат
projected_points = projected_points[:, :2] / projected_points[:, 3].reshape(-1, 1)

# Рисование результата
fig, ax = plt.subplots()
for point in projected_points:
    ax.scatter(point[0], point[1], color='blue')

# Соединение точек для отображения куба
edges = [
    (0, 1), (1, 2), (2, 3), (3, 0),  # Передняя грань
    (4, 5), (5, 6), (6, 7), (7, 4),  # Задняя грань
    (0, 4), (1, 5), (2, 6), (3, 7)   # Рёбра
]
for edge in edges:
    p1, p2 = projected_points[edge[0]], projected_points[edge[1]]
    ax.plot([p1[0], p2[0]], [p1[1], p2[1]], color='blue')

plt.gca().set_aspect('equal')
plt.show()
```

---

**Основные выводы:**

- Двухточечное проецирование используется для более реалистичного отображения объектов под углом.
- Оно требует дополнительных расчётов для обработки двух точек схода, но результат выглядит убедительнее.
- Метод широко применяется в архитектурной визуализации и компьютерной графике.

## 26. Трехточечное перспективное проецирование

Трёхточечное перспективное проецирование — это метод проекции, при котором линии, параллельные всем трём координатным осям (X, Y, Z), сходятся в трёх различных точках схода. Этот тип проецирования используется для отображения объектов с наклонёнными или смещёнными относительно всех осей гранями, создавая реалистичную трёхмерную перспективу.

---

**Основные характеристики**

1. **Три точки схода:**
   - Линии, параллельные каждой из осей (X, Y, Z), сходятся в своих уникальных точках схода, расположенных на "горизонте" сцены.
2. **Отсутствие параллельных линий:**
   - Все грани объекта кажутся наклонёнными, и параллельные линии объекта пересекаются на горизонте.
3. **Реалистичное отображение наклонённых объектов:**
   - Применяется для создания изображений объектов с выраженным наклоном, например, при изображении высоких зданий с перспективой сверху вниз или снизу вверх.

---

**Формулы трёхточечного проецирования**

Пусть:

- Центр проекции — точка $C(x_c, y_c, z_c)$;
- Точка объекта — P(x, y, z);
- Плоскость проекции находится в z = 0.

Координаты точки P′ на плоскости рассчитываются следующим образом:

1. **Формулы проекции:**
$$x' = \frac{x \cdot d}{d + z}, \quad y' = \frac{y \cdot d}{d + z}$$
где d — расстояние от центра проекции до плоскости проекции.
2. **Общая матрица проецирования (в однородных координатах):**
$$M = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$
После применения матрицы нормализуйте координаты, деля на w:
$$x' = \frac{x}{w}, \quad y' = \frac{y}{w}$$

---

**Алгоритм построения трёхточечного проецирования**

1. **Определите центр проекции:**
   - Задайте точку камеры $C(x_c, y_c, z_c)$, с которой наблюдается объект.
2. **Определите положение плоскости проекции:**
   - Обычно используется плоскость z = 0, но её можно сместить для получения других эффектов.
3. **Примените матрицу преобразования:**
   - Переведите координаты объекта в однородные координаты и умножьте их на матрицу проецирования.
4. **Нормализуйте координаты:**
   - Разделите x и y на w, чтобы получить конечные экранные координаты.

---

**Преимущества и недостатки**

**Преимущества:**

- Реалистичное изображение объектов, наклонённых относительно всех трёх осей.
- Удобно для отображения сложных трёхмерных сцен.

**Недостатки:**

- Более сложные расчёты по сравнению с одноточечным и двухточечным проецированием.
- Возможность значительного искажения объекта при неправильном выборе точки камеры.

---

**Применение трёхточечного проецирования**

- Архитектура: отображение высотных зданий или сооружений под углом.
- Компьютерные игры: создание трёхмерных миров с наклонёнными объектами.
- Искусство: рисование объектов с драматичной перспективой.

---

**Пример кода на Python**

```
import numpy as np
import matplotlib.pyplot as plt

# Исходные точки объекта (куб)
points = np.array([
    [1, 1, 1],
    [1, -1, 1],
    [-1, -1, 1],
    [-1, 1, 1],
    [1, 1, -1],
    [1, -1, -1],
    [-1, -1, -1],
    [-1, 1, -1]
])
```

```python
# Параметры проекции
center = np.array([2, 3, 5])  # Центр проекции
d = 5  # Расстояние до плоскости проекции (z=0)

# Функция трёхточечного проецирования
def project_point(point, center, d):
    x, y, z = point
    cx, cy, cz = center
    # Перевод координат
    x_proj = d * (x - cx) / (z - cz)
    y_proj = d * (y - cy) / (z - cz)
    return x_proj, y_proj

# Применяем проекцию к каждой точке
projected_points = np.array([project_point(p, center, d) for p in points])

# Рисование результата
fig, ax = plt.subplots()
for point in projected_points:
    ax.scatter(point[0], point[1], color='blue')

# Соединение точек для отображения куба
edges = [
    (0, 1), (1, 2), (2, 3), (3, 0),  # Передняя грань
    (4, 5), (5, 6), (6, 7), (7, 4),  # Задняя грань
    (0, 4), (1, 5), (2, 6), (3, 7)   # Рёбра
]
for edge in edges:
    p1, p2 = projected_points[edge[0]], projected_points[edge[1]]
    ax.plot([p1[0], p2[0]], [p1[1], p2[1]], color='blue')

plt.gca().set_aspect('equal')
plt.title('Трёхточечное перспективное проецирование')
plt.show()
```

---

**Основные выводы:**

- Трёхточечное проецирование применяется для реалистичного отображения наклонённых объектов.
- Метод сложнее в реализации, но позволяет добиться впечатляющих результатов в трёхмерной визуализации.
- Подходит для сцен, где важен эффект высоты и перспективы.

## 27. Масштабирование в окне

Масштабирование в окне — это процесс изменения размера изображения или объектов внутри заданного прямоугольного окна просмотра (viewport). Оно позволяет пользователю увеличивать или уменьшать масштаб изображения без искажения пропорций, при этом сохраняется отображение объектов внутри ограниченной области.

---

**Основные характеристики**

1. **Окно просмотра:**
   - Прямоугольная область на экране, в которой отображаются графические объекты.
   - Размер окна задаёт границы, в пределах которых масштабируется содержимое.
2. **Факторы масштабирования:**
   - Горизонтальный ($S_x$) и вертикальный ($S_y$) коэффициенты определяют, как изменяется размер объекта.
   - Если $S_x = S_y$, масштабирование является равномерным, иначе оно называется неравномерным.
3. **Точка привязки:**
   - Опорная точка (обычно центр окна) остаётся на месте, вокруг неё производится изменение размера.

---

**Формула масштабирования**

1. **Общая формула:**
   Пусть точка P(x, y) является координатой объекта. После масштабирования она перемещается в точку P'(x', y'):
   $$x' = x \cdot S_x, \quad y' = y \cdot S_y$$
2. **С учётом точки привязки $(x_0, y_0)$:**
   Если масштабирование производится относительно точки привязки, новые координаты вычисляются как:
   $$x' = x_0 + (x - x_0) \cdot S_x, \quad y' = y_0 + (y - y_0) \cdot S_y$$
3. **Матрица масштабирования в однородных координатах:**
   $$M = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$
   Применение этой матрицы к точке в однородных координатах (x, y, 1) даёт масштабированные координаты.

---

**Алгоритм масштабирования**

1. **Определите окно просмотра:**
   - Установите область окна, внутри которой будет производиться масштабирование.
2. **Выберите масштабные коэффициенты $S_x$ и $S_y$:**
   - Задайте величину увеличения или уменьшения по каждой оси.
3. **Примените формулы масштабирования:**
   - Для каждой точки объекта пересчитайте её координаты с учётом выбранных коэффициентов.
4. **Отобразите масштабированное изображение:**
   - Нарисуйте обновлённые точки внутри окна просмотра.

---

**Преимущества и недостатки**

**Преимущества:**

- Простота реализации.
- Позволяет удобно управлять видимостью объектов в ограниченной области.
- Может быть легко объединено с другими преобразованиями, такими как повороты и сдвиги.

**Недостатки:**

- При слишком большом увеличении возможна потеря детализации.
- При уменьшении объектов их визуальное восприятие может ухудшиться.

---

**Применение масштабирования**

- Редактирование графики в графических редакторах.
- Управление зумом в пользовательских интерфейсах (например, карты, чертежи).
- Компьютерные игры для создания эффекта приближения или удаления сцены.
- Отображение данных в научных или инженерных приложениях.

---

**Пример кода на Python**

```
import numpy as np
import matplotlib.pyplot as plt

# Исходные точки объекта (квадрат)
points = np.array([
    [1, 1],
    [1, -1],
    [-1, -1],
    [-1, 1],
    [1, 1]  # Замыкаем квадрат
```

```
])

# Коэффициенты масштабирования
S_x = 2  # Увеличение по оси X
S_y = 1.5  # Увеличение по оси Y

# Матрица масштабирования
scale_matrix = np.array([
    [S_x, 0],
    [0, S_y]
])

# Применяем масштабирование
scaled_points = points @ scale_matrix.T

# Рисование результата
fig, ax = plt.subplots()

# Оригинальный объект
ax.plot(points[:, 0], points[:, 1], label="Оригинал", color="blue")

# Масштабированный объект
ax.plot(scaled_points[:, 0], scaled_points[:, 1], label="Масштабировано", color="red")

ax.legend()
plt.gca().set_aspect('equal')
plt.title('Масштабирование в окне')
plt.grid()
plt.show()
```

---

**Основные выводы:**

- Масштабирование позволяет изменять размер объекта относительно окна просмотра.
- При равномерном масштабировании пропорции сохраняются, а при неравномерном — могут искажаться.
- Этот процесс широко используется в компьютерной графике для управления видимостью и размером объектов.

## 28. Алгоритмы отсечения.

Алгоритмы отсечения используются в компьютерной графике для удаления тех частей объектов (точек, линий, многоугольников), которые выходят за границы окна просмотра. Это позволяет оптимизировать процесс отрисовки, отображая только те элементы сцены, которые находятся в видимой области.

---

**Основные характеристики**

1. **Окно просмотра:**
   - Прямоугольная область, внутри которой должны находиться отображаемые объекты.
2. **Цель отсечения:**
   - Удалить невидимые части объектов, чтобы уменьшить количество вычислений и повысить производительность.
3. **Типы отсечения:**
   - **Отсечение точек:** Проверка, находятся ли точки внутри окна.
   - **Отсечение линий:** Удаление частей линии, выходящих за границы окна.
   - **Отсечение полигонов:** Удаление частей многоугольников за пределами окна.

---

**Алгоритмы отсечения**

1. **Алгоритм Коэна-Сазерленда**
   - Предназначен для отсечения отрезков.
   - Разделяет пространство на девять областей с помощью кодирования.
   - **Основные этапы:**
     - Каждой вершине отрезка присваивается код (outcode), указывающий, где она находится относительно окна.

- Если обе вершины имеют код 0000 (внутри окна) — отрезок полностью виден.
- Если outcode обоих концов отрезка имеет хотя бы один общий бит, отрезок полностью невиден.
- Иначе применяется итеративное отсечение по границам окна.

**Формулы для границ окна:**

$$x = x_1 + (x_2 - x_1) \cdot \frac{y_{\text{граница}} - y_1}{y_2 - y_1}$$

$$y = y_1 + (y_2 - y_1) \cdot \frac{x_{\text{граница}} - x_1}{x_2 - x_1}$$

2. **Алгоритм Лиен-Барски**
   - Предназначен для отсечения отрезков.
   - Использует параметрическое представление линии:
     $$x = x_1 + t \cdot (x_2 - x_1), \quad y = y_1 + t \cdot (y_2 - y_1)$$
   - Определяет значение t для точек пересечения отрезка с границами окна.
   - Если $t \in [0,1]$, точка пересечения находится на отрезке.
   - Более эффективен, чем алгоритм Коэна-Сазерленда, так как работает с минимальным числом проверок.
3. **Алгоритм отсечения полигонов Сазерленда-Ходжмана**
   - Используется для отсечения многоугольников.
   - Применяется пошаговое отсечение по каждой стороне окна.
   - **Основные этапы:**
     - Проверяется положение каждой вершины относительно текущей стороны окна.
     - Если вершина внутри, она сохраняется. Если снаружи, вычисляются точки пересечения.
   - Итеративно строится новый многоугольник, лежащий полностью внутри окна.
4. **Алгоритм Вейлера-Атера**
   - Применяется для сложных случаев отсечения, например, для многоугольников с пересекающимися границами.
   - Использует понятие "входных" и "выходных" точек пересечения.
   - Формирует два подмножества вершин — видимый и невидимый многоугольники.

---

**Преимущества и недостатки**

**Преимущества:**

- Сокращение объёма вычислений при отрисовке.
- Эффективная обработка больших сцен с множеством объектов.
- Улучшение производительности графических приложений.

**Недостатки:**

- Усложнение реализации в случае сложных объектов (например, пересекающихся полигонов).
- Возможность появления артефактов при неточных вычислениях.

---

**Применение алгоритмов отсечения**

- Компьютерные игры: отображение только видимых объектов.
- Системы проектирования (CAD): работа с ограниченными областями чертежей.
- Геоинформационные системы (ГИС): отображение карт внутри заданного окна.
- Графические редакторы: обрезка изображений.

---

**Основные выводы:**

- Алгоритмы отсечения являются важной частью оптимизации в компьютерной графике.
- Каждый алгоритм предназначен для определённых задач, будь то отсечение линий или полигонов.

- Правильный выбор алгоритма зависит от сложности сцены и требований к производительности.

## 29. Двумерный алгоритм Коэна—Сазерленда.

Двумерный алгоритм Коэна—Сазерленда используется для отсечения отрезков относительно прямоугольного окна. Он разделяет пространство на области и применяет к каждому отрезку кодирование для определения его видимости. Этот алгоритм эффективен для обработки большого количества линий в сценах.

---

**Основные характеристики**

1. **Окно отсечения:**
   - Прямоугольная область, заданная координатами минимальной $(x_{min}, y_{min})$ и максимальной $(x_{max}, y_{max})$ границ.
2. **Видимость отрезков:**
   - Отрезок может быть:
     - **Полностью видимым:** Полностью внутри окна.
     - **Полностью невидимым:** Полностью за пределами окна.
     - **Частично видимым:** Частично пересекает границы окна.
3. **Кодирование областей (outcode):**
   Каждая точка пространства получает 4-битный код, который определяет её положение относительно окна:
   - **Биты:**
     - Бит 1 (слева от окна).
     - Бит 2 (справа от окна).
     - Бит 3 (ниже окна).
     - Бит 4 (выше окна).
   - Кодирование:
     - 0000 — точка внутри окна.
     - Остальные комбинации — точка снаружи.

---

**Этапы алгоритма**

1. **Определение кодов (outcode):**
   - Для каждой вершины отрезка вычисляется код относительно границ окна.
2. **Проверка видимости:**
   - Если оба кода равны 0000 — отрезок полностью видим.
   - Если логическое "И" (&) двух кодов не равно 0 — отрезок полностью невиден.
   - Иначе отрезок частично видим, и требуется его отсечение.
3. **Отсечение:**
   - Рассчитываются точки пересечения отрезка с границами окна.
   - Используются уравнения прямой для нахождения точек пересечения.

---

**Формулы пересечения**

1. **Вычисление пересечения с горизонтальными границами:**
$$x = x_1 + (x_2 - x_1) \cdot \frac{y_{\text{граница}} - y_1}{y_2 - y_1}$$
2. **Вычисление пересечения с вертикальными границами:**
$$y = y_1 + (y_2 - y_1) \cdot \frac{x_{\text{граница}} - x_1}{x_2 - x_1}$$

---

**Преимущества и недостатки**

**Преимущества:**

- Эффективен для обработки большого числа отрезков.
- Простая реализация с минимальными вычислениями.

**Недостатки:**

- Не подходит для сложных случаев, например, отсечения кривых или многоугольников.
- Требует дополнительных вычислений при частично видимых отрезках.

---

**Пример кода на Python**

```python
import matplotlib.pyplot as plt

# Границы окна
x_min, x_max = -5, 5
y_min, y_max = -5, 5

# Функция для вычисления outcode
def compute_outcode(x, y):
    code = 0
    if x < x_min:
        code |= 1  # Левее окна
    elif x > x_max:
        code |= 2  # Правее окна
    if y < y_min:
        code |= 4  # Ниже окна
    elif y > y_max:
        code |= 8  # Выше окна
    return code

# Алгоритм Коэна-Сазерленда
def cohen_sutherland_clip(x1, y1, x2, y2):
    outcode1 = compute_outcode(x1, y1)
    outcode2 = compute_outcode(x2, y2)
    accept = False

    while True:
        if outcode1 == 0 and outcode2 == 0:
            # Полностью внутри окна
            accept = True
            break
        elif outcode1 & outcode2 != 0:
            # Полностью вне окна
            break
        else:
            # Отрезок частично внутри окна
            x, y = 0, 0
            outcode_out = outcode1 if outcode1 != 0 else outcode2
            if outcode_out & 8:  # Выше окна
                x = x1 + (x2 - x1) * (y_max - y1) / (y2 - y1)
                y = y_max
            elif outcode_out & 4:  # Ниже окна
                x = x1 + (x2 - x1) * (y_min - y1) / (y2 - y1)
                y = y_min
            elif outcode_out & 2:  # Правее окна
                y = y1 + (y2 - y1) * (x_max - x1) / (x2 - x1)
                x = x_max
            elif outcode_out & 1:  # Левее окна
                y = y1 + (y2 - y1) * (x_min - x1) / (x2 - x1)
                x = x_min

            # Обновляем координаты
            if outcode_out == outcode1:
                x1, y1 = x, y
                outcode1 = compute_outcode(x1, y1)
            else:
                x2, y2 = x, y
                outcode2 = compute_outcode(x2, y2)

    if accept:
        return (x1, y1, x2, y2)
    else:
        return None

# Пример отрезков
lines = [(-7, -6, 6, 7), (-6, 6, 6, -6), (-3, -3, 3, 3)]
```

```
# Визуализация
fig, ax = plt.subplots()
for line in lines:
    result = cohen_sutherland_clip(*line)
    if result:
        x1, y1, x2, y2 = result
        ax.plot([x1, x2], [y1, y2], 'r-')
    ax.plot([line[0], line[2]], [line[1], line[3]], 'b--')

# Границы окна
ax.plot([x_min, x_max, x_max, x_min, x_min],
        [y_min, y_min, y_max, y_max, y_min], 'g-')

plt.title('Двумерный алгоритм Коэна-Сазерленда')
plt.gca().set_aspect('equal')
plt.grid()
plt.show()
```

---

**Основные выводы:**

- Двумерный алгоритм Коэна—Сазерленда — один из наиболее популярных и эффективных методов для отсечения отрезков.
- Простая структура и логика позволяют использовать его в реальных приложениях, таких как графические редакторы и системы отображения данных.

## 30. Алгоритм Лианга—Барски

Алгоритм Лианга—Барски является оптимизированным методом отсечения отрезков относительно прямоугольного окна. В отличие от алгоритма Коэна—Сазерленда, который использует кодирование областей, этот алгоритм основан на параметрической записи отрезка и сравнении его пересечений с границами окна.

---

**Основные характеристики**

1. **Отрезок в параметрической форме:**
   Отрезок представляется как:
   $$x = x_1 + t \cdot (x_2 - x_1), \quad y = y_1 + t \cdot (y_2 - y_1)$$
   где $t \in [0, 1]$.
2. **Границы окна:**
   Определяются $(x_{min}, y_{min})$ и максимальными $(x_{max}, y_{max})$ координатами.
3. **Проверка пересечений:**
   Используется сравнение параметра t для определения видимости отрезка.

---

**Этапы алгоритма**

1. **Определение направлений (p) и расстояний (q):**
   Для каждой из четырёх сторон окна вычисляются:
   $$p_1 = -(x_2 - x_1), \quad q_1 = x_1 - x_{\min}$$
   $$p_2 = x_2 - x_1, \quad q_2 = x_{\max} - x_1$$
   $$p_3 = -(y_2 - y_1), \quad q_3 = y_1 - y_{\min}$$
   $$p_4 = y_2 - y_1, \quad q_4 = y_{\max} - y_1$$
2. **Рассчитывается параметр t:**
   $$t = \frac{q}{p}, \quad p \neq 0$$
3. **Классификация пересечений:**
   - Если p>0, отрезок пересекает входную границу.

- Если p<0, отрезок пересекает выходную границу.
- Если p=0, отрезок параллелен границе:
  - Если q<0, отрезок полностью вне окна.

4. **Определение видимости:**
   - Поддерживаются два параметра $t_{in}$ (вход) и $t_{out}$ (выход), которые обновляются на основе пересечений.
   - Если $t_{in} \leq t_{out}$ , отрезок частично или полностью видим.

---

**Формулы**

1. **Границы параметра t:**
   - $t_{in} = \max(0, t_{min})$
   - $t_{out} = \min(1, t_{max})$
2. **Пересечения:**
   - Если $t_{in} > t_{out}$, отрезок полностью невидим.

---

**Преимущества и недостатки**

**Преимущества:**

- Более производительный по сравнению с Коэном—Сазерлендом, так как минимизирует количество вычислений.
- Эффективен для сцен с большим количеством прямых линий.

**Недостатки:**

- Сложнее в реализации.
- Не подходит для сложных форм, таких как многоугольники или кривые.

---

**Пример кода на Python**

```
import matplotlib.pyplot as plt

# Границы окна
x_min, x_max = -5, 5
y_min, y_max = -5, 5

# Алгоритм Лианга–Барски
def liang_barsky_clip(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    t_in = 0.0
    t_out = 1.0

    def clip_test(p, q):
        nonlocal t_in, t_out
        if p < 0:
            t = q / p
            if t > t_out:
                return False
            elif t > t_in:
                t_in = t
        elif p > 0:
            t = q / p
            if t < t_in:
                return False
            elif t < t_out:
                t_out = t
        elif q < 0:
            return False
        return True

    # Проверка пересечений с границами окна
    if (clip_test(-dx, x1 - x_min) and
        clip_test(dx, x_max - x1) and
        clip_test(-dy, y1 - y_min) and
        clip_test(dy, y_max - y1)):
        x1_clip = x1 + t_in * dx
```

```
            y1_clip = y1 + t_in * dy
            x2_clip = x1 + t_out * dx
            y2_clip = y1 + t_out * dy
            return (x1_clip, y1_clip, x2_clip, y2_clip)
    return None

# Пример отрезков
lines = [(-7, -6, 6, 7), (-6, 6, 6, -6), (-3, -3, 3, 3)]

# Визуализация
fig, ax = plt.subplots()
for line in lines:
    result = liang_barsky_clip(*line)
    if result:
        x1, y1, x2, y2 = result
        ax.plot([x1, x2], [y1, y2], 'r-')
    ax.plot([line[0], line[2]], [line[1], line[3]], 'b--')

# Границы окна
ax.plot([x_min, x_max, x_max, x_min, x_min],
        [y_min, y_min, y_max, y_max, y_min], 'g-')

plt.title('Алгоритм Лианга–Барски')
plt.gca().set_aspect('equal')
plt.grid()
plt.show()
```

---

**Основные выводы:**

- Алгоритм Лианга—Барски является мощным инструментом для отсечения отрезков относительно прямоугольного окна.
- Он более эффективен, чем другие методы, такие как алгоритм Коэна—Сазерленда, особенно при работе с большим количеством линий.

## 31. Особенности растеризации прямой линии

Растеризация прямой линии — это процесс преобразования координатной (математической) формы прямой в пиксели на экране. В компьютерной графике задача растеризации прямой линии возникает при необходимости отобразить отрезок или линию на сетке пикселей. Это важный этап в работе графических приложений, таких как графические редакторы, игры и визуализация данных.

---

**Основные характеристики**

1. **Отрезок прямой:**
   Прямая линия на экране обычно представляется как последовательность пикселей, которые располагаются на пути линии в двумерном пространстве. Отрезок между двумя точками $P_1(x_1, y_1)$и $P_2(x_2, y_2)$растеризуется, и для этого используется определенный алгоритм.
2. **Математическая формулировка линии:**
   Прямая линия между двумя точками может быть опиана уравнением:
   $$y = mx + b$$
   где m — это угловой коэффициент, а b — значение y на оси при x=0. Важно, что на экране мы можем работать с целочисленными координатами пикселей, что иногда вызывает ошибки округления.
3. **Цель растеризации:**
   На каждом шаге алгоритма необходимо определить, какой пиксель из сетки будет "занят" линией, основываясь на математическом уравнении линии.

---

**Алгоритмы растеризации**

1. **Алгоритм Брезенхэма (Bresenham's Line Algorithm):**
   Этот алгоритм является наиболее популярным и эффективным для растеризации прямых линий. Он использует целочисленные вычисления для определения ближайшего пикселя,

который лежит на линии, что минимизирует ошибки округления и повышает производительность.

2. **Алгоритм Брезенхэма для "толстых" линий:**
   Если необходимо отобразить линии с заданной толщиной, алгоритм Брезенхэма может быть модифицирован для растеризации таких линий, создавая несколько пикселей вокруг основной линии для имитации толщины.
3. **Алгоритм DDA (Digital Differential Analyzer):**
   Этот алгоритм использует арифметические операции с плавающей точкой для растеризации прямой. Он менее эффективен, чем алгоритм Брезенхэма, так как требует вычислений с плавающей точкой, что увеличивает вычислительную нагрузку.

---

**Преимущества и недостатки**
**Преимущества алгоритма Брезенхэма:**

- Использует целочисленные операции, что делает алгоритм быстрым и эффективным.
- Простой в реализации.
- Предотвращает ошибки округления, часто возникающие при использовании операций с плавающей точкой.

**Недостатки:**

- Сложность в реализации для линий с наклоном 45 градусов, когда шаг по осям одинаков.
- Алгоритм может потребовать адаптации для работы с линиями различной толщины или кривыми.

---

**Основные выводы**

- Растеризация прямой линии — это ключевая задача для построения графики в компьютерных системах.
- Алгоритм Брезенхэма является оптимальным методом растеризации для большинства прямых линий благодаря его эффективности и точности.
- Применение различных алгоритмов растеризации, таких как DDA и алгоритм Брезенхэма, позволяет улучшить качество изображения и уменьшить вычислительные затраты.

## 32. Алгоритм Брезенхема для прямой линии

**Алгоритм Брезенхема** — это дискретный метод построения прямой линии на сетке пикселей. Он минимизирует вычисления, используя только целые числа.

**Основная идея:**

Вместо вычисления координат всех точек линии с использованием дробных значений, алгоритм определяет, какой пиксель ближе всего к идеальной линии, и выбирает его для закраски.

**Этапы выполнения:**

1. **Входные данные:**
   - Начальная $(x_0, y_0)$ и конечная $(x_1, y_1)$ точки.
2. **Подготовка:**
   - Рассчитываются приросты по осям:
     $$dx = x_1 - x_0, \quad dy = y_1 - y_0$$
   - Определяется основной шаг (по $x$ или $y$) в зависимости от направления линии.
3. **Начальная ошибка:**
   - Для основного направления по оси $x$:
     $$D = 2dy - dx$$
4. **Итеративное построение:**
   - Начальная точка закрашивается.
   - На каждом шаге:
     - Проверяется ошибка $D$:
       - Если $D > 0$, происходит смещение по обеим осям $(x, y)$, и ошибка корректируется:
         $$D = D + 2(dy - dx)$$
       - Иначе смещается только $x$, а ошибка обновляется:
         $$D = D + 2dy$$
     - Переход к следующему пикселю до достижения конечной точки.

**Особенности:**

- Использует только целые числа, что делает его быстрым.
- Поддерживает построение линии во всех восьми октантах.
- Простота адаптации для различных углов наклона.

↓

Алгоритм эффективен для отображения линий с минимальными артефактами.

## 33. Способы растеризации окружности. Алгоритм Брезенхема для окружности

### Способы растеризации окружности

Растеризация окружности — процесс приближения идеальной окружности на дискретной пиксельной сетке. Основные методы:

1. **Алгебраический метод**:
   - Использует уравнение окружности $x^2 + y^2 = r^2$.
   - Для каждого значения $x$ вычисляется $y$ и наоборот.
   - Недостатки: требует работы с плавающей точкой, может быть медленным.
2. **Параметрический метод**:
   - Уравнение окружности задаётся параметрически:
     $$x = r\cos\theta, \quad y = r\sin\theta$$
   - Путём изменения угла $\theta$ рисуются точки.
   - Недостатки: необходимость вычисления тригонометрических функций.
3. **Алгоритм Брезенхема для окружности**:
   - Оптимизированный метод растеризации, использующий целочисленные вычисления.
   - Обеспечивает высокую производительность и точность.

**Алгоритм Брезенхема для окружности** строит её растеризованное представление, основываясь на симметрии относительно осей и диагоналей.

**Основная идея:**

- Уравнение окружности:
  $$x^2 + y^2 = r^2$$
- Вместо непосредственного вычисления корней, алгоритм определяет, в какой из соседних точек сетки переместиться, чтобы минимизировать отклонение.

**Этапы выполнения:**

1. **Входные данные**:
   Радиус $r$ и центр окружности $(x_c, y_c)$.
2. **Инициализация**:
   - Начальная точка: $(x, y) = (0, r)$.
   - Начальная ошибка:
     $$D = 3 - 2r$$
3. **Итерации**:
   - Для каждой точки в первой восьмой части окружности:
     - Рисуются точки во всех восьми симметричных положениях:
       $(x_c \pm x, y_c \pm y)$ и $(x_c \pm y, y_c \pm x)$.
     - Если $D > 0$:
       $$D = D + 4(x - y) + 10, \quad y = y - 1$$
     - Иначе:
       $$D = D + 4x + 6$$
     - Увеличить $x$ на 1.
4. **Остановка**:
   Алгоритм заканчивается, когда $x \geq y$.

**Преимущества:**

- Использует только целые числа.
- Высокая производительность благодаря симметрии.

**Особенности:**

- Алгоритм рисует только одну восьмую окружности, остальные части строятся за счёт симметрии.

## 34. Заполнение сплошных областей. Тест принадлежности точки многоугольнику

**Заполнение сплошных областей**

Заполнение сплошных областей — это процесс закрашивания замкнутой области, заданной многоугольником, окружностью или другим контуром. Основные алгоритмы:

**1. Алгоритм построчного заполнения (Scanline Filling):**

- Заполняет область построчно, определяя пересечения строки с границами.
- Этапы:
    1. Находятся пересечения строки с рёбрами многоугольника.
    2. Отсортированные точки пересечения используются для закраски сегментов строки.
- Преимущества: эффективно для сложных многоугольников.

**2. Алгоритм заливки с затравкой (Flood Fill):**

- Начинает закрашивание из стартовой точки и распространяется, пока не достигнет границы.
- Типы:
    - **4-связный (4-connected):** рассматривает соседей сверху, снизу, слева и справа.
    - **8-связный (8-connected):** включает диагональных соседей.
- Применяется в инструментах рисования, например, "заливка" в графических редакторах.

**3. Boundary Fill (Заливка по границе):**

- Закрашивает область внутри заданной границы, начиная с точки внутри.
- Проверяет соседние пиксели и заполняет их, если они не совпадают с цветом границы.

**Тест принадлежности точки многоугольнику**

Определяет, находится ли заданная точка P(x,y) внутри многоугольника. Основные методы:

**1. Алгоритм "Чётности пересечений" (Odd-Even Rule):**

- Проводится луч из точки P в произвольном направлении.
- Считается количество пересечений луча с рёбрами многоугольника:
    - Если пересечений нечётное количество — точка внутри.
    - Если чётное — точка снаружи.

**2. Алгоритм "Уголового подсчёта" (Winding Number):**

- Для каждого ребра многоугольника вычисляется направление обхода относительно точки PP.
- Подсчитывается количество оборотов многоугольника вокруг точки:
    - Если число не равно нулю — точка внутри.
    - Если равно нулю — точка снаружи.

**Пример применения:**

- Проверка попадания точки в кликабельную область на экране.
- Определение внутренних и внешних точек для операций с многоугольниками (например, в GIS или CAD-системах).

Эти методы широко применяются в графике, геометрическом моделировании и игровых движках.

## 35. Стиль заполнения. Особенности применения различных стилей.

**Стиль заполнения (Fill Style)** – это способ задания цвета, узора или текстуры при закрашивании внутренней области графического примитива (например, прямоугольника, многоугольника, круга и т.д.). Выбор стиля заполнения значительно влияет на визуальное восприятие объекта.

**Основные стили заполнения:**

1. **Сплошное заполнение (Solid Fill):**
   - Область примитива закрашивается сплошным цветом.
   - Наиболее простой и широко используемый стиль.
   - **Особенности применения:**
     - Используется в случаях, когда требуется четкое разделение объектов по цвету.
     - Подходит для диаграмм, графиков и элементов пользовательского интерфейса.
2. **Заполнение узором (Pattern Fill):**
   - Примитив заполняется повторяющимся узором, который состоит из набора точек, линий, штрихов или графических элементов.
   - Узоры могут быть стандартными (предопределенными) или пользовательскими.
   - **Особенности применения:**
     - Применяется для декоративного оформления.
     - Используется для передачи дополнительной информации, например, выделения разных областей на чертежах или схемах.
3. **Градиентное заполнение (Gradient Fill):**
   - Плавный переход между двумя или более цветами.
   - Типы градиентов:
     - Линейный (Linear Gradient) – переход вдоль линии.
     - Радиальный (Radial Gradient) – переход от центра к краям.
     - Конический (Conic Gradient) – переход по кругу.
   - **Особенности применения:**
     - Создает эффект объема и глубины.
     - Используется в дизайне, иллюстрациях и интерфейсах.
4. **Текстурное заполнение (Texture Fill):**
   - В качестве заполнения используется изображение или текстура.
   - **Особенности применения:**
     - Используется для имитации материалов (например, древесины, металла, ткани).
     - Популярен в 3D-моделировании и графическом дизайне.
5. **Полупрозрачное заполнение (Transparent Fill):**
   - Цвет заполнения имеет определенный уровень прозрачности (alpha channel).
   - **Особенности применения:**
     - Позволяет видеть подложку или перекрывающиеся элементы.
     - Используется для создания сложных визуальных эффектов в инфографике или интерфейсах.

**Применение различных стилей:**

- В **научной визуализации** (например, графики, диаграммы): предпочтение отдается сплошному заполнению для ясности.
- В **компьютерных играх** и **3D-графике**: активно используются текстурное и градиентное заполнение для реалистичности.
- В **графическом дизайне**: текстурное и градиентное заполнение применяется для повышения эстетики.
- В **инженерной графике**: часто используются узоры для обозначения различных материалов на чертежах.

## 36. Текстуры в трехмерной графике

**Текстуры** – это изображения, которые накладываются на поверхности 3D-объектов для придания реалистичности или стилизации. Они заменяют сложную геометрию визуальными деталями.

**Основные типы текстур:**

1. **Diffuse** – цвет поверхности.
2. **Normal Map** – иллюзия неровностей без дополнительных полигонов.
3. **Specular/Roughness** – отражение и блеск.
4. **Opacity** – прозрачность.
5. **Displacement** – изменение геометрии.
6. **Ambient Occlusion** – тени в труднодоступных местах.
7. **Emissive** – свечение.

**Применение:**

- Добавление деталей (например, кирпичная стена, кожа).
- Создание реалистичных материалов (металл, ткань, стекло).
  Текстуры мапятся на объект с помощью **UV-развёртки**.

## 37. Представление кривых линий

В компьютерной графике кривые линии используются для моделирования плавных контуров и очертаний объектов. Наиболее распространённые методы представления кривых линий связаны с использованием полиномиальных и сплайновых кривых.

**Сплайновые кривые**

**Сплайн** — это кривая, удовлетворяющая критериям гладкости и построенная на основе набора базовых (опорных) точек.

**Кривые Безье**

Кривые Безье представляют собой один из основных инструментов для работы с кривыми линиями. Они были разработаны для проектирования автомобильных кузовов и активно применяются в системах автоматизированного проектирования (CAD) и компьютерной графике.

**Свойства кривых Безье:**

1. **Выпуклая оболочка**: Кривая всегда располагается внутри выпуклой оболочки, определённой опорными точками.
2. **Гладкость**: Кривая Безье является непрерывной и гладкой.
3. **Локальный контроль**: Перемещение одной из опорных точек влияет только на соответствующую часть кривой.
4. **Лёгкость вычислений**: Для расчёта используются полиномы Бернштейна, обеспечивающие стабильность вычислений.

**Формула кубической кривой Безье:**

Для четырёх опорных точек $P_0, P_1, P_2, P_3$, кубическая кривая Безье описывается уравнением:

$$R(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t) P_2 + t^3 P_3, \quad t \in [0,1].$$

**Построение составной кривой**

Для построения сложных кривых линии Безье разбиваются на сегменты, соединяющие группы из четырёх опорных точек. Чтобы кривая оставалась плавной, необходимо, чтобы каждые три соседние точки на границе сегментов лежали на одной прямой.

**Применение**

- **Векторная графика** (Adobe Illustrator, CorelDRAW): кривые Безье лежат в основе инструментов рисования.
- **Компьютерная анимация**: используются для описания траекторий движения.
- **CSS-анимация**: определяют плавность переходов.
- **3D-моделирование**: применяются для создания контуров объектов.

## 38. Глобальная интерполяция

**Определение:** Глобальная интерполяция — метод построения кривой, которая проходит через все заданные опорные точки. Обычно это осуществляется с использованием полинома высокой степени, описывающего всю кривую.

**Преимущества глобальной интерполяции:**

1. Кривая точно проходит через все опорные точки.
2. Подходит для задач, где важна точность в каждом опорном узле.

**Недостатки:**

1. **«Волнение» кривой:** Полиномы высокой степени имеют тенденцию к сильным колебаниям между точками, особенно если точки расположены неравномерно.
2. **Числовая нестабильность:** При большом количестве точек возникают трудности в вычислении коэффициентов из-за плохой обусловленности системы уравнений.
3. **Экстраполяция:** За пределами заданного интервала полиномы могут неадекватно приближать значения, уходя в бесконечность или внося большие ошибки.

Для уменьшения недостатков часто применяются локальные методы, такие как сплайны, где каждый сегмент описывается отдельным низкостепенным полиномом. Например, сплайновая интерполяция Catmull-Rom позволяет строить гладкие кривые, которые:

- проходят через опорные точки;
- обеспечивают геометрическую непрерывность;
- исключают излишнюю гибкость полинома, ограничивая вычисления локальными областями.

## 39. Локальная интерполяция. Кусочно-линейная интерполяция

**Определение:** Локальная интерполяция — метод построения интерполяционной кривой, в котором кривая создаётся по частям (локально) на каждом интервале между соседними опорными точками. Вместо использования одного глобального полинома высокой степени для всех точек, каждый сегмент определяется отдельной функцией низкой степени.

**Преимущества локальной интерполяции:**

1. **Стабильность:** Нет проблем с "волнением" кривой, характерным для глобальной интерполяции.
2. **Лёгкость вычислений:** На каждом интервале используются низкостепенные функции, что упрощает расчёты.
3. **Гибкость:** Можно использовать разные подходы на разных интервалах.

**Недостатки:**

- Может быть менее гладкой, чем глобальная интерполяция, особенно если используются простые функции (например, кусочно-линейная интерполяция).

## Кусочно-линейная интерполяция

**Определение:** Кусочно-линейная интерполяция — частный случай локальной интерполяции, где на каждом интервале между соседними опорными точками строится линейная функция.

**Формула для кусочно-линейной интерполяции:** Для двух соседних точек $(x_i, y_i)$ и $(x_{i+1}, y_{i+1})$, линейная функция определяется как:

$$y = y_i + \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i), \quad x_i \leq x \leq x_{i+1}.$$

**Преимущества:**

1. Простота реализации.
2. Быстрота вычислений.
3. Подходит для аппроксимации функций с небольшими изменениями между точками.

**Недостатки:**

1. Кривая не гладкая: в местах стыков интервалов появляются разрывы в первой производной (угловатость).
2. Для функций с быстрыми изменениями может потребоваться много опорных точек для точного представления.

## 40. Локальная интерполяция. Квадратичная интерполяция.

**Определение:** Квадратичная интерполяция — это метод локальной интерполяции, где для каждого интервала между соседними опорными точками используется полином второй степени (квадратичный полином). Квадратичная интерполяция обеспечивает более плавную аппроксимацию, чем кусочно-линейная, так как кривая становится гладкой внутри каждого интервала.

**Формула:** Полином второго порядка имеет вид:

$$P(x) = a_0 + a_1 x + a_2 x^2,$$

где коэффициенты $a_0$, $a_1$, $a_2$ определяются из условий, задаваемых для каждой пары соседних точек.

Для интервала $[x_i, x_{i+1}]$, коэффициенты рассчитываются так, чтобы:

1. Полином проходил через точки $(x_i, y_i)$ и $(x_{i+1}, y_{i+1})$.
2. Гладкость обеспечивалась за счёт согласования с соседними интервалами (опционально).

**Способы построения:**

1. **Без учёта производной:** Только значения в двух соседних точках используются для определения коэффициентов.
2. **С учётом производной:** Добавляется условие гладкости, учитывающее значения первой производной в концах интервала.

**Преимущества:**

1. Квадратичная интерполяция лучше приближает нелинейные функции по сравнению с линейной.
2. Может быть достаточно гладкой (при согласовании производных на стыках интервалов).

**Недостатки:**

1. Сложнее реализовать, чем линейную интерполяцию.
2. Может быть избыточной для данных с незначительными изменениями.

## 41. Параметрические сплайны в форме Эрмита

**Определение:** Параметрический сплайн Эрмита — это метод интерполяции, в котором кривая задаётся полиномами третьей степени (кубическими) на каждом интервале между опорными точками. Эти полиномы определяются как функции параметра $t \in [0, 1]$, а не явной зависимостью $y(x)$, что позволяет задать кривую в многомерном пространстве, например, в 2D или 3D.

**Формулы:** На интервале между двумя точками $\mathbf{P}_0$ и $\mathbf{P}_1$ с заданными касательными $\mathbf{T}_0$ и $\mathbf{T}_1$, параметрический сплайн в форме Эрмита записывается как:

$$\mathbf{P}(t) = (2t^3 - 3t^2 + 1)\mathbf{P}_0 + (-2t^3 + 3t^2)\mathbf{P}_1 + (t^3 - 2t^2 + t)\mathbf{T}_0 + (t^3 - t^2)\mathbf{T}_1,$$

где:

- $t \in [0, 1]$ — параметр, изменяющийся вдоль сегмента,
- $\mathbf{P}_0, \mathbf{P}_1$ — начальная и конечная точки сегмента,
- $\mathbf{T}_0, \mathbf{T}_1$ — касательные в этих точках.

**Свойства:**

1. **Гладкость:** Кривая и её первая производная непрерывны.
2. **Гибкость:** Форма кривой регулируется не только положением точек, но и заданными касательными.
3. **Параметризация:** Сплайны могут быть заданы для каждой координаты ($x(t)$, $y(t)$, $z(t)$) отдельно, что позволяет строить произвольные траектории в пространстве.

**Построение:**

1. Определите набор опорных точек $\mathbf{P}_i$ и касательных $\mathbf{T}_i$. Касательные могут быть заданы:
   - Явно (если известны значения),
   - Автоматически (например, как центральные разности между соседними точками).
2. Для каждого сегмента $[\mathbf{P}_i, \mathbf{P}_{i+1}]$ определите функцию $\mathbf{P}(t)$, используя вышеописанную формулу.
3. Соедините все сегменты для получения общей кривой.

**Применение:**

1. Графическая визуализация (анимации, игры, моделирование движений).
2. Построение траекторий в CAD-системах.
3. Интерполяция в геометрии и робототехнике.

## 42. Параметрические сплайны в форме Безье

Параметрические сплайны в форме Безье являются важным инструментом в компьютерной графике. Они представляют собой гладкие кривые, построенные на основе набора базовых (опорных) точек. Основные положения:

1. **Определение:** Сплайн — это кривая, удовлетворяющая критериям гладкости. В форме Безье сплайн определяется набором базовых точек $P_0, P_1, ..., P_n$. Кубическая кривая Безье выражается формулой:

$$R(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t)P_2 + t^3 P_3,\ t \in [0,1].$$

2. **Свойства**:
   - Кривая проходит внутри выпуклой оболочки, заданной опорными точками.
   - Начальная и конечная точки кривой совпадают с первыми и последними базовыми точками ($P_0, P_3$).
   - Направления в начале и конце кривой задаются отрезками $P_0P_1$ и $P_2P_3$.
3. **Построение**:
   - Для вычислений используются полиномы Бернштейна, что позволяет создать кривую любой степени.
   - Чтобы избежать высокой степени полиномов при большом числе точек, используют составные кривые из сегментов. Стыки сегментов обеспечивают геометрическую непрерывность при условии, что точки на границе лежат на одной прямой.
4. **Алгоритм де Кастельжо**:
   - Рекурсивный метод для построения кривой. Для заданного значения параметра $t$ координаты точки на кривой вычисляются путём линейной интерполяции между базовыми точками.
5. **Практическое применение**:
   - Широко используются в дизайне, анимации и системах автоматизированного проектирования (например, AutoCAD, CorelDRAW).
   - Подходят для описания гладких контуров и аппроксимации функций.

Кривые Безье позволяют эффективно строить и управлять параметрическими формами, обеспечивая высокую гибкость и точность.

## 43
## B-сплайны (B-splines) и их особенности

**B-сплайны** (Basis splines) — это математические функции, используемые для построения гладких кривых и поверхностей. Они являются линейной комбинацией базисных функций, каждая из которых влияет на форму только части кривой, что обеспечивает локальное управление.

---

### Основные особенности B-сплайнов:

1. **Гладкость и гибкость:**
   - B-сплайны обеспечивают высокую степень гладкости кривых.
   - При увеличении степени полинома (например, кубические B-сплайны) кривые становятся более гладкими.
2. **Локальный контроль:**
   - Изменение одного из контрольных точек влияет только на часть кривой, а не на всю кривую целиком.
   - Это свойство достигается благодаря свойству базисных функций, которые имеют ограниченную область действия.
3. **Кусочно-полиномиальная структура:**
   - B-сплайны строятся как кусочные полиномы определенной степени, которые соединены друг с другом в заданных узлах (точках, называемых **узловым вектором**).
4. **Узловой вектор (Knots):**
   - Узловой вектор определяет, где соединяются полиномы, и контролирует поведение сплайна. Это массив значений, который задает "вес" каждой базисной функции.
   - Узлы могут быть равномерно распределены (равные промежутки) или неравномерно (для более сложных форм).
5. **Степень полинома:**
   - B-сплайн задается полиномами степени k−1, где k — количество базисных функций. Например, кубический B-сплайн (k=4) состоит из полиномов третьей степени.
6. **Контрольные точки:**
   - Геометрия кривой определяется контрольными точками. Сами контрольные точки могут не лежать на кривой, но они влияют на ее форму.
7. **Линейная комбинация:**
   - B-сплайн — это линейная комбинация базисных функций $N_{(t)}, k(t)$, где каждая функция умножается на вес или координату соответствующей контрольной точки.

$$C(t) = \sum_{i=0}^{n} P_i \cdot N_{i,k}(t)$$

Здесь $P_i$ — контрольные точки, а $N_{i,k}(t)$ — базисные функции степени $k-1$.

8. **Глобальная непрерывность:**
   - B-сплайны обладают свойством C^k-непрерывности (гладкости), где k — степень базисных функций.
9. **Общие случаи:**
   - Если узлы многократно повторяются, гладкость в этих точках уменьшается, а кривая может даже пересекать узлы.
   - B-сплайны включают в себя линейные, квадратичные, кубические и сплайны более высокой степени.
10. **Преимущества над полиномами:**
    - B-сплайны более устойчивы к численным ошибкам.
    - Избегают эффекта Рунге (осцилляций в интерполяции).

---

### Применения B-сплайнов:

- **Компьютерная графика:** Создание гладких кривых и поверхностей.

- **CAD/CAM системы:** Построение геометрии деталей.
- **Анимация:** Управление движениями объектов.
- **Аппроксимация данных:** Построение моделей на основе экспериментальных данных.
- **Решение дифференциальных уравнений:** Для численного анализа.

---

B-сплайны являются важным инструментом в вычислительной математике, предлагая баланс между гибкостью, точностью и управляемостью.

## 44

Фундаментальные сплайны — это математические функции, используемые для интерполяции и аппроксимации кривых и поверхностей. Они находят широкое применение в компьютерной графике, CAD-системах и других областях, где требуется создание гладких кривых.

### Сплайны Catmull-Rom

Сплайны Catmull-Rom — это один из типов интерполяционных сплайнов, который обеспечивает гладкую кривую, проходящую через заданные контрольные точки. Они являются частью более общего класса сплайнов, называемого "интерполяционными сплайнами", и имеют следующие характеристики:

1. Интерполяция: Сплайн проходит через все заданные контрольные точки. Это отличает их от некоторых других типов сплайнов, таких как B-сплайны, которые могут не проходить через контрольные точки.
2. Гладкость: Сплайны Catmull-Rom обладают непрерывными производными до второго порядка, что обеспечивает гладкость кривой.
3. Параметризация: Кривые Catmull-Rom могут быть параметризованы по длине или по времени, что позволяет контролировать скорость движения по кривой.
4. Формула: Для построения сегмента сплайна Catmull-Rom между четырьмя контрольными точками $P_0$, $P_1$, $P_2$, $P_3$ используется следующая формула:

$$C(t) = 0.5 \cdot ((2P_1 + (-P_0 + P_2)t + (2P_0 - 5P_1 + 4P_2 - P_3)t^2 + (-P_0 + 3P_1 - 3P_2 + P_3)t^3))$$

где $t$ — параметр, который изменяется от 0 до 1 для каждого сегмента между контрольными точками.

5. Применение: Сплайны Catmull-Rom широко используются в анимации, компьютерной графике и в приложениях, где необходимо создавать плавные кривые, такие как векторная графика и моделирование.

### Преимущества и недостатки

Преимущества:

• Легкость в использовании и понимании.
• Обеспечивают гладкие и визуально приятные кривые.
• Простота в реализации.

Недостатки:

• Могут создавать нежелательные колебания (осцилляции) при использовании большого количества контрольных точек.
• Не всегда обеспечивают оптимальные свойства для всех типов задач.

Сплайны Catmull-Rom являются мощным инструментом для создания гладких кривых и часто используются в сочетании с другими методами для достижения желаемых визуальных эффектов.

## 45

Бикубические поверхности — это тип поверхностей, используемых в компьютерной графике, CAD-системах и других областях, где требуется создание гладких и непрерывных 3D-форм. Они являются обобщением бикубических сплайнов, которые представляют собой двумерные функции, определяемые на квадратной сетке контрольных точек.

### Основные характеристики бикубических поверхностей:

1. Определение: Бикубическая поверхность задается с помощью 16 контрольных точек, расположенных в виде 4x4 сетки. Каждая контрольная точка влияет на форму поверхности, обеспечивая гибкость в моделировании.

2. Гладкость: Бикубические поверхности обладают непрерывными производными до третьего порядка, что обеспечивает высокую степень гладкости. Это делает их особенно полезными для создания органических форм и сложных объектов.
3. Интерполяция и аппроксимация: Бикубические поверхности могут интерполировать заданные контрольные точки или аппроксимировать их, в зависимости от конкретной реализации.
4. Параметризация: Поверхность определяется с помощью параметров u и v, которые изменяются в диапазоне от 0 до 1. Эти параметры позволяют перемещаться по поверхности и вычислять ее координаты в зависимости от значений параметров.
5. Формула: Бикубическая поверхность может быть задана следующим образом:

$$S(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} P_{ij} B_i(u) B_j(v)$$

где $P_{ij}$ — контрольные точки, а $B_i(u)$ и $B_j(v)$ — бикубические базисные функции, которые обеспечивают плавный переход между контрольными точками.
6. Базисные функции: Бикубические базисные функции имеют свойства непрерывности и гладкости, что позволяет создавать плавные переходы между контрольными точками.

▎Преимущества бикубических поверхностей:

• Гладкость: Высокая степень гладкости делает их идеальными для создания сложных и органических форм.
• Контроль: Позволяют легко управлять формой поверхности путем изменения контрольных точек.

• Универсальность: Широко применяются в различных областях, включая графику, анимацию и CAD.

▎Недостатки бикубических поверхностей:

• Сложность вычислений: Вычисление бикубических поверхностей может быть более ресурсоемким по сравнению с простыми поверхностями.
• Устойчивость к осцилляциям: При использовании большого количества контрольных точек могут возникать нежелательные колебания поверхности.

▎Применение:

Бикубические поверхности находят применение в таких областях, как:
• Компьютерная графика (моделирование объектов, анимация).
• CAD (проектирование и создание инженерных моделей).
• Визуализация данных и научные вычисления.

В целом, бикубические поверхности являются мощным инструментом для создания гладких и сложных 3D-форм, обеспечивая высокую степень контроля над их геометрией.

## 46

Удаление невидимых линий и поверхностей (Hidden Line and Surface Removal) — это важная задача в компьютерной графике, которая позволяет улучшить визуализацию 3D-объектов, скрывая те части, которые не видны для наблюдателя. Существует несколько алгоритмов для решения этой задачи, каждый из которых имеет свои особенности и применения.

▎Основные алгоритмы удаления невидимых линий и поверхностей:

1. Алгоритм Z-буфера (Depth Buffering):
   • Принцип работы: Каждому пикселю на экране сопоставляется значение глубины (Z-координаты). При отрисовке каждого объекта сравнивается его глубина с уже имеющимся значением в Z-буфере. Если новая глубина меньше (объект ближе к камере), то пиксель обновляется.
   • Преимущества: Простота реализации и возможность работы с произвольными сценами.
   • Недостатки: Требует дополнительной памяти для хранения Z-буфера, что может быть проблемой при больших сценах.
2. Алгоритм удаления невидимых поверхностей с использованием сортировки по глубине (Depth Sorting):
   • Принцип работы: Все объекты сортируются по их расстоянию от камеры, и отрисовываются в порядке от самых дальних до самых близких.
   • Преимущества: Простота и интуитивность.
   • Недостатки: Проблемы с пересечениями объектов и сложностью в случае перекрывающихся объектов.

3. Алгоритм отсечения (Clipping):
• Принцип работы: На основе геометрических свойств сцены и камеры отсеиваются невидимые части объектов еще до их отрисовки.
• Преимущества: Уменьшает количество полигонов, которые нужно обрабатывать, что может повысить производительность.
• Недостатки: Сложность реализации и необходимость учитывать множество условий.
4. Алгоритм BSP-деревьев (Binary Space Partitioning):
• Принцип работы: Сцена разбивается на множество полигонов, которые организуются в BSP-дерево. Это позволяет эффективно определять, какие поверхности видимы из определенной точки.
• Преимущества: Эффективен для сложных сцен с большим количеством объектов.
• Недостатки: Требует предварительной обработки данных и может быть сложным в реализации.
5. Алгоритмы на основе сеток (Mesh-based Algorithms):
• Принцип работы: Используются структуры данных, такие как сетки или графы, чтобы представлять сцены и определять видимость объектов.
• Преимущества: Позволяют эффективно управлять сложными сценами.
• Недостатки: Могут требовать значительных вычислительных ресурсов.
6. Рэйтрейсинг (Ray Tracing):
• Принцип работы: Для каждого пикселя на экране проводится "луч" из камеры в сцену. Если луч пересекает объект, определяется его цвет и текстура. Невидимые поверхности игнорируются.
• Преимущества: Высокое качество изображения и реалистичное освещение.
• Недостатки: Высокие вычислительные затраты и время рендеринга.

Общие особенности:

• Сложность реализации: Некоторые алгоритмы требуют более сложной реализации и предварительной обработки данных.
• Производительность: Выбор алгоритма часто зависит от требований к производительности и качеству визуализации.
• Тип сцены: Разные алгоритмы могут быть более эффективными для различных типов сцен (например, сложные сцены с множеством объектов или простые сцены).
• Качество изображения: Некоторые алгоритмы могут обеспечивать более высокое качество изображения, но требуют больше времени на обработку.

В зависимости от конкретных требований приложения и особенностей сцены, выбирается наиболее подходящий алгоритм удаления невидимых линий и поверхностей

## 47

Алгоритм Z-буфера (или Depth Buffering) — это один из самых распространённых методов удаления невидимых поверхностей в компьютерной графике. Он используется для определения, какие части объектов видимы из определённой точки зрения, и позволяет корректно отображать 3D-сцены на 2D-экране.

Принцип работы алгоритма Z-буфера:

1. Инициализация Z-буфера:
• Создаётся Z-буфер, который имеет такую же разрешающую способность, как и кадр (изображение), и инициализируется значениями, представляющими максимальную глубину (например, бесконечность). Это означает, что изначально все пиксели считаются "дальними".
2. Отрисовка объектов:
• При отрисовке каждого объекта (или примитива) сцены для каждого пикселя, который должен быть закрашен, вычисляется его значение глубины (Z-координата).
• Если значение глубины этого пикселя меньше, чем текущее значение в Z-буфере для этого пикселя, то:
• Обновляется цвет пикселя в кадре.
• Обновляется значение глубины в Z-буфере.
3. Вывод изображения:
• После завершения отрисовки всех объектов на экран выводится финальное изображение, где видны только те части объектов, которые находятся ближе к камере.

Преимущества алгоритма Z-буфера:

• Простота реализации: Алгоритм довольно прост в реализации и не требует сложной предварительной обработки данных.
• Гибкость: Может работать с произвольными сценами и не требует сортировки объектов.
• Поддержка сложных геометрий: Эффективно обрабатывает сцены с множеством пересекающихся объектов.

Недостатки алгоритма Z-буфера:

• Память: Требует дополнительной памяти для хранения Z-буфера, что может быть проблемой при больших разрешениях или сложных сценах.
• Точность: Ограниченная точность представления глубины может приводить к артефактам, таким как "зазоры" (z-fighting), когда два объекта находятся очень близко друг к другу.
• Производительность: В некоторых случаях может быть менее производительным по сравнению с другими методами, особенно при больших объемах данных или сложных сценах.

Применение:

Алгоритм Z-буфера широко используется в различных областях компьютерной графики, включая видеоигры, 3D-моделирование и визуализацию. Он является основным методом рендеринга в большинстве современных графических API и игровых движков.

Заключение:Алгоритм Z-буфера является мощным инструментом для решения задачи удаления невидимых поверхностей и остаётся одним из основных методов рендеринга в компьютерной графике благодаря своей простоте и эффективности.

## 48

Алгоритм Робертса — это один из методов обнаружения границ в изображениях, который используется в области компьютерного зрения и обработки изображений. Он основан на вычислении градиента изображения с помощью простых масок (операторов), которые выделяют изменения яркости, что позволяет определить границы объектов.

Принцип работы алгоритма Робертса:

1. Градиент изображения:
   • Алгоритм Робертса использует два оператора, которые называются операторами Робертса. Эти операторы вычисляют градиент изображения в двух направлениях: по диагонали.
2. Операторы:
   • Оператор для вычисления градиента по диагонали (слева сверху направо вниз):

$$G_x = \begin{matrix} 1 & 0 \\ 0 & -1 \end{matrix}$$

• Оператор для вычисления градиента по диагонали (справа сверху налево вниз):

$$G_y = \begin{matrix} 0 & 1 \\ -1 & 0 \end{matrix}$$

3. Применение операторов:
   • Эти операторы применяются к каждому пикселю изображения с помощью свертки. Для каждого пикселя вычисляются значения градиента $G_x$ и $G_y$.
4. Вычисление величины градиента:
   • Величина градиента (интенсивность границы) может быть вычислена как:

$$G = \sqrt{(G_x^2 + G_y^2)}$$

Также можно использовать приближенную формулу:

$$G = |G_x| + |G_y|$$

5. Пороговая обработка:
   • После вычисления величины градиента применяется пороговая обработка, чтобы выделить значимые границы. Пиксели с величиной градиента выше определённого порога считаются границами.

Преимущества алгоритма Робертса:

• Простота: Алгоритм легко реализовать и он требует небольшое количество вычислений.
• Эффективность: Хорошо работает для обнаружения резких изменений яркости, что делает его эффективным для выделения границ объектов.

▎Недостатки алгоритма Робертса:
• Чувствительность к шуму: Алгоритм может быть чувствителен к шуму в изображении, что может привести к ложным срабатываниям.
• Ограниченная направленность: Операторы Робертса ориентированы на диагональные границы и могут не так эффективно обрабатывать горизонтальные и вертикальные границы.
▎Применение:
Алгоритм Робертса часто используется в задачах обработки изображений, таких как выделение объектов, анализ форм и распознавание образов. Он может быть полезен в предварительной обработке изображений перед применением более сложных методов анализа.
▎Заключение:
Алгоритм Робертса является простым и эффективным методом для обнаружения границ в изображениях, который находит применение в различных областях компьютерного зрения и обработки изображений.

## 49

Алгоритм Робертса находит множество практических применений в различных областях, связанных с обработкой изображений и компьютерным зрением. Вот несколько примеров:
▎1. Обнаружение границ объектов в медицинских изображениях
В медицинской визуализации, например, при анализе рентгеновских снимков или МРТ, алгоритм Робертса может использоваться для выделения границ органов или патологий. Это помогает врачам в диагностике, позволяя лучше визуализировать структуры и аномалии.
▎2. Применение в системах видеонаблюдения
Алгоритм может быть использован для обнаружения движущихся объектов в видеопотоке. Выделяя границы объектов, система может отслеживать движение людей или транспортных средств, что полезно для анализа поведения и обеспечения безопасности.
▎3. Обработка фотографий
В графических редакторах алгоритм Робертса может использоваться для создания эффектов резкости или выделения контуров объектов на фотографиях. Это позволяет дизайнерам и фотографам улучшать визуальное восприятие изображений.
▎4. Распознавание образов
В системах распознавания образов алгоритм Робертса может быть использован как один из этапов предварительной обработки изображений. Выделение границ помогает улучшить точность последующих этапов, таких как классификация или идентификация объектов.
▎5. Автономные системы и робототехника
В робототехнике алгоритм может применяться для картографирования окружающей среды. Обнаруживая границы объектов, робот может лучше ориентироваться в пространстве и избегать препятствий.
▎Пример кода на Python с использованием OpenCV
Вот пример кода, который демонстрирует применение алгоритма Робертса для обнаружения границ на изображении с использованием библиотеки OpenCV:

```
import cv2
import numpy as np

# Загрузка изображения
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)

# Применение оператора Робертса
roberts_x = np.array([[1, 0], [0, -1]])
roberts_y = np.array([[0, 1], [-1, 0]])

# Свертка с операторами Робертса
gradient_x = cv2.filter2D(image, -1, roberts_x)
gradient_y = cv2.filter2D(image, -1, roberts_y)

# Вычисление величины градиента
gradient_magnitude = cv2.magnitude(gradient_x, gradient_y)
```

```
# Применение пороговой обработки
_, edges = cv2.threshold(gradient_magnitude, 50, 255, cv2.THRESH_BINARY)

# Отображение результатов
cv2.imshow('Original Image', image)
cv2.imshow('Edges (Roberts)', edges)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

### Заключение

Алгоритм Робертса является мощным инструментом для обнаружения границ и может быть использован в самых различных приложениях, от медицины до робототехники. Его простота и эффективность делают его популярным выбором среди специалистов по обработке изображений.

**50**

Модели отражения света играют важную роль в компьютерной графике, физике и инженерии, так как они помогают понять, как свет взаимодействует с поверхностями различных материалов. Вот несколько основных моделей отражения света:

### 1. Модель идеального отражения (зеркальное отражение)

• Описание: В этой модели предполагается, что свет полностью отражается от гладкой поверхности, как от зеркала. Угол падения равен углу отражения.
• Применение: Используется для описания металлических поверхностей или стекол.

### 2. Модель диффузного отражения

• Описание: Эта модель предполагает, что свет рассеивается во всех направлениях после попадания на шероховатую поверхность. Это создает мягкое, рассеянное освещение.
• Применение: Применяется для описания матовых и шероховатых материалов, таких как бумага или некрашеная древесина.

### 3. Модель смешанного отражения (Phong)

• Описание: Модель Phong сочетает в себе как диффузное, так и зеркальное отражение. Она использует три компонента:
  • Ambient (окружающее): Общее освещение окружающей среды.
  • Diffuse (диффузное): Отражение света от поверхности.
  • Specular (зеркальное): Отражение света от гладких участков.
• Формула:

$$I = I_{amβient} + I\_(diffuse) + I\_(specular)$$

• Применение: Широко используется в рендеринге 3D-графики.

### 4. Модель Блинна-Фонга (Blinn-Phong)

• Описание: Это модификация модели Phong, которая использует половинный вектор между направлением к источнику света и нормалью поверхности для вычисления зеркального отражения. Это позволяет получить более реалистичные результаты с меньшими вычислительными затратами.
• Применение: Часто используется в реальном времени для игр и интерактивной графики.

### 5. Модель Кука-Торренса

• Описание: Эта модель более сложная и учитывает микроструктуру поверхности. Она основана на теории микроскопического отражения и учитывает такие факторы, как шероховатость поверхности и угол падения света.
• Применение: Применяется для фотореалистичного рендеринга, особенно в современных рендерерах и игровых движках.

### 6. Модель Френеля

• Описание: Модель Френеля описывает, как свет отражается и преломляется на границе двух сред с различными показателями преломления. Угол отражения зависит от угла падения и может быть рассчитан с использованием уравнений Френеля.
• Применение: Важно для рендеринга стеклянных и водных поверхностей.

### Заключение

Выбор модели отражения света зависит от конкретных требований задачи и желаемого уровня реализма. Современные графические движки часто комбинируют несколько моделей для достижения оптимального баланса между производительностью и качеством изображения.

## 51

### Зеркальное отражение

Описание:
Зеркальное отражение — это процесс, при котором световые лучи, падая на гладкую поверхность, отражаются от неё под тем же углом, под которым они на неё попали. Это явление можно объяснить законом отражения, который гласит: угол падения равен углу отражения.
Основные характеристики:
• Угол падения и угол отражения: Если угол падения (угол между падающим лучом и нормалью к поверхности) составляет θ, то угол отражения также будет равен θ.
• Гладкость поверхности: Зеркальное отражение наблюдается на гладких поверхностях, таких как стекло или металл. Чем более гладкая поверхность, тем более четким будет отражение.
• Отображение: Зеркальное отражение создает четкое изображение объекта, находящегося перед зеркалом.
Формула:
Для описания зеркального отражения можно использовать простую формулу для расчета углов:

$$\boldsymbol{\theta}\_(\text{отраж}) = \boldsymbol{\theta}\_(\text{пад})$$

где:
• θ_(отраж) — угол отражения,
• θ_(пад) — угол падения.

Применение:
• Оптика: Зеркальное отражение используется в оптических приборах, таких как зеркала, телескопы и микроскопы.
• Компьютерная графика: В 3D-моделировании и рендеринге зеркальное отражение помогает создать реалистичные изображения объектов, таких как вода или стекло.
• Архитектура и дизайн: Зеркала используются для визуального увеличения пространства и создания интересных эффектов в интерьере.
Примеры:
1. Зеркала: Обычные зеркала в ванных комнатах или на туалетных столиках.
2. Стеклянные поверхности: Отражения зданий в стеклянных фасадах.
3. Вода: Спокойная водная поверхность может отражать окружающие объекты, создавая эффект зеркала.

### Заключение

Зеркальное отражение — это ключевое явление в физике и оптике, которое имеет множество практических применений. Оно важно не только для понимания основ светопередачи, но и для создания визуально привлекательных изображений в различных областях.

## 52

### Диффузное отражение

Описание:Диффузное отражение — это процесс, при котором световые лучи, падая на шероховатую или неровную поверхность, рассеиваются в разных направлениях. В отличие от зеркального отражения, при диффузном отражении не образуется четкого изображения, а свет рассеивается равномерно.
Основные характеристики:
• Шероховатость поверхности: Диффузное отражение происходит на неровных и шероховатых поверхностях, таких как бумага, бетон или матовая краска.
• Равномерное распределение света: Свет, отражаясь от такой поверхности, рассеивается в различных направлениях, что делает её менее яркой и более мягкой по сравнению с зеркальными отражениями.
• Отсутствие четкого изображения: Из-за рассеяния света не формируется четкое изображение объекта, находящегося перед такой поверхностью.
Формула:

Для диффузного отражения можно использовать закон Блюра, который описывает интенсивность света, отражающегося от поверхности, но в общем виде формулы нет, так как отражение зависит от многих факторов (угол падения, свойства материала и т.д.).
Применение:
• Освещение: Диффузные отражатели используются для создания мягкого освещения в интерьере, чтобы избежать резких теней.
• Фотография: В фотографии диффузоры помогают смягчить свет и улучшить качество изображений.
• Декор: Матовые поверхности и текстуры используются в дизайне для создания визуального интереса и уменьшения бликов.
Примеры:
1. Матовая краска: Стены, окрашенные матовой краской, рассеивают свет и не создают бликов.
2. Шершавая бумага: Бумага для заметок или упаковки рассеивает свет и выглядит менее блестящей.
3. Текстиль: Ткани с шероховатой текстурой, такие как бархат или фланель, также демонстрируют диффузное отражение.

### Заключение

Диффузное отражение — важный аспект взаимодействия света с поверхностями, который находит применение в различных областях, включая архитектуру, дизайн и искусство. Оно помогает создавать мягкое и равномерное освещение, а также эстетически привлекательные текстуры.

## 53

### Смешанное отражение

Описание:Смешанное отражение — это процесс, при котором свет падает на поверхность и отражается как в виде зеркального (специфического) отражения, так и в виде диффузного (рассеянного) отражения. Это происходит на поверхностях, которые имеют как гладкие, так и шероховатые участки или имеют текстуру, способствующую различным типам отражения.
Основные характеристики:
• Комбинация эффектов: На поверхности могут быть участки, которые отражают свет зеркально (например, гладкие участки), и участки, которые рассеивают свет (шероховатые).
• Зависимость от угла падения: Угол, под которым свет падает на поверхность, может влиять на то, какое количество света будет отражено в виде зеркального или диффузного отражения.
• Разнообразие текстур: Разные материалы и текстуры могут создавать различные эффекты смешанного отражения.
Применение:
• Архитектура и дизайн интерьеров: Смешанное отражение используется для создания интересных визуальных эффектов в интерьере. Например, использование стеклянных и матовых поверхностей в одном пространстве.
• Фотография: В фотографии смешанное отражение может создать уникальные визуальные эффекты, добавляя глубину и текстуру изображению.
• Искусство: Художники могут использовать смешанное отражение для создания объемных и многослойных эффектов на своих работах.
Примеры:
1. Гладкая плитка с текстурой: Плитка может иметь гладкую поверхность с рельефными узорами, что позволяет одновременно наблюдать как зеркальное, так и диффузное отражение.
2. Металлические поверхности: Полированные металлические поверхности могут отражать свет зеркально, но при этом иметь матовые участки, которые рассеивают свет.
3. Стеклянные предметы: Стеклянные вазы или посуды могут создавать смешанное отражение, где часть света отражается четко, а часть рассеивается из-за текстуры или загрязнений.

### Заключение

Смешанное отражение — это сложный и интересный процесс взаимодействия света с поверхностями, который позволяет создавать разнообразные визуальные эффекты и текстуры. Оно находит широкое применение в различных областях, включая архитектуру, дизайн, фотографию и искусство.

## 54) Вычисление нормалей

**Нормаль** — это вектор, перпендикулярный поверхности или грани, используемый для расчёта освещения и отражения.

**Шаги вычисления:**

1. **Для треугольника (плоской грани):**
   - Используйте два ребра грани $\vec{u}$ и $\vec{v}$.
   - Вычислите нормаль с помощью векторного произведения:
     $$\vec{n} = \vec{u} \times \vec{v}$$
   - Нормализуйте вектор:
     $$\vec{n} = \frac{\vec{n}}{|\vec{n}|}$$

1. **Для сложной поверхности**:
   - Рассчитайте нормали для каждой грани.
   - Для вершины нормаль — усреднение нормалей всех граней, связанных с вершиной.
2. **Использование нормалей**:
   - Для расчёта освещения (модель освещения, например, Фонга).
   - Для текстурных эффектов (нормал-маппинг).

**Итог**: Нормали обеспечивают корректное отображение света, теней и отражений, что делает их ключевыми в рендеринге 3D-объектов.

## 55) Закрашивание методом Гуро

**Метод Гуро** — это техника закрашивания в компьютерной графике, используемая для плавного отображения света и теней на 3D-объектах.

### Алгоритм:

1. **Расчёт нормалей**:
   - Нормаль вычисляется для каждой вершины многоугольника как усреднение нормалей граней, к которым она принадлежит.
2. **Интерполяция интенсивности**:
   - Интенсивность освещения рассчитывается в каждой вершине на основе нормали и источника света (модель освещения Фонга).
   - Значения интенсивности линейно интерполируются вдоль рёбер и внутри грани.
3. **Закрашивание**:
   - Цвет каждой точки грани вычисляется по интерполированным значениям интенсивности.

### Особенности:

- **Преимущества**: Плавный переход между освещёнными участками, высокая скорость выполнения.
- **Недостатки**: Ограничена видимостью изменений внутри грани, что может приводить к нереалистичным отражениям.

**Итог**: Метод Гуро используется для создания более реалистичного освещения на 3D-объектах по сравнению с плоским закрашиванием, но менее точен, чем метод Фонга (см. 56 пункт).

## 56) Закрашивание методом Фонга

**Метод Фонга** — это техника закрашивания в компьютерной графике, обеспечивающая более плавное и точное освещение на 3D-объектах по сравнению с методом Гуро.

### Алгоритм:

1. **Расчёт нормалей**:
   - Нормаль вычисляется для каждой вершины как усреднение нормалей граней, к которым она принадлежит.
2. **Интерполяция нормалей**:

- Нормали линейно интерполируются вдоль рёбер и внутри полигона, чтобы определить нормаль для каждой точки грани.

3. **Расчёт освещения**:
   - Освещённость каждой точки вычисляется с использованием интерполированной нормали и модели освещения (например, Фонга).

## Особенности:

- **Преимущества**: Более реалистичное отображение бликов и теней, особенно на больших или угловатых полигонах.
- **Недостатки**: Требует больше вычислений, чем метод Гуро.

**Итог**: Метод Фонга используется для создания реалистичного освещения и отражений в 3D-графике, особенно в приложениях, где важна высокая точность рендеринга.

## 57) Модели преломления света

**Преломление света** — это изменение направления светового луча при переходе между средами с разной плотностью. В компьютерной графике преломление моделируется для создания реалистичных эффектов, таких как стекло, вода или кристаллы.

## Основные модели:

1. **Закон Снеллиуса**:
   - Описывает угол преломления:

$$n_1 \sin\theta_1 = n_2 \sin\theta_2$$

- $n_1, n_2$: показатели преломления двух сред.
- Используется для расчёта направления луча при переходе между средами.

2. **Модель Френеля**:
   - Определяет доли отражённого и преломлённого света в зависимости от угла падения и показателей преломления.
   - Применяется для реалистичного отображения отражений на границах материалов.
3. **Трассировка лучей (Ray Tracing)**:
   - Преломление моделируется с помощью прослеживания пути луча через несколько сред.
   - Учитывает закон Снеллиуса и потери света.
4. **Метод объёмного рендеринга**:
   - Используется для симуляции преломления в неоднородных средах (например, дым, газ).

## Применение:

Модели преломления используются в визуализации стекла, воды, прозрачных объектов и оптических эффектов в 3D-графике.

**Итог**: Преломление рассчитывается с учётом законов физики, таких как закон Снеллиуса и модель Френеля, для создания реалистичных изображений в компьютерной графике.

## 58) Вычисление вектора преломленного луча

Преломленный луч вычисляется на основе **закона Снеллиуса** и направлений векторов, описывающих падающий луч и нормаль к поверхности.

## Алгоритм:

1. **Дано**:

- $\vec{I}$: единичный вектор падающего луча.
- $\vec{N}$: нормаль к поверхности (единичный вектор).
- $n_1, n_2$: показатели преломления первой и второй сред.
- $\eta = \frac{n_1}{n_2}$: относительный показатель преломления.

2. **Вычисление угла преломления:**
   - Закон Снеллиуса:
$$\eta \cdot \sin\theta_t = \sin\theta_i$$
3. **Формула для преломленного вектора:**
$$\vec{T} = \eta \cdot \vec{I} - \left(\eta \cdot (\vec{I} \cdot \vec{N}) + \sqrt{1 - \eta^2 \cdot (1 - (\vec{I} \cdot \vec{N})^2)}\right) \cdot \vec{N}$$
   Где:
   - $\vec{T}$: преломлённый вектор.
   - $\vec{I} \cdot \vec{N}$: скалярное произведение падающего вектора и нормали.
4. **Полное внутреннее отражение:**
   - Если дискриминант $(1 - \eta^2 \cdot (1 - (\vec{I} \cdot \vec{N})^2))$ отрицателен, преломление отсутствует, и происходит только отражение.

## Итог:
Вектор преломленного луча рассчитывается с учётом закона Снеллиуса и геометрии поверхности. Он используется в методах трассировки лучей для реалистичного отображения преломления света.

## 59) Трассировка лучей в компьютерной графике
**Трассировка лучей (Ray Tracing)** — это метод рендеринга, моделирующий поведение света для создания реалистичных изображений.

## Основная идея:
- Лучи от камеры (наблюдателя) прослеживаются до объектов сцены, учитывая отражение, преломление, тени и взаимодействие с источниками света.

## Шаги алгоритма:
1. **Инициализация луча**:
   - Луч запускается от камеры через пиксель изображения.
2. **Пересечение объектов**:
   - Проверяется пересечение луча с объектами сцены.
3. **Расчёт освещения**:
   - Освещённость точки определяется с учётом:
     - Прямого света.
     - Отражений (вторичные лучи).
     - Преломления (на прозрачных материалах).
     - Теней (теневые лучи).
4. **Комбинирование эффектов**:
   - С использованием моделей освещения (например, Фонга) рассчитываются цвета и яркость.
5. **Рекурсия**:
   - Для отражённых и преломленных лучей процесс повторяется, пока не достигнут предел глубины трассировки.

## Преимущества:
- Реалистичное освещение, отражения, тени, преломления.

- Подходит для физически точного рендеринга.

**Недостатки:**

- Высокая вычислительная сложность.
- Требует оптимизаций для интерактивной графики (например, в играх).

**Итог**: Трассировка лучей используется для создания реалистичных изображений в 3D-графике, включая анимацию, кино и игры, благодаря способности точно моделировать взаимодействие света.