Машинная графика
Computer Graphics

Лекция 11.

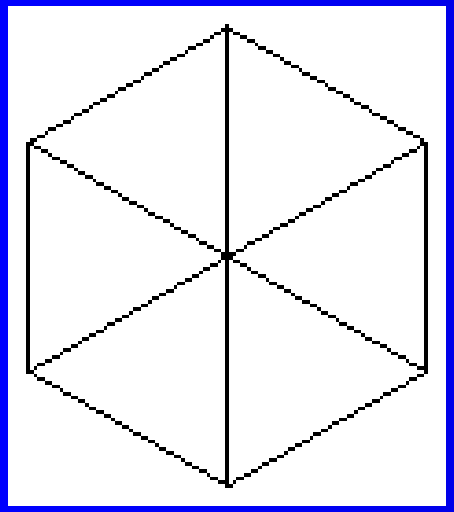
«Удаление невидимых линий и поверхностей»

План лекции

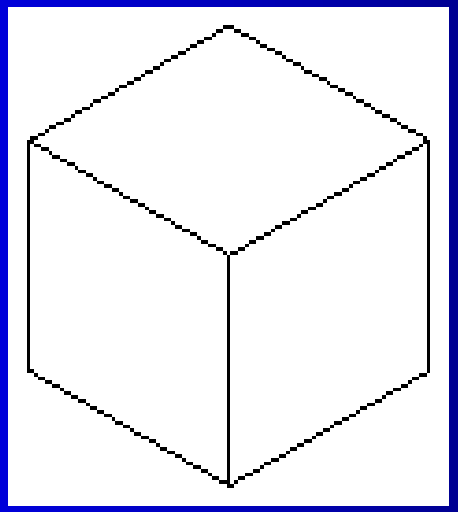
- Постановка задачи. Классификация подходов
- Алгоритм Робертса
- Алгоритм художника
- Алгоритм плавающего горизонта
- Алгоритм Z-буфера
- Алгоритм A-буфера

Постановка задачи

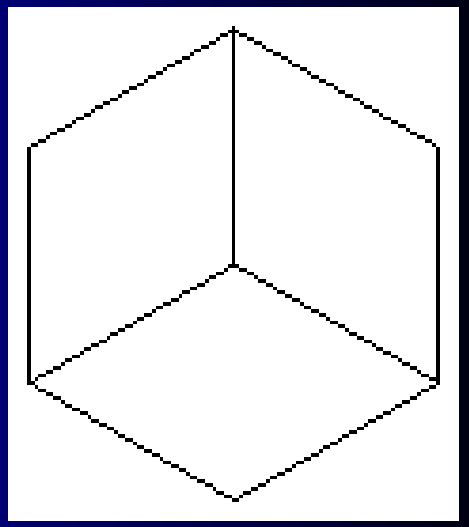
Если преобразовывать 3D объект в 2D плоскость, посредством проецирования (любым способом), возникает проблема неоднозначности получаемого результата. Например:



Изометрия



Вид сверху

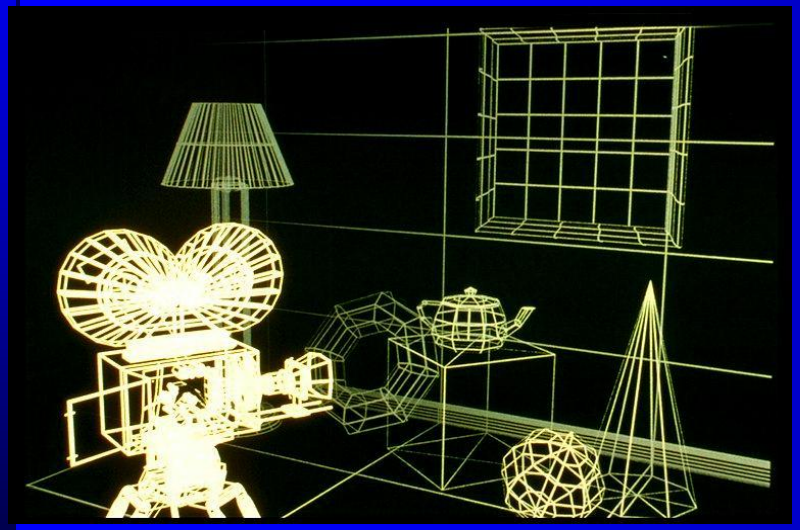


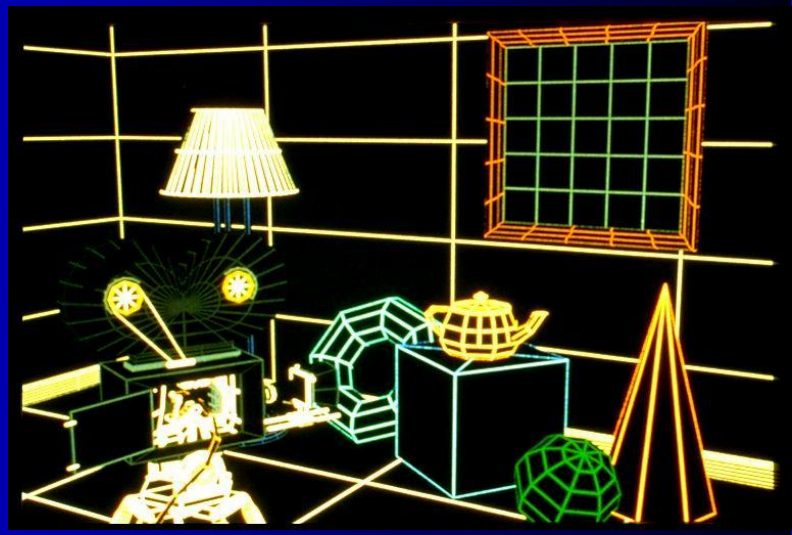
Вид снизу

Постановка задачи (the visibility problem)

В общем случае две задачи (отсечение не в счёт):

- Удалить все не лицевые поверхности.
- Удалить все видимые поверхности, закрываемые другими объектами.





Постановка задачи и классификация алгоритмов

Соответственно, следует прежде всего определить – какие рёбра и грани видимы у конкретного объекта.

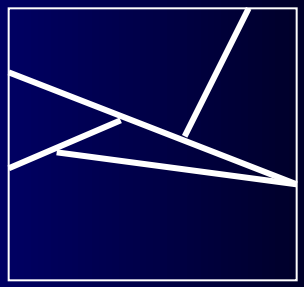
Если объектов на сцене несколько, то требуется так же определить какие части объектов закрываются другими объектами и соответственно невидимы наблюдателю.

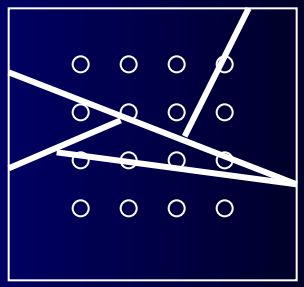
Оптимального решения в общем виде для данной задачи не существует.

Большинство алгоритмов удаления невидимых линий и поверхностей используют алгоритмы сортировки и когерентности.

Сортировка облегчает сравнение по глубине. Когерентность учитывает регулярность структур сцены (как объектов, так и 2D представления). Кроме того, между объектами на сцене часто можно установить постоянные связи.

Классификация алгоритмов, Sutherland, Sproull, Schumacher (1974)





Алгоритмы различаются:

по анализу пространства видимости:

-в пространстве изображения (2D)

-в пространстве объекта (3D)

Основные преимущества и недостатки:

3D алгоритмы – векторные, точность больше.

2D алгоритмы – растровые, проще, но может случаться alliasing (отрицательный эффект дискретизации из-за невыполнения условий теоремы Шеннона) и при изменении шага дискретизации – приходится пересчитывать всю сцену заново.

Классификация алгоритмов

Основная идея: 3D

Оперируем геом. примитивами, проверяем пересечение объектов друг с другом, в результате – список видимых объектов и их частей. Объекты из списка могут отображаться с любой точностью, но вычисления видимых частей – с высокой.

Сложность алгоритмов зависит от количества объектов: в среднем для n объектов –> $O(n^2)$

Основная идея: 2D

Находим ближ. точки сцены к наблюдателю и для каждого пикселя изображения отображаем только такие точки. Точность на уровне разрешения устройства. Не требуется высокой точности вычислений, но $\exists$ зависимость сложности алгоритмов от числа пикселей и количества объектов: для

c пикселей и n объектов –> $O(c \times n)$.

Классификация алгоритмов

Особенности: 3D

Объект анализируется 1 раз. Пиксель сцены может перерисовываться несколько раз.

Алгоритмы:

Робертса (1963)

Вейлера-Айзертона (1977)

Хедали (1982)

BSP-деревьев (1969-91)

Октодеревьев (1982)

Особенности: 2D

Пиксель сцены рисуется 1 раз, отношения между объектами для данного пикселя анализируются несколько раз.

Алгоритмы:

Z-буфера (1974), A-буфера (1984)

Плавающего горизонта (1972)

«Художника» (????)

Варнона (1968)

Построчного сканирования (1967)

Трассировки лучей (1968)

Айзертона (1983)

Плоскость в 3D пространстве

Уравнение произвольной плоскости в трехмерном пространстве:

$$Ax + By + Cz + D = 0$$

Матричная форма плоскости:

$$[x\ y\ z\ 1][P]^T = 0,$$

где $[P]^T = [a\ b\ c\ d]$ – плоскость.

Точка (x,y,z) расположена за плоскостью, если:

$$Ax + By + Cz + D < 0$$

Модель объекта-многогранника 3D

- состоит из N граней
- описывается матрицей T размера 4xN, каждый n-й столбец которой содержит описание n-й (n=[1,N]) плоскости AnX+BnY+CnZ+Dn=0 в объектной системе координат XYZ

$$T = \begin{bmatrix} A1 & \dots & An & \dots & AN \\ B1 & \dots & Bn & \dots & BN \\ C1 & \dots & Cn & \dots & CN \\ D1 & \dots & Dn & \dots & DN \end{bmatrix}$$

минимальное число граней K=4 наблюдается у тетраэдра

Определение факта невидимости плоскости объекта

Не все грани, входящие в состав объекта, будут видны наблюдателю. Невидимой будет та, нормаль **N** к которой ориентирована в ту же сторону (с некоторым допуском), что и вектор **R**, направленный от наблюдателя к объекту.

Иными словами, для невидимых граней скалярное произведение данных векторов положительно: **NR**> 0.

Более того, если модель объекта преобразована к системе координат проекции, а направление наблюдения параллельно оси OZ, то требуется рассмотреть только Z-координату вектора нормали **N**.

В правосторонней системе координат (R совпадает с отрицательным направлением OZ), грань невидима, если $C<0$, где C - координата Z вектора нормали **N** проверяемой плоскости.

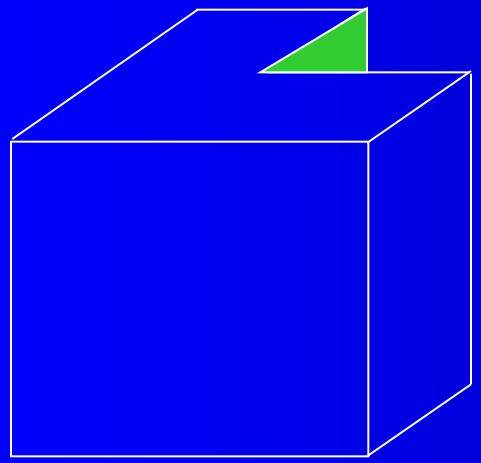
Если $C=0$, то грань так же не видна – расположена параллельно взгляду, т.е. для невидимых граней: $C\leq 0$

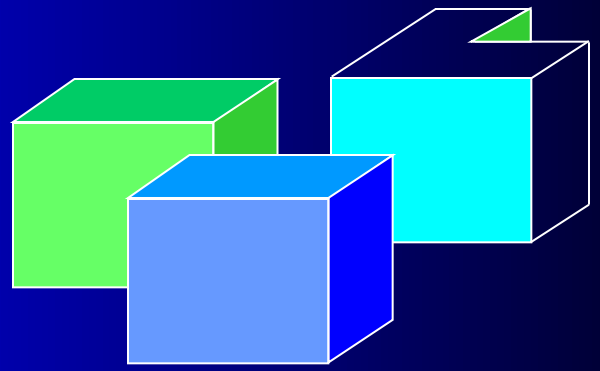
Визуализация плоскостей

Для выпуклых многоугольников алгоритм ясен – видимые плоскости сортируются по глубине и отображаются в последовательности «от дальнего к ближнему».

С невыпуклыми многоугольниками – ситуация сложнее.

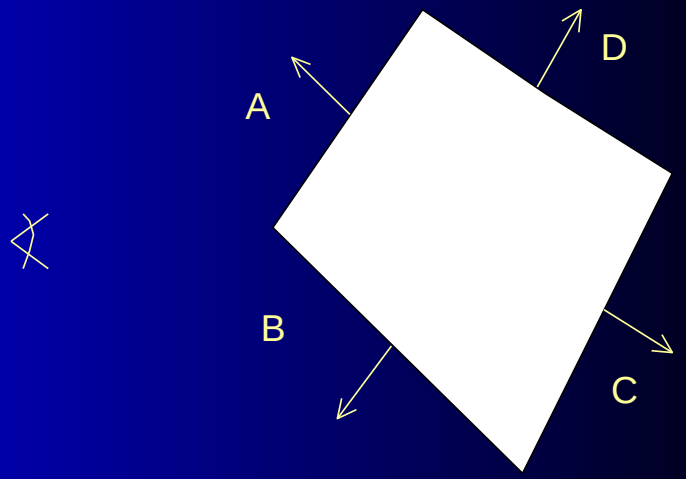
При визуализации сцен, состоящих из нескольких объектов – сложность повышается ещё больше.





Лицевые и нелицевые грани

Если грани являются границей тела (или нескольких тел), то для каждой из них можно определить вектор внешней нормали.



Нормали к граням A и B смотрят в сторону наблюдателя (наблюдатель находится в положительном полупространстве по отношению к плоскости, проходящей через соответствующую грань). Такие грани называются лицевыми (front-faced).

Для граней C и D нормали направлены от наблюдателя, их называют нелицевыми (back-faced).

Свойства нелицевых граней

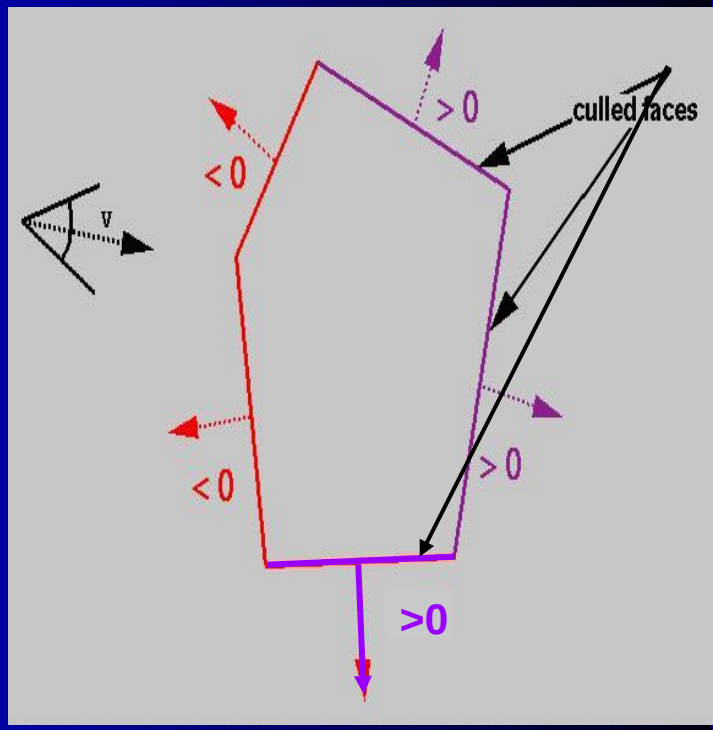
В случае, когда грани являются границей тела (или нескольких тел), то ни одна из нелицевых граней не может быть видна даже частично – любая из них всегда будет закрываться от наблюдателя лицевыми гранями.

При определении видимости все нелицевые грани можно всегда отбрасывать, что сокращает число рассматриваемых граней примерно вдвое (в общем случае количество лицевых граней примерно равно количеству лицевых, т.е. составляет половину от общего числа граней).

Когда вся сцена состоит из одного выпуклого объекта, то все лицевые грани и только они будут видны, причем полностью.

Алгоритм Робертса

Алгоритм может быть с успехом применен для изображения множества выпуклых многогранников на одной сцене в виде проволочной модели с удаленными невидимыми линиями, алгоритм непригоден непосредственно для передачи падающих теней и других сложных визуальных эффектов.



Требования к сцене:

В алгоритме Робертса требуется, чтобы все изображаемые тела или объекты были выпуклыми.

Невыпуклые тела должны быть разбиты на выпуклые части.

В алгоритме выпуклое многогранное тело с плоскими гранями должно представляться набором пересекающихся плоскостей.

Алгоритм Робертса

Алгоритм выполняется в 3 этапа:

На 1-м этапе объекты анализируются индивидуально с целью удаления нелицевых плоскостей.

На 2-м этапе проверяется экранирование оставшихся в каждом теле рёбер всеми другими телами с целью обнаружения их невидимых отрезков.

На 3-м этапе вычисляются отрезки, которые образуют новые рёбра при протыкании объектов друг друга.

В алгоритме предполагается, что объекты состоят из плоских полигональных граней, которые в свою очередь ограничены рёбрами, а рёбра – отдельными вершинами. Все вершины, рёбра и грани связаны с конкретным объектом.

Алгоритм Робертса

Первый этап (удаление невидимых граней):

Для каждого объекта сцены:

- Сформировать многоугольники граней, рёбра на основе списка вершин объекта.
- Вычислить уравнение плоскости для каждой грани объекта
- Получить матрицу граней объекта
- Произвести её проективное преобразование (например ЦП)
- Вычислить и запомнить габариты описывающего проекцию объекта прямоугольника: X_{max},X_{min},Y_{max},Y_{min}
- Определить нелицевые плоскости:
  - Вычислить скалярное произведение вектора R, на грань из матрицы объекта.
  - Если скалярное произведение больше нуля – грань невидима.
  - Удалить весь описывающий многоугольник, лежащий в данной плоскости- это избавляет от необходимости отдельно рассматривать невидимые линии, образуемые пересечением пар невидимых плоскостей.

Если в сцене лишь 1 объект – алгоритм завершён, иначе – 2-й этап.

Алгоритм Робертса

Второй этап (проверка экранирования объектов друг другом):

Сформировать список «приоритета» объектов, упорядоченных по Z координате. Наиболее удалённый объект будет стоять первым в списке (в правой системе координат – он будет иметь максимальную координату Z).

Для каждого тела из списка:

- Проверить экранирование всех лицевых рёбер всеми другими телами сцены, стоящими ниже в списке.
- Проверка производится сперва по описывающим проекции граней прямоугольникам.
- Если не прямоугольники не пересекаются (в плоскости XY), то переход к следующему объекту.
- Иначе проверка на протыкание.

Алгоритм Робертса

Проверка на протыкание:

Сравнить макс. значение Z проверяемого объекта с минимальным значением Z «экранируемого» объекта.

Если $Z_{max} < Z_{min}$, то протыкание невозможно. Перейти к следующему экранируемому объекту. В противном случае - проверить видимое протыкание.

На протыкание проверяется передняя грань, боковые и верхние грани.

Если условия протыкания выполнены – установить флаг протыкания (для активизации третьего шага) и занести оба объекта в список протыканий.

2-й этап (продолжение):

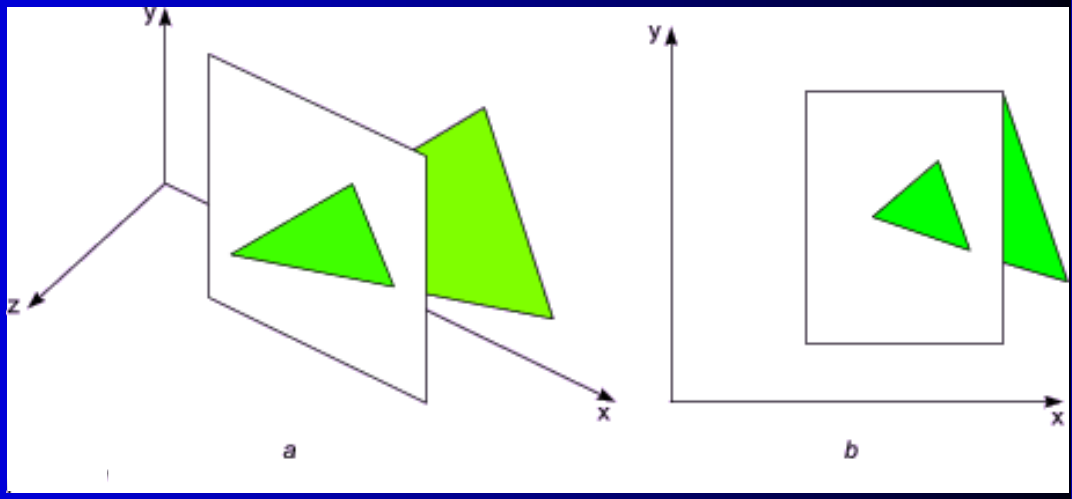
Проверяется возможность частичного экранирования.

Если ребро полностью видимо-> далее. Иначе вычисляются видимые участки рёбер (сохраняются отдельно для их проверки с объектами более низкого приоритета).

Алгоритм Робертса

Возможны случаи:

- грань не закрывает ребро;
- грань полностью закрывает ребро;
- грань частично закрывает ребро (в этом случае ребро разбивается на несколько частей, из которых видимыми являются не более двух).



3-й этап (протыкание):

Сформировать все возможные рёбра, соединяющие точки протыкания, для пар объектов, связанных отношением протыкания.

Проверить экранирование всех соединяющих рёбер гранями обоих объектов, связанных отношением протыкания.

Проверить экранирование оставшихся соединяющих рёбер всеми прочими объектами сцены. Запомнить видимые отрезки.

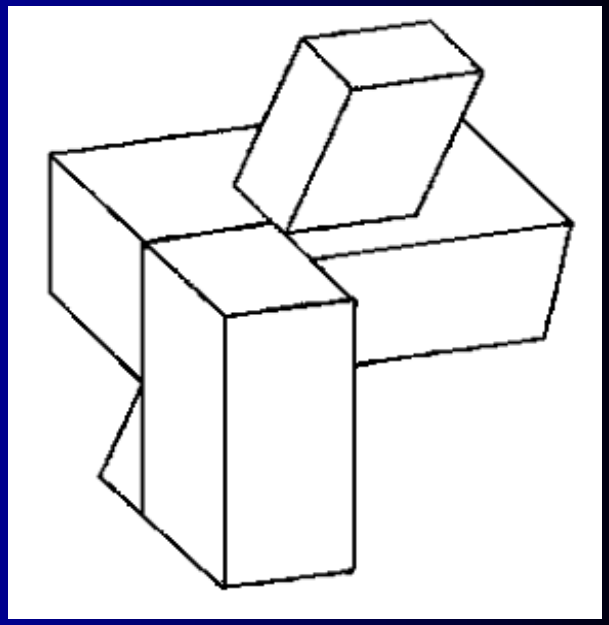
Визуализировать все видимые грани и отрезки рёбер.

Алгоритм Робертса

Недостатки алгоритма Робертса :

- неспособность без привлечения других подходов реализовать падающие тени
- невозможность передачи зеркальных эффектов и преломления света
- строгая ориентация метода только на выпуклые многогранники

Преимущество – одно:
относительная простота.

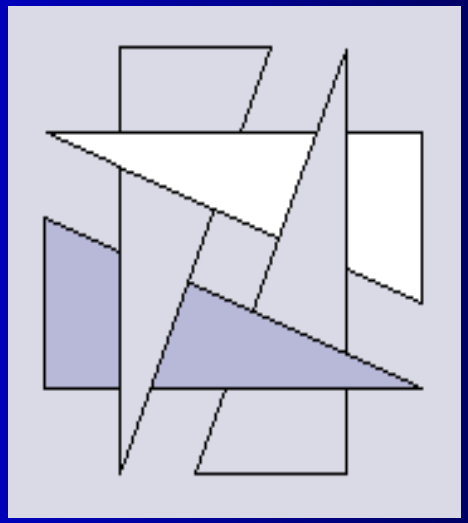


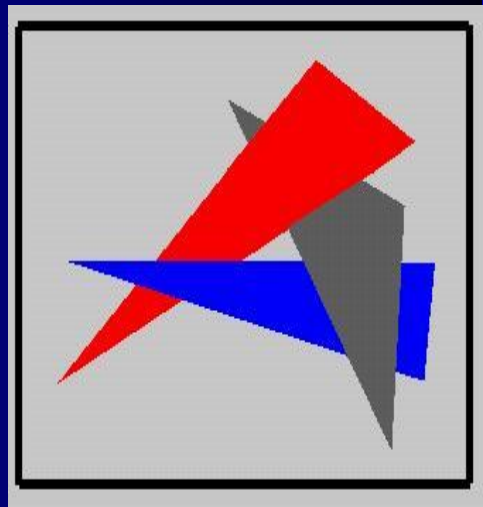
Алгоритм художника

- В отличии от алгоритма z-буфера, который может обрабатывать многоугольники, поступающие в произвольном порядке, в алгоритме "художника" многоугольники сначала упорядочиваются в направлении от заднего плана к переднему. В случае когда пары многоугольников не удается достаточно просто упорядочить, они подразделяются на части до тех пор, пока получившиеся части не позволят это сделать. Это, наверно, самый простой способ прорисовать перекрывающиеся грани.
- Пусть имеется некий набор граней (т.е. сцена), который требуется нарисовать. Отсортируем грани по удаленности от камеры и отрисуем все грани, начиная с самых удаленных. Довольно распространенная характеристика удаленности для грани ABC - это среднее значение z, mid_z = (A.z+B.z+C.z)/3. Вот и весь алгоритм. Просто, и обычно достаточно быстро.
- Существует, правда, несколько проблем.

Алгоритм художника

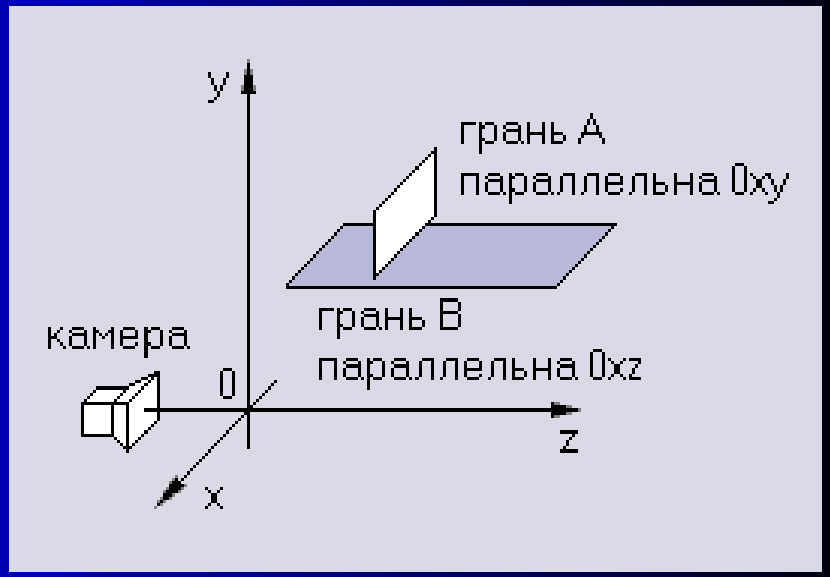
- Во-первых, при некотором расположении граней этот алгоритм вообще не может дать правильного результата - в каком порядке грани не рисуй, получится неправильно. Вот стандартные примеры.





Алгоритм художника

Во-вторых, при некотором расположении граней и использовании среднего значения z как характеристики удаленности алгоритм тоже дает неправильный результат. Пример на рисунке. В этом случае горизонтальную грань надо отрисовывать второй, но по среднему значению z она лежит дальше и таким образом получается, что ее надо отрисовывать первой.



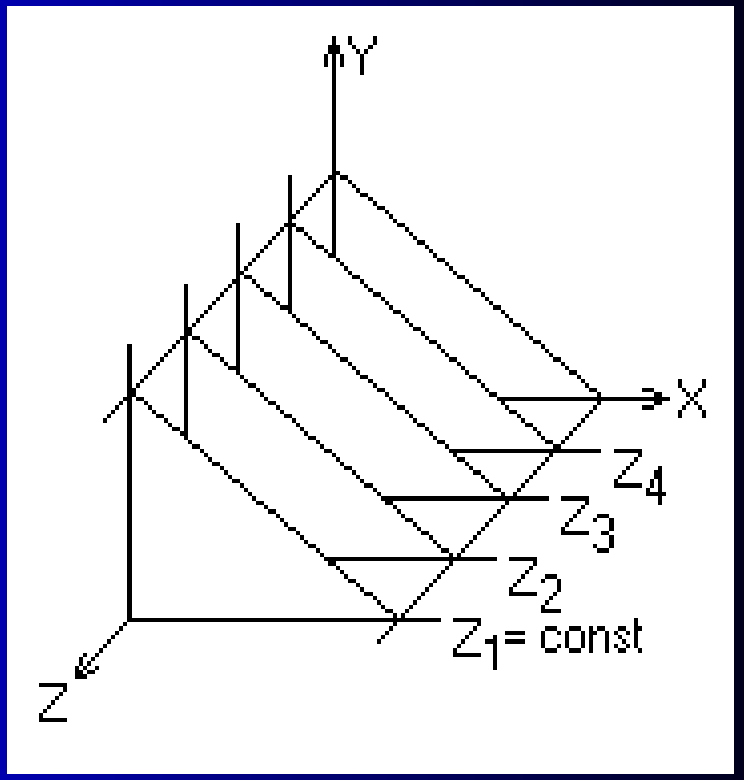
Возможные пути решения этой проблемы - какие-то изменения характеристики удаленности, либо моделирование, не вызывающее таких ситуаций.

Алгоритм художника

- И наконец, при использовании этого алгоритма отрисовываются вообще все грани сцены, и при большом количестве загораживающих друг друга граней мы будем тратить большую часть времени на отрисовку невидимых в конечном итоге частей. То есть совершенно впустую. В таком случае целесообразно использовать какие-то другие методы (например BSP-деревья и PVS, порталы, и т.д).

Алгоритм плавающего горизонта

- Главная идея данного алгоритма заключается в сведении трехмерной задачи к двумерной путем пересечения исходной поверхности последовательностью параллельных секущих плоскостей, имеющих постоянные значения координат x, y или z.

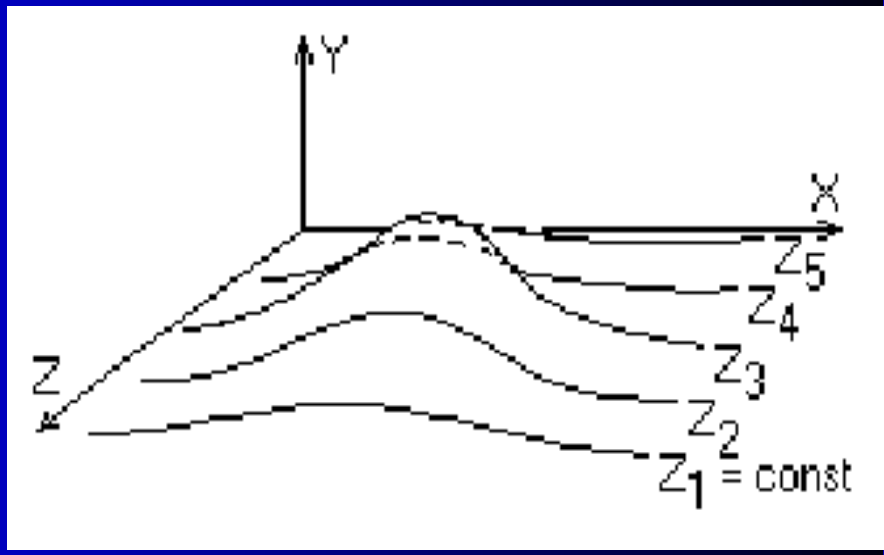


В частности, обычно, параллельные плоскости определяются постоянными значениями z.

Алгоритм плавающего горизонта

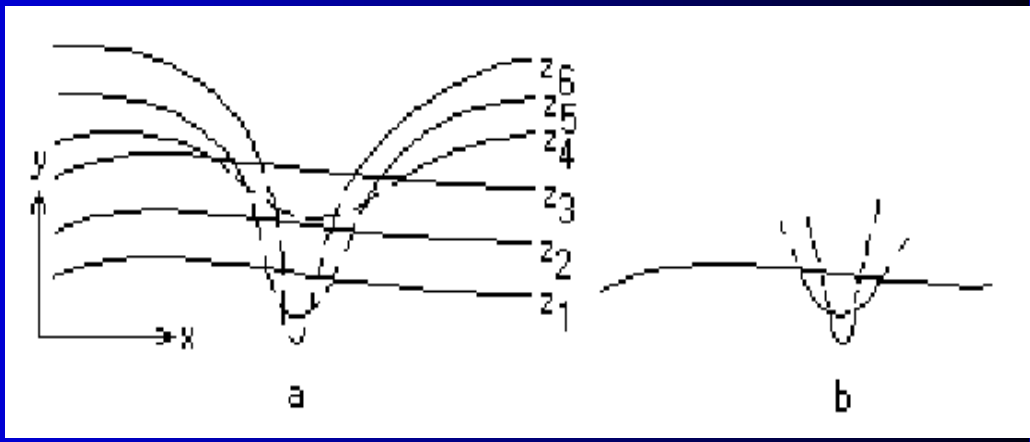
- Функция F (x, y, z) = 0 сводится к последовательности кривых, лежащих в каждой из этих параллельных плоскостей, например к последовательности
- y = f (x, z) или
- x = (y, z)

  где *z* постоянна на каждой из заданных параллельных плоскостей.



Алгоритм плавающего горизонта

Для хранения максимальных значений y для каждого значения x используется массив, длина которого равна числу различимых точек (пикселей) по оси x.



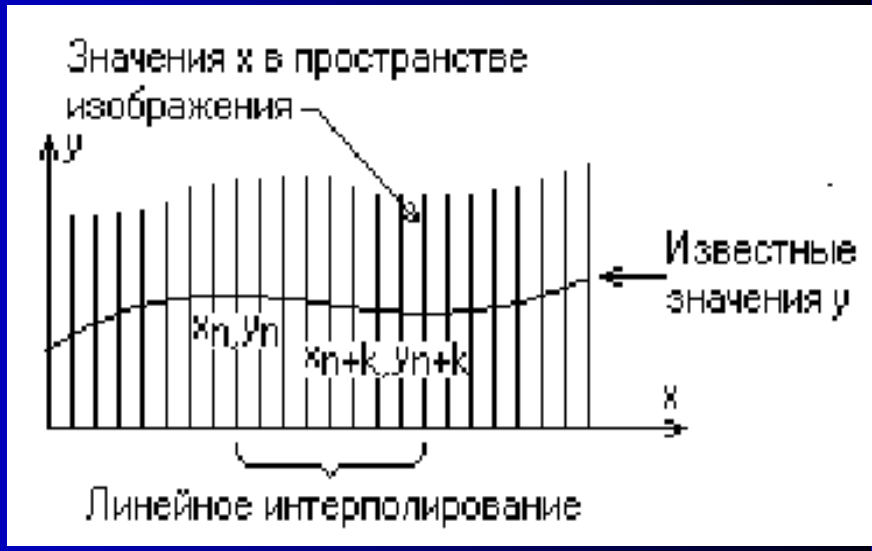
Значения, хранящиеся в этом массиве, представляют собой текущие значения «горизонта». Поэтому по мере рисования каждой очередной кривой этот горизонт «всплывает». Фактически этот алгоритм удаления невидимых линий работает каждый раз с одной линией.

Алгоритм работает очень хорошо до тех пор, пока какая-нибудь очередная кривая не окажется ниже самой первой из кривых.

Для этого случая предусмотрен нижний «горизонт».

Алгоритм плавающего горизонта

В приведённом варианте алгоритме предполагается, что значение функции, т. е. y, известно для каждого значения x в пространстве изображения. Однако если для каждого значения x нельзя указать (вычислить) соответствующее ему значение y, то невозможно



поддерживать массивы верхнего и нижнего плавающих горизонтов.

В таком случае используется линейная интерполяция значений y между известными значениями для того, чтобы заполнить массивы верхнего и нижнего плавающих горизонтов.

Алгоритм плавающего горизонта

Если видимость кривой меняется, то метод с такой простой интерполяцией не даст корректного результата. Предполагая, что операция по заполнению массивов проводится после проверки видимости, получаем, что при переходе текущей кривой от видимого к невидимому состоянию (сегмент AB на рис.), точка (xn+k, yn+k) объявляется невидимой. Тогда участок кривой между точками (xn, yn) и (xn+k, yn+k) не изображается и операция по заполнению массивов не производится.

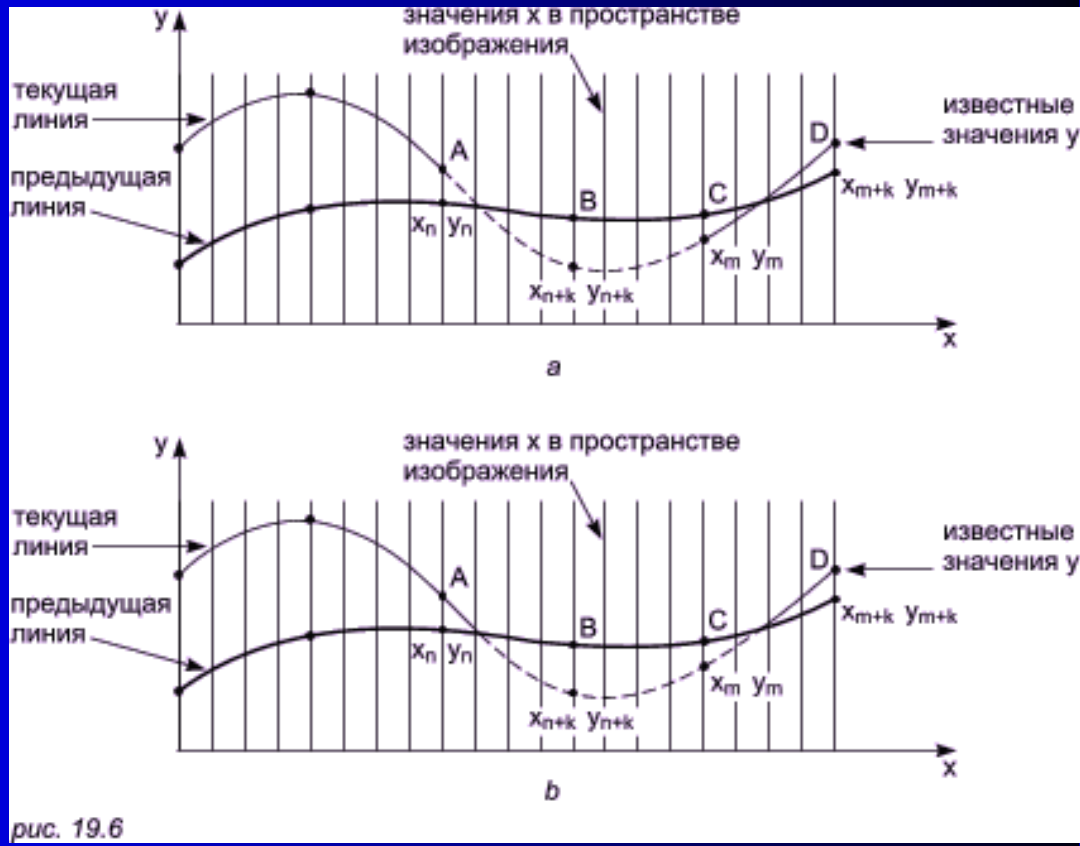


рис. 19.6

Алгоритм плавающего горизонта

Образуется зазор между текущей и предыдущей кривыми Если на участке текущей кривой происходит переход от невидимого состояния к видимому (сегмент CD на рис.), то точка (xm+k, ym+k) объявляется видимой, а участок кривой между точками (xm, ym) и (xm+k, ym+k) изображается и операция по заполнению массивов проводится. Поэтому изображается и невидимый кусок сегмента CD.

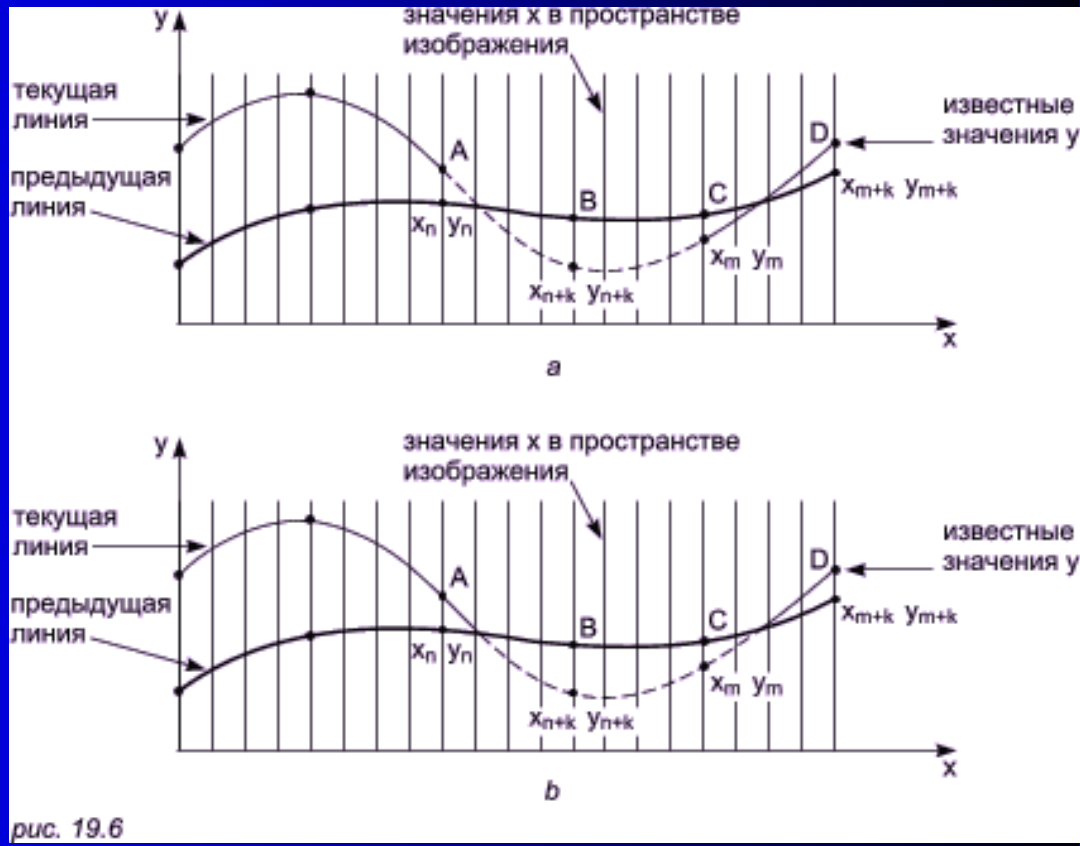


рис. 19.6

Кроме того, массивы плавающих горизонтов не будут содержать точных значений y. А это вызовет дополнительные нежелательные эффекты для последующих кривых. Следовательно, необходимо решать задачу о поиске точек пересечения сегментов.

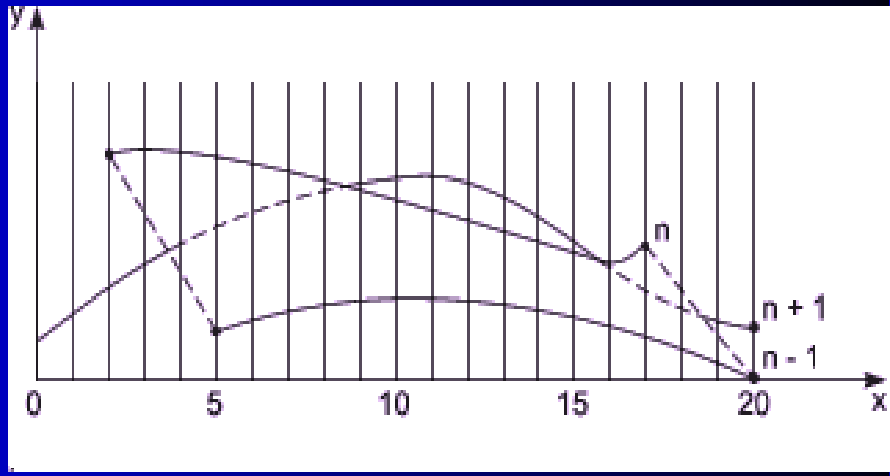
Алгоритм плавающего горизонта

Формальная запись алгоритма:

- Если на текущей плоскости при некотором заданном значении x соответствующее значение y на кривой больше максимума или меньше минимума по y для всех предыдущих кривых при этом x, то текущая кривая видима. В противном случае она невидима.
- Если на участке от предыдущего (x_n) до текущего (x_{n+k}) значения x видимость кривой изменяется, то вычисляется точка пересечения (x_i).
- Если на участке от x_n до x_{n+k} сегмент кривой полностью видим, то он изображается целиком; если он стал невидимым, то изображается фрагмент от x_n до x_i; если же он стал видимым, то изображается фрагмент от x_i до x_{n+k}.
- Заполнить массивы верхнего и нижнего плавающих горизонтов.

Алгоритм плавающего горизонта

- Приведенная версия алгоритма так же не свободна от некоторых недостатков, в частности в случаях, когда кривая, лежащая в одной из более удаленных от точки наблюдения плоскостей, появляется слева или справа из-под множества кривых, лежащих в плоскостях, которые ближе к указанной точке наблюдения.



Такой эффект приводит к появлению зазубренных боковых ребер. Проблема с зазубренностью боковых ребер решается включением в массивы верхнего и нижнего горизонтов ординат, соответствующих штриховым линиям на рисунке.

Алгоритм плавающего горизонта

- Устранить недостаток можно, создав ложные боковые ребра. Приведем алгоритм, реализующий эту идею для обеих ребер.
- Обработка левого бокового ребра:
  - Если P_n является первой точкой на первой кривой, то запомним P_n в качестве P_{n-1} и закончим заполнение. В противном случае создадим ребро, соединяющее P_n и P_{n-1}.
  - Занесем в массивы верхнего и нижнего горизонтов ординаты этого ребра и запомним P_n в качестве P_{n-1}.
- Обработка правого бокового ребра:
  - Если P_n является последней точкой на первой кривой, то запомним P_n в качестве P_{n-1} и закончим заполнение. В противном случае создадим ребро, соединяющее P_n и P_{n-1}.
  - Занесем в массивы верхнего и нижнего горизонтов ординаты этого ребра и запомним P_n в качестве P_{n-1}.

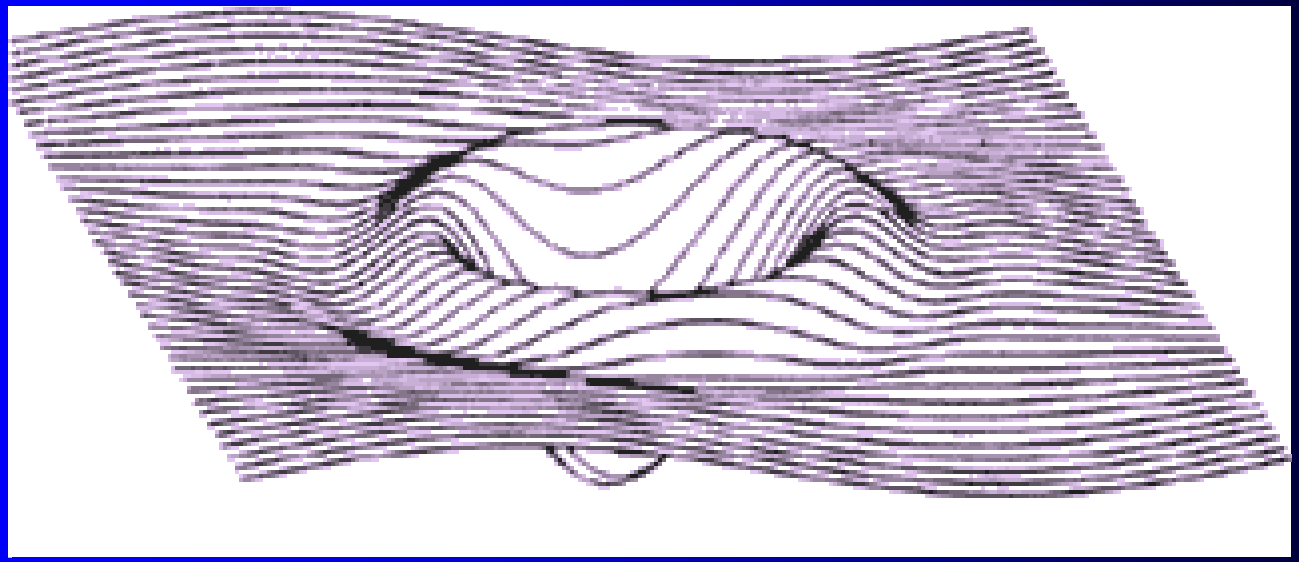
Алгоритм плавающего горизонта

Полная версия алгоритма выглядит следующим образом:

- Для каждой плоскости z = const.
- Обработать левое боковое ребро.
- Для каждой точки, лежащей на кривой из текущей плоскости:
  - Если при некотором заданном значении x соответствующее значение y на кривой больше максимума или меньше минимума по y для всех предыдущих кривых при этом x, то кривая видима (в этой точке). В противном случае она невидима.
  - Если на сегменте от предыдущего (x_n) до текущего (x_{n+k}) значения x видимость кривой изменяется, то вычисляется пересечение (x_i).
  - Если на участке от x_n до (x_{n+k}) сегмент кривой полностью видим, то он изображается целиком; если он стал невидимым, то изображается его часть от x_n до x_i; если же он стал видимым, то изображается его часть от x_i до x_{n+k}.
- Заполнить массивы верхнего и нижнего плавающих горизонтов.
- Обработать правое боковое ребро.

Алгоритм плавающего горизонта

Типичный результат работы алгоритма:



Алгоритм Z-буфера

Алгоритм предложен Эдом Кэтмулом в 1974 г. и основывается на обобщенной идее буфера кадра. Обычный буфер кадра хранит коды цвета для каждого пикселя в пространстве изображения. Идея алгоритма состоит в том, чтобы для каждого пикселя дополнительно хранить еще и координату Z или глубину.

Любая визуализация представляет собой двумерную проекцию на плоскость экрана. Таким образом, любая изображаемая точка в конечном итоге имеет лишь 2 координаты - X и Y. Суть описываемого алгоритма заключается в том, что каждой точке приписывается еще третье значение - расстояние от плоскости проекции (либо - в случае перспективной проекции - от наблюдателя) до “прототипа” этой точки, лежащего на поверхности детали. Это значение и берется как мнимая Z-координата точки (пикселя).

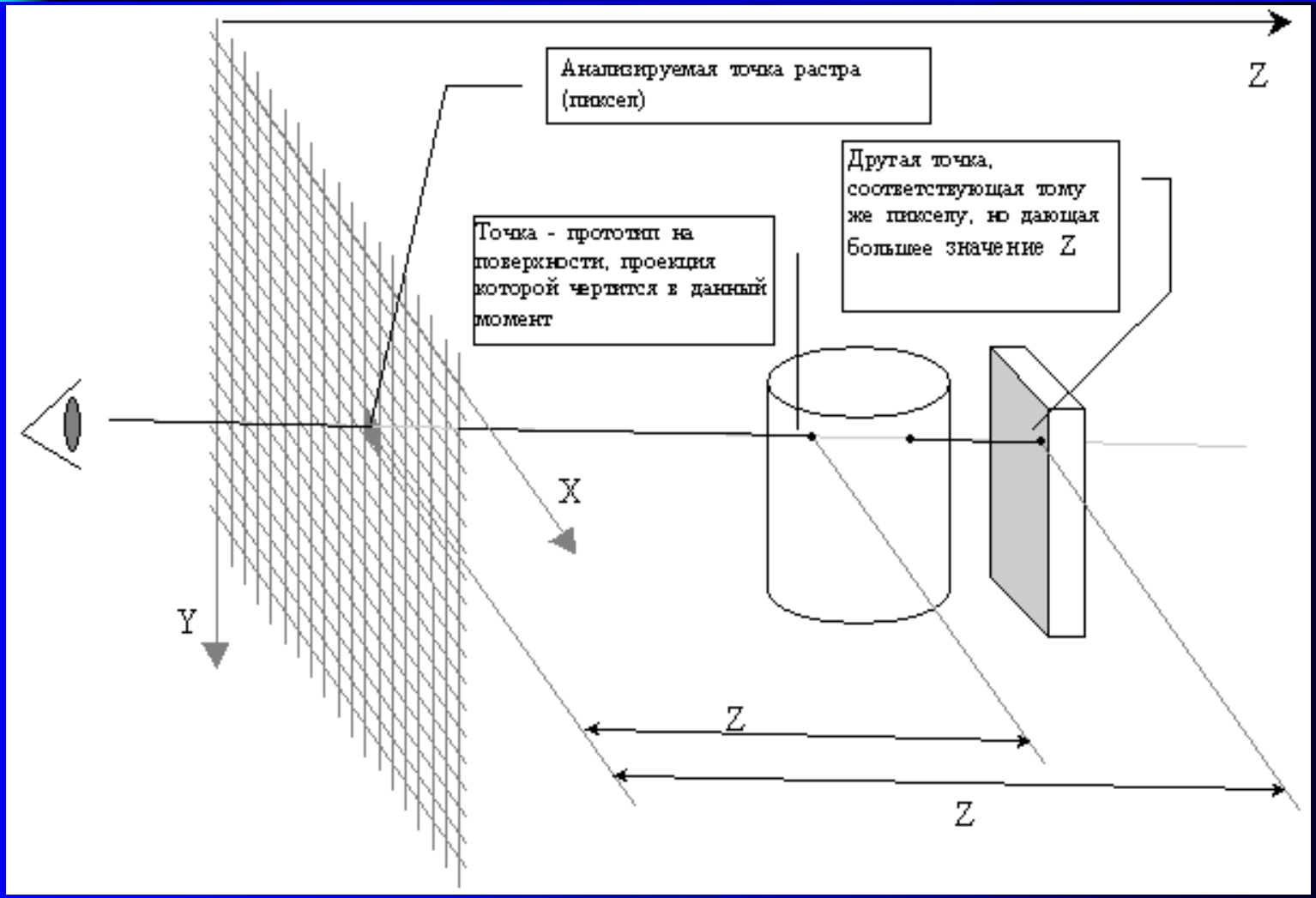
Алгоритм Z-буфера (суть алгоритма)

На этапе инициализации все точки экрана получают некоторое начальное значение координаты Z (обычно для него берется расстояние до максимально удаленной точки сцены). Таким образом, помимо системы координат программы создается система координат экрана, где каждая точка на экране получает Z-значение. Совокупность этих значений заносится в двумерную матрицу и называется Z-буфером экрана.

Общее правило при использовании Z-буфера:

Если при вычерчивании на экране пикселя расстояние от него до соответствующей ему точки - прототипа меньше, чем значение, хранящееся в Z-буфере для этого пикселя, то точка вычерчивается, а в Z-буфер заносится новое (меньшее) значение. Если же расстояние до прототипа больше, чем Z-значение для данного пикселя, то точка игнорируется (пиксель сохраняет прежний цвет).

Алгоритм Z-буфера (суть алгоритма)



Алгоритм Z-буфера (формальная запись)

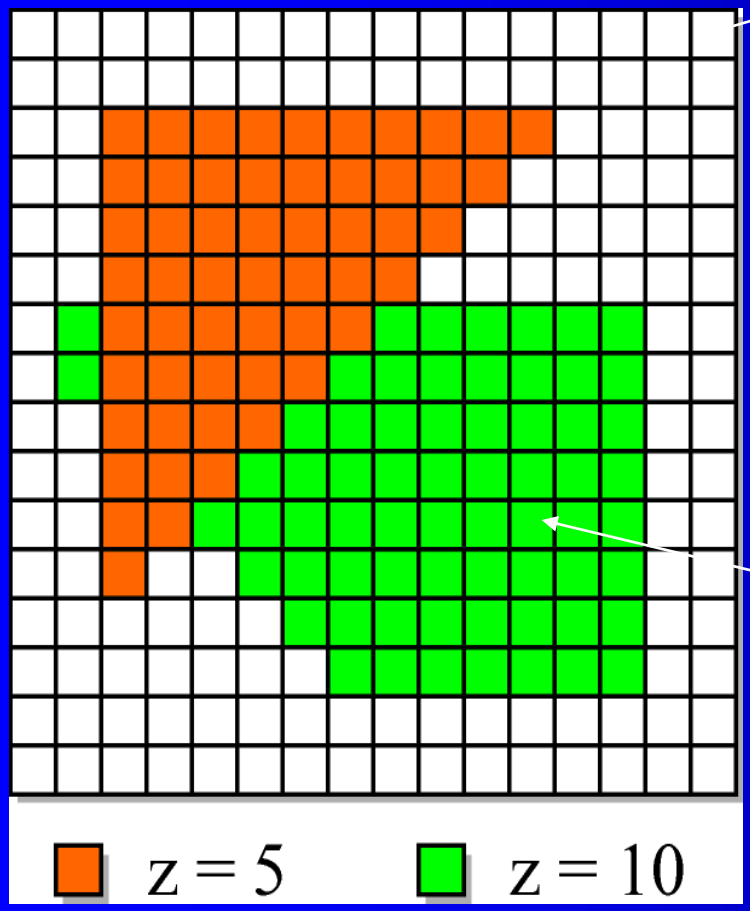
Визуализация сцены с использованием Z-буфера:

· Инициализировать кадровый буфер и Z-буфер. Кадровый буфер закрашивается фоном. Z-буфер закрашивается максимальным значением Z.

· Выполнить преобразование каждого многоугольника сцены в растровую форму. При этом для каждого пикселя вычисляется его глубина z. Если вычисленная глубина меньше, чем значение, уже имеющееся в Z-буфере, то буфер кадра заносятся атрибуты пикселя, а в Z-буфер -> его глубина, иначе никаких занесений не выполнять.

· Выполнить, если это было предусмотрено, усреднение изображения с понижением разрешения.

Z-Buffer Algorithm



Визуализация состояния Z-буфера

Изначально, буфер инициализируется значением $z = z_{max}$

```
For each pixel in polygon:
    if (pixel z < buffer z) then
        buffer z = pixel z
        fill pixel in raster
```

Зелёный полигон расположен позади оранжевого

Если мы закрашиваем пиксель, то мы запоминаем для него расстояние до ближайшего видимого объекта.

Алгоритм Z-буфера (построчный вариант)

Если используется построчный алгоритм заливки граней, то легко сделать пошаговое вычисление Z-координаты очередного пикселя, дополнительно храня Z-координаты предыдущих пикселей в строке и вычисляя приращение *dz* при перемещении вдоль X на *dx* =1. Если известно уравнение плоскости, в которой лежит обрабатываемый многоугольник, то можно обойтись без хранения Z-координат вершин. Пусть уравнение плоскости имеет вид:

$$A \cdot x + B \cdot y + C \cdot z + D = 0$$

Тогда при C не равном нулю:

$$z = -(A \cdot x + B \cdot y + D)/C$$

Приращение Z-координаты пикселя при шаге по X на *dx*, помня, что Y очередной обрабатываемой строки - константа.

$dz = -(A \cdot (x+dx) + B \cdot y + D)/C + (A \cdot x + B \cdot y + D)/C = -A \cdot dx/C$

но *dx = 1*, поэтому $dz = -A/C$.

Алгоритм Z-буфера

Основной недостаток алгоритма с Z-буфером - большой объем требуемой памяти. Если сцена подвергается видовому преобразованию и отсекается до фиксированного диапазона координат z значений, то можно использовать z-буфер с фиксированной точностью. Информацию о глубине нужно обрабатывать с большей точностью, чем координатную информацию на плоскости (x, y); обычно бывает достаточно 20 бит. Буфер кадра размером 512 * 512 * 24 бит в комбинации с z-буфером размером 512 * 512 * 20 бит требует почти 1.5 мегабайт памяти.

Однако снижение цен на память делает экономически оправданным создание специализированных запоминающих устройств для z-буфера и связанной ним аппаратуры. Для уменьшения памяти можно разбивать изображение на несколько прямоугольников или полос. В пределе можно использовать Z-буфер в виде одной строки. Понятно, что это приведет к увеличению времени, так как каждый прямоугольник будет обрабатываться столько раз, на сколько областей разбито пространство изображения. Уменьшение затрат времени в этом случае может быть обеспечено предварительной сортировкой многоугольников на плоскости.

Алгоритм Z-буфера

Другие недостатки алгоритма с Z-буфером заключаются в том, что так как пиксели в буфер заносятся в произвольном порядке, то возникают трудности с реализацией эффектов прозрачности или просвечивания и устранением лестничного эффекта с использованием предфильтрации, когда каждый пиксель экрана трактуется как точка конечного размера и его атрибуты устанавливаются в зависимости от того какие части точек объектов сцены проецируются на конкретный пиксель экрана.

Однако другой подход к устранению лестничного эффекта, основанный на постфильтрации - усреднении значений пикселя с использованием изображения с бóльшим разрешением реализуется сравнительно просто за счет увеличения расхода памяти (и времени).

Реализация Z-буфера в OpenGL®

- Разрешение использования *z*-буфера:

```
glEnable(GL_DEPTH_TEST)
```

- Начальная инициализация *z*-буфера:

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH)
```

- Перед отрисовкой кадра обязательная очистка *z*-буфера:

```
glClear(GL_DEPTH_BUFFER_BIT)
```

- После отрисовки сцены имеется возможность чтения текущих значений *z*-буфера в виде массива типа float (значения в диапазоне [0,1]):

```
glReadPixels(x,y,w,h,GL_DEPTH_COMPONENT,GL_FLOAT,array)
```

Реализация Z-буфера в OpenGL®

- Обычно используются линейные коэффициенты возможных значений глубины:
  - это позволяет создавать Z-буфер с фиксированной точностью по всей глубине (т.е. целочисленный)
  - Если зафиксировать z_{min} и z_{max} сцены, то тогда можно использвать целочисленные значения глубины z в диапазоне - 0…65535 (16-bit):

$$\bar{z} = 65535\left(\frac{z - z_{min}}{z_{max} - z_{min}}\right)$$

- Однако при этом следует обязательно определить переднюю и заднюю отсекающие плоскости объёма видимости:

```
glFrustrum(xmin, xmax, ymin, ymax, zmin, zmax)
glOrtho(xmin, xmax, ymin, ymax, zmin, zmax)
```

определяется проекцией

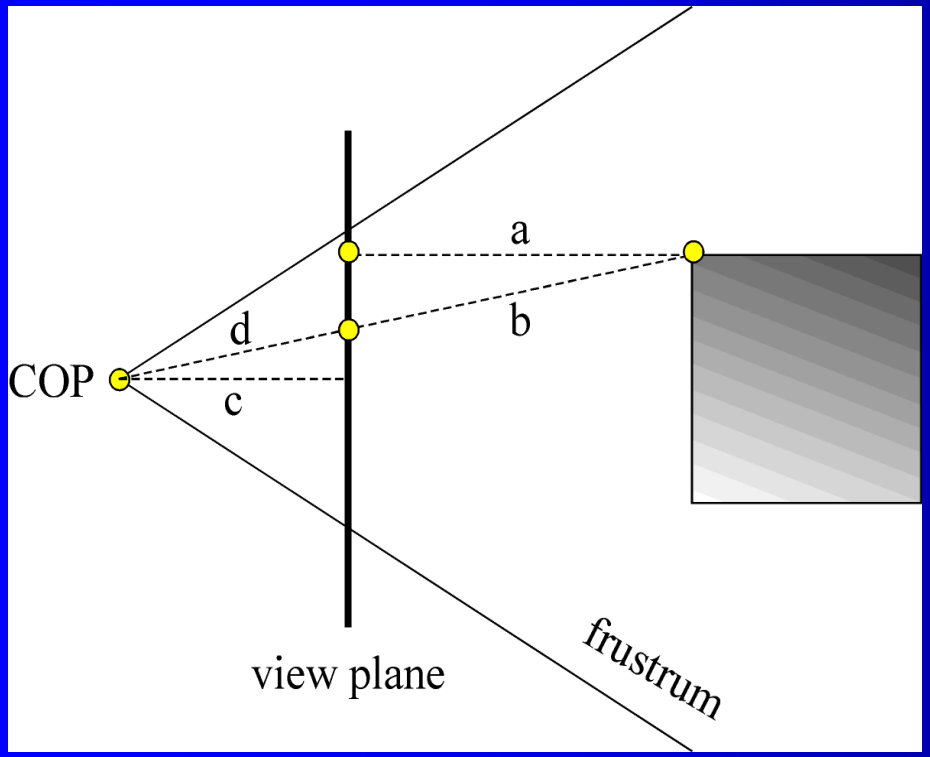
Реализация Z-буфера в OpenGL®

- Однако возможно использование и не линейного диапазона глубины z-буфера:

$$\bar{z} = f\left(\frac{1}{z}\right)$$

- Данная разновидность позволяет обеспечить большую точность на не больших удалениях от наблюдателя и меньшую – для объектов, расположенных в дали.
- Если в каких-то случаях требуется определить действительное значение расстояние от наблюдателя (плоскости проекции) до объекта по значению подобной координаты из *z*-буфера, то:
  - Сперва используется f^{-1} для получения актуального значения координаты *z*
  - Затем производится коррекция *z* для (от) перспективного преобразования

Реализация Z-буфера в OpenGL®



$$\bar{z} = \frac{1}{2}\left(\frac{f+n}{f-n} + \frac{2fn}{z(f-n)}\right) + \frac{1}{2}$$

$\Rightarrow$ Мы легко можем вычислить

$z = a$

Но нам требуется расстояние b:

$$\frac{b}{a} = \frac{d}{c} \Rightarrow b = \frac{ad}{c} = ad$$

если расстояние от ЦП до плоскости проекции = 1

$\Rightarrow$Соответственно значения d должны быть вычислены предварительно для каждого объёма видимости который мы хотим использовать

Почему z-буфер так популярен ?

Преимущества

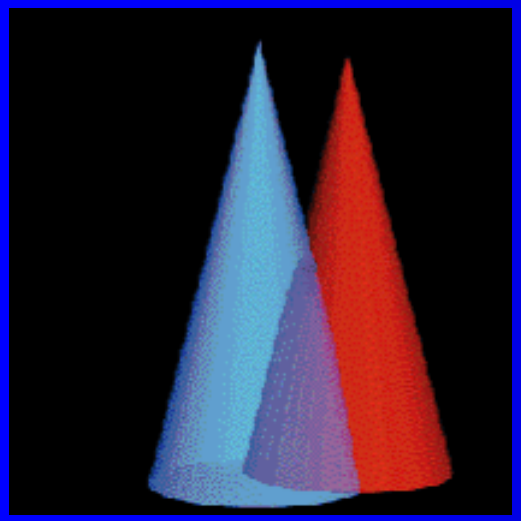
- Прост в реализации на «железе».
  - Добавление дополнительного z интерполятора для каждого примитива.
  - Память для z-буфера уже не дорогая.
- Разнородность использующихся примитивов – не ограничиваемся только полигонами.
- Нелимитированная возможная сложность сцены.
- Отсутствие необходимости вычисления пересечений объектов сцены друг с другом.

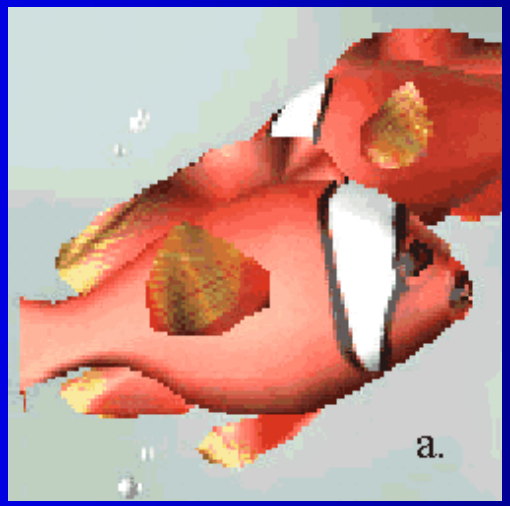
Недостатки

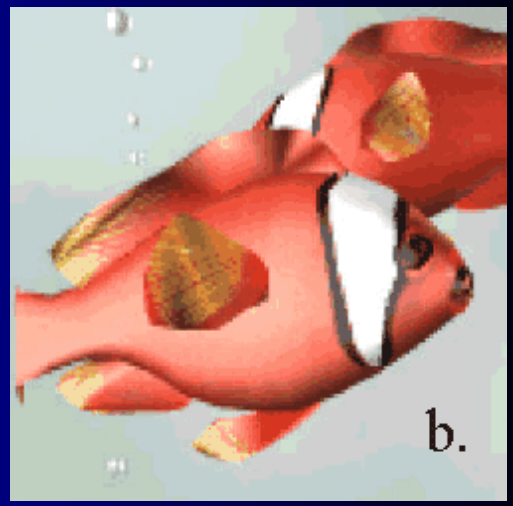
- Дополнительная память и дополнительные требования к каналу передачи данных (полосе пропускания)
- Напрасная трата времени на отрисовку невидимых объектов
- Недостаток точности Z- координат

Алгоритм А-буфера

Алгоритм является расширением метода буфера глубины (в названии использована противоположная от Z буква алфавита). Данное расширение предназначено специально для обеспечения эффектов усреднения по области, прозрачности, оптимизации наложения полигонов. Область буфера в алгоритме называется буфером накопления, так как в ней в дополнение к значениям глубин хранятся различные данные о поверхности.







Алгоритм A-буфера

Z-буфер работает только с непрозрачными поверхностями и не может накапливать коды цветов для нескольких поверхностей. Алгоритм A-буфера расширяет буфер глубины таким образом, чтобы каждая позиция в буфере могла соответствовать целому списку поверхностей. Это позволяет вычислить цвет пикселя как комбинацию цветов различных поверхностей при расчёте прозрачности или защиты от наложения.

Каждая позиция в A-буфере имеет два поля.

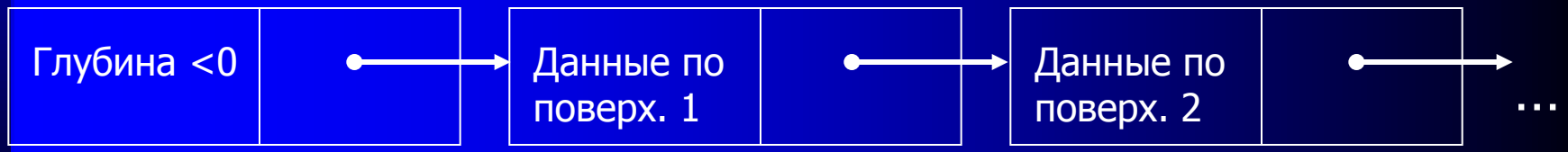
- поле глубины: здесь хранится действительное значение (положительное, отрицательное или ноль).
- поле данных о поверхности: здесь находятся данные о поверхности или указатель.

Если поле глубины неотрицательное, то число записанное в этой позиции – глубина поверхности, проектирующейся в соответствующую область пикселя.

Алгоритм A-буфера

Следующее за ним поле данных о поверхности содержит цвет поверхности для этой точки и процент охвата пикселя.

Если поле глубины для позиции в A-буфере отрицательно, значит цвет пикселя определяется вкладами нескольких поверхностей. Следующее за ним поле содержит указатель на связанный список данных о поверхности.



Алгоритм A-буфера

Данные по поверхности включают следующие поля:

- значения интенсивностей RGB-компонентов
- параметр непрозрачности (процент прозрачности)
- глубина
- процент охвата площади
- идентификатор поверхности
- другие параметры, требуемые для визуализации поверхности.

Схема работы алгоритма аналогична оной в алгоритме Z-буфера. Однако, благодаря возможности хранения доп. информации ∃ большое количество модификаций алгоритма, в отличие от алг. Z-буфера. При реализации прозрачности алгоритм учитывает параметры прозрачности каждой поверхности и её процент охвата каждого пикселя, соответственно цвет каждого пикселя итогового изображения получается как среднее вкладов перекрывающихся поверхностей.