5.3. Трехмерные алгоритмы удаления скрытых линий и поверхностей

Методы удаления невидимых частей сцены можно классифицировать:

- По выбору удаляемых частей:
  - удаление невидимых линий,
  - удаление невидимых ребер,
  - удаление невидимых поверхностей,
  - удаление невидимых объемов.
- По порядку обработки элементов сцены:
  - удаление в произвольном порядке,
  - удаление в порядке, определяемом процессом визуализации.
- По системе координат:
  - алгоритмы работающие в пространстве объектов, когда каждая из n граней объекта сравнивается с остальными $n-1$ гранями (объем вычислений растет как n^2),
  - алгоритмы работающие в пространстве изображения, когда для каждого пиксела изображения определяется какая из n граней объекта видна (при разрешении экрана $m \times m$ объем вычислений растет как $m^2 \times n$).

5.3.1. Алгоритм Варнока (разбиения области)

Алгоритм работает в пространстве изображения и анализирует область на экране дисплея (окно) на наличие в них видимых элементов. Если в окне нет изображения, то оно просто закрашивается фоном. Если же в окне имеется элемент, то проверяется достаточно ли он прост для визуализации. Если объект сложный, то окно разбивается на более мелкие, для каждого из которых повторяется алгоритм. Рекурсивный процесс разбиения может продолжаться до тех пор пока не будет достигнут предел разрешения экрана.

Тесты на сложность визуализации элемента могут быть различными. Здесь приведем классическую схему алгоритма.

Можно выделить 4 случая взаимного расположения окна и многоугольника (рис. 5.32):

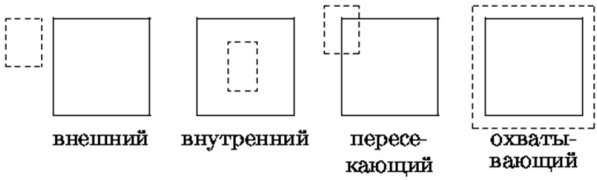

Рис. 5.32. Случаи расположение окна и многоугольника.

- многоугольник целиком вне окна,
- многоугольник целиком внутри окна,
- многоугольник пересекает окно,
- многоугольник охватывает окно.

В четырех случаях можно сразу принять решение о правилах закраски области экрана:

- все многоугольники сцены — внешние по отношению к окну. В этом случае окно закрашивается фоном;
- имеется всего один внутренний или пересекающий многоугольник. В этом случае все окно закрашивается фоном и затем часть окна, соответствующая внутреннему или пересекающему окну закрашивается цветом многоугольника;
- имеется единственный охватывающий многоугольник. В этом случае окно закрашивается его цветом.
- имеется несколько различных многоугольников и хотя бы один из них охватывающий. Если при этом охватывающий многоугольник расположен ближе остальных к наблюдателю, то окно закрашивается его цветом.

В любых других случаях процесс разбиения окна продолжается.

Легко видеть, что при разрешении 1024×1024 и делении стороны окна пополам требуется не более 10 разбиений. Если достигнуто максимальное разбиение, но не обнаружено ни одного из приведенных выше четырех случаев, то для точки с центром в полученном минимальном окне (размером в пиксел) вычисляются глубины оставшихся многоугольников и закраску определяет многоугольник, наиболее близкий к наблюдателю. При этом для устранения лестничного эффекта можно выполнить дополнительные разбиения и закрасить пиксел с учетом всех многоугольников, видимых в минимальном окне.

Первые три случая идентифицируются легко. Последний же случай фактически сводится к поиску охватывающего многоугольника, перекрывающего все остальные многоугольники, связанные с окном. Проверка на такой многоугольник может быть выполнена следующим образом: в угловых точках окна вычисляются z_s-координаты для всех многоугольников, связанных с окном. Если все четыре такие z_s-координаты охватывающего многоугольника ближе к наблюдателю, чем все остальные, то окно закрашивается цветом соответствующего охватывающего многоугольника. Если же нет, то мы имеем сложный случай и разбиение следует продолжить.

Очевидно, что после разбиения окна охватывающие и внешние многоугольники наследуются от исходного окна. Поэтому необходимо проверять лишь внутренние и пересекающие многоугольники.

Из изложенного ясно, что важной частью алгоритма является определение расположения многоугольника относительно окна.

Проверка на то что многоугольник внешний или внутренний относительно окна для случая прямоугольных окон легко реализуется использованием прямоугольной оболочки многоугольника и сравнением координат. Для внутреннего многоугольника должны одновременно выполняться условия:

$$\begin{cases} x_L \geqslant x_{\min}, \\ x_R \leqslant x_{\max}, \\ y_B \geqslant y_{\min}, \\ y_T \leqslant y_{\max}, \end{cases}$$

где x_L, x_R, y_B, y_T — ребра оболочки, $x_{\min}$, $x_{\max}$, $y_{\min}$, $y_{\max}$ — ребра окна.

Для внешнего многоугольника достаточно выполнение любого из следующих условий:

$$\begin{cases} x_L > x_{\max}, \\ x_R < x_{\min}, \\ y_B > y_{\max}, \\ y_T < y_{\min}, \end{cases}$$

Таким способом внешний многоугольник, охватывающий угол окна не будет идентифицирован как внешний (см. рис. 5.33).

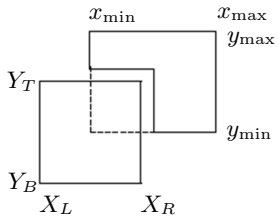

Рис. 5.33. Многоугольник, охватывающий угол.

Проверка на пересечение окна многоугольником может быть выполнена проверкой на расположение всех вершин окна по одну сторону от прямой, на которой расположено ребро многоугольника. Пусть ребро многоугольника задано точками $P_1 = (x_1, y_1, z_1)$ и $P_2 = (x_2, y_2, z_2)$, а очередная вершина окна задается точкой $P_3 = (x_3, y_3, z_3)$. Псевдоскалярное произведение вектора P_1P_3 на вектор P_1P_2, равное $(x_3 - x_1)(y_2 - y_1) - (y_3 - y_1)(x_2 - x_1)$ будет меньше 0, равно 0 или больше 0, если вершина лежит слева, на или справа от прямой P_1P_2. Если знаки различны, то окно и многоугольник пересекаются. Если же все знаки одинаковы, то окно лежит по одну сторону от ребра, т.е. многоугольник может быть либо внешним, либо охватывающим.

5.3.2. Алгоритм удаления поверхностей с Z-буфером

Алгоритм предложен Эдом Кэтмулом и представляет собой обобщение буфера кадра. Обычный буфер кадра хранит коды цвета для каждого пиксела в пространстве изображения. Идея алгоритма состоит в том, чтобы для каждого пиксела дополнительно хранить еще и координату z точки, изображенной в этом пикселе. При попытке изображения точки сцены в некотором пикселе растровой области сравнивается значение её z-координаты с z-координатой пиксела, находящейся в буфере. Если z-координата точки меньше, чем координата уже изображенной точки, т.е. она ближе к наблюдателю, то точка изображается и её z-координата заносятся в буфер, если нет, то ничего не делается.

Дополнительный массив, выполняющий роль буфера, в который заносятся координаты z изображенных точек называют Z-буфером.

Алгоритм с использованием Z-буфера наиболее простой из всех алгоритмов удаления невидимых поверхностей, но требует большого объема памяти. Например, если

необходимо иметь разрядность порядка 20 бит на 1 пиксел, то при изображении нормального телевизионного размера в 768×576 пикселов для хранения z_s-координат необходим объем памяти порядка 1 МБ, а суммарный объем памяти при 3 байтах для значений RGB составит более 2.3 МБ.

Время работы алгоритма зависит от сложности сцены. Многоугольники, составляющие сцену, могут обрабатываться в произвольном порядке. Для сокращения затрат времени нелицевые многоугольники могут быть удалены. По сути дела алгоритм с Z-буфером — некоторая модификация уже рассмотренного алгоритма заливки многоугольника. Если используется построчный алгоритм заливки, то легко сделать пошаговое вычисление z_s-координаты очередного пиксела, дополнительно храня z_s-координаты его вершин и вычисляя приращение Δz z_s-координаты при перемещении вдоль оси Ox_s на $\Delta x = 1$.

Общая схема алгоритма с Z-буфером:

1. Инициализировать растровую область и Z-буфер. Растровая область закрашивается фоном. Z-буфер заполняется максимальным значением z.
2. Для каждой строки растра

   - Пересматривается список активных ребер (*AEL*) на предмет сокращения или пополнения.
   - Для каждой пары активных ребер выполняется возможное заполнение строки растра и Z-буфера (процедура ShowLine, алгоритм 11).
   - Производится обработка списка активных ребер в целях подготовки к следующей строке растра (пересчитываются величины x и z для каждого активного ребра).

При переборе строк растра в порядке увеличения координаты y список активных ребер можно рассматривать как список пятерок: для каждого ребра, заданного координатами (x_1, y_1, z_1) и (x_2, y_2, z_2), где $y_1 < y_2$, в этом списке сохраняются величины $\left(x_1, z_1, y_2, \dfrac{x_2 - x_1}{y_2 - y_1}, \dfrac{z_2 - z_1}{y_2 - y_1}\right)$. Первые два элемента в этой пятерке — значения x и z для проекции ребра на текущую строку растра (первоначально x_1 и z_1). Третий элемент — значение координаты y, при котором данное ребро нужно удалить из списка активных ребер. Два последних элемента пятерки — приращения значений x и y при переходе к следующей строке растра (при увеличении y на 1).

В общем виде метод Z-буфера приведен в алгоритме 12 с использованием процедуры, сформулированной в алгоритме 11.

Алгоритм 11: Процедура SHOWLINE для метода Z-буфера

Вход: y_0 — координата y строки растра, (x_1, z_1) — координаты x и z первой точки, (x_2, z_2) — координаты x и z второй точки. C — цвет соответствующего многоугольника.
Выход: Измененные Z-буфер и область рисования.
начало алгоритма
  $x = x_1$; $z = z_1$; $\Delta z = \dfrac{z_2 - z_1}{x_2 - x_1}$;
  цикл пока $x \leqslant x_2$ **выполнять**
    если $z < Z[x, y_0]$, **то**
      Присвоить $Z[x, y_0] = z$ и закрасить в растровой области точку с координатами (x, y_0) цветом C;
    Присвоить $x = x + 1$, $z = z + \Delta z$;
конец алгоритма

Основной недостаток алгоритма с Z-буфером — дополнительные затраты памяти. Для их уменьшения 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.

Другие недостатки алгоритма с Z-буфером заключаются в том, что так как пикселы в буфер заносятся в произвольном порядке, то возникают трудности с реализацией эффектов прозрачности или просвечивания и устранением лестничного эффекта с использованием предфильтрации, когда каждый пиксел экрана трактуется как точка конечного размера и его атрибуты устанавливаются в зависимости от того какая часть пиксела изображения попадает в пиксел экрана.

Алгоритм 12: Отсечение невидимых граней с использованием Z-буфера

Вход: $\mathscr{P}$ — список многоугольников трехмерной сцены.
начало алгоритма
- · Заполнить растровую область рисования цветом фона. Определить вещественнозначный двумерный массив Z с размерностями (и индексами элементов) области рисования;
- **цикл пока** *список* $\mathscr{P}$ *не пуст* **выполнять**
  - · Взять из списка $\mathscr{P}$ очередной многоугольник P с цветом C;
  - · Сформировать список S ребер многоугольника. Упорядочить список S по возрастанию значения y_1;
  - · Найти y_{min} и y_{max} — минимальное и максимальное значение координаты y точек вершин многоугольника;
  - · $AEL = \varnothing$, $y_t = y_{min}$, $y_{S\,next} = y_{min}$;
  - **цикл пока** $y_t \leqslant y_{max}$ **выполнять**
    - **если** $y_t = y_{S\,next}$, **то**
      - · Добавить в AEL все пятерки $\left(x_1, z_1, y_2, \dfrac{x_2 - x_1}{y_2 - y_1}, \dfrac{z_2 - z_1}{y_2 - y_1}\right)$, составленные для каждого отрезка из S, у которого $y_1 = y_t$ и $y_1 \neq y_2$;
      - · Для всех ребер в S, у которых $y_1 = y_t$ и $y_1 = y_2$ выполнить ShowLine(y_t, (x_1, z_1), (x_2, z_2));
      - · Удалить из S все ребра, у которых $y_1 = y_t$;
      - · Для отрезков в S найти $y_{S\,next}$ — минимальное значение y_1 у отрезков в S;
      - · Отсортировать AEL по возрастанию первого элемента и по возрастанию четвертого элемента;
      - · Найти $y_{AEL\,next}$ — минimальное значение третьего элемента в пятерках в AEL;
    - · $i = 1$;
    - **цикл пока** $i \leqslant |AEL|$ **выполнять**
      - · Выполнить ShowLine(y_t, (x_i, z_i), (x_{i+1}, z_{i+1}), C), где $(x_i, z_i, y_i, \Delta_i x, \Delta_i z)$ обозначает i-й элемент списка AEL;
      - · $i = i + 2$;
    - · $y_t = y_t + 1$;
    - **если** $y_t \geqslant y_{AEL\,next}$, **то**
      - · Удалить из AEL пятерки с третьим элементом меньшим или равным y_t;
      - · Обновить значение $y_{AEL\,next}$;
    - · В каждой пятерке $(x_j, z_j, y_j, \Delta_j x, \Delta_j z)$ в AEL заменить x_j на $x_j + \Delta_j x$, z_j на $z_j + \Delta_j z$;

конец алгоритма

5.3.3. Алгоритм с использованием S-буфера

Метод S-буфера (или алгоритм построчного сканирования с использованием Z-буфера) является оптимизацией предыдущего метода в целях уменьшения размера Z-буфера до одной строки растра. В данной оптимизации многоугольники обрабатываются не последовательно, а в совокупности. Считаем, что в списке активных многоугольников (*APL*) для каждой строки растра (сканирующей строки) занесены все многоугольники, проецируемые на эту строку. В списке активных ребер заносятся все

ребра активных многоугольников, проецируемые на строку растра. Чтобы отделять в списке активных ребер ребра одного многоугольника от ребер другого, к каждой пятерке элементов, соответствующей ребру многоугольника, добавим идентификатор многоугольника и его цвет. Таким образом в AEL вместо пятерок будем записывать семерки элементов.

Для каждой строки растра в таком случае выполняется следующая последовательность действий:

- Пересматривается список активных многоугольников.
- Пересматривается список активных ребер на предмет сокращения или пополнения.
- Инициализируется S-буфер. Соответствующая строка растра закрашивается фоном. S-буфер закрашивается максимальным значением z.
- Для каждой пары ребер активных многоугольников выполняется возможное заполнение строки растра и S-буфера (процедура ShowLine, алгоритм 11).
- Производится обработка списка активных ребер в целях подготовки к следующей строке растра.

В общем виде метод S-буфера приведен в алгоритме 14 с использованием процедуры, сформулированной в алгоритме 13.

Алгоритм 13: Процедура ShowLine для метода S-буфера

Вход: y_0 — координата y строки растра, (x_1, z_1) — координаты x и z первой точки, (x_2, z_2) — координаты x и z второй точки. C — цвет соответствующего многоугольника.
Выход: Измененные Z-буфер и область рисования.
начало алгоритма
  $x = x_1$; $z = z_1$; $\Delta z = \dfrac{z_2 - z_1}{x_2 - x_1}$;
  цикл пока $x \leqslant x_2$ **выполнять**
    если $z < Z[x]$, **то**
      Присвоить $Z[x] = z$ и закрасить в растровой области точку с координатами (x, y_0) цветом C;
    Присвоить $x = x + 1$, $z = z + \Delta z$;
конец алгоритма

Алгоритм 14: Отсечения невидимых граней с использованием S-буфера

Вход: $\mathscr{P}$ — список многоугольников трехмерной сцены
начало алгоритма
- Для каждого многоугольника в $\mathscr{P}$ вычислить $y_{P\min}$ и $y_{P\max}$ — значения минимальной и максимальной координаты y для вершин многоугольника;
- Определить значения $y_{P\,next}$, $y_{\max}$ — минимальное значение $y_{P\min}$ и максимальное значение $y_{P\max}$ для многоугольников в $\mathscr{P}$;
- $y_t = y_{P\,next}$;

цикл пока $y_t \leqslant y_{\max}$ **выполнять**
- **если** $y_t = y_{P\,next}$, **то**
  - Занести в список APL все многоугольники из $\mathscr{P}$, для которых $y_{P\min} = y_t$, удалив их из $\mathscr{P}$;
  - $y_{APL\,next} = y_t$;
  - **если** *список* $\mathscr{P}$ *пуст,* **то** присвоить $y_{P\,next} = \infty$;
  - **иначе**
    - Определить значение $y_{P\,next}$ — минимальное значение $y_{P\min}$ для многоугольников в $\mathscr{P}$
- **если** $y_t \geqslant W_{cy}$ **то**
  - Инициализировать строку растра, соответствующую координате y_t;
  - Заполнить строку растра цветом фона. Определить вещественнозначный массив Z с количеством элементов равным количеству пикселов в строке растра;
- **если** $y_t = y_{APL\,next}$, **то**
  - Добавить в AEL все семерки $\left(P, C, x_1, z_1, y_2, \dfrac{x_2 - x_1}{y_2 - y_1}, \dfrac{z_2 - z_1}{y_2 - y_1}\right)$, составленные для ребер многоугольников из APL, у которых $y_1 = y_t$ и $y_1 \neq y_2$;
  - Для всех ребер многоугольников в APL, у которых $y_1 = y_t$ и $y_1 = y_2$ выполнить ShowLine(y_t, (x_1, z_1), (x_2, z_2), C), где C — цвет соответствующего многоугольника; ;
  - Для ребер многоугольников в APL найти $y_{APL\,next}$ — минимальное значение y_1 такое, что $y_1 > y_t$. Если таких ребер нет в APL, то присвоить $y_{APL\,next} = \infty$;
  - Упорядочить AEL по значению 1-го элемента семерки, затем по возрастанию 3-го, и затем по возрастанию 6-го;
  - Найти $y_{AEL\,next}$ — минимальное значение пятого элемента в семерках в AEL;
- $i = 1$;
- **цикл пока** $i \leqslant |AEL|$ **выполнять**
  - Выполнить ShowLine(y_t, (x_i, z_i), (x_{i+1}, z_{i+1}), C_i), где $(P_i, C_i, x_i, z_i, y_i, \Delta_i x, \Delta_i z)$ обозначает i-й элемент списка AEL;
  - $i = i + 2$;
- $y_t = y_t + 1$;
- **если** $y_t \geqslant y_{AEL\,next}$, **то**
  - удалить из AEL семерки с пятым элементом меньшим или равным y_t;
  - Обновить значение $y_{AEL\,next}$;
- В каждой семерке $(P_j, C_j, x_j, z_j, y_j, \Delta_j x, \Delta_j z)$ в AEL заменить x_j на $x_j + \Delta_j x$, z_j на $z_j + \Delta_j z$;

конец алгоритма

5.3.4. Алгоритм Уоткинса

Алгоритм Уоткинса (интервальный метод построчного сканирования) работает с совокупностью проекций активных многоугольников на очередную строку растра, как и предыдущий алгоритм. Но в этом алгоритме не используется дополнительный буфер, а анализируются тот набор отрезков, который определяется списком активных ребер. Данный набор отрезков является результатом пересечения сканирующей плоскости (плоскости, параллельной плоскости xOz и проходящей через текущую строку растра) с многоугольниками трехмерной сцены (см. рис. 5.34).

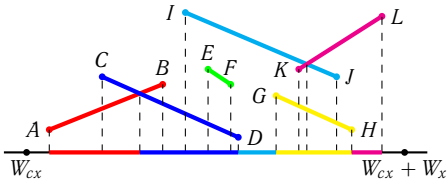

Рис. 5.34. Отрезки, проецируемые на строку растра

Основная процедура в алгоритме — процедура обработки строки растра (процедура ProcessLine, алгоритм 15), в которой выясняется — каким цветом будет закрашиваться каждый из интервалов строки растра. Интервалы определяются точками — концами отрезков, в порядке возрастания координаты x этих точек. Так, например, 12 точек (обозначенных символами латинского алфавита) на рисунке 5.34 соответствуют элементам списка AEL. Эти точки, вместе с начальной и конечной точками строки растра, образуют 13 интервалов.

Интервалы обрабатываются по порядку слева направо. Порядок следования интервалов определяется списком AEL, отсортированным в порядке возрастания текущей координаты x для каждого ребра. Так, для приведенного примера, ребра в списке AEL будут следовать в порядке: A, C, B, I, E, F, D, G, K, J, H, L. Все многоугольники, проецируемые на определенный интервал называются активными по отношению к этому интервалу. Так например для интервала $[K, J]$ активными будут три многоугольника, соответствующие отрезкам GH, IJ и KL; для интервала $[A, C]$ будет активным только один многоугольник, соответствующий отрезку AB; для интервалов [*начало*, A] и [L, *конец*] нет активных многоугольников.

Алгоритм 15: ProcessLine

начало алгоритма
- $x_{left} = -\infty$; $x_{right} = W_{cx}$ $polygonCount = 0$; $i = 1$;
- **цикл пока** *не достигнут конец списка AEL и* $x_{left} < W_{cx} + W_x$ **выполнять**
  - Пусть $(P_i, x_i, y_i, \Delta_i x)$ — очередной элемент AEL;
  - $x_{right} = x_i$;
  - **если** $x_{right} > W_{cx}$ **то**
    - **если** $x_{left} < W_{cx}$ **то** присвоить $x_{left} = W_{cx}$;
    - **если** $x_{right} > W_{cx} + W_x$ **то** присвоить $x_{right} = W_{cx} + W_x$;
    - **если** $polygonCount = 0$ **то** присвоить цвет фона переменной C;
    - **иначе если** $polygonCount = 1$ **то** присвоить $C = C_i$ — цвет многоугольника P_i;
    - **иначе**
      - Присвоить $intersectionStack = \varnothing$;
      - **повторять вычисления**
        - **если** $intersectionStack \neq \varnothing$ **то**
          - Начертить отрезок $[(x_{left}, y_0), (x_{right}, y_0)]$ цветом C;
          - Присвоить $x_{left} = x_{right}$;
          - Извлечь значение из стека $intersectionStack$ и присвоить его переменной x_{right};
        - **цикл пока** HasIntersection$(x_{left}, x_{right}, x_{int})$ **выполнять**
          - Занести x_{right} в стек $intersectionStack$;
          - Присвоить $x_{right} = x_{int}$;
        - Для всех P, таких, что $Active(P)$, найти z_{mid}

$$z_{mid} = -\frac{a(x_{left} + x_{right}) + 2by_t + 2d}{2c};$$

        - Присвоить переменной C цвет многоугольника с минимальным z_{mid};
      - **пока** $intersectionStack \neq \varnothing$;
  - Изменить $Active(P_i) = !Active(P_i)$;
  - **если** $Active(P_i)$ **то** $polygonCount = polygonCount + 1$;
  - **иначе** $polygonCount = polygonCount - 1$;
  - Начертить отрезок $[(x_{left}, y_0), (x_{right}, y_0)]$ цветом C;
  - Присвоить $x_{left} = x_{right}$, $i = i + 1$;
- **если** $x_{left} < W_{cx} + W_x$ **то**
  - **если** $x_{left} < W_{cx}$ **то** присвоить $x_{left} = W_{cx}$;
  - Присвоить $x_{right} = W_{cx} + W_x$;
  - Начертить отрезок $[(x_{left}, y_0), (x_{right}, y_0)]$ цветом фона;

конец алгоритма

Обработка интервалов проводится в следующем порядке:

- Если в интервале нет активных многоугольников, то интервал закрашивается цветом фона (интервалы $[начало, A]$ и $[L, конец]$).
- Если в интервале только один активный многоугольник, — интервал закрашивается цветом многоугольника (интервалы $[A, C]$, $[B, I]$, $[D, G]$, $[H, L]$).
- Если в интервал попадает больше одного многоугольника то:
  - Если в интервале нет пересечений отрезков (интервалы $[I, E]$, $[E, F]$,

$[F, D]$, $[G, K]$, $[J, H]$), то среди всех отрезков ищется отрезок, с минимальным значением координаты z для точки середины интервала.

– Если в интервале есть пересечения отрезков (интервалы $[C, B]$ и $[K, J]$), то: находится координата x точки пересечения; интервал делится точкой пересечения на части; для каждой из частей интервала повторяется алгоритм.

В общем виде интервальный метод построчного сканирования приведен в алгоритме 17 с использованием процедур, описанных в алгоритмах 15 и 16.

Алгоритм 16: HASINTERSECTION Проверка наличия пересечений на интервале

Вход: x_{left}, x_{right}. Параметр x_{int} может использоваться для возвращения одного из результатов.

Выход: $False$ — если на отрезке от x_{left} до x_{right} отсутствуют пересечения активных многоугольников. $True$ — в противном случае. В случае возврата $True$ параметр x_{int} содержит значение координаты x для точки пересечения.

начало алгоритма

цикл для каждого *многоугольника* P_j *такого, что* $Active(P_j)$ **выполнить**

· Вычислить

$$z_{j\,left} = -\frac{a_j x_{left} + b_j y_t + d_j}{c_j}; \quad z_{j\,right} = -\frac{a_j x_{right} + b_j y_t + d_j}{c_j};$$

цикл для каждого *многоугольника* P_k *такого, что* $Active(P_k)$ **выполнить**

· Вычислить

$$z_{k\,left} = -\frac{a_k x_{left} + b_k y_t + d_k}{c_k}; \quad z_{k\,right} = -\frac{a_k x_{right} + b_k y_t + d_k}{c_k};$$

· Вычислить $\Delta z_{left} = z_{j\,left} - z_{k\,left}$; $\Delta z_{right} = z_{j\,right} - z_{k\,right}$;

если $sign(\Delta z_{left}) \neq sign(\Delta z_{right})$ **то**

· Вычислить

$$x_{int} = \frac{x_{right}\Delta z_{left} - x_{left}\Delta z_{right}}{\Delta z_{left} - \Delta z_{right}};$$

· Выдать $True$ и **закончить алгоритм**;

· Выдать $False$;

конец алгоритма

Алгоритм 17: Алгоритм Уоткинса

Вход: $\mathscr{P}$ — список многоугольников трехмерной сцены
начало алгоритма
- Для каждого многоугольника в $\mathscr{P}$ вычислить $y_{P\min}$ и $y_{P\max}$ — значения минимальной и максимальной координаты y для вершин многоугольника;
- Определить значения $y_{P\,next}$, $y_{\max}$ — минимальное значение $y_{P\min}$ и максимальное значение $y_{P\max}$ для многоугольников в $\mathscr{P}$;
- $y_t = y_{P\,next}$;

цикл пока $y_t \leqslant y_{\max}$ **выполнять**
  если $y_t = y_{P\,next}$, **то**
    - Занести в список APL все многоугольники из $\mathscr{P}$, для которых $y_{P\min} = y_t$, удалив их из $\mathscr{P}$;
    - Для каждого нового многоугольника P_j в APL установить значение $Active(P_j) = False$ и определить значения пятерки $(C_j, a_j, b_j, c_j, d_j)$, где C_j — цвет многоугольника, а a_j, b_j, c_j, d_j — коэффициенты уравнения несущей плоскости для многоугольника P_j. Если $c_j = 0$ — удалим многоугольник из APL;
    - $y_{APL\,next} = y_t$;

    если *список* $\mathscr{P}$ *пуст,* **то** присвоить $y_{P\,next} = \infty$;
    иначе
      Определить значение $y_{P\,next}$ — минимальное значение $y_{P\min}$ для многоугольников в $\mathscr{P}$

  если $y_t = y_{APL\,next}$, **то**
    - Добавить в AEL все четверки $\left(P, x_1, y_2, \dfrac{x_2 - x_1}{y_2 - y_1}\right)$, составленные для ребер многоугольников из APL, у которых $y_1 = y_t$ и $y_1 \neq y_2$;
    - Добавить в AEL все четверки $(P, x_1, y_2, 0)$, составленные для ребер многоугольников из APL, у которых $y_1 = y_t$ и $y_1 = y_2$;
    - Для ребер многоугольников в APL найти $y_{APL\,next}$ — минимальное значение y_1 такое, что $y_1 > y_t$. Если таких ребер нет в APL, то присвоить $y_{APL\,next} = \infty$;
    - Упорядочить AEL по значению 2-го элемента четверки;
    - Найти $y_{AEL\,next}$ — минимальное значение третьего элемента в четверках в AEL;

  Выполнить процедуру ProcessLine
  - $y_t = y_t + 1$;

  если $y_t \geqslant y_{AEL\,next}$, **то**
    - удалить из AEL четверки с третьим элементом меньшим либо равным y_t;
    - Обновить значение $y_{AEL\,next}$;

  - В каждой четверке $(P_j, x_j, y_j, \Delta_j x)$ в AEL заменить x_j на $x_j + \Delta_j x$;

конец алгоритма

5.3.5. Алгоритм художника с использованием BSP-дерева

Алгоритм художника предполагает, что трехмерная сцена состоит из набора многоугольников, упорядоченных по степени приближения к наблюдателю. То есть трехмерную сцену можно представить упорядоченным списком многоугольников, где на первом месте находится самый удаленный от наблюдателя многоугольник, а на последнем — самый близкий к наблюдателю. В этом случае можно по порядку (от пер-

вого до последнего) изобразить многоугольники на экране, подобно «художнику», наносящему краски на холст. Тогда наиболее приближенный к наблюдателю многоугольник будет изображен в последнюю очередь и окажется полностью видимым. При этом он может перекрывать собой любой из многоугольников, изображенных ранее.

Такой подход является вполне удовлетворительным, если в трехмерной сцене отсутствуют многоугольники «протыкающие друг друга» или набор многоугольников, среди которого нельзя установить какой из многоугольников ближе другого (как на рисунке 5.35), а следовательно, сортировка «по приближенности к наблюдателю» невозможна. Выходом из таких ситуаций является разделение многоугольников,

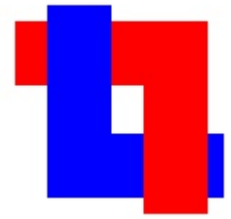

Рис. 5.35. Многоугольники,

входящих в такую группу, на более мелкие части, для которых сортировка станет возможной.

Один из методов, реализующих предварительную сортировку многоугольников для алгоритма художника и организующих разбиения многоугольников в случае необходимости — алгоритм использующий BSP-дерево.

Алгоритм построения BSP-дерева — двоичное разбиение пространства (BSP — binary space partition) — это метод, который впервые был сформулирован в 1969 году. Изначально предлагалось использовать метод для сортировки полигонов в сцене. В то время отсутствовала аппаратная метода Z-буфера, а программная реализация была слишком медленной. Однако метод двоичного разбиения пространства остался актуальным даже после создания аппаратного Z-буфера (поскольку, к сожалению, метод Z-буфера не решает многих проблем, например, отображение полупрозрачных объектов). Кроме сортировки, метод широко применяется и в других областях, к примеру, проверка на столкновение, в некоторых алгоритмах компьютерных сетей, в методе

излучательности.

Скорость работы метода двоичного разбиения пространства достигается за счёт разбиения исходного пространства и проведения предварительных вычислений. На вход метода поступает набор некоторых объектов (в случае сортировки полигонов в сцене этими объектами являются полигоны), а потом с помощью рекурсивного алгоритма создаётся двоичное дерево таким образом, что можно совершать обход дерева в порядке от более удалённых к менее удалённым или от менее удалённых к более удалённым многоугольникам.

Принцип работы алгоритма заключается в том, что все полигоны пространства (в общем случае n-мерного) разбиваются на группы, лежащие в разных выпуклых подпространствах относительно некоторой гиперплоскости (гиперплоскость — это пространство размерности $n - 1$). Не нарушая общности рассуждений будем рассматривать двумерное пространство (далее сделаем некоторые поправки для трёхмерного пространства). В этом случае гиперплоскость будет представлять собой прямую линию. Для наглядности рассмотрим пример расположения многоугольников, представленный на рисунке 5.36.

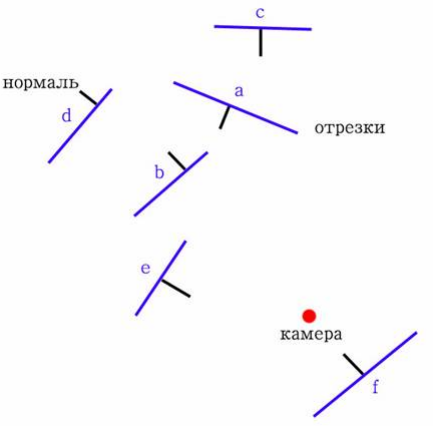

Рис. 5.36. Первоначальное расположение многоугольников

У нас есть плоскость (пространство), на которой расположены отрезки (участки гиперплоскостей) с заданными нормалями. Первым этапом алгоритма является определение разделяющей плоскость прямой. Будем выбирать произвольный отрезок

(например, отрезок **b**) и дополнять его до прямой — это и будет наша гиперплоскость (см. рис. 5.37):

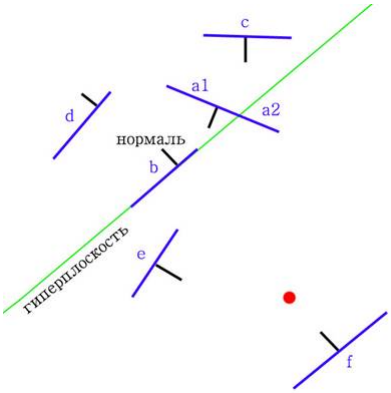

Рис. 5.37. Разделение первоначальной сцены первой гиперплоскостью

Благодаря прямой, мы получили плоскость, разбитую на две полуплоскости. Нормаль к прямой совпадает с нормалью отрезка, через который она проведена. По направлению нормали мы будем определять переднюю и заднюю полуплоскости: если нормаль находится в полуплоскости, то данная полуплоскость — передняя; иначе — задняя. Теперь нужно определить, каким полуплоскостям принадлежат отрезки. Таким образом все отрезки разбиваются на три группы: отрезки, лежащие в передней полуплоскости (**c** и **d**), отрезки, лежащие в задней полуплоскости (**e** и **f**), и отрезки, лежащие на прямой (только **b**). Если отрезок принадлежит обоим полуплоскостям, то он делится на два (так **a** делится на **a1** и **a2**). Если узлу двоичного дерева приписать прямую и все отрезки, лежащие на ней, а две оставшиеся группы приписать его дочерним поддеревьям, то получим образование структуры, представленной на рисунке 5.38.

Теперь нужно рекурсивно повторить алгоритм для каждого поддерева. То есть, мы имеем в качестве пространства переднее полуподпространство (в него входят отрезки **d**, **a1**, **c**). Выбираем любой из этих отрезков, например, **a1** и проводим через него гиперплоскость: получаем в качестве переднего поддерева отрезок **d**, а заднего — **c** (см. рис. 5.39):

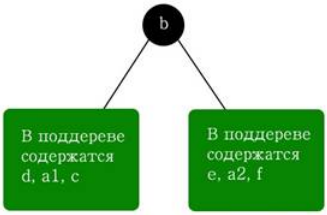

Рис. 5.38. Первый шаг построения BSP-дерева

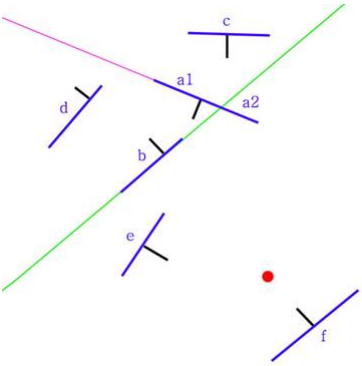

Рис. 5.39. Разделение сцены второй гиперплоскостью

Получаем структуру, представленную на рисунке 5.40.

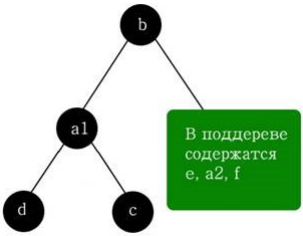

Рис. 5.40. Второй шаг построения BSP-дерева

Аналогичным образом поступаем с задним поддеревом узла **b**. Например, если выбрать за гиперплоскость сначала **a2**, а затем **e**, то получим разбиение исходной

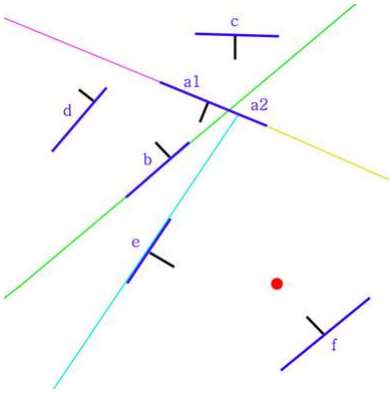

Рис. 5.41. Разбиение сцены

плоскости прямыми, как показано на рисунке 5.41 и в структуре на рисунке 5.42.

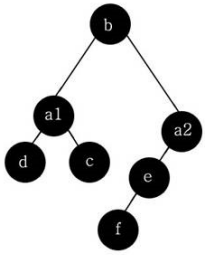

Рис. 5.42. Окончательное BSP-дерево

Итак, мы построили дерево двоичного разбиения пространства. Теперь можно, зная положение камеры, обойти все дерево по полигонам от самого дальнего до самого ближнего к камере. Начинается обход дерева, естественно, с его корня. Порядок обхода каждого узла задается положением камеры относительно прямой, соответствующей данному узлу, следующими правилами:

- Если камера находится в передней полуплоскости относительно прямой, соответствующей данному узлу, то обходим сначала заднее поддерево, потом все полигоны, которые находятся в данном узле, и в последнюю очередь переднее поддерево.
- Наоборот, если камера в задней полуплоскости, то обходим узел в порядке

от переднего поддерева к заднему.

- Если же камера находится на данной прямой, то сначала обходятся поддеревья в любом порядке, а полигоны самого узла не обходятся вовсе (т.к. они, фактически, не видны наблюдателю).