Государственное образовательное учреждение
высшего профессионального образования
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ЛЕСА»

С. В. Польский

КОМПЬЮТЕРНАЯ ГРАФИКА

Рекомендовано к изданию Редакционно-издательским советом
университета в качестве учебно-методического пособия
по выполнению расчётно-графической работы
для студентов специальности 230101 Вычислительные машины,
комплексы, системы и сети

Москва
Издательство Московского государственного университета леса
2008

УДК 681.3.06
ББК 32.973
П53

Рецензенты: профессор кафедры прикладной математики,
доктор физико-математических наук Е.В. Мышенков

профессор кафедры архитектурной графики,
кандидат технических наук А.С. Летин

Работа подготовлена на кафедре вычислительной техники

Польский, С. В.

П53 Компьютерная графика : учебн.-методич. пособие. – М. : ГОУ ВПО МГУЛ, 2008. – 38 с.

В работе описаны основные алгоритмы удаления невидимых линий и поверхностей. Рассмотрены особенности их практической реализации, при этом особое внимание уделено оценке их производительности и методам её повышения. Также приведено задание на выполнение расчётно-графической работы, посвящённой реализации и оценке производительности подобных алгоритмов.

УДК 681.3.06
ББК 32.973

ВВЕДЕНИЕ

Одной из основных задач компьютерной графики является визуализация трёхмерных сцен. Подобные задачи возникают в системах автоматизированного проектирования, пакетах моделирования физических процессов, средствах компьютерной анимации и виртуальной реальности.

При отображении трёхмерной сцены на экране некоторые из объектов сцены могут заслонить другие объекты. Заслонённые части объектов невидимы и не должны рисоваться, или должны рисоваться иначе, чем видимые части, например, пунктиром. Если этого не делать, то изображение будет выглядеть неправильно.

Такие задачи решают с помощью алгоритмов удаления невидимых линий и поверхностей. Если сцена отображается в каркасном виде, линиями, то нужно удалять невидимые линии. Каркасное изображение обычно строится из отрезков – рёбер, и алгоритм должен выделить части отрезков, заслонённых объектами сцены. Если объекты сцены отображаются в виде закрашенных поверхностей, то нужно удалять невидимые части этих поверхностей. Обычно в качестве поверхностей используются выпуклые многоугольники, чаще всего – треугольники.

В расчётно-графической работе необходимо реализовать один из алгоритмов удаления невидимых линий и поверхностей и произвести анализ его производительности. В соответствующих разделах приведены описания основных алгоритмов: Робертса, Варнака, Z-буфера, художника, трассировки лучей, построчного сканирования.

Все алгоритмы удаления невидимых линий и поверхностей можно разделить на две группы.

В одних, сначала определяется видимость для участков линий или поверхностей, а затем рисуются видимые части. К таким алгоритмам относится, например, алгоритм Робертса. В подобных алгоритмах необходимо сравнить каждый объект сцены (линию или поверхность) с остальными объектами, способными заслонить его.

В других, для каждого пиксела изображения определяется, какой из объектов сцены в нём виден. К таким алгоритмам относятся, например, алгоритмы трассировки лучей или Z-буфера. В этом случае нужно для каждого пиксела выбрать ближайший к наблюдателю объект сцены.

В обоих случаях требуется много вычислений, из-за того, что необходимо перебирать все объекты сцены. Для сокращения объёма вычислений используется свойство когерентности (англ. coherence – связность) расположенных рядом объектов. Можно выделить три вида когерентности.

Когерентность в картинной плоскости (плоскости экрана) – расположенные рядом пикселы, скорее всего, имеют одинаковые свойства, на-

пример, принадлежат одному и тому же объекту сцены, или видимы или, наоборот, заслонены одним и тем же объектом сцены.

Когерентность в объектном пространстве (пространстве сцены) – расположенные рядом объекты сцены, скорее всего, имеют одинаковые свойства, например, видимы или, наоборот, заслонены одним и тем же объектом сцены.

Когерентность во времени – при перемещении наблюдателя, на соседних кадрах будут видимы примерно одни и те же объекты сцены.

В некоторых алгоритмах когерентность используется явно. Например, в алгоритме Варнака картинная плоскость рекурсивно делится на области расположенных рядом пикселей, и, если вся область видима или, наоборот, заслонена, то дальнейшее её разделение не требуется. Другие алгоритмы можно модифицировать, используя свойство когерентности, существенно повышая их производительность. В разделах, посвящённых конкретным алгоритмам, подробно рассматриваются некоторые из способов модификации.

1. АЛГОРИТМЫ ПРЕДВАРИТЕЛЬНОГО УДАЛЕНИЯ НЕВИДИМЫХ ЛИНИЙ И ПОВЕРХНОСТЕЙ

1.1. Удаление невидимых объектов по пирамиде зрения

Прежде, чем удалять линии и поверхности, заслонённые объектами сцены, следует сначала отбросить все объекты, не попадающие в окно. Это можно сделать в картинной плоскости, отсекая спроецированные объекты границами окна (см. рис. 1), либо в объектном пространстве, отсекая их по плоскостям пирамиды зрения (см. рис. 2).

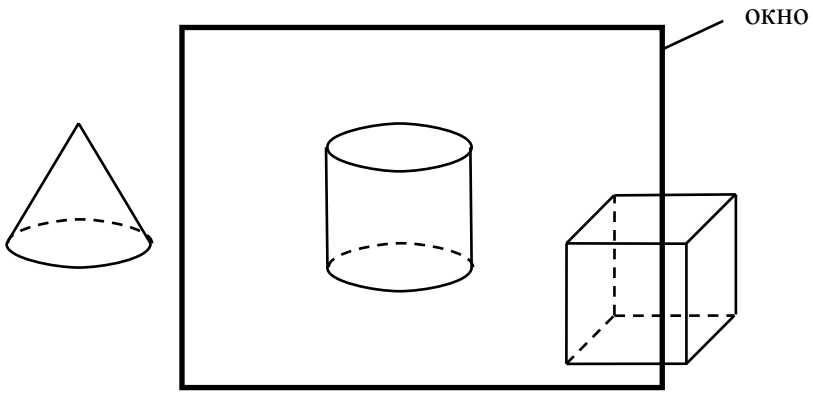


Рис. 1. Отсечение объектов по границам экрана

Пирамида зрения представляет собой объём пространства, который виден наблюдателю через окно. При перспективном проецировании она

ограничена плоскостями, построенными на точке наблюдателя и сторонах окна. Кроме того, она часто ограничивается ещё двумя плоскостями, параллельными окну – ближней и дальней [2].

Ближняя плоскость необходима для того, чтобы отсечь объекты находящиеся очень близко к наблюдателю и сзади него. Даже если основное отсечение происходит в картинной плоскости, без участия пирамиды зрения, отсечение ближней плоскостью необходимо, чтобы правильно спроецировать вершины объектов на картинную плоскость. Перспективное проецирование требует деления на координату глубины (z), и в случае, когда она отрицательная, проекция получается неверной, а когда нулевая - происходит ошибка деления на ноль.

Дальняя плоскость используется для сокращения числа отображаемых объектов, ей отсекаются объекты, расположенные дальше некоторого расстояния от наблюдателя. Чтобы не было видно резкой границы, на которой объекты исчезают, используются различные спецэффекты, например, туман, или плавный переход в прозрачность, но они выходят за рамки рассматриваемых алгоритмов удаления невидимых линий и поверхностей.

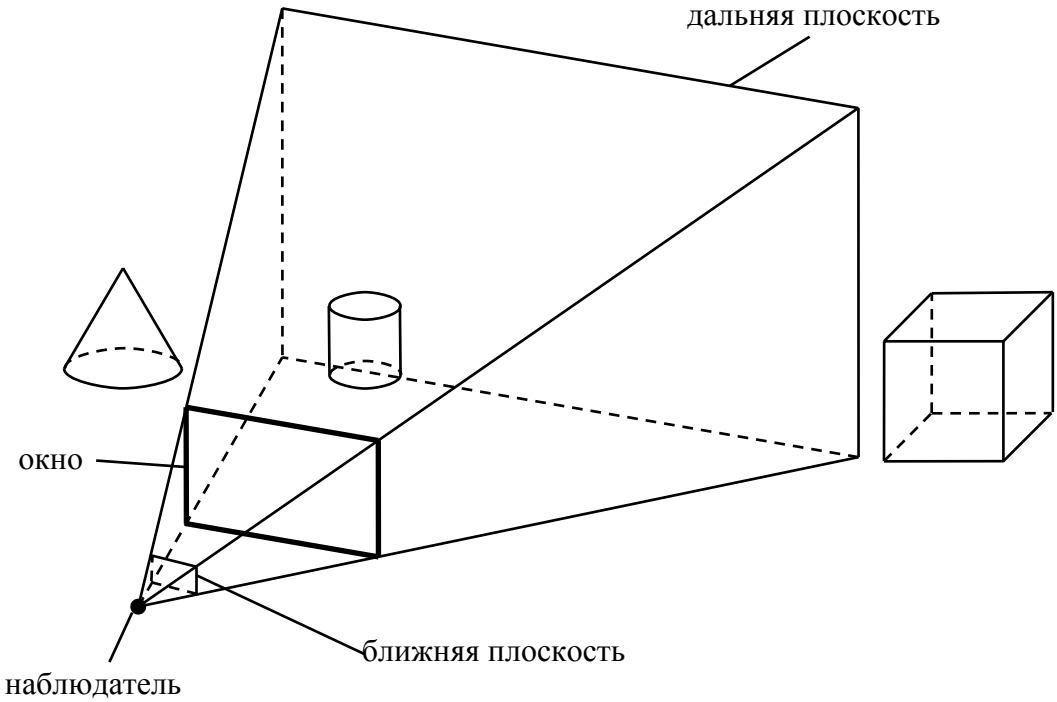


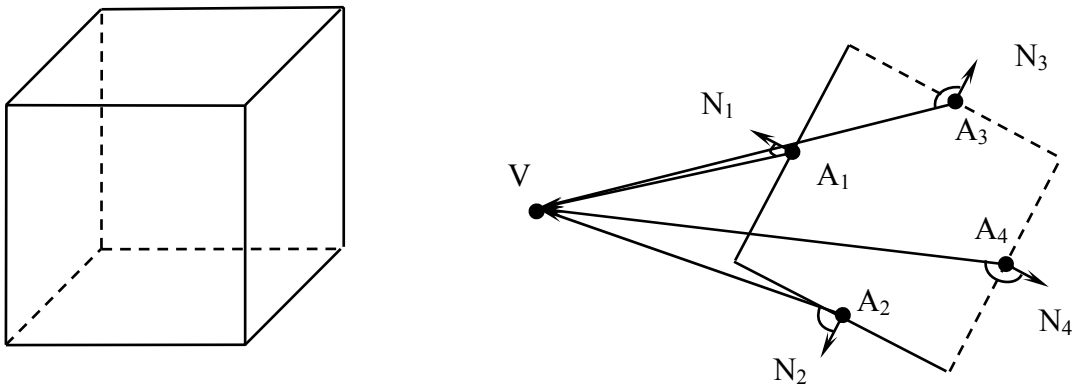
Рис. 2. Отсечение объектов по пирамиде зрения

Отсечение пирамидой зрения по сравнению с отсечением окном позволяет уменьшить количество вычислений связанных с проецированием объектов, так как часть объектов отсекается ещё до проецирования, но и требует больше элементарных операций. В пространстве сцены необходимо работать с трёхмерными векторами, тогда как при отсечении в картинной плоскости – с двухмерными.

Обычно по пирамиде зрения отсекают описанные вокруг объектов простые геометрические тела, например, сферы. И только, если сфера оказывается полностью или частично внутри пирамиды зрения, тогда соответствующий ей объект проецируется на картинную плоскость и его ребра и грани отсекаются по границам окна.

1.2. Удаление нелицевых граней

Если объект задан гранями, то часть этих граней будет заслонена самим объектом. Это в первую очередь те грани, нормали которых направлены в противоположную сторону от наблюдателя, угол между нормалью и вектором от грани к наблюдателю больше 90º. Такие грани называются нелицевыми. Грани, нормали которых направлены к наблюдателю, называются лицевыми гранями. На рисунке (см. рис. 3) слева показана проекция куба на картинную плоскость экрана, передняя, правая и верхняя грани лицевые, задняя, левая и нижняя – нелицевые. Справа показан тот же куб, но сверху. Видно, что для нелицевых (пунктирных) граней, угол между нормалью N и направлением на наблюдателя V больше 90º.



а) проекция в картинной плоскости б) вид сверху

Рис. 3. Лицевые и нелицевые грани объекта

Определить, является ли грань нелицевой можно по знаку скалярного произведения нормали $\overline{N}$ и вектора из любой точки на грани $\overline{A}$ на наблюдателя $\overline{V}$. Выражение $\overline{N} \cdot (\overline{V} - \overline{A}) < 0$ означает, что косинус угла между векторами отрицательный, и угол больше 90º, а значит грань нелицевая [1,3,4]. Нормаль $\overline{N}$ и точка на грани $\overline{A}$ задают плоскость грани, а знак выражения $\overline{N} \cdot (\overline{V} - \overline{A})$ говорит о том, с какой стороны от этой плоскости лежит точка наблюдателя $\overline{V}$, положительный – над плоскостью, и грань лицевая, отрицательный – под плоскостью, грань нелицевая, ноль – на плоскости, грань проецируется в отрезок.

По нелицевым граням можно также определить, какие рёбра будут невидимы, это рёбра, принадлежащие двум нелицевым граням. Если требуется отобразить один единственный выпуклый объект, то удаления нелицевых граней будет достаточно для получения правильного изображения. Если объектов несколько, или объект невыпуклый, то лицевые грани могут заслонять друг друга и требуются дополнительные алгоритмы для удаления невидимых частей сцены.

2. АЛГОРИТМ РОБЕРТСА

Лоренц Робертс, американский учёный, один из проектировщиков сети ARPAnet, впоследствии развившейся в Internet. Алгоритм Робертса [1,2,3,4] позволяет определить, какие рёбра или части рёбер объектов сцены видимы, а какие заслонены гранями других объектов. Объекты проецируются на картинную плоскость, и анализ видимости происходит на плоскости. Каждое ребро последовательно сравнивается со всеми гранями, обычно, выпуклыми многоугольниками. Могут быть следующие варианты их взаимного расположения:

- проекции ребра и грани не пересекаются, грань не заслоняет ребро (см. рис. 4а);
- проекции ребра и грани пересекаются, но ребро лежит ближе грани, грань не заслоняет ребро (см. рис. 4б);
- проекции ребра и грани пересекаются, и ребро лежит дальше грани, грань заслоняет всё ребро (см. рис. 4в) или часть ребра (см. рис. 4г, д);
- ребро пересекает грань (см. рис. 4е).

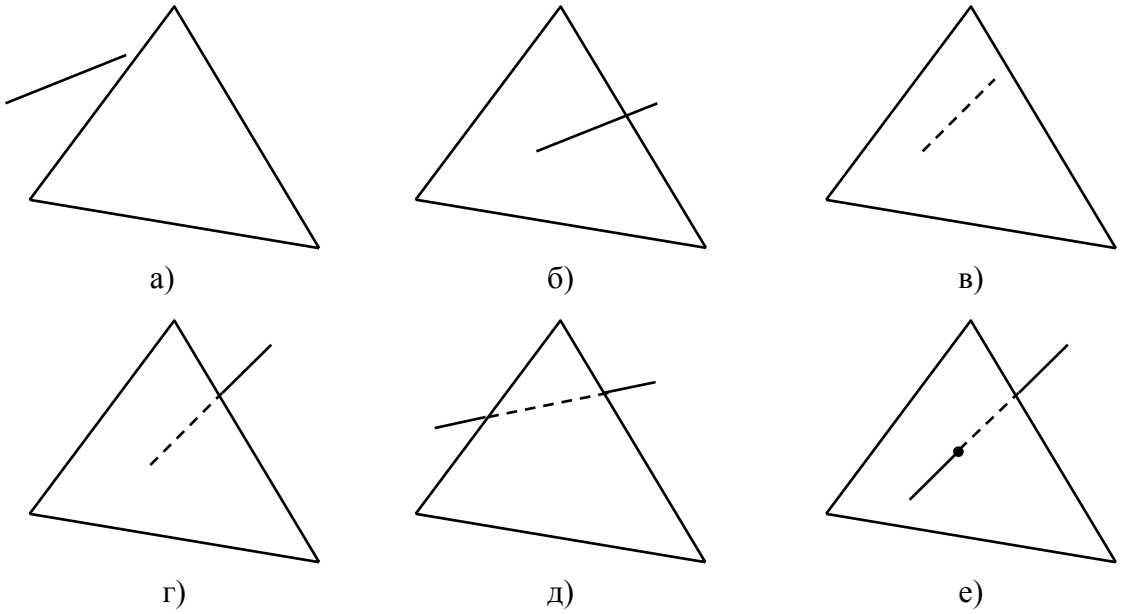


Рис. 4. Взаимное расположение ребра и грани

Если ребро не заслоняется гранью, то оно сравнивается со следующей гранью. Если ребро заслоняется полностью, то оно невидимо и срав-

нивать его со следующими гранями не нужно. Если грань заслоняет часть ребра, то нужно разделить отрезок на части, невидимую часть отбросить, а видимые сравнивать дальше с остальными гранями. Выпуклая грань может заслонить лишь одну часть отрезка, поэтому не заслонёнными останутся не более двух частей (см. рис. 4г, д, е).

Количество сравнений ребра с гранью можно оценить исходя из количества исходных граней N. Количество рёбер будет O(N), например, для треугольных граней будет максимум 3N рёбер. Так как требуется сравнивать каждое из рёбер с каждой гранью, то потребуется $O(N)\cdot N = O(N^2)$ сравнений. Из-за того, что грань может разделить ребро на части количество сравнений может быть больше. Так, в худшем случае, после проверки N-1 граней, если каждая из них закроет участок ребра, то ребро поделится на N частей, каждую из которых нужно будет сравнить со следующей N-ой гранью. Таким образом, для одного ребра в худшем случае потребуется $1+2+3+\ldots+N = O(N^2)$ сравнений, а для всех O(N) рёбер $O(N)\cdot O(N^2) = O(N^3)$ сравнений.

Производительность можно повысить, если не делить ребро после каждого сравнения, а запоминать невидимые интервалы ребра в отдельном массиве. После сравнения со всеми гранями, в этом массиве будет максимум столько же интервалов, сколько граней, т.е. O(N). Далее, границы интервалов можно отсортировать в порядке возрастания, и затем, перебрав границы в этом порядке, выделить видимые части ребра – интервалы отрезка, не совпадающие ни с одним из заслонённых интервалов. Например (см. рис. 5), отрезок AB заслоняют грани a,b,c,d. Каждая грань заслоняет соответствующий интервал отрезка $[a_1,a_2]$, $[b_1,b_2]$, $[c_1,c_2]$, $[d_1,d_2]$, всему отрезку соответствует интервал [0,1]. После сортировки границы интервалов расположатся следующим образом: 0, a_1, a_2, b_1, b_2, c_1, d_1, c_2, d_1, d_2, 1, а после перебора выделяются видимые интервалы: $[0, a_1]$, $[b_2, c_1]$, $[d_2, 1]$.

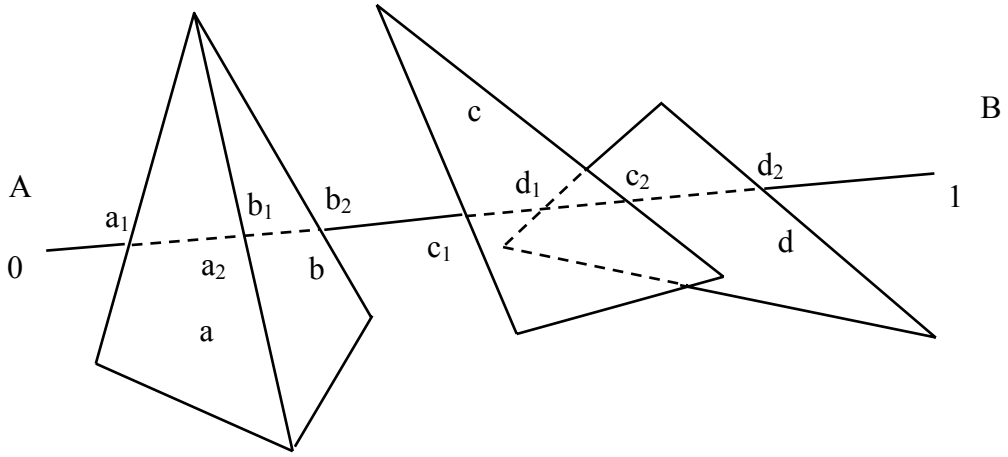
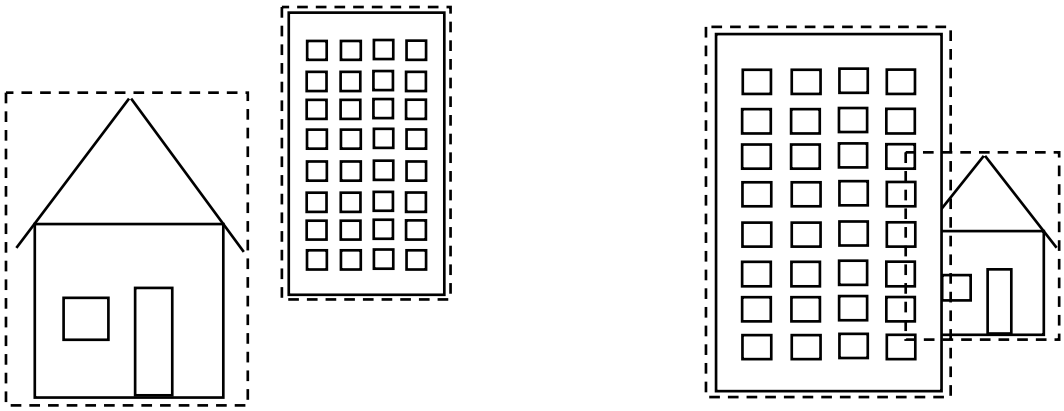


Рис. 5. Сортировка интервалов ребра, заслонённых гранями

Сортировка потребует времени $O(N\cdot\log N)$, таким образом, общее время для всех рёбер оценивается как $O(N)\cdot O(N\cdot\log N) = O(N^2\cdot\log N)$.

Также можно повысить производительность, если использовать когерентность в пространстве. Грани сцены редко бывают отдельными, обычно они составляют некоторые объекты (дома, машины, люди), состоящие из десятков-сотен граней, расположенных рядом. Вокруг таких



а) оболочки не пересекаются

б) оболочки пересекаются

Рис. 6. Сравнение описанных оболочек

объектов можно описать простые геометрические тела – оболочки. Например, прямоугольники на экране (см. рис. 6), и сначала сравнивать эти оболочки между собой. Если оболочки объектов не пересекаются (см. рис. 6а), то и грани одного объекта не могут заслонить рёбра другого, и нет необходимости сравнивать их между собой. Таким образом, достаточно простое сравнение оболочек позволяет избежать сравнения сотен рёбер одного объекта с сотнями граней другого. Если же оболочки пересекаются (см. рис. 6б), то этого сравнения не избежать.

Можно разделить грани объекта на более мелкие группы и описать вокруг них свои оболочки, таким образом построить некоторую иерархию оболочек. Сначала сравниваются оболочки вокруг целых объектов, если они пересекаются, то сравниваются оболочки вокруг их частей и так далее, только в самом конце сравниваются рёбра и грани. Такой иерархический подход позволяет сократить количество операций сравнения в ситуациях, когда объекты лишь частично закрывают друг друга.

3. АЛГОРИТМ ВАРНАКА

Джон Варнак, американский учёный, один из основателей компании Adobe Systems. Алгоритм Варнака [2,3,4] позволяет определить, какие грани или части граней объектов сцены видимы, а какие заслонены гранями других объектов. Так же как и в алгоритме Робертса анализ видимости происходит в картинной плоскости. В качестве граней обычно выступают выпуклые многоугольники, алгоритмы работы с ними эффективнее, чем с произвольными многоугольниками. Окно, в котором необходимо отобразить сцену, должно быть прямоугольным. Алгоритм работает рекурсивно,

на каждом шаге анализируется видимость граней и, если нельзя легко определить видимость, окно делится на 4 части и анализ повторяется отдельно для каждой из частей (см. рис. 7).

Случаи, когда можно легко определить видимость, следующие:

а) в окне нет ни одной грани, в этом случае ничего рисовать не нужно;

б) в окне ровно одна грань, в этом случае достаточно отобразить эту грань, так как нет других граней, которые могли бы её заслонить;

в) размеры окна равны размерам одного пиксела, в этом случае из граней выбирается ближайшая и её цветом закрашивается этот пиксел;

г) ближайшая грань охватывает всё окно, в этом случае она заслоняет все остальные грани и достаточно отобразить только её.

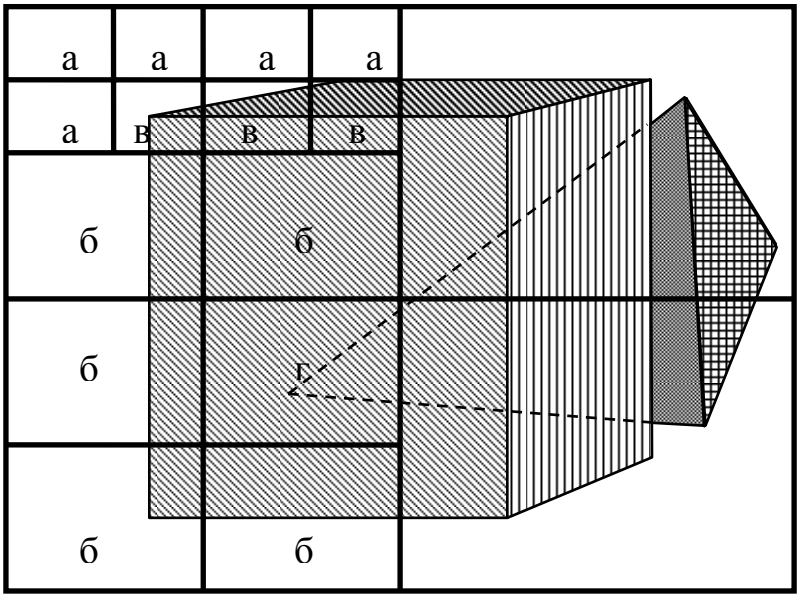


Рис. 7. Определение видимости алгоритмом Варнака

Алгоритм можно разделить на следующие элементарные задачи:

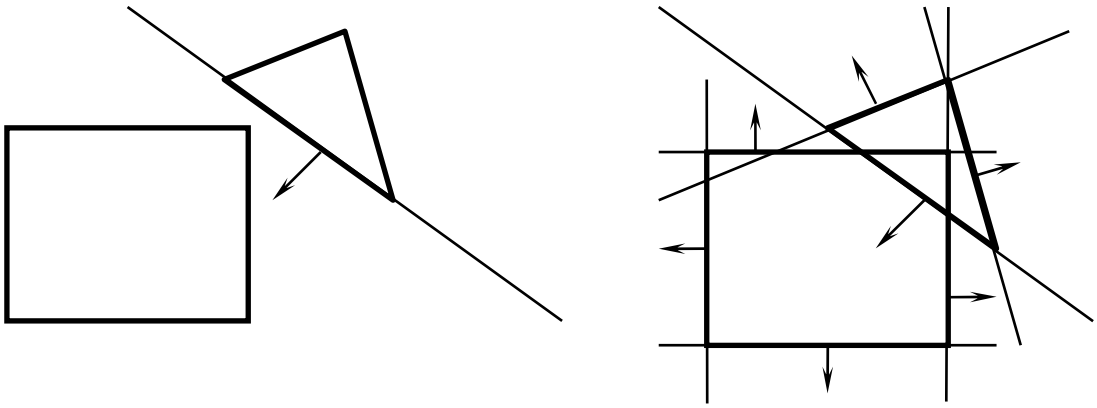
- определение, в какие части окна попадает грань;
- определение, что грань охватывает окно;
- определение, что грань является ближайшей.

Существует два различных подхода к решению этих задач. Сначала рассмотрим вариант решения первой задачи, когда при разделении окна грани разрезаются на части вертикальной и горизонтальной прямыми, делящими окно. Получившиеся многоугольники полностью лежат в соответствующих частях окна, и заносятся в соответствующий этой части набор. При последующем делении окна они разрезаются на более мелкие части. Определить, что такой многоугольник охватывает всё окно можно по условию, что каждая из четырёх вершин окна совпадает с одной из вершин многоугольника. Определить, что грань ближайшая можно по глубине вершин многоугольника. При перспективной проекции в качестве пара-

метра глубины удобно использовать обратную величину $w = \frac{1}{z}$, где z – координата глубины, так как w в отличие от z можно линейно интерполировать в экранных координатах. Чем больше w, тем ближе вершина к наблюдателю. Так как грань не обязательно параллельна экрану, то глубина у вершин разная. Найдя минимум и максимум глубины всех вершин можно определить интервал глубины [Wmin, Wmax]. В случае, когда окно равно пикселу, достаточно выбрать грань с максимальным Wmax, она и будет ближайшей. Если же нужно убедиться, что грань ближе всех остальных граней, то необходимо проверить условие, что Wmin этой грани больше Wmax всех остальных граней.

Второй подход не требует деления граней на части, для вычислений всегда используются исходные грани.

Для решения первой задачи нужно определить факт пересечения двух выпуклых многоугольников – грани и окна. Это можно сделать, например, по следующему условию: многоугольники не пересекаются, если все вершины одного из них лежат с внешней стороны от прямой, образующей ребро другого (см. рис. 8). Нужно перебрать все рёбра грани и сравнить с вершинами окна, и все четыре границы окна сравнить с вершинами грани.



а) есть разделяющая прямая, многоугольники не пересекаются

б) нет разделяющей прямой, многоугольники пересекаются

Рис. 8. Определение пересечения выпуклых многоугольников

Для решения второй задачи достаточно проверить, что все 4 угла окна лежат внутри или на границе грани. Это можно определить по расстояниям от углов окна до прямых, образующих рёбра граней.

Третья задача решается сложнее, чем в первом варианте. Грань не лежит полностью в окне, поэтому рассчитывать интервал её глубины по вершинам будет неправильно. Необходимы вершины той её части, которая лежит в окне. Это вершины грани, попадающие в окно, вершины окна, попадающие в грань и точки пересечения рёбер грани с рёбрами окна (см. рис. 9).

По этим точкам рассчитывается интервал глубины [Wmin, Wmax] и используется для определения ближайшей грани.

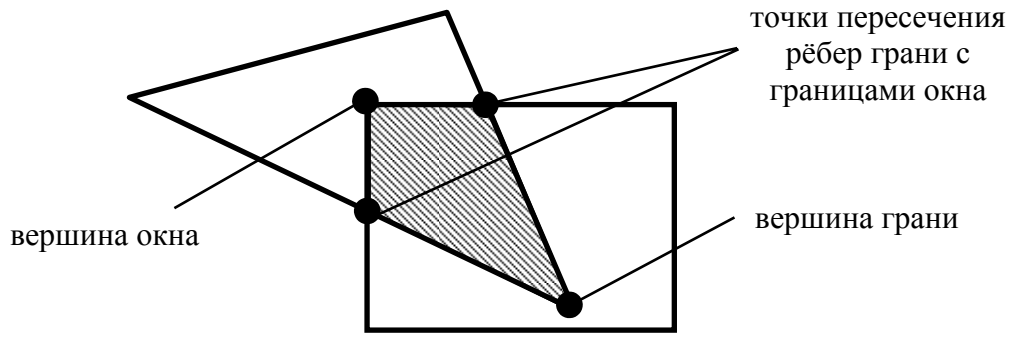


Рис. 9. Определение вершин пересечения грани с окном

У каждого из подходов есть свои достоинства и недостатки. Подход с разрезанием граней проще в реализации, но требует дополнительной памяти для хранения частей граней. Второй подход лишён этого недостатка, но более сложен в реализации. Производительность при обоих подходах примерно одинакова. Её можно оценить, исходя из количества граней N и количества пикселей C как O(CN), так как в худшем случае все грани могут быть размером с окно и окно в худшем случае придётся делить до пиксела. Уменьшить количество граней нельзя, но можно сократить количество делений окна. Например, его можно сократить, если дополнительно

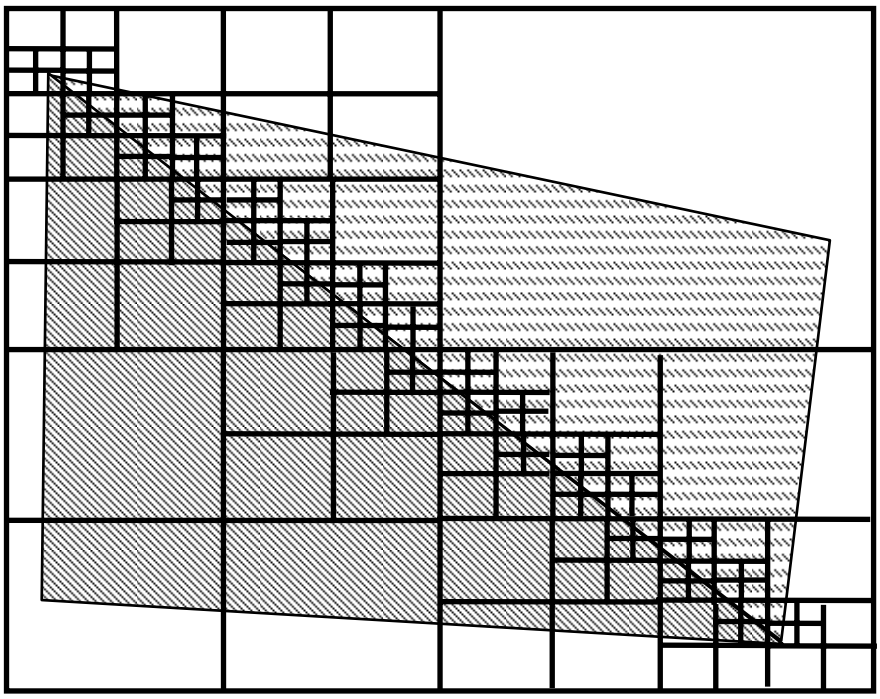


Рис. 10. Разделение окна вдоль общего ребра двух граней

обрабатывать случай на общем ребре двух граней. По описанному алгоритму окно, в которое попадает общее ребро двух граней, будет делиться

до пиксела. Это происходит потому, что не выполняется ни один из четырёх случаев, в окне несколько граней и ни одна из этих граней не охватывает окно (см. рис. 10).

Можно избежать лишних делений, если выделить общие рёбра граней, и добавить ещё один простой случай: две грани с общим ребром полностью охватывают окно и являются ближайшими. Ребро в этом случае должно пересекать окно в двух точках, вершины окна с одной стороны ребра должны лежать внутри грани лежащей с этой стороны ребра, а вершины окна с другой стороны ребра должны лежать внутри второй грани (см. рис. 11).

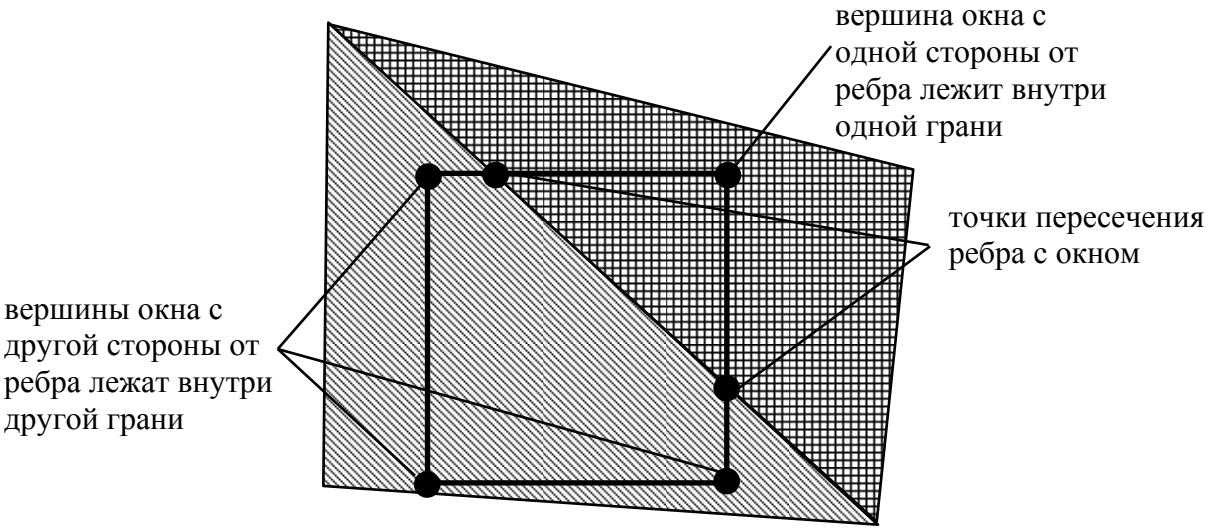


Рис. 11. Определение случая с двумя гранями с общим ребром

Интервал глубины [Wmin, Wmax] должен рассчитываться по вершинам обеих граней, попавших в окно частей.

4. АЛГОРИТМ Z-БУФЕРА

Алгоритм Z-буфера [2,3,4] позволяет определить, какие пикселы граней сцены видимы, а какие заслонены гранями других объектов. Z-буфер – это двухмерный массив, его размеры равны размерам окна, таким образом, каждому пикселу окна, соответствует ячейка Z-буфера. В этой ячейке хранится значение глубины пиксела (см. рис. 12). Перед растеризацией сцены Z-буфер заполняется значением, соответствующим максимальной глубине. В случае, когда глубина характеризуется значением w, максимальной глубине соответствует нулевое значение. Анализ видимости происходит при растеризации граней, для каждого пиксела рассчитывается глубина и сравнивается со значением в Z-буфере, если рисуемый пиксел ближе (его w больше значения в Z-буфере), то пиксел рисуется, а значение в Z-буфере заменяется его глубиной. Если пиксел дальше, то пиксел не ри-

суется и Z-буфер не изменяется, текущий пиксел дальше того, что нарисован ранее, а значит невидим.

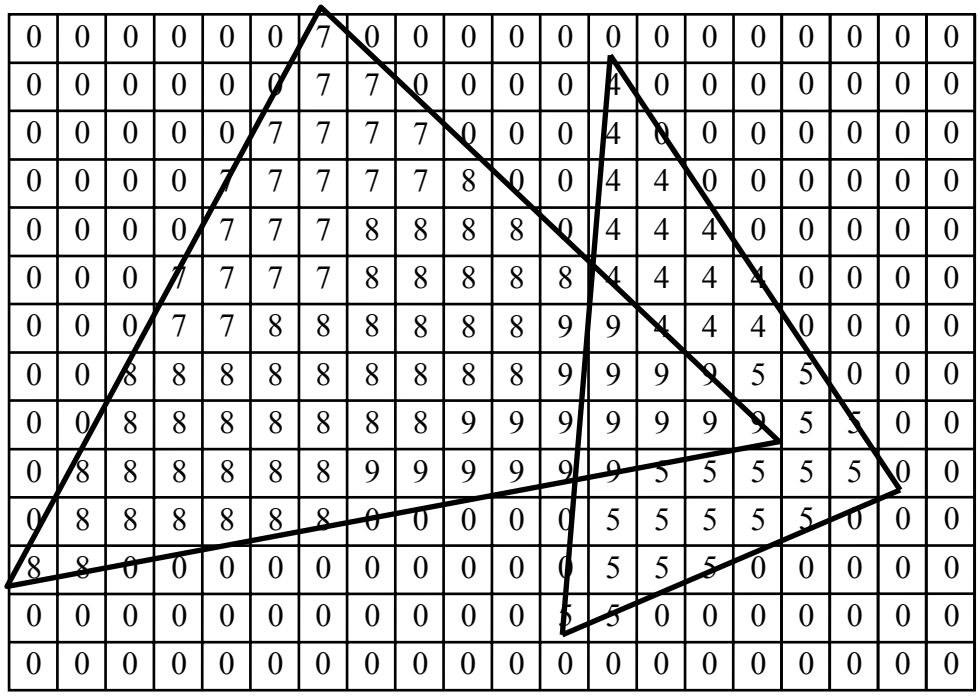


Рис. 12. Значения в z-буфере

Производительность можно оценить как O(CN), где N – количество граней, C – количество пикселей, так как в худшем случае все грани могут быть размером с окно. Производительность можно увеличить, если растеризовать грани в порядке от ближних к дальним. Тогда ближние, видимые, грани заполнят Z-буфер своими значениями глубины, и при растеризации дальних граней будет больше случаев, когда рисуемый пиксел дальше, чем нарисованный ранее, и пиксел не будет рисоваться, а глубина изменяться. Если грани рисуются одним цветом, то производительность возрастёт не сильно, но если на грань накладывается текстура или учитывается освещение, то рисование пиксела требует сложных вычислений, и такой метод может дать значительный прирост производительности.

5. АЛГОРИТМ ХУДОЖНИКА

Алгоритм художника [2,3,4] позволяет определить, какие пикселы граней сцены видимы, а какие заслонены гранями других объектов. Для этого все грани сортируются и рисуются в порядке от дальних к ближним, в результате чего ближние грани автоматически заслоняют дальние (см. рис. 13).

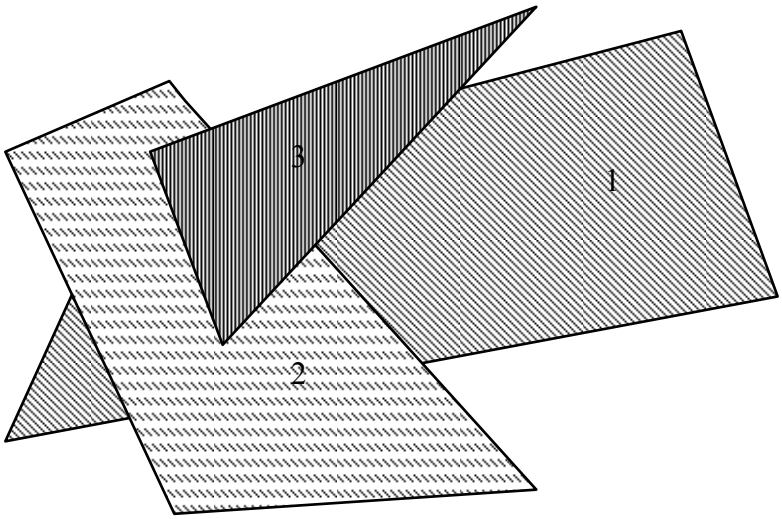
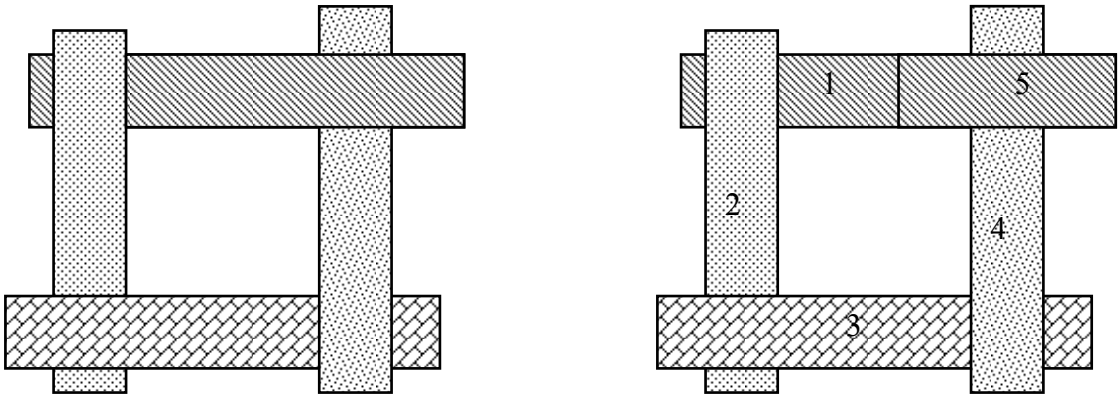


Рис. 13. Отрисовка граней по алгоритму художника

Сортировка граней от дальних к ближним задача нетривиальная, недостаточно отсортировать грани по координате, соответствующей глубине. Каждой грани обычно соответствует некоторый интервал глубины, и если интервалы двух граней пересекаются, то нужны дополнительные проверки, чтобы определить, какая грань ближе. Кроме того, встречаются ситуации, когда грани отсортировать невозможно, например, когда грани образуют цикл (см. рис. 14а). В этом случае нужно разделить одну из граней на части (см. рис. 14б), и тогда становится возможным определить порядок рисования.



а) невозможно определить порядок рисования

б) одна из граней разделена на части

Рис. 14. Циклически закрывающие друг друга грани

Одним из методов сортировки граней является метод бинарного разделения пространства. В качестве исходных данных выступает набор граней в пространстве и точка, в которой находится наблюдатель. Если разделить пространство плоскостью одной из граней, то грани, лежащие в полупространстве, в котором находится наблюдатель, будут ближе к нему, чем те, что лежат в другом полупространстве. Поэтому порядок рисования граней будет следующим:

- грани, лежащие в полупространстве, где нет наблюдателя;
- грани, лежащие на разделяющей плоскости;
- грани, лежащие в полупространстве, где находится наблюдатель.

Аналогично сортируются и грани в полупространствах, в каждом выбирается грань, её плоскостью полупространство делится на две части и так до тех пор, пока в каждой части не останется ни одной грани.

В качестве примера рассмотрим случай на плоскости (см. рис. 15), вместо граней – отрезки (a, b, c, d, e, f, g, h), вместо плоскостей – прямые (можно представить, что все грани вертикальные, а на рисунке показан их вид сверху).

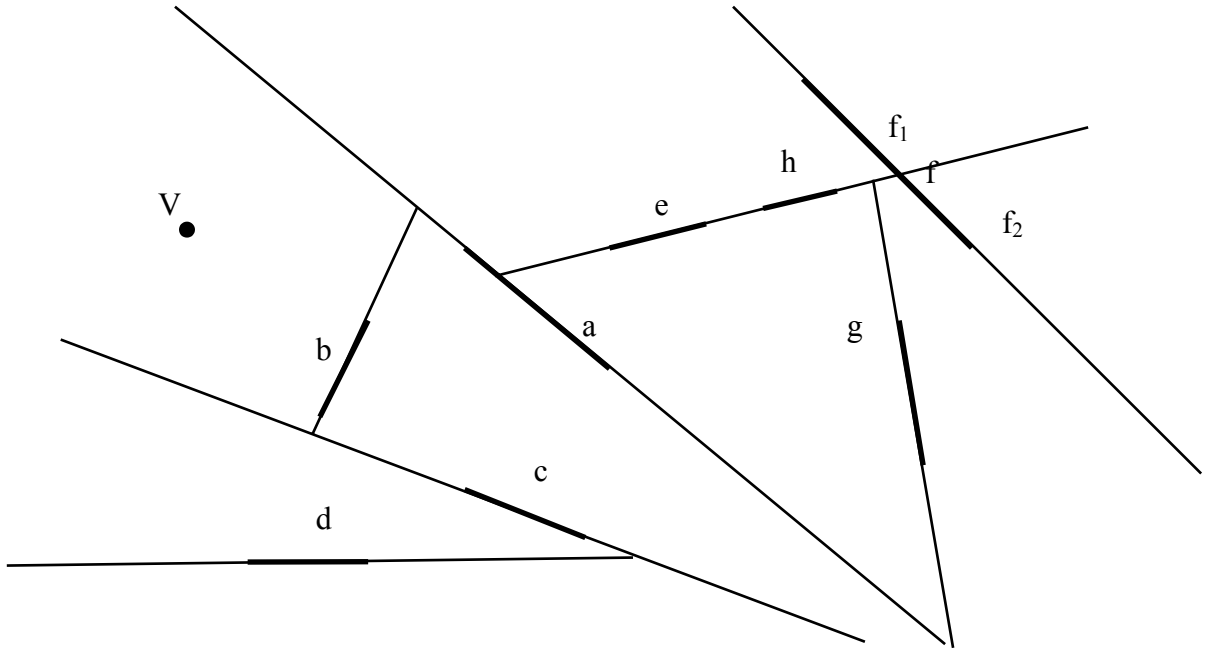


Рис. 15. Сортировка граней бинарным разделением пространства

Наблюдатель находится в точке V. Плоскость первой грани (a) делит грани на три группы: (b, c, d) – лежат в полупространстве наблюдателя, (e, g, h, f) – в другом полупространстве, грань (a) – на разделяющей плоскости. Порядок рисования (e, g, h, f), (a), (b, c, d). Грани в каждом из полупространств нужно отсортировать межу собой. В одном выбирается плоскость грани (e), она делит грани на группы: (f_1) – лежит в полупространстве наблюдателя, (f_2, g) – в другом полупространстве, грани (e, h) – на разделяющей плоскости. Грань (f) попала в оба полупространства и была разделена на две части (f_1 и f_2), грань (h) попала на разделяющую плоскость и

должна рисоваться вместе с гранью (e), их взаимный порядок рисования может быть любым, они не могут заслонить друг друга, так как лежат в одной плоскости. Аналогично, в части пространства с гранями (f_2, g) выбирается плоскость g, в части пространства с гранями (b, c, d) – грань (c). После всех разделений порядок рисования граней будет: f_2, g, e, h, f_1, a, d, c, b.

Время, необходимое на сортировку зависит от количества граней на всех этапах разделения пространства, например, при первом делении нужно сравнить с разделяющей плоскостью все N граней. Количество граней может возрасти из-за разделения граней на части, поэтому лучше выбирать такие плоскости, которые делят минимум граней. Также алгоритм будет работать эффективнее, если с каждой стороны от разделяющей плоскости будет примерно одинаковое количество граней. Рассмотрим два крайних случая, в первом – все грани оказываются с одной стороны от плоскости, кроме одной, плоскость которой является разделяющей. Тогда количество сравнений на первом шаге будет N, на втором N-1, на третьем N-2 и так далее до 1, общее количество делений будет N, а сравнений $O(N^2)$. Во втором случае – граней с обеих сторон поровну. Количество сравнений на первом шаге будет N, на втором N/2 + N/2 (с каждой стороны по N/2 граней), на третьем N/4 + N/4 + N/4 + N/4 и так далее до 1 + 1 + … + 1, общее количество делений будет logN, а сравнений $O(N \cdot \log N)$.

6. АЛГОРИТМ ТРАССИРОВКИ ЛУЧЕЙ

Алгоритм трассировки лучей [2,3,4] позволяет определить, какие пикселы граней сцены видимы, а какие заслонены гранями других объектов. Определение происходит непосредственно для каждого пиксела. Для этого ищется пересечение луча, исходящего из точки наблюдателя через центр пиксела, с гранями сцены и выбирается ближайшая к наблюдателю грань, она и будет видима в данном пикселе (см. рис. 16).

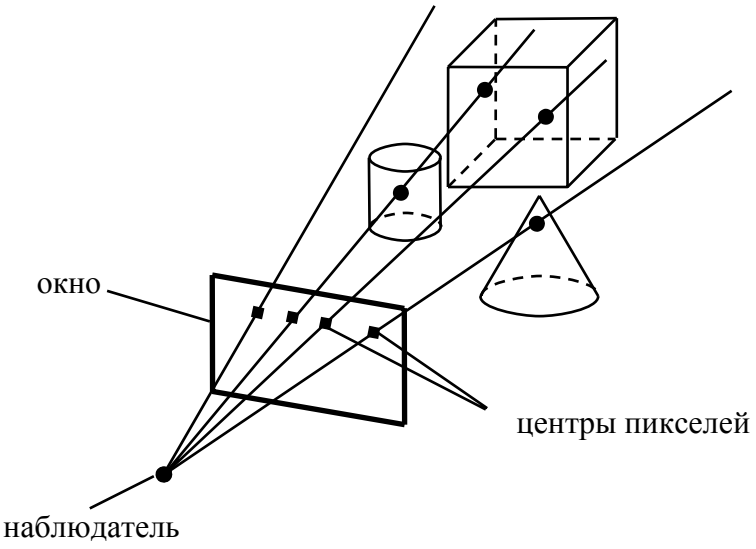
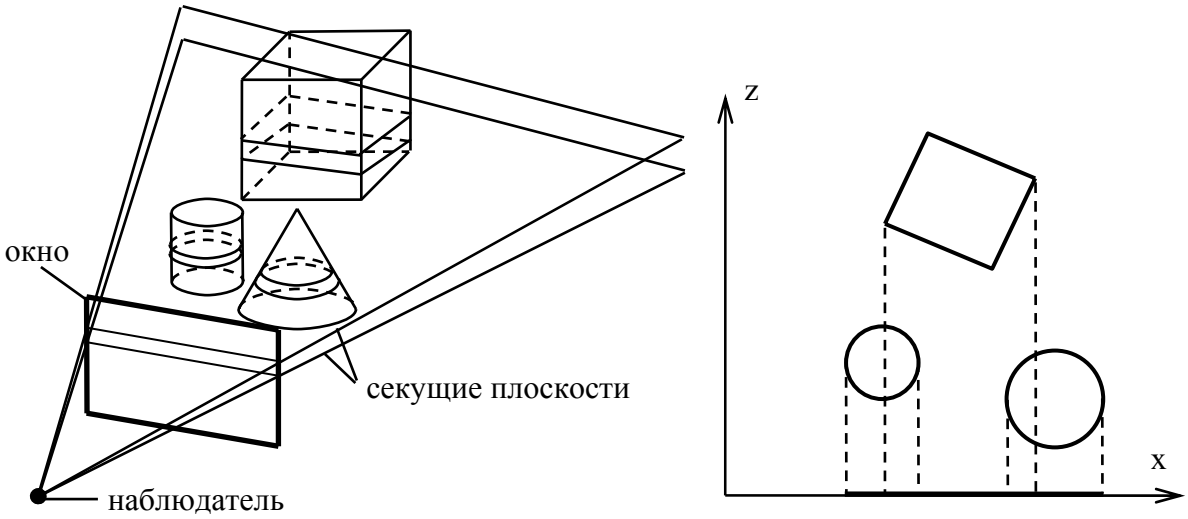


Рис. 16. Трассировка лучей

Скорость работы алгоритма можно оценить как O(CN), где N – количество граней, C – количество пикселей, так как необходимо луч для каждого пиксела проверить на пересечение с каждой из граней. Для повышения производительности можно воспользоваться когерентностью в пространстве. Грани обычно образуют поверхность объектов, то есть располагаются в пространстве компактными группами. Вокруг таких групп можно описать простые оболочки, например, сферы, и сначала проверять луч на пересечение с оболочкой, и только если он с ней пересекается, искать пересечение с отдельными гранями. Также можно использовать когерентность в картинной плоскости. Если для текущего пиксела была найдена ближайшая грань, то она скорее всего будет ближайшей и для соседних пикселей. Таким образом, для соседнего пиксела можно сначала проверить луч на пересечение с этой гранью. Если оно имеет место, то с остальными гранями достаточно проверять пересечение не всего луча, а только отрезка от наблюдателя до найденной точки пересечения. Это позволит быстрее отбросить грани или описанные тела, находящиеся дальше от наблюдателя, чем эта точка пересечения.

7. МЕТОДЫ ПОСТРОЧНОГО СКАНИРОВАНИЯ

Суть метода построчного сканирования [2,3,4] заключается в том, что через каждую строку пикселей окна проводится плоскость, дающая сечение сцены. При пересечении поверхностей объектов этими плоскостями образуются некоторые линии, например, при пересечении граней – отрезки, и задача видимости решается для этого набора отрезков (см. рис. 17). Таким образом, задача определения видимых участков граней в трёхмерном пространстве преобразуется в набор задач определения видимости отрезков на плоскости, которые решаются проще и эффективнее.



а) сечение в пространстве

б) сечение на плоскости

Рис. 17. Метод построчного сканирования

Пересечение плоскости и выпуклой грани является отрезком, на экране он отображается в виде горизонтального отрезка пикселей – спана.

7.1. Алгоритм построчного Z-буфера

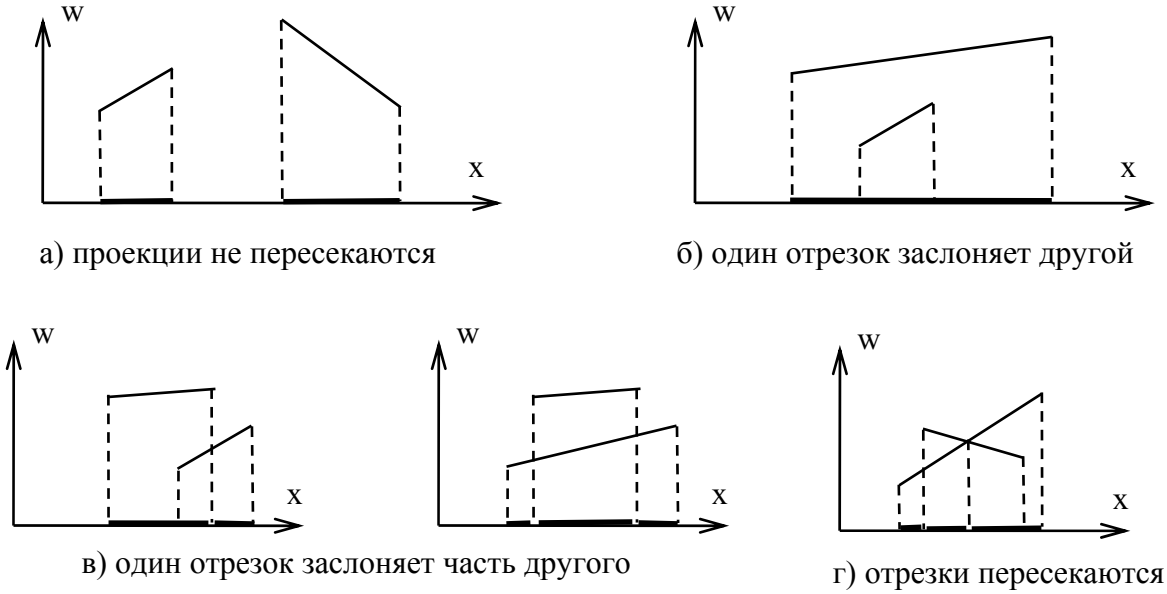
Алгоритм работает аналогично обычному Z-буферу (см. раздел 4), единственное отличие в том, что вместо двухмерного массива значений глубины, достаточно одномерного массива размером в строку пикселей. Каждому пикселу соответствует свой элемент массива, перед рисованием строки в него заносится максимальное значение глубины, а при рисовании пикселей спанов их глубина сравнивается с глубиной в Z-буфере и, если пиксел ближе, он рисуется, а его глубина заносится в Z-буфер. Построчный алгоритм более эффективен, чем полноэкранный, так как Z-буфер требует значительно меньшего объёма памяти и полностью умещается в кэше процессора. Чтобы ещё немного повысить производительность можно воспользоваться когерентностью соседних строк, грани видимые в одной строке, скорее всего, видимы и в соседних. Если рисовать сначала те грани, которые были видимы в предыдущей строке, а затем все остальные, то Z-буфер будет сразу заполняться наиболее близкими значениями глубины этих граней, а большинство пикселей остальных граней окажутся дальше, что позволит сократить количество операций рисования пикселей и модификации значений глубины в Z-буфере.

7.2. Алгоритм S-буфера

S-буфер – это буфер отрезков видимых наблюдателю (S – от segment, отрезок). Отрезки, полученные в результате пересечения граней плоскостью, поочерёдно заносятся в S-буфер. При этом очередной отрезок сначала сравнивается с отрезками буфера. Могут возникнуть следующие случаи:

- проекции отрезков не пересекаются (см. рис. 18а), ни один из отрезков не заслоняет другого;
- новый отрезок может полностью заслонить отрезок буфера (см. рис. 18б), в этом случае заслонённый отрезок удалятся из буфера;
- новый отрезок может заслонить часть отрезка буфера (см. рис. 18в), в этом случае отрезок буфера делится на части и заслонённая часть удаляется из буфера;
- отрезок из буфера может полностью заслонить новый отрезок (см. рис. 18б), в этом случае новый отрезок не виден и не заносится в буфер;
- отрезок из буфера может заслонить часть нового отрезка (см. рис. 18в), в этом случае новый отрезок делится на части, заслонённая часть удаляется, а оставшиеся сравниваются с остальными отрезками буфера;

- отрезки пересекаются (см. рис. 18г), в этом случае оба отрезка делятся на части, заслонённые части удаляются, оставшиеся части отрезка буфера заносятся обратно в буфер, оставшиеся части нового отрезка сравниваются с остальными отрезками буфера.



а) проекции не пересекаются

б) один отрезок заслоняет другой

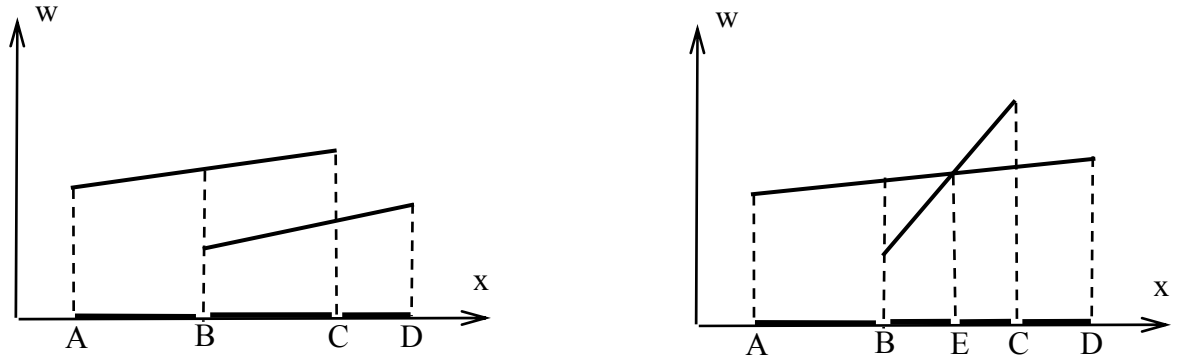
в) один отрезок заслоняет часть другого

г) отрезки пересекаются

Рис. 18. Сравнение двух отрезков

На рисунке 18 ось x – горизонтальная ось окна, ось w – параметр глубины, чем больше w, тем точка ближе к наблюдателю.

Определить, что проекции отрезков не пересекаются можно по условию, что максимальная координата x одного отрезка не превышает минимальную координату x другого. Когда проекции пересекаются, нужно определить, какие части отрезков заслонены (см. рис. 19). Если отсортировать точки концов отрезков (A, B, C, D) вдоль оси x, то можно выделить три интервала (A-B, B-C, C-D). На первом интервале видна часть отрезка, которому принадлежит точка A, на третьем интервале видна часть отрезка, которому принадлежит точка D.



а) один отрезок заслоняет часть другого

б) отрезки пересекаются

Рис. 19. Определение заслонённых частей отрезков

Чтобы определить, какие части видимы на втором интервале, нужно рассчитать глубину точек отрезков на его концах. Для двух точек она известна, так как точки являются концами отрезков, для двух оставшихся можно рассчитать, используя уравнение прямой $w = w1 + \frac{w2 - w1}{x2 - x1}(x - x1)$, где x1,w1,x2,w2 – координаты двух концов отрезка, x, w – координаты точки пересечения. Если глубина обеих точек одного отрезка больше чем у точек второго, то второй отрезок заслонён первым на этом интервале (см. рис. 19а). Если в начале интервала больше глубина одного отрезка, а в конце – другого, то отрезки пересекаются (см. рис. 19б). В этом случае необходимо найти точку пересечения отрезков, например, приравняв вышеприведённое уравнение прямой для прямых обоих отрезков. Интервал B-C поделится точкой пересечения E на два интервала (B-E, E-C). На первом будет виден отрезок, глубина которого в точке B больше, чем у другого, на втором – отрезок, глубина которого в точке C больше, чем у другого.

8. ВЫПОЛНЕНИЕ РАСЧЁТНО-ГРАФИЧЕСКОЙ РАБОТЫ

8.1. Задание на расчётно-графическую работу

Тема: оценка производительности алгоритмов удаления невидимых линий и поверхностей.

Цель: реализовать один из алгоритмов удаления невидимых линий и поверхностей и оценить его производительность при различных параметрах сцены и изображения.

Исходные данные:
- программа-каркас;
- файлы данных сцены;
- индивидуальные номера варианта алгоритма и варианта сцены.

Варианты алгоритмов:
1. Алгоритм Z-буфера;
2. Алгоритм построчного сканирования с одномерным Z-буфером;
3. Алгоритм построчного сканирования с S-буфером;
4. Алгоритм Робертса;
5. Алгоритм трассировки лучей;
6. Алгоритм Варнака;
7. Алгоритм художника.

Варианты сцены:
Файлы Data\scene_X.*, где X - номер варианта от 1 до 8.

8.2. Описание программы-каркаса

Программа-каркас выполнена на языке C++ и состоит из нескольких модулей.

Main.cpp, main.h – основной модуль, в котором сосредоточены функции, зависящие от операционной системы и компилятора. Он отвечает за запуск программы, вывод изображения на экран, подсчёт времени.

Geometry.cpp, geometry.h – модуль, в котором сосредоточены функции работы с геометрией – чтения данных сцены из файлов, преобразований между системами координат, отсечения, нахождения пересечений.

Raster.cpp, raster.h – модуль, в котором находятся функции растеризации отрезков и граней.

Bitmap.cpp, bitmap.h – модуль, содержащий функцию записи изображения в файл в формате bmp.

User.cpp – модуль, который содержит функции формирования изображения сцены.

List.h – шаблоны классов для работы с двухсвязными списками

Vector2D.h, vector3D.h – классы для работы с двухмерными и трёхмерными векторами

Имеются варианты программы для Visual C++ 6.0 (в каталоге VisionVC), для Borland C Builder 6.0 (в каталоге VisionBCB) и для Borland C++ 3.1 (в каталоге VisionBC). Для своей работы программа требует процессор Intel Pentium MMX или выше.

Сцена содержит следующие данные.

Количество и координаты точек (**NumberVertices** и **Vertices3D**) в системе координат камеры. Начало координат соответствует положению камеры, ось X – вправо, Y – вниз, Z – вперёд.

Количество и параметры ребёр (**NumberEdges** и **Edges**). Каждое ребро содержит два индекса точек и индекс цвета.

Количество и параметры граней (**NumberFaces** и **Faces**). Каждая грань – треугольник, содержит три индекса точек и индекс цвета.

Количество и параметры объектов (**NumberObjects** и **Objects**). Объект – это набор граней, образующих единую поверхность. Задаётся номером первой грани в массиве граней и количеством граней, грани одного объекта идут подряд.

Количество и значения цветов в формате RGB (**NumberColors** и **Colors**). Цвета нужны исключительно для отображения изображения и записи его в файл, во всех внутренних функциях используются только индексы цветов в этом массиве. Цвет с нулевым индексом – цвет фона, остальные – цвета рёбер и граней.

Размеры изображения зависят от установленного разрешения и находятся в переменных **RWidth** и **RHeight**.

Требуется изменить только файл **user.cpp**, все остальные файлы менять не следует. В этом модуле содержится основная функция **UserMain**, в которой вызываются функции чтения файлов данных сцены, формирования изображения, оценки времени его формирования и вывода изображения и данных о времени в файлы.

Время оценивается для 5-ти сцен, отличающихся количеством граней и загружаемых вызовами **LoadScene**("Data\\scene_X",i+1), где X – номер варианта, i+1 – номер сцены, а также для 5-ти разрешений экрана. В результате получается таблица из 25-ти значений, которая выводится в файл **results.txt**, пример приведён в таблице 1.

Таблица 1

Результаты работы программы

Количество пикселей в изображении	Количество граней в сцене				
	113	229	348	463	581
1749	112374	135622	221268	268727	337186
2560	152924	199238	357141	409090	533160
4000	168974	299221	456591	646327	772428
6996	263801	510177	813610	1045630	1309974
16000	584997	1169266	1833199	2453510	3045587

В таблице содержится время работы алгоритма при соответствующем количестве граней и пикселей. Единица измерения времени – одна тысяча тактов процессора.

Эту таблицу следует отобразить в виде набора графиков, отображающих зависимость времени работы алгоритма от количества граней в сцене (см. рис. 20). Каждый график соответствует определённому разрешению изображения – строке таблицы.

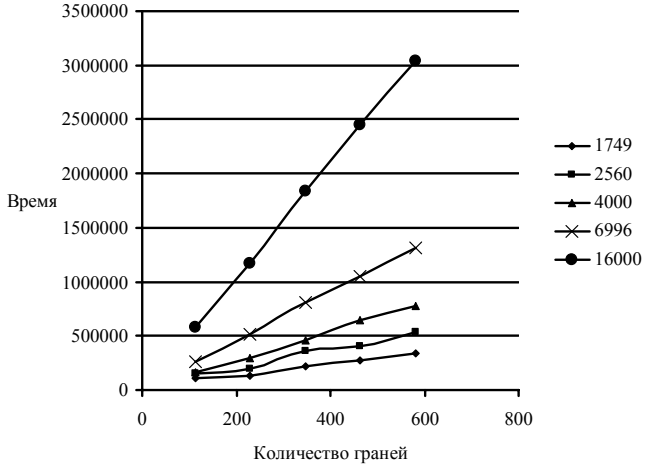


Рис. 20. График зависимости времени работы алгоритма от разрешения изображения и количества граней

Изображение сцены с максимальным количеством граней и максимальным разрешением выводится в файл **result.bmp**. Пример изображения приведён на рисунке 21.

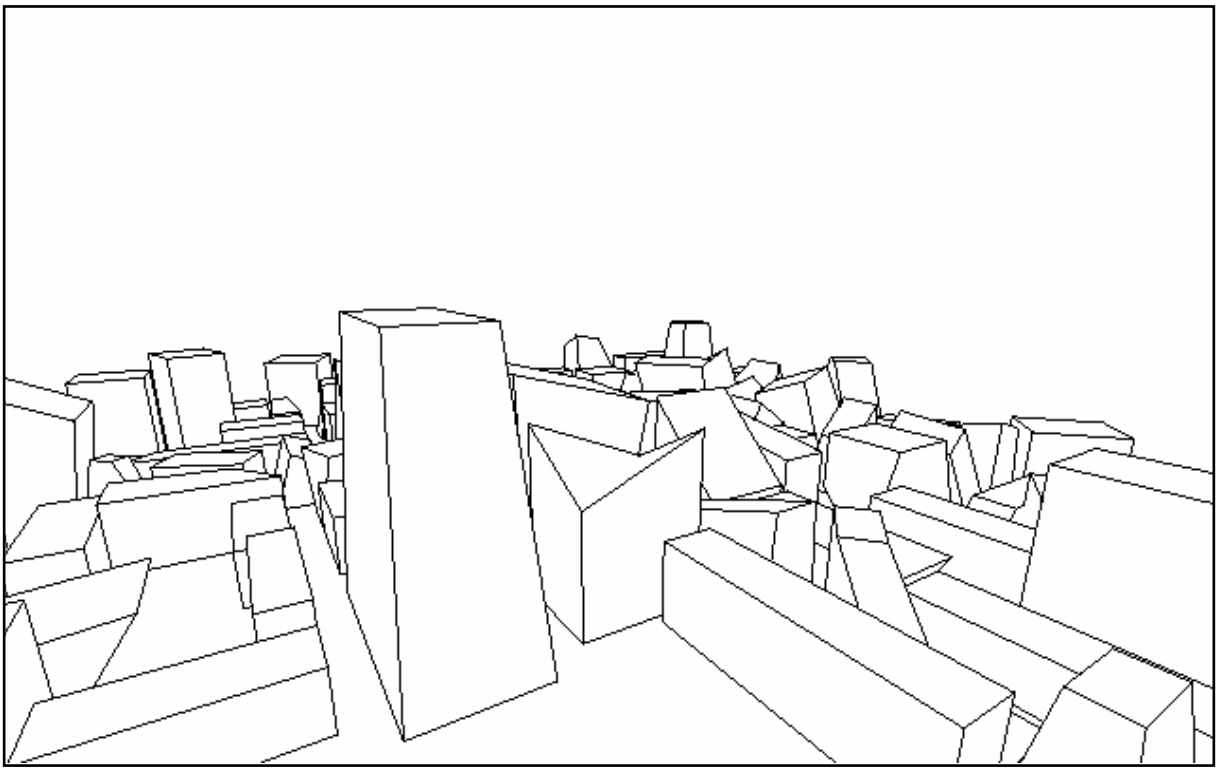


Рис. 21. Пример изображения сцены

Для отладки предусмотрена функция **UserTest**, в которой читается одна сцена и строится её изображение. Вызов функции **UserTest** находится в начале **UserMain**, по умолчанию он отключен.

В файле **user.cpp** есть несколько примеров функций построений сцены без удаления невидимых частей объектов – **DrawSceneSolid**, **DrawSceneWire** и **DrawSceneScan**. Их можно использовать в качестве шаблона для реализации конкретного алгоритма.

Функции **DrawSceneSolid** и **DrawSceneWire** работают по следующей схеме:

1. Изображение заполняется цветом фона.

2. Формируется примитив (грань или ребро).

3. Этот примитив отсекается по ближней плоскости, чтобы отсечь всё, что находится ближе, чем **MINIMUM_Z** по оси z от начала координат, где находится камера. Дальняя плоскость не используется, все точки сцены заведомо ближе константы **MAXIMUM_Z**.

4. Далее примитив проецируется на экран. Для получения экранных координат точек примитива используется функция **Project**, которая проецирует трёхмерную точку на экран. Точка в экранных координатах, кроме

координат x,y имеет также координату w, которая является линейной функцией от $\frac{1}{z}$. Диапазон её значений от 0 до 65535, для точек с глубиной от **MAXIMUM_Z**, при которой w = 0, до **MINIMUM_Z**, при которой w = 65535. Её можно использовать в качестве глубины в расчётах, чем больше w, тем ближе точка к наблюдателю.

5. Спроецированный примитив отсекается по четырём границам изображения. Граница изображения представляет собой прямоугольник с координатами левого верхнего угла (0,0), а правого нижнего (**RWidth**, **RHeight**).

6. Отсечённый примитив растеризуется.

Функция **DrawSceneScan** немного отличается, пункты 1-5 совпадают, а растеризация граней происходит не сразу, а с помощью алгоритма активных рёбер. После отсечения формируются рёбра граней и заносятся в списки тех строк изображения, в которых они начинаются. У каждой строки свой список рёбер, **LineEdges** – массив указателей на первые элементы списков. После занесения рёбер всех граней в списки в отдельном цикле по строкам изображения определяются и растеризуются спаны граней, которые пересекаются этой строкой.

8.3. Реализация алгоритмов на основе программы-каркаса

8.3.1. Алгоритм Z-буфера

В качестве шаблона можно использовать функции рисования сцены **DrawSceneSolid** и рисования грани **RasterPolygon**, добавив код проверки глубины пиксела по Z-буферу.

8.3.2. Алгоритм построчного сканирования с одномерным Z-буфером

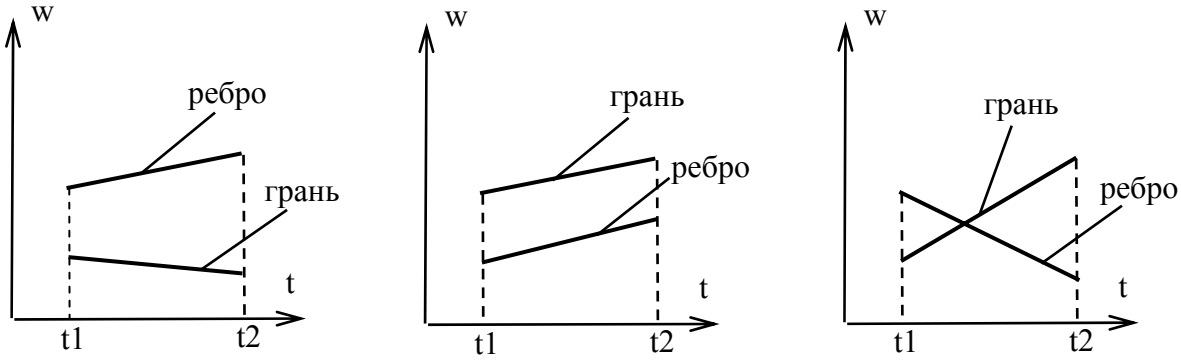
В качестве шаблона можно использовать функцию рисования сцены **DrawSceneScan**, добавив код проверки глубины пиксела по Z-буферу.

8.3.3. Алгоритм построчного сканирования с S-буфером

В качестве шаблона можно использовать функцию рисования сцены **DrawSceneScan**. Вместо рисования спана нужно заносить его в S-буфер, и только после того как все спаны строки будут в него занесены, нужно нарисовать содержащиеся в нём участки спанов.

8.3.4. Алгоритм Робертса

В качестве шаблона можно использовать функцию рисования сцены **DrawSceneWire**. Для проверки взаимного расположения ребра и грани можно использовать функцию **PartSegmentInsidePolygon**, которая определяет пересечение проекции грани и отрезка на плоскости, и если проекции пересекаются, возвращает интервал отрезка, на котором они пересекаются. Интервал возвращается в виде двух чисел (t1, t2) – значений параметра t на концах отрезка в параметрическом уравнении отрезка $\overline{P} = \overline{A} + (\overline{B} - \overline{A})t$, где $\overline{A}$ и $\overline{B}$ – вершины отрезка. Если проекции не пересекаются, то грань не заслоняет отрезок. Если пересекаются, то необходимо сравнить глубину отрезка и грани. Это можно сделать по методу, использованному в S-буфере для сравнения двух отрезков (см. раздел 7.2). Один отрезок – проекция ребра, второй – отрезок в плоскости грани, проекция которого совпадает с проекцией ребра. Достаточно рассмотреть только ту часть ребра, проекция которой совпадает с проекцией грани, так как только её может заслонить эта грань. Тогда достаточно рассчитать и сравнить глубину w для двух концов этих двух отрезков. Если обе вершины ребра ближе, чем обе вершины отрезка грани, то грань не заслоняет ребро (см. рис. 22а). Если обе вершины отрезка грани ближе вершин ребра, то грань заслоняет общую часть ребра (см. рис. 22б). Если для одной вершины ближе ребро, а для другой – грань, то грань и ребро пересекаются и грань заслоняет лишь часть ребра, от вершины, где она ближе, до точки пересечения (см. рис. 22в).



а) ребро ближе чем грань б) грань ближе чем ребро в) ребро пересекает грань

Рис. 22. Определение частей ребра, заслонённых гранью

Для вершин ребра глубину можно найти по параметрическому уравнению, Для грани – по уравнению плоскости грани ax+by+cw+d = 0, которую можно построить по любым трём вершинам грани.

8.3.5. Алгоритм трассировки лучей

Для определения направления луча, проходящего через центр пиксела можно использовать функцию **PixelDirection**, начальные точки всех лучей находятся в точке камеры – начале координат. Для определения пересечения луча с гранью можно использовать функцию **CheckRayTriangle**,

которая возвращает расстояние от начальной точки луча до точки пересечения, или отрицательное число, если луч не пересекает грань.

8.3.6. Алгоритм Варнака

В качестве шаблона можно использовать функцию рисования сцены **DrawSceneSolid**. Для разделения граней при делении окна на части можно использовать функцию отсечения многоугольника прямой **TruncPolygon2D**. Многоугольник должен быть представлен в виде циклического списка вершин. В результате выполнения функции остаётся его часть, лежащая под прямой – со стороны, противоположной направлению нормали. Если все вершины лежат над или на прямой, то функция возвращает значение **false** и пустой список вершин.

8.3.7. Алгоритм художника

В качестве шаблона можно использовать функцию рисования сцены **DrawSceneSolid**. Для разделения граней при двоичном разделении пространства можно использовать функцию отсечения многоугольника плоскостью **TruncPolygon3D**. Многоугольник должен быть представлен в виде циклического списка вершин, в результате выполнения функции остаётся его часть, лежащая под плоскостью – со стороны, противоположной направлению нормали. Если все вершины лежат над плоскостью или на плоскости, то функция возвращает значение **false** и пустой список вершин.

8.3.8. Пример реализации алгоритма

В приложении приведён пример текста модуля **user.cpp** с реализацией алгоритма блочного Z-буфера. Это одна из модификаций алгоритма Z-буфера, позволяющая уменьшить объём памяти, необходимой для хранения значений глубины, что в свою очередь позволяет повысить производительность, так как буфер небольшого объёма умещается в кэш процессора, и работа с ним происходит эффективнее. Окно, в котором необходимо построить изображение, разбивается на блоки одинакового размера, и для каждого блока формируется список граней, которые в него попадают. Каждый блок рисуется независимо от остальных с помощью алгоритма Z-буфера. При этом для хранения значений глубины достаточно массива с размером, соответствующим, количеству пикселей в одном блоке.

В приведённом тексте выделены строки, которые были изменены или добавлены в первоначальный текст модуля **user.cpp**, а неиспользуемые функции удалены. В качестве шаблона использована функция рисования сцены **DrawSceneSolid**, она переименована в **DrawSceneTileZBuffer**, и вызывается из функций **UserTest** и **UserMain**.

Сначала вычисляется количество блоков в окне и выделяется память для хранения списков полигонов, попавших в соответствующие блоки и значений Z-буфера для блока. Затем, все грани проецируются и заносятся в

списки блоков, пересекающихся с описанным вокруг проекции грани прямоугольником. Далее для каждого блока отдельно происходит формирование изображения. Сначала в Z-буфер заносится максимальная глубина, затем все грани из списка данного блока отсекаются по границам блока и рисуются функцией **RasterPolygonTileZBuffer**. Эта функция является модификацией функции рисования грани **RasterPolygon** программы-шаблона. В ней добавлены проверки на выход координат рисуемых пикселей за пределы блока, а также сравнение значения глубины со значением в Z-буфере и его модификация.

8.4. Требования к реализации алгоритма

Обязательные требования к реализации алгоритма:

- реализация должна соответствовать приведённому выше описанию алгоритма;
- реализация должна формировать правильное изображение – все невидимые части не видны, все видимые – видны;
- реализация должна освобождать все выделенные в ней блоки памяти;
- реализация не должна вызывать сообщений об ошибках.

Желательно применить какие-либо дополнительные методы для повышения производительности работы основного алгоритма, например, используя свойство когерентности. В этом случае следует реализовать две функции, одна – со стандартным алгоритмом, вторая – с улучшенным. И привести результаты по обеим, для каждой своя таблица и графики, чтобы можно было оценить улучшение.

8.5. Оформление отчета

В отчет входят:

- титульный лист;
- задание с индивидуальными номерами вариантов;
- подробное словесное описание алгоритма;
- список и краткое описание классов и функций программы-каркаса, использованных в реализации алгоритма;
- листинг модуля **user.cpp**, в котором реализован алгоритм;
- содержимое файла **results.txt**;
- изображение **result.bmp**;
- графики зависимостей времени работы алгоритма, построенные по данным файла **results.txt**.

ЗАКЛЮЧЕНИЕ

Основной целью написания данного пособия было снабдить студентов, изучающих компьютерную графику, полной и подробной информацией, достаточной для выполнения расчётно-графической работы. Главным отличием от других изданий является детальное описание алгоритмов удаления невидимых линий и поверхностей, позволяющее легко реализовать эти алгоритмы на практике. При описании алгоритмов особое внимание уделено производительности алгоритмов и способам её повышения.

Проблема производительности была и остаётся одной из основных проблем вычислительной техники, так как одновременно с ростом возможностей современных вычислительных систем возрастает и сложность решаемых задач. При выполнении расчётно-графической работы можно на практике оценить производительность реализованного алгоритма и, при необходимости, модифицировать алгоритм с целью её повышения. Подобный опыт позволит в будущем разрабатывать более эффективные алгоритмы, как в области компьютерной графики, так и в других областях.

Оглавление

Библиографический список

1. **Никулин, Е.А.** Компьютерная геометрия и алгоритмы машинной графики. – С.Пб: БХВ–Петербург, 2003. – 560с.

2. **Роджерс, Д.** Алгоритмические основы машинной графики.: Пер. с англ. – М.: Мир, 1989, 504 с.

3. **Шикин, Е.В., Боресков, А.В.** Компьютерная графика. Динамика, реалистические изображения. – М.: ДИАЛОГ–МИФИ, 1998. – 288 с.

4. **Шикин, Е.В., Боресков, А.В.** Компьютерная графика. Полигональные модели. – М.: ДИАЛОГ– МИФИ, 2001. – 464 с.

Приложение

Текст модуля **user.cpp** с реализацией алгоритма блочного Z-буфера

```
#include "main.h"
#include "raster.h"
#include "geometry.h"
#include "bitmap.h"
#include <stdio.h>

//--------------------------------------------------------------------
// Размеры блока
const long TileWidth = 32;
const long TileHeight = 32;

//--------------------------------------------------------------------
// Отображение полигона с проверкой глубины по блочному Z-буферу
// in_Poly – список вершин полигона
// in_color – цвет полигона
// in_tile_offset_x, in_tile_offset_y – смещение блока в окне
// in_zbuffer – массив Z-буфера
void RasterPolygonTileZBuffer(
    TCircleList<TVertex2D> &in_Poly,
    short in_color,
    long in_tile_offset_x,
    long in_tile_offset_y,
    float *in_zbuffer)
{
    TEdgeList *Edge1=0,*Edge2=0;
    // Занести данные рёбер полигона в списки строк растра
    long ys,ye;
    PrepareRasterPolygon(in_Poly,0,ys,ye);
    for(long y=ys;y<ye;y++)
    {
        //проверка, что рёбра завершились
        if (Edge1 && (y>=Edge1->end))    SafeDelete(Edge1);
        if (Edge2 && (y>=Edge2->end))    SafeDelete(Edge2);
        //обновление координат для очередной строки
        if (Edge1) { Edge1->x+=Edge1->dx; Edge1->w+=Edge1->dw; }
        if (Edge2) { Edge2->x+=Edge2->dx; Edge2->w+=Edge2->dw; }
        //добавление новых рёбер
        while(LineEdges[y])
```

```
{
    if (!Edge1)          Edge1=LineEdges[y];
    else if (!Edge2)     Edge2=LineEdges[y];
    elseassert(0,"Больше двух рёбер на строку");
    LineEdges[y]=LineEdges[y]->next;
}
assert(Edge1 && Edge2,"Меньше двух рёбер на строку");
if (!Edge1 || !Edge2) continue;
// Если спан не попадает в блок - не рисуем его
if (y < in_tile_offset_y)        continue;
if (in_tile_offset_y + TileHeight < y)      continue;
// Edge1 должно быть всегда слева от Edge2
if (Edge1->x>Edge2->x)       Swap(Edge1,Edge2);
// Начало и конец спана
long xs=(long)floor(Edge1->x+0.5);
long xe=(long)floor(Edge2->x+0.5);
// Проверка на выход за пределы растра
if (xs<0)   xs=0;
if (xe>RWidth)   xe=RWidth;
// Если спан выходит за границы блока - ограничиваем его
if (xs<in_tile_offset_x)        xs=in_tile_offset_x;
if (xe>in_tile_offset_x + TileWidth)
    xe=in_tile_offset_x + TileWidth;
// Спан вне растра
if (xs>=xe)         continue;
//рисуем спан
float dw=(Edge2->w-Edge1->w)/(Edge2->x-Edge1->x);
float w=Edge1->w+dw*(xs+0.5f-Edge1->x);
for(long x=xs;x<xe;x++)
{
    // сравниваем глубину со значением в z-буфере
    long zbuffer_offset =     (y - in_tile_offset_y) * TileWidth +
                              (x - in_tile_offset_x);
    if (in_zbuffer[zbuffer_offset] < w)
    {
        // заносим глубину
        in_zbuffer[zbuffer_offset] = w;
        // рисуем пиксел
        PutPixel(x,y,in_color);
    }
    //обновляем w-координату для следующего пиксела
    w+=dw;
}
```

```
    }
    SafeDelete(Edge1);
    SafeDelete(Edge2);
}

//---------------------------------------------------------------------
// Структура для хранения данных о проекции полигона
struct TPolygonInfo
{
    TCircleList<TVertex2D> Corners;
    short Color;
};

//---------------------------------------------------------------------
// Нарисовать сцену с использованием блочного Z-Буфера
void DrawSceneTileZBuffer()
{
    // Очистить изображение и все дополнительные данные в начале кадра
    InitRasterFrame();
    // Плоскость отсечения полигонов за камерой (ближняя плоскость)
    TPlane plane(0,0,-1,-MINIMUM_Z);

    // Количество блоков в окне
    long tile_size_x = (RWidth + TileWidth - 1) / TileWidth;
    long tile_size_y = (RHeight + TileHeight - 1) / TileHeight;
    // Списки полигонов, попавших в блоки
    TCircleList<TPolygonInfo> *tile_polygones =
        new TCircleList<TPolygonInfo>[tile_size_x * tile_size_y];
    // Список всех спроецированых полигонов
    TCircleList<TPolygonInfo> projected_polygones;
    // Массив Z-буфера
    float ZBuffer[TileWidth * TileHeight];
    // Заносим все грани в списки соответствующих блоков
    for(long i=0;i<NumberFaces;i++)
    {
        const TFace &face=Faces[i];
        // Формируем полигон в пространстве
        TCircleList<TVertex3D> Poly3D;
        Poly3D.append(new TVertex3D(Vertices3D[face.vertex[0]]));
        Poly3D.append(new TVertex3D(Vertices3D[face.vertex[1]]));
        Poly3D.append(new TVertex3D(Vertices3D[face.vertex[2]]));
```

```
    // Отсекаем по ближней плоскости
    if (TruncPolygon3D(Poly3D,plane))
    {
        // Создаём объект для хранения полигона
        TPolygonInfo *Poly2D = new TPolygonInfo();
        // Заносим цвет полигона
        Poly2D->Color = face.color_id;
        // Проецируем полигон на экран
        ProjectPolygon(Poly3D, Poly2D->Corners);
        // Заносим полигон в список, чтобы в конце удалить
        projected_polygones.append(Poly2D);

        // Рассчитываем координаты прямоугольника,
        // описанного вокруг полигона
        TVector2D min_rect(1e20f, 1e20f), max_rect(-1e20f);
        for(TCircleListIterator<TVertex2D> j(Poly2D->Corners);
            !j.end();j.step())
        {
            const TVertex2D *vertex = j.get();
            if (vertex->x < min_rect.x)        min_rect.x = vertex->x;
            if (vertex->x > max_rect.x)        max_rect.x = vertex->x;
            if (vertex->y < min_rect.y)        min_rect.y = vertex->y;
            if (vertex->y > max_rect.y)        max_rect.y = vertex->y;
        }

        // Заносим полигон в списки соответствующих блоков
        long tile_min_x = Max((long)floor(min_rect.x / TileWidth), 0L);
        long tile_max_x = Min((long)floor(max_rect.x / TileWidth),
                                        tile_size_x - 1L);
        long tile_min_y = Max((long)floor(min_rect.y / TileHeight), 0L);
        long tile_max_y = Min((long)floor(max_rect.y / TileHeight),
                                        tile_size_y - 1L);
        for(long tile_y=tile_min_y;tile_y<=tile_max_y;++tile_y)
            for(long tile_x=tile_min_x;tile_x<=tile_max_x;++tile_x)
              tile_polygones[tile_y*tile_size_x+tile_x].append(Poly2D);
    }
}


// Для каждого блока
for(long tile_y = 0; tile_y < tile_size_y; ++tile_y)
    for(long tile_x = 0; tile_x < tile_size_x; ++tile_x)
    {
```

```
            // Границы блока
            long tile_min_x = tile_x * TileWidth;
            long tile_max_x = Min((tile_x + 1) * TileWidth, RWidth);
            long tile_min_y = tile_y * TileHeight;
            long tile_max_y = Min((tile_y + 1) * TileHeight, RHeight);
            // Прямые для отсечения по границам блока
            TLine left(-1,0,-tile_min_x), right(1,0,tile_max_x);
            TLine top(0,-1,-tile_min_y), bottom(0,1,tile_max_y);
            // Заносим в Z-буфер максимальную глубину
            for(long i = 0; i < TileWidth * TileHeight; ++i)
                ZBuffer[i] = 0.0f;
            // Рисуем все полигоны, попавшие в список блока
            TCircleList<TPolygonInfo> &list =
                tile_polygones[tile_y * tile_size_x + tile_x];
            // Пока в списке есть полигоны
            while(list.start())
            {
                // Данные о полигоне
                const TPolygonInfo *polygon = list.start()->Object;
                // Делаем копию списка вершин полигона
                TCircleList<TVertex2D> CellCorners;
                CellCorners.copy(polygon->Corners);
                // Отсекаем полигон по границам блока
                if (    TruncPolygon2D(CellCorners,left)
                   &&   TruncPolygon2D(CellCorners,right)
                   &&   TruncPolygon2D(CellCorners,top)
                   &&   TruncPolygon2D(CellCorners,bottom)    )
                   {
                        // Растеризуем полигон
                        RasterPolygonTileZBuffer(
                              CellCorners, polygon->Color,
                              tile_min_x, tile_min_y, ZBuffer);
                   }
                // Удаляем полигон из списка
                delete list.remove(list.start());
            }
        }
    // Удаляем списки блоков
    delete[] tile_polygones;

    //Удаляем все спроецированные полигоны
    projected_polygones.clear();
}
```

```
//---------------------------------------------------------------------
// Пользовательская тестовая функция
void UserTest()
{
    //читаем вариант scene_X, X - номер сцены из задания
    if (!LoadScene("Data\\scene_1",5))       return;

    // Установить разрешение изображения
    SetResolutionFactor(0);

    // рисуем сцену
    DrawSceneTileZBuffer();

    SaveBMP("result.bmp",ImageArray,Colors,RWidth,RHeight);

    // выводим изображение на экран
    UpdateView(true);

    // очищаем данные сцены
    ClearScene();
}

//---------------------------------------------------------------------
// Кол-во вариантов сцены
#define NUMBER_SCENES            5
// Кол-во вариантов разрешения растра
#define NUMBER_RESOLUTIONS 5

//---------------------------------------------------------------------
// Пользовательская функция
void UserMain()
{
    // Пользовательская тестовая функция
//  UserTest(); return;
    // времена рисования сцены в различных вариантах
    double time[NUMBER_SCENES][NUMBER_RESOLUTIONS];
    // кол-во пикселов в заданном варианте
    long number_pixels[NUMBER_RESOLUTIONS];
    // кол-во полигонов в заданном варианте
    long number_polygones[NUMBER_SCENES];
    for(int i=0;i<NUMBER_SCENES;i++)
    {
```

```
        //читаем вариант scene_X, X - номер сцены из задания
        if (!LoadScene("Data\\scene_1",i+1))       return;
        number_polygones[i]=NumberFaces;
        for(int j=0;j<NUMBER_RESOLUTIONS;j++)
        {
            // Установить разрешение изображения
            SetResolutionFactor(NUMBER_RESOLUTIONS+1-j);
            number_pixels[j]=RWidth*RHeight;
            double takts=GetTakts();
            // рисуем сцену
            DrawSceneTileZBuffer();
            // определяем время отрисовки сцены
            time[i][j]=GetTakts()-takts;
            // выводим изображение на экран
            UpdateView(true);
        }
        // изображение последнего варианта сцены сохраняем в файле
        if (i==(NUMBER_SCENES-1))
            SaveBMP("result.bmp",ImageArray,Colors,RWidth,RHeight);
        // очищаем данные сцены
        ClearScene();
    }
    // записываем файл времён
    FILE *file = fopen("results.txt","wt");
    assert(file,"Ошибка при открытии файла");
    if (file)
    {
        fprintf(file,"Pixels\\Polygones ");
        for(int i=0;i<NUMBER_SCENES;i++)
            fprintf(file,"    %5d",number_polygones[i]);
        fprintf(file,"\n");
        for(int j=0;j<NUMBER_RESOLUTIONS;j++)
        {
            fprintf(file," %5d          ",number_pixels[j]);
            for(int i=0;i<NUMBER_SCENES;i++)
            {
                fprintf(file," %8d",long(time[i][j]/1000));
            }
            fprintf(file,"\n");
        }
    }
    fclose(file);
}
```

Учебное издание

Польский Сергей Владимирович

Компьютерная графика. Учебно-методическое пособие по выполнению расчётно-графической работы

Редактор И.И. Кожемяко
Компьютерный набор и верстка С. В. Польского

По тематическому плану внутривузовских изданий учебной литературы на 2008 г., доп.

Подписано в печать 06.03.2008. Формат 60×90 1/16. Бумага 80 г/м^2
Гарнитура «Таймс». Ризография. Усл. печ. л. 1,5.
Тираж _____ экз. Заказ № _____.

Издательство Московского государственного университета леса.
141005, Мытищи-5, Московская обл., 1-я Институтская, 1, МГУЛ.
E-mail: izdat@mgul.ac.ru