Кафедра Систем Искусственного Интеллекта

КОМПЬЮТЕРНАЯ ГЕОМЕТРИЯ И ГРАФИКА

Методические указания для выполнения лабораторных работ

Сиротин Э.Е. Перфильев Д.А. Кушнаренко А.В.
Руководитель проекта: Сарафанов А.В.

Лабораторная работа №1.

Исследование алгоритмов генерации отрезков

Цель: анализ различных алгоритмов генерации отрезков на прямоугольном растре.

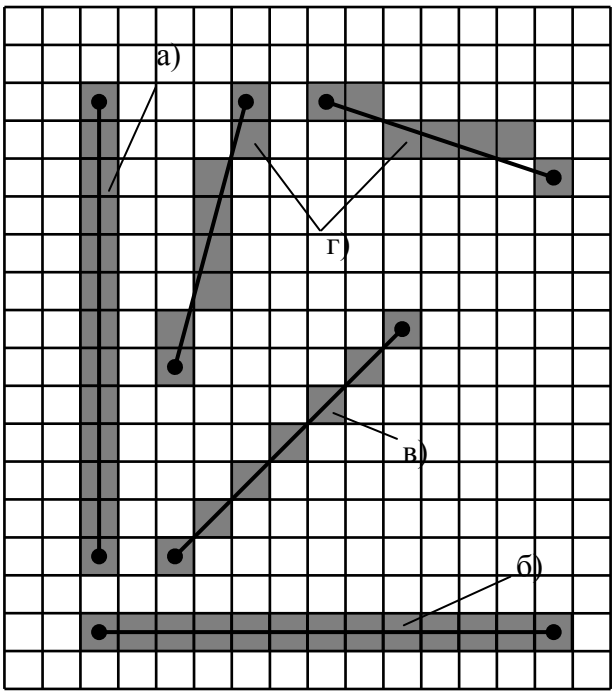
Задача: Ознакомиться с работой алгоритма несимметричный ЦДА и алгоритма Брезенхема, реализовав их на языке программирования, и разработать на их базе программу рисования заданных плоских фигур на прямоугольном растре.

Результат: программа рисования заданных плоских фигур на прямоугольном растре. Отчет в печатном виде, оформленный в соответствии с приложением 2.

Генерация изображений отрезков на прямоугольном растре

Одним из графических примитивов в двумерной графике является отрезок прямой линии, задаваемый координатами своих концевых точек. Отображение таких отрезков в растровой графике отнюдь не просто. Связано это с тем, что изображение отрезка на прямоугольном растре является совокупностью горизонтальных, либо вертикальных отрезков и отдельных точек рис. 1.1.

Задачей алгоритма генерации отрезков прямых линий является определение координат пикселей, которые необходимо подсветить для получения изображения заданного отрезка. Существует несколько алгоритмов генерации отрезков. Эти алгоритмы выбирают пиксели так, чтобы их координаты были наиболее близки к координатам точек, принадлежащих реальному отрезку. Наибольшее распространение в компьютерной графике получили инкрементные алгоритмы генерации отрезков. В этих алгоритмах, отрезок рисуется последовательно пиксель за пикселем. Координаты следующего пикселя определяются исходя из координат текущего, и некоторой дополнительной информации. Рассмотрим два инкрементных алгоритма.



а) Вертикальный отрезок.

б) Горизонтальный отрезок.

в) Отрезок, проведенный под углом 45º.

г) Отрезки, проведенные под произвольными углами.

Рисунок 1.1. Растровое представление отрезков прямых линий.

Алгоритм №1. Несимметричный цифровой дифференциальный анализатор (несимметричный ЦДА)

В основу работы этого алгоритма положен тот факт, что производная функции, представляющей прямую линию на плоскости, является величиной постоянной, т.е. $\frac{dy}{dx} = const$. Если у нас имеется отрезок прямой, заданный координатами своих концевых точек (x_1, y_1) и (x_2, y_2), то величину этой производной можно рассчитать следующим образом: $\frac{dy}{dx} = \frac{\Delta x}{\Delta y} = \frac{x_1 - x_2}{y_1 - y_2}$.

Алгоритм «несимметричный ЦДА» использует эту величину, чтобы рассчитать значение смещения по одной из координат, при смещении по другой на 1. Этот алгоритм можно описать следующей последовательностью действий (подразумевается, что хотя бы одно из значений Δx или Δy не равно нулю).

1. Рассчитаем величину d и определим, ось, вдоль которой будем продвигаться, следующим образом:

   Если $\Delta x = 0$, то наш отрезок является вертикальным, тогда $d = 0$, продвигаться будем вдоль оси y.

Если $\Delta y = 0$, то наш отрезок является горизонтальным, тогда $d = 0$, и продвигаться будем вдоль оси x.

Если $\Delta x \geq \Delta y$, тогда, $d = \frac{\Delta y}{\Delta x}$ продвигаться будем вдоль оси x.

В противном случае, $d = \frac{\Delta x}{\Delta y}$, и продвигаться будем вдоль оси y.

2. Подсветим пиксель с координатами (x_1, y_1).
3. Увеличим координату, соответствующую оси, вдоль которой продвигаемся, на 1 (-1), а другую на d, и подсветим пиксель, соответствующий этим координатам.
4. Повторять пункт 3, до тех пор, пока не нарисуем весь отрезок.

Рассмотрим работу алгоритма на следующих примерах рис. 1.2.

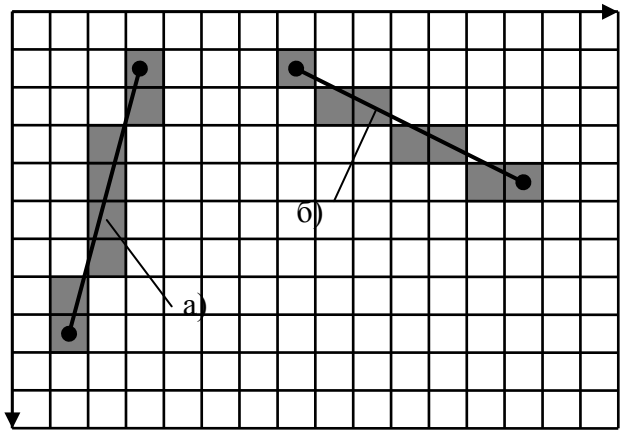


Рисунок 1.2. Генерация отрезков несимметричным ЦДА.

Отрезок а): $x_1 = 3, y_1 = 1; x_2 = 1, y_2 = 8$

1. Поскольку $\Delta y > \Delta x$, то $d = \frac{\Delta x}{\Delta y} = \frac{3-1}{1-8} = -\frac{2}{7}$, продвигаться будем вдоль оси *y*.
2. Подсветим пиксель с координатами (3, 1).
3. Увеличим y_1 на 1, а x_1 на $-\frac{2}{7}$. Получим $y_1 = 2, x_1 = 2\frac{5}{7}$. Подсветим пиксель с координатами (3 ($2\frac{5}{7} \approx 3$), 2).

4. Далее повторяем пункт 3, пока не нарисуем весь отрезок.

На следующем шаге получим $y_1 = 3,\ x_1 = 2\frac{3}{7}$. Подсветим пиксель (3, 2). Далее получим $y_1 = 4,\ x_1 = 2\frac{1}{7}$. Подсветим пиксель (4, 2) и т.д.

Отрезок б): $x_1 = 7, y_1 = 1;\ x_2 = 13, y_2 = 4$

1. Поскольку $\Delta x > \Delta y$, то $d = \frac{\Delta y}{\Delta x} = \frac{1-4}{7-13} = \frac{3}{6} = 0{,}5$, продвигаться будем вдоль оси *x*.
2. Подсветим пиксель с координатами (7, 1).
3. Увеличим x_1 на 1, а y_1 на 0,5 Получим $y_1 = 1{,}5,\ x_1 = 8$. Подсветим пиксель с координатами (8, 2 (1,5≈2)).

4. Далее повторяем пункт 3, пока не нарисуем весь отрезок.

На следующем шаге получим $y_1 = 2,\ x_1 = 9$. Подсветим пиксель (9, 2). Далее получим $y_1 = 2{,}5,\ x_1 = 10$. Подсветим пиксель (10, 3) и т.д.

Алгоритм №2. Алгоритм Брезенхема

Рассмотренный выше алгоритм неудобен тем, что в общем случае нам приходится использовать вещественные переменные и операцию деления, что отрицательно сказывается на его быстродействии. Однако можно разработать такой алгоритм генерации отрезков, в котором будет использоваться только целочисленная арифметика, и не будет операций деления и умножения. Быстродействие такого алгоритма также возрастет. Данный алгоритм получил название «алгоритм Брезенхема» по фамилии его автора.

Будем разрабатывать алгоритм нахождения координат точек между A и B использующий вещественные числа, но таким образом, что это ограничение потом можно будет легко устранить, и использовать только целочисленные переменные. Для этого выполним следующие операции.

Начнем рисовать отрезок с точки A(Ax, Ay). В цикле будем задавать приращение 1 для переменной Ay и будем оставлять Ax либо неизменным, либо также увеличивать на 1. При этом последний выбор будем осуществлять так, чтобы новая точка сетки (Ax, Ay) располагалась как можно ближе к прямой

линии, проходящей через точки A и B рис. 1.3.

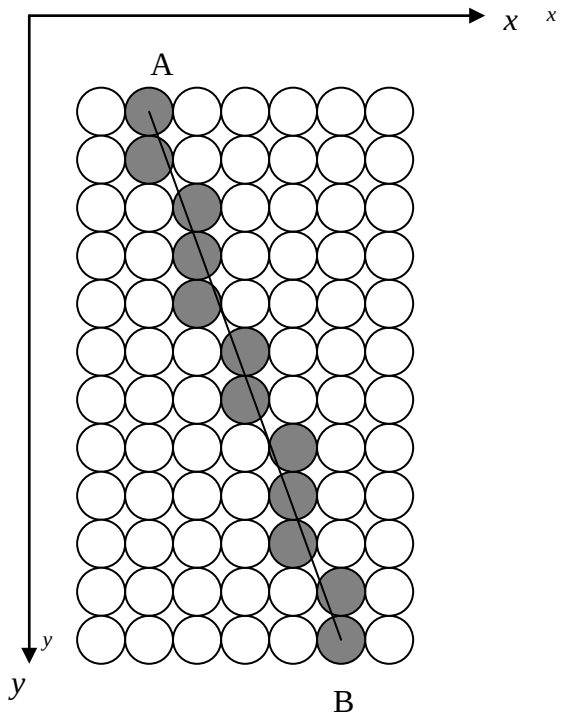


Рисунок 1.3. Пример отрезка.

Это означает, что расстояние по горизонтали между новой выбранной точкой и этой линией не должно превышать значения 0,5.

Введем переменную *d* для обозначения этого расстояния. Потребуем, чтобы $-0{,}5 < d \le 0{,}5$.

Последнее неравенство обеспечивает условие для определения необходимости давать приращение переменной Ax.

Теперь можно написать первый вариант реализации алгоритма нахождения координат точек между A и B:

1. Установим значения для переменных *d* и *t*: $d = 0$, $t = \frac{\Delta x}{\Delta y} = \frac{Ax - Bx}{Ay - By}$.

2. Подсветим пиксель с координатами (Ax, Ay).

3. Увеличим *d* на величину *t*, а Ay на 1 (-1).

4. Если $d > 0{,}5$, то увеличим Ax на 1, и вычтем 1 из *d*.

5. Будем повторять пункты 2-4, пока не нарисуем весь отрезок.

Отклонение, вначале устанавливается равным нулю и изменяется на каждом шаге. Поскольку оно показывает, насколько левее точной прямой

линии лежит вычисленная точка, то значение *d* увеличивается на значение *t*, если *y* увеличивается на 1 и *x* остается без изменения. Это условие не выполняется, если значение *d* превышает 0,5. В этот момент нужно увеличить значение *x* на 1. Соответственно, и отклонение *d* должно быть уменьшено на единицу.

Теперь посмотрим, как можно избавиться от вещественных значений переменных *t* и *d* и постоянной 0,5.

Значение переменной *t* вычисляется следующим образом:

$$t = \frac{Ax - Bx}{Ay - By}$$

где числитель и знаменатель представляют собой целые числа.

Величина *d* вычисляется как конечная сумма элементов, каждый из которых равен либо *t*, либо -1. Поэтому *d* также можно записать в виде частного со знаменателем равным $Ay - By$.

Значит, можно перейти к целочисленным переменным *t* и *d*, путем умножения на значение знаменателя. Также просто избавиться от константы 0,5. Для этого нужно дополнительно умножить значение знаменателя на 2. Эта операция может быть заменена простым поразрядным сдвигом на 1 бит влево. Таким образом $t = 2\Delta x$, где $\Delta x = Ax - Bx$.

Напишем новый вариант реализации нашего алгоритма.

1. Установим значения для переменных *d*, *t* и Δ: $d = 0$, $t = 2\Delta x = 2(Ax - Bx)$, $\Delta = 2\Delta y = 2(Ay - By)$.

2. Подсветим пиксель с координатами (Ax, Ay).

3. Увеличим *d* на величину *t*, а Ay на 1.

4. Если $d > \Delta y$, то увеличим Ax на 1, и вычтем Δ из *d*.

5. Будем повторять пункты 2-4, пока не нарисуем весь отрезок.

Однако наша функция будет правильно работать только для того случая, который изображен на рис. 1.3. Для того, чтобы алгоритм правильно работал для любого отрезка, его необходимо дополнить. В результате получим следующий алгоритм:

1. Отсортируем вершины A и B так, чтобы выполнялось условие $Ay \le By$.

2. Рассчитаем значения $\Delta x = Bx - Ax$, и $\Delta y = By - Ay$.

3. Если $\Delta x \ge 0$, то установим значение $dx = 1$, иначе: $dx = -1$, и изменим знак у Δx на противоположный.

4. Установим $d = 0$. Если $\Delta y \ge \Delta x$, то установим $t = 2\Delta x$, $\Delta = 2\Delta y$. Иначе $t = 2\Delta y$, $\Delta = 2\Delta x$. (операцию умножения на 2 везде заменяем поразрядным сдвигом влево на 1 бит).

5. Если $\Delta y \ge \Delta x$, то перейти к пункту 6, иначе к пункту 11.

6. Подсветим пиксель с координатами (Ax, Ay).

7. Увеличим Ay на 1, а d на величину t.

8. Если $d > \Delta y$, то увеличить Ax на величину dx, и вычесть величину Δ из d.

9. Повторять пункты 6-8, пока не нарисуем весь отрезок.

10. Завершить выполнение алгоритма.

11. Подсветим пиксель с координатами (Ax, Ay).

12. Увеличим Ax на 1, а d на величину t.

13. Если $d > \Delta x$, то увеличить Ay на 1, и вычесть величину Δ из d.

14. Повторять пункты 6-8, пока не нарисуем весь отрезок.

15. Завершить выполнение алгоритма.

Задания к лабораторной работе №1

Перед выдачей заданий, студентов необходимо поделить на пары. Если количество студентов нечетное, то оставшемуся без пары студенту предлагается выполнять вариант №1.

Все задания к лабораторной работе №1 необходимо выполнять, используя псевдопиксель, в виде закрашенного сплошным цветом квадрата размером n x n пикселей. Для выполнения заданий необходимо в первую очередь реализовать соответствующие алгоритмы генерации отрезков на

используемом языке программирования. Затем, используя эти алгоритмы, выполнить соответствующий вариант задания.

При выполнении вариантов заданий ЗАПРЕЩАЕТСЯ пользоваться встроенными функциями рисования отрезков!

Вариант №1.

Нарисовать наибольший **равносторонний** треугольник, который поместится в области вывода. Одна из сторон треугольника должна идти вдоль нижней границы области вывода рис. 1.4. Боковые стороны необходимо нарисовать красными линиями, используя алгоритм Брезенхема. Основание нарисовать зеленой линией алгоритмом несимметричного ЦДА. Размер псевдопикселя для всех отрезков составляет 20х20 пикселей.

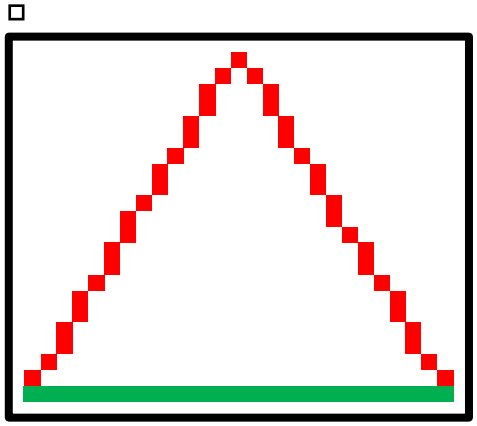


Рисунок 1.4. Наибольший равносторонний треугольник.

Добейтесь, чтобы размеры треугольника изменялись при изменении размеров области вывода (окна).

Вариант №2.

Нарисовать наибольший **правильный** шестиугольник, который поместится в области вывода и построить все его диагонали рис. 1.5. Стороны шестиугольника необходимо нарисовать оранжевым цветом, используя алгоритм несимметричный ЦДА и псевдопиксель размером 20х20 пикселей. Диагонали изобразить тремя различными цветами (красным зеленым и синим) алгоритмом Брезенхема псевдопикселями размером 10х10 пикселей.

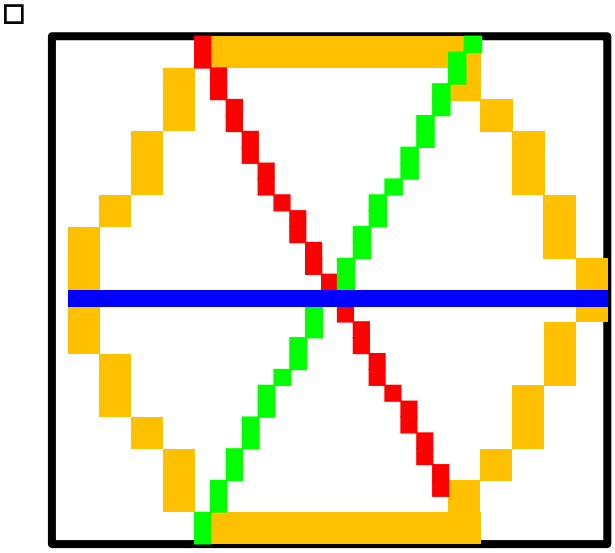


Рисунок 1.5. Наибольший правильный шестиугольник.

Добейтесь, чтобы размеры рисунка изменялись при изменении размеров области вывода (окна).

Вариант №3.

Нарисовать наибольший прямоугольник **с соотношением сторон 2/3**, который поместится в области вывода и построить все его диагонали рис. 1.6. Прямоугольник расположить по центру экрана. Короткие стороны необходимо расположить параллельно горизонтальной стороне экрана. Прямоугольник необходимо нарисовать красным цветом, используя алгоритм несимметричный ЦДА и псевдопиксель размером 20x20 пикселей. Диагонали изобразить двумя различными цветами (зеленым и синим) алгоритмом Брезенхема псевдопикселями размером 10x10 пикселей.

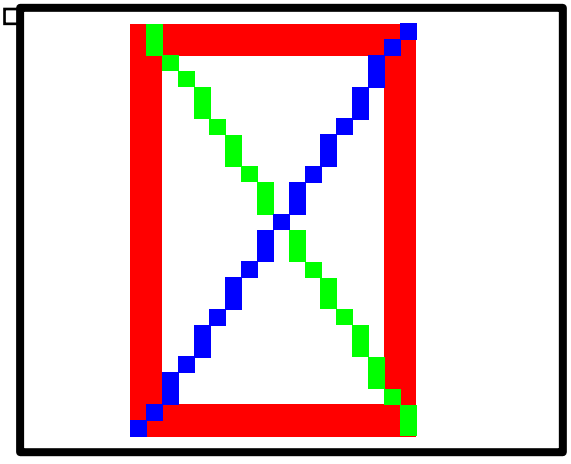


Рисунок 1.6. Наибольший прямоугольник с соотношением сторон 2/3.

Добейтесь, чтобы размеры рисунка изменялись при изменении размеров области вывода (окна).

Вариант №4.

Нарисовать наибольшую **правильную** пятиконечную звезду, которая поместится в области вывода и пятиугольник внутри нее рис. 1.7. Звезду необходимо нарисовать красным цветом, используя алгоритм Брезенхема и псевдопиксель размером 20х20 пикселей. Пятиугольник нарисовать синим цветом алгоритмом несимметричный ЦДА псевдопикселями размером 10х10 пикселей.

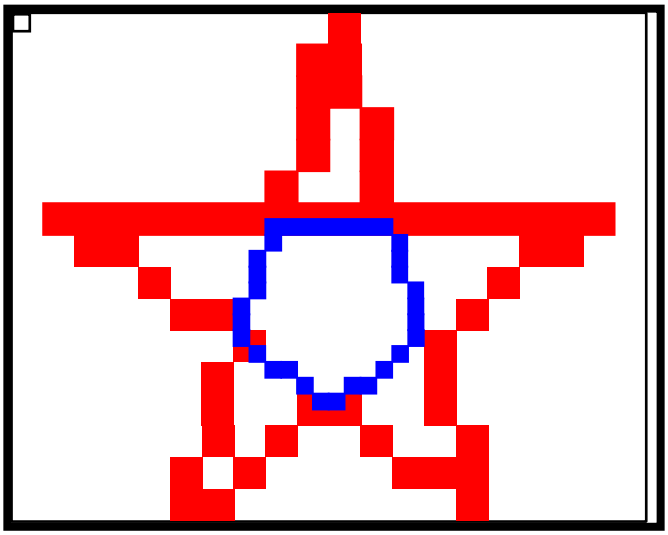


Рисунок 1.7. Наибольшая правильная пятиконечная звезда и пятиугольник.

Добейтесь, чтобы размеры рисунка изменялись при изменении размеров области вывода (окна).

Вариант №5.

Нарисовать наибольшую **правильную** шестилучевую звезду, которая поместится в области вывода рис. 1.8. Стороны верхнего треугольника необходимо нарисовать зеленым цветом, используя алгоритм несимметричный ЦДА и псевдопиксель размером 10х10 пикселей. Стороны нижнего треугольника необходимо нарисовать красным цветом, используя алгоритм

Брезенхема и псевдопиксель размером 20х20 пикселей.

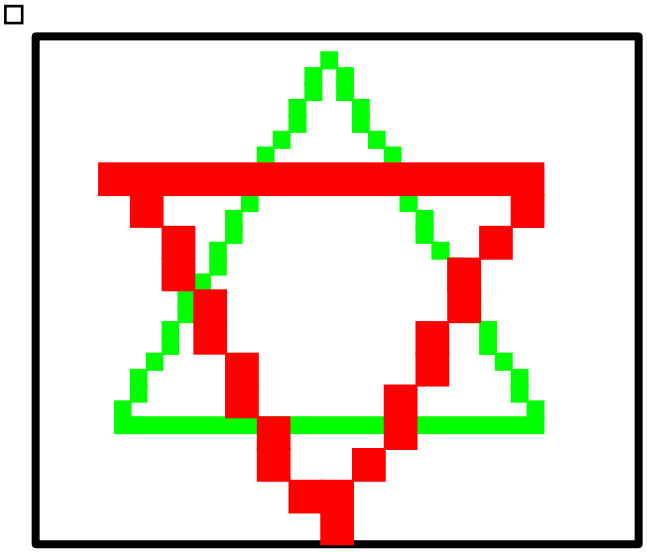


Рисунок 1.8. Наибольшая правильная шестилучевая звезда.

Добейтесь, чтобы размеры рисунка изменялись при изменении размеров области вывода (окна).

Вариант №6.

Нарисовать зеленым цветом наибольшее изображение октаэдра, которое поместится в области вывода рис. 1.9 (внешний контур изображения октаэдра является **правильным шестиугольником**). Стороны октаэдра наиболее близкие к наблюдателю (жирные линии) нарисовать, используя алгоритм несимметричный ЦДА и псевдопиксель размером 20х20 пикселей. Стороны дальние необходимо нарисовать, используя алгоритм Брезенхема и псевдопиксель размером 10х10 пикселей.

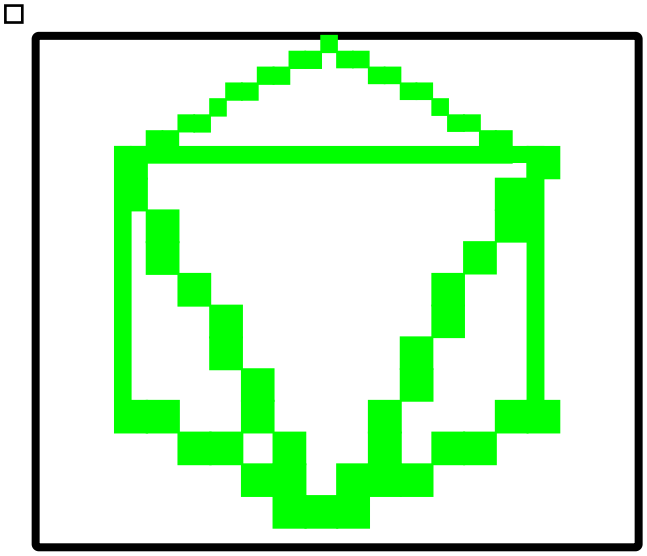


Рисунок 1.9. Наибольший октаэдр.

Добейтесь, чтобы размеры рисунка изменялись при изменении размеров области вывода (окна).

Вариант №7.

Нарисовать красным цветом наибольшее изображение куба, которое поместится в области вывода рис. 1.10 (внешний контур изображения куба является **правильным шестиугольником**). Ребра куба наиболее близкие к наблюдателю (жирные линии) нарисовать, используя алгоритм Брезенхема и псевдопиксель размером 20х20 пикселей. Стороны дальние необходимо нарисовать, используя алгоритм несимметричный ЦДА и псевдопиксель размером 10х10 пикселей.

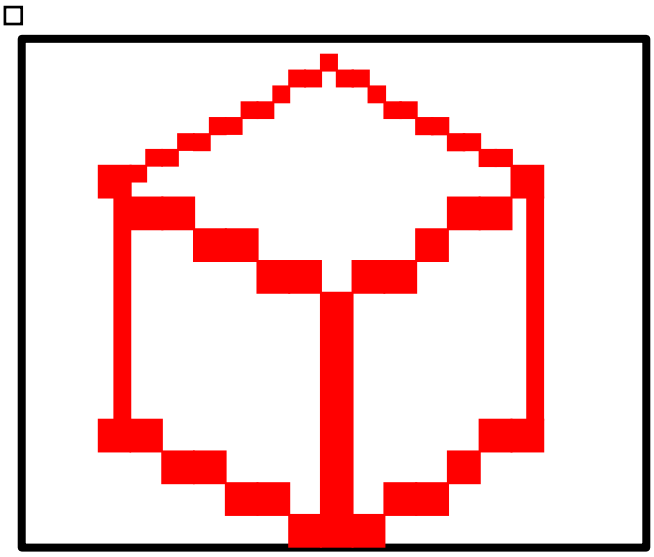


Рисунок 1.10. Наибольший куб.

Добейтесь, чтобы размеры рисунка изменялись при изменении размеров области вывода (окна).

Вариант №8.

На базе алгоритма несимметричный ЦДА разработать алгоритм рисования пунктирной линии. Пунктирная линия состоит из отрезков, разделенных промежутками. Длина отрезка не может быть больше 10 псевдопикселей. Длина промежутка составляет половину длины отрезка. Линия начинается и заканчивается отрезком. Все отрезки и промежутки должны иметь примерно одинаковую длину. Используя алгоритм Брезенхема и псевдопиксель 20x20, нарисовать зеленым цветом наибольший прямоугольник, который поместится в области вывода. Диагонали прямоугольника провести пунктирной линией красного цвета, используя псевдопиксель 5x5 рис. 1.11.

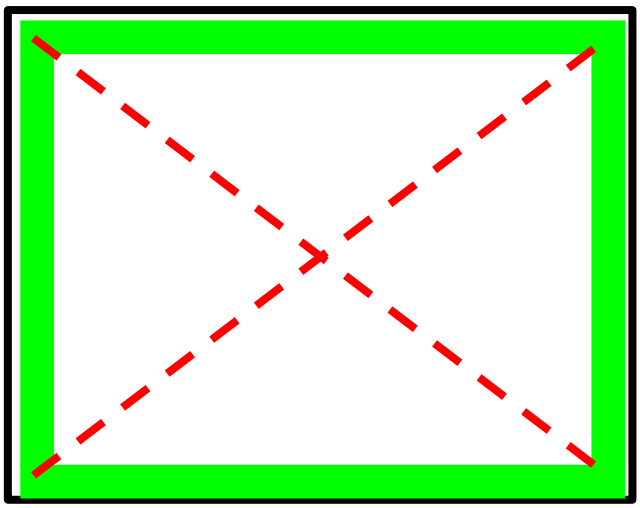


Рисунок 1.11. Наибольший прямоугольник с диагоналями.

Добейтесь, чтобы размеры рисунка изменялись при изменении размеров области вывода (окна).

Вариант №9.

Нарисовать красным цветом наибольшее изображение икосаэдра, которое поместится в области вывода рис. 1.12 (наружный контур изображения

икосаэдра является **правильным шестиугольником**, внутренний треугольник – **равносторонний**). Ребра икосаэдра, наиболее близкие к наблюдателю, (жирные линии) нарисовать, используя алгоритм несимметричный ЦДА и псевдопиксель размером 20х20 пикселей. Ребра дальние необходимо нарисовать, используя алгоритм Брезенхема и псевдопиксель размером 10х10 пикселей.

Добейтесь, чтобы размеры рисунка изменялись при изменении размеров области вывода (окна).

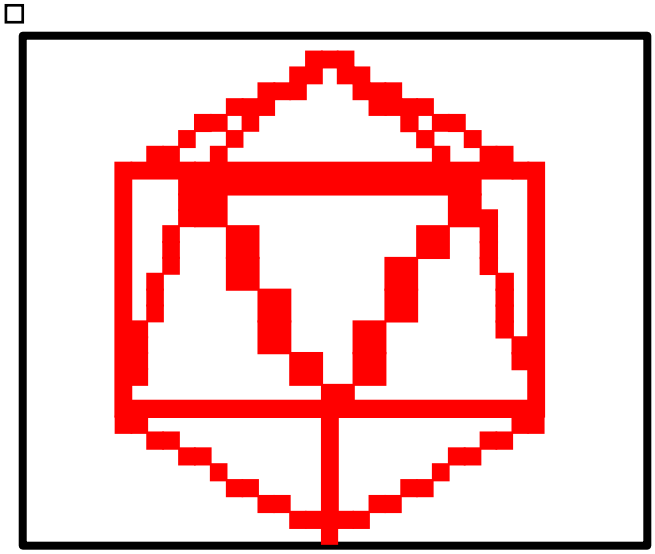


Рисунок 1.12. Наибольший икосаэдр.

Вариант №10.

Нарисовать наибольшее изображение вида сверху на тетраэдр, которое поместится в области вывода рис. 1.13 (внешний контур изображения является **равносторонним треугольником**). Ребра тетраэдра наиболее близкие к наблюдателю **(жирные линии)** нарисовать красным цветом, используя алгоритм несимметричный ЦДА и псевдопиксель размером 20х20 пикселей . Стороны дальние необходимо нарисовать зеленым цветом, используя алгоритм Брезенхема и псевдопиксель размером 10х10 пикселей.

Добейтесь, чтобы размеры рисунка изменялись при изменении размеров области вывода (окна).

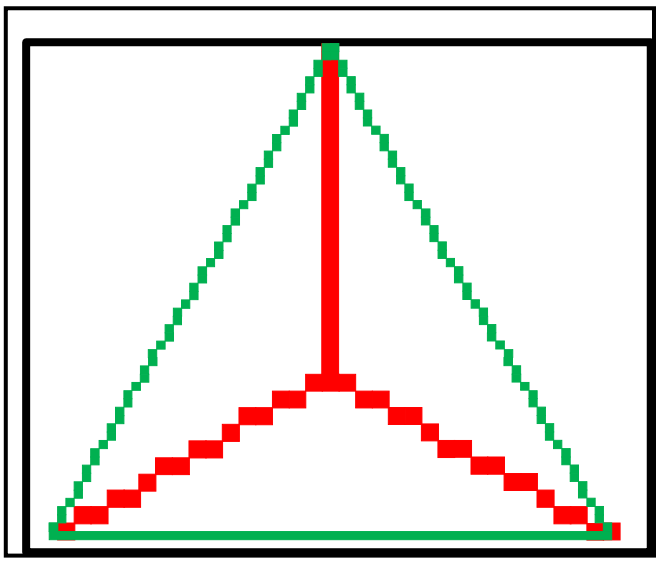


Рисунок 1.13. Наибольший тетраэдр.

Добейтесь, чтобы размеры рисунка изменялись при изменении размеров области вывода (окна).

Лабораторная работа №2. Геометрические преобразования на плоскости

Цель: Научиться выполнять геометрические преобразования над плоскими фигурами.

Задача: Используя матричное представление геометрических преобразований на плоскости, разработать программу анимации плоских фигур.

Результат: программа анимации заданных плоских фигур. Отчет в печатном виде, оформленный в соответствии с приложением 2.

Способы представления геометрических преобразований на плоскости

Рассмотрим следующую систему уравнений:

$$\begin{cases} x' = x + a; \\ y' = y. \end{cases}$$

Эти уравнения можно интерпретировать двояким образом:

1. Все точки на плоскости *xy* перемещаются вправо на расстояние a рис. 2.1.а.
2. Координатные оси x и y перемещаются влево на расстояние a рис. 2.1.б.

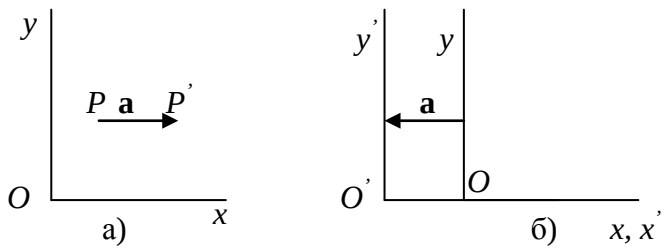


Рисунок 2.1. а) — перенос точек; б) — перенос системы координат

Этот простой пример иллюстрирует принцип, применимый и в более сложных ситуациях. Мы и далее будем рассматривать системы уравнений, интерпретируя их либо как преобразования всех точек в фиксированной системе координат, либо как изменение самой системы координат.

Пусть необходимо повернуть точку $P(x, y)$ вокруг начала координат O на угол. Изображение новой точки на рис. 2.2 обозначим через $P'(x', y')$.

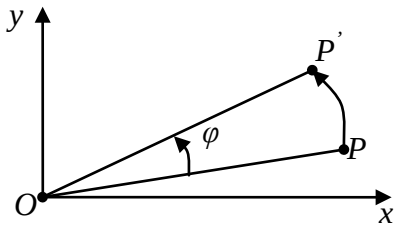


Рисунок 2.2. Поворот вокруг точки О на угол φ

Уравнения, задающие поворот точек на плоскости относительно начала координат:

$$\begin{cases} x' = x\cos\varphi - y\sin\varphi; \\ y' = x\sin\varphi + y\cos\varphi \end{cases}$$

Эта система уравнений описывает поворот вокруг точки О — начала системы координат. Но часто это не то, что нам нужно. Если требуется выполнить поворот относительно заданной точки (x_0, y_0), то в этих уравнениях можно заменить x на $(x - x_0)$, y на $(y - y_0)$, x' на, y' – на $(y' - y_0)$:

$$\begin{cases} x' - x_0 = (x - x_0)\cos\varphi - (y - y_0)\sin\varphi; \\ y' - y_0 = (x - x_0)\sin\varphi + (y - y_0)\cos\varphi \end{cases}$$

$$\begin{cases} x' = x_0 + (x - x_0)\cos\varphi - (y - y_0)\sin\varphi; \\ y' = y_0 + (x - x_0)\sin\varphi + (y - y_0)\cos\varphi \end{cases}$$

Система уравнений:

$$\begin{cases} x' = ax; \\ y' = by \end{cases}$$

описывает изменение масштаба относительно точки О – начала системы координат. При этом координата x всех точек плоскости изменяется в a раз, а координата y – в b раз. Если $a = b$ то искажения изображения объектов не происходит, и тогда говорят о равномерном масштабировании. В противном случае, изображение объектов искажается, и такое преобразование называется неравномерным масштабированием.

Если требуется выполнить масштабирование относительно заданной точки (x_0, y_0), то в этих уравнениях также можно заменить x на $(x - x_0)$, y на

$(y - y_0)$, x’ на $(x' - x_0)$, y’ – на $(y' - y_0)$:

$$\begin{cases} x' - x_0 = a(x - x_0); \\ y' - y_0 = b(y - y_0). \end{cases}$$

$$\begin{cases} x' = x_0 + a(x - x_0); \\ y' = y_0 + b(y - y_0). \end{cases}$$

Система уравнений поворота точки относительно начала координат может быть записана в виде одного матричного уравнения:

$$\begin{bmatrix} x' & y' \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{bmatrix}$$

Можно записать в матричной форме и систему уравнений для масштабирования:

$$\begin{bmatrix} x' & y' \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}$$

В ситуациях, когда совмещаются несколько преобразований, было бы удобно иметь единое матричное произведение для каждого элементарного преобразования. На первый взгляд это кажется невозможным, если преобразование включает операцию переноса. Но, при использовании для представления точек на плоскости **однородных координат,** это реально.

Двумерные однородные координаты.

Двумерные однородные координаты точки имеют следующий вид:

$$\begin{bmatrix} X & Y & W \end{bmatrix}$$

Здесь W - произвольный множитель не равный 0.

Двумерные декартовые координаты (x, y) точки получаются из однородных, делением последних на множитель W:

$$x = X/W;\ y = Y/W;\ W \neq 0$$

Однородные координаты можно представить как промасштабированные с коэффициентом W значения двумерных координат, расположенные в плоскости с $Z = W$.

В силу произвольности значения W в однородных координатах не существует единственного представления точки, заданной в декартовых

координатах.

Начнем с простого переноса. Пусть точка $P(x,y)$ переносится в точку $P'(x',y')$, где

$$x' = x + a;$$
$$y' = y + b.$$

Эти уравнения можно переписать в виде:

$$[x' \quad y' \quad 1] = [x \quad y \quad 1]\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{bmatrix}$$

Такую запись принято называть записью в системе "однородных координат".

Запись каждого преобразования в форме произведения матриц позволяет совмещать несколько преобразований в одном. Чтобы показать такое совмещение преобразований, объединим поворот с двумя переносами. Поворот на угол φ вокруг начала координат О был описан ранее. Заменим это уравнение следующим:

$$[x' \quad y' \quad 1] = [x \quad y \quad 1]\begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Теперь выведем новую версию уравнений для описания поворота на угол φ вокруг точки (x_0, y_0); это уравнение может быть выражено формулой:

$$[x' \quad y' \quad 1] = [x \quad y \quad 1]\mathbf{R},$$

где через **R** обозначена матрица размером 3x3. Для нахождения этой матрицы **R** будем считать, что преобразование состоит из трех шагов с промежуточными точками (u_1, v_1) и (u_2, v_2).

Преобразование для переноса точки О начала координат в точку (x_0, y_0):

$$[u_1 \quad v_1 \quad 1] = [x \quad y \quad 1]\mathbf{T}',$$

где $\mathbf{T}' = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -x_0 & -y_0 & 1 \end{bmatrix}$

Поворот на угол φ относительно точки O начала координат:

$$[u_2 \quad v_2 \quad 1] = [u_1 \quad v_1 \quad 1]\mathbf{R_0},$$

где $\mathbf{R_0} = \begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Возвращение точки O начала координат на прежнее место:

$$[x' \quad y' \quad 1] = [u_2 \quad v_2 \quad 1]\mathbf{T},$$

где $\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x_0 & y_0 & 1 \end{bmatrix}$

Возможность комбинации этих шагов основана на свойстве ассоциативности матричного умножения, то есть (AB)C = A (BC) для любых трех матриц A, B и C, имеющих размерности, допускающие такое умножение. Для любой части этого уравнения мы можем просто записать ABC. Теперь найдем:

$$[x' \quad y' \quad 1] = [u_2 \quad v_2 \quad 1]\mathbf{T} = ([u_1 \quad v_1 \quad 1]\mathbf{R_0})\mathbf{T} =$$

$$= (([x_1 \quad y_1 \quad 1]\mathbf{T'})\mathbf{R_0})\mathbf{T} = [x_1 \quad y_1 \quad 1]\mathbf{T'R_0T} = [x_1 \quad y_1 \quad 1]\mathbf{R}$$

где $\mathbf{R} = \mathbf{T'R_0T}$

Это и будет искомая матрица, которая после выполнения двух матричных умножений дает:

$$\mathbf{R} = \begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

где введены обозначения:

$$c_1 = x_0 - x_0\cos\varphi + y_0\sin\varphi;$$

$$c_2 = y_0 - x_0\sin\varphi - y_0\cos\varphi$$

Подобным образом можно записать любую последовательность геометрических преобразований на плоскости. Для этого нужно определить, какие преобразования переноса, поворота и масштабирования относительно начала координат необходимо выполнить. Затем сформировать матрицы для

каждого из этих преобразований, и перемножить их в той последовательности, в какой выполняются преобразования. Полученная в результате матрица и будет матрицей заданного сложного преобразования.

Задания к лабораторной работе №2

Перед выдачей заданий, студентов необходимо поделить на пары. Если количество студентов нечетное, то оставшемуся без пары студенту предлагается выполнять вариант №1.

Фигуру, подвергающуюся преобразованию, задайте массивом векторов, где каждый вектор представляет координаты одной из вершин фигуры.

Необходимое преобразование задавайте матрицей 3x3. Матрицу преобразования, являющегося комбинацией нескольких преобразований, формируйте путем последовательных перемножений матриц соответствующих преобразований.

Следует программировать задачу в общем виде, используя параметры (угол поворота, величину перемещения, коэффициент масштабирования) и предполагая произвольную фигуру, но отлаживать программу на примере, указанном в задаче в скобках!

На рисунках, приведенных в вариантах заданий, жирными линиями изображено первоначальное положение фигуры.

Включив в программу необходимые задержки (используйте таймер), добейтесь желаемой скорости смены – изображений. В конце некоторых заданий есть дополнения. Их необходимо выполнять только после того, как будет готова программа, для базового задания.

Вариант №1

Поворачивать любую фигуру, образованную замкнутой ломаной линией (секундную стрелку), вокруг одной из ее вершин (оси вращения) на небольшой угол *r* по часовой стрелке (соответствующий секунде) рис.2.3. Совершить *n* поворотов .

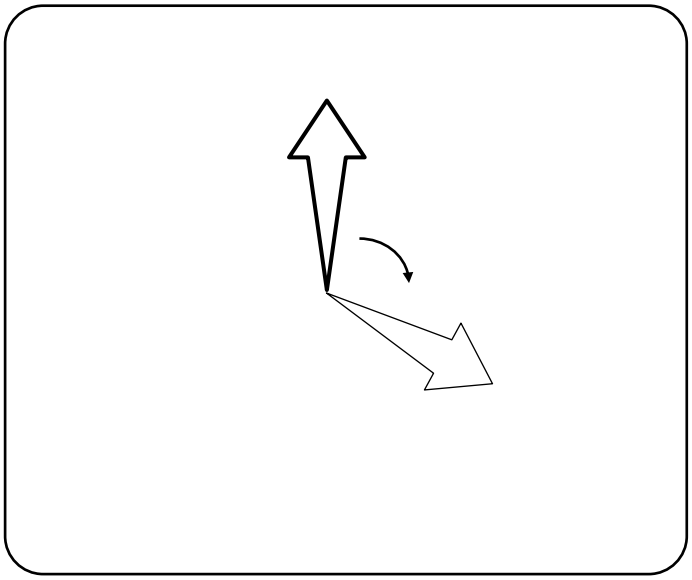


Рисунок 2.3. Секундная стрелка

Дополнение:

На основе полученной программы создайте часы - будильник, звонящие по достижении заданного времени. К рисунку добавьте часовую, минутную стрелки и стрелку будильника.

Вариант №2

Поворачивать любую фигуру, образованную отрезками прямых (велосипедное колесо), и расположенную у левого края экрана, вокруг заданной точки фигуры (оси колеса), на угол *r* по часовой стрелке (на половину угла между соседними спицам), сдвигая, фигуру, по горизонтали вправо на величину *t*. Фигуру перемещать до правого края экрана. Один, из отрезков (спицу колеса) изобразить контрастным цветом. Величину *t* рассчитайте так, чтобы создавалось впечатление катящегося колеса, и проведите пунктирную горизонталь, через точки вращения рис. 2.4.

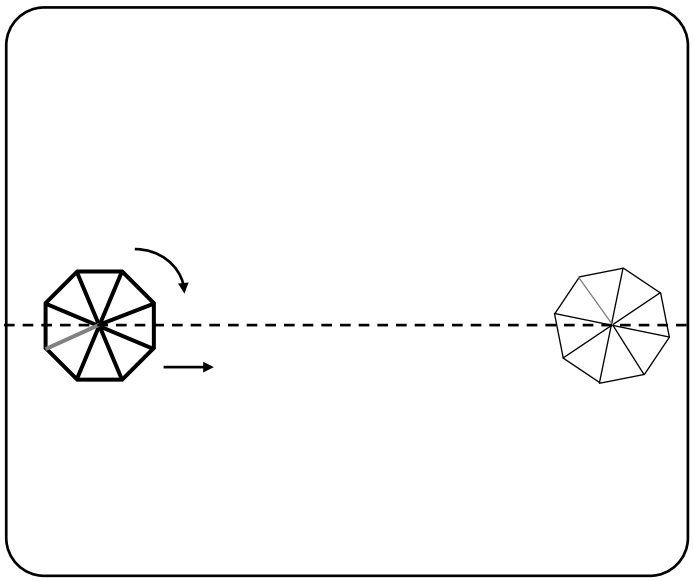


Рисунок 2.4. Колесо.

Дополнение:

Создайте картину движущегося по экрану велосипеда.

Вариант №3

Задана любая фигура двумя замкнутыми ломаными линиями (треугольник внутри квадрата). Требуется поворачивать фигуру вокруг заданной точки *z* (центра квадрата) на угол *r* (15 градусов) по часовой стрелке, незначительно уменьшая относительно точки и часть фигуры, образованную первой ломаной линией (треугольник), и во столько же раз увеличивая другую часть фигуры (квадрат). Преобразования повторять, пока уменьшающаяся часть не превратится в точку.

Картина должна напоминать "вертушку", состоящую из удаляющегося при вращении треугольника и приближающегося квадрата рис. 2.5. Затем повторить процесс, только уменьшать вторую часть фигуры и увеличивать первую. Повторять чередование, пока пользователь не завершит работу программы.

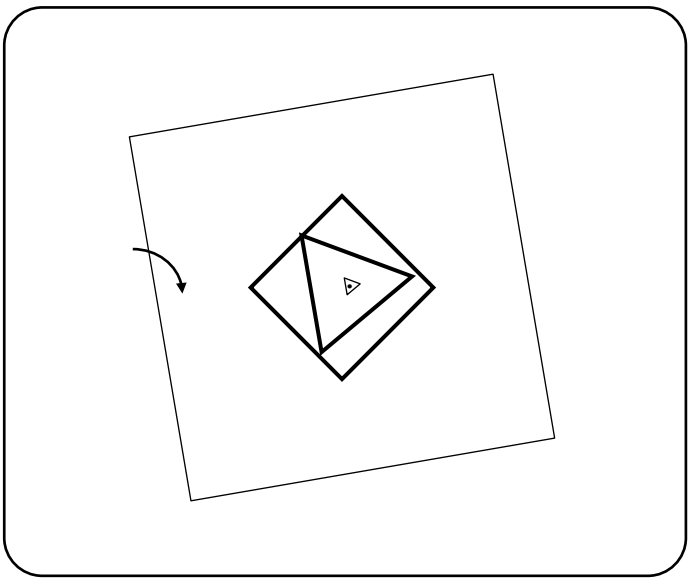


Рисунок 2.5. Вертушка.

Вариант №4

Поворачивать любую фигуру, образованную отрезками прямых (маятник в крайнем левом положении, отклоненный, на 45 градусов от вертикали), вокруг точки в центре верхней половины экрана (оси маятника) на небольшой угол *r* (5 градусов) против часовой стрелки. Совершить *k* поворотов (достигнув крайнего правого положения). Затем, аналогично, выполнить *k* поворотов по часовой стрелке (до крайнего левого положения). Выполнить колебания фигуры, пока пользователь не завершит работу программы.

Процесс должен напоминать колебание маятника рис. 2.6.

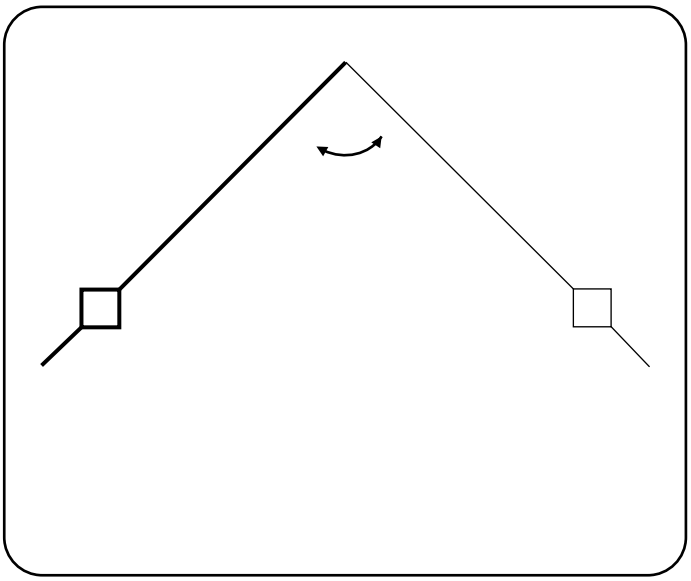


Рисунок 2.6. Маятник.

Дополнение:

Изменяйте скорость поворота маятника в соответствии с гармоническим законом колебаний. Создайте программу часов с маятником.

Вариант №5

Поворачивать любую фигуру, образованную отрезками прямых (лист дерева) и находящуюся в правой половине экрана, вокруг точки *P*, лежащей в центре верхнего края экрана, на угол *r* (10 градусов) по часовой стрелке, каждый раз незначительно увеличивая фигуру относительно точки *P*. Повороты выполнять до тех пор, пока фигура не займет примерно симметричное первоначальному положение в левой половине экрана. Затем, аналогично, поворачивать фигуру против часовой стрелки вокруг той же точки *P*. Чередовать такие колебания фигуры, пока она остается в пределах экрана.

Картина должна напоминать падение осеннего листа, сорвавшегося с дерева рис. 2.7.

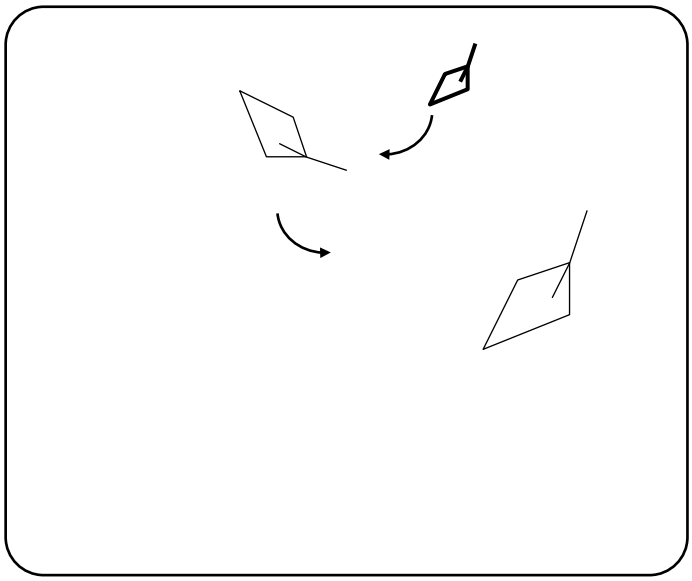


Рисунок 2.7. Падение листа.

Вариант №6

Задана любая фигура, состоящая из отрезков (треугольник и прямоугольник с общей стороной, напоминающие кабину колеса обозрения); одна из точек фигуры особо выделена (точка подвески кабины). Поворачивать фигуру так, чтобы особая точка поворачивалась вокруг центра экрана на угол *r* (15 градусов) против часовой стрелки при каждом повороте. Совершать полные обороты фигуры вокруг центра экрана, пока пользователь не завершит работу программы. Картина должна напоминать вращение кабины колеса обозрения рис. 2.8.

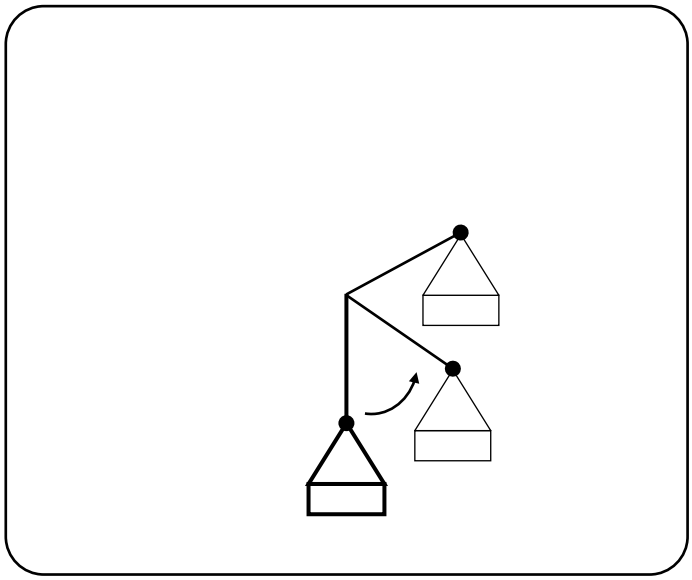


Рисунок 2.8. Колесо обозрения.

Дополнение:

Создайте полную картину вращающегося колеса обозрения с шестью кабинами.

Вариант №7

Поворачивать любую фигуру, заданную замкнутой ломаной линией с яркой точкой на фигуре (многоугольник, похожий на круг, с ярким "пятном" на окружности), вокруг центра фигуры на угол *r* (30 градусов) по часовой стрелке, сдвигая фигуру так, чтобы она перемещалась по диагонали экрана, и незначительно увеличивая размеры фигуры в а раз (1.2 раза) относительно своего центра, но так, чтобы пятно, вращаясь вместе с фигурой, оставалось на

прежнем расстоянии от ее центра. Картина должна напоминать катящийся с горы нарастающий снежный ком, а пятно напоминать о первоначальных размерах кома. Нарисовать гору и остановить ком в углу экрана рис. 2.9.

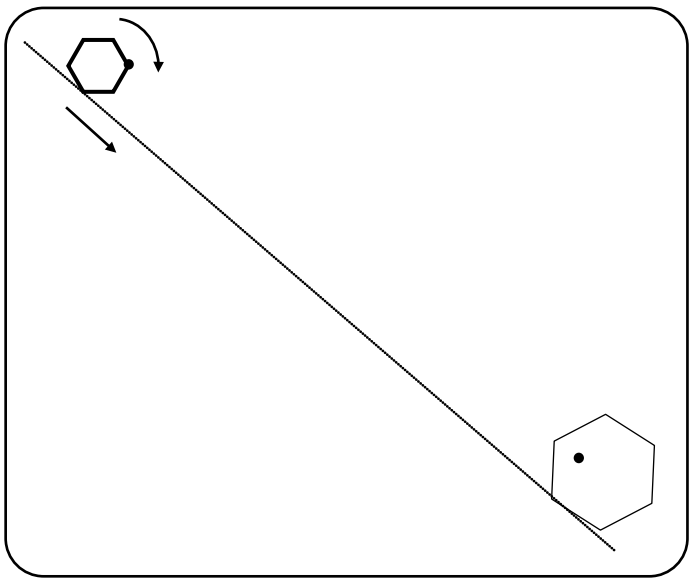


Рисунок 2.9. Снежный ком.

Вариант №8

Задана любая фигура замкнутой ломаной линией (квадрат). Поворачивать фигуру вокруг своего центра на угол *r* (35 градусов) против часовой стрелки и одновременно вокруг центра экрана на угол *s* (20 градусов) по часовой стрелке. Одну из сторон фигуры выделить другим цветом. Совершать полные обороты фигуры вокруг центра экрана, пока пользователь не завершит работу программы. Картина должна напоминать карусель с вращающейся кабиной рис. 2.10.

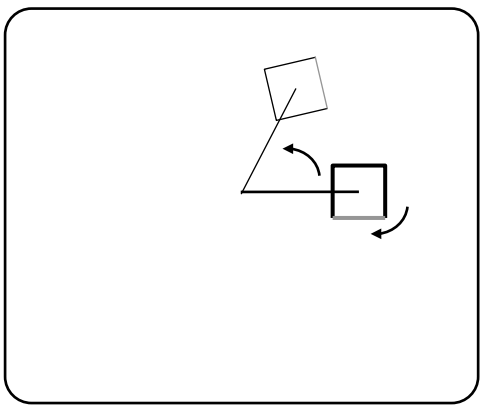


Рисунок 2.10. Карусель с вращающейся кабиной.

Дополнение:

Создайте полную картину карусели с четырьмя вращающимися кабинами.

Вариант №9

Поворачивать любую фигуру, заданную замкнутой ломаной линией (вытянутый прямоугольник, стоящий у левого края экрана на наклонной линии), на угол *r* (90 градусов) по часовой стрелке вокруг первой вершины фигуры, затем на тот же угол вокруг второй вершины и т. д. *k* раз (каждый раз в качестве точки поворота выбирать нижний правый угол очередного прямоугольника, Одну из сторон фигуры (короткую) выделить другим цветом рис. 2.11.

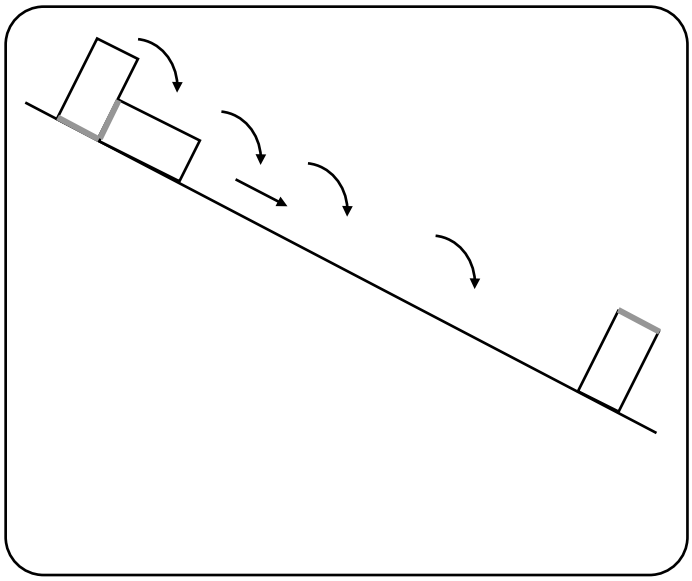


Рисунок 2.11. Кувырок.

Дополнение:

Добейтесь того, чтобы прямоугольник стоял на выделенной стороне дольше, чем на остальных, и докувыркался до правого края экрана.

Вариант №10

Поворачивать любую фигуру, заданную замкнутой ломаной линией (правильный шестиугольник, расположенный в центре экрана), на угол *r* (30

градусов) по часовой стрелке вокруг центра фигуры. Вторую фигуру (правильный пятиугольник меньшего размера), поворачивать вокруг центра первой фигуры по часовой стрелке на угол m (15 градусов) одновременно поворачивая вокруг своего центра по часовой стрелке на такой же угол. Одну из сторон второй фигуры (первоначально направленную к шестиугольнику) выделить другим цветом рис. 2.12.

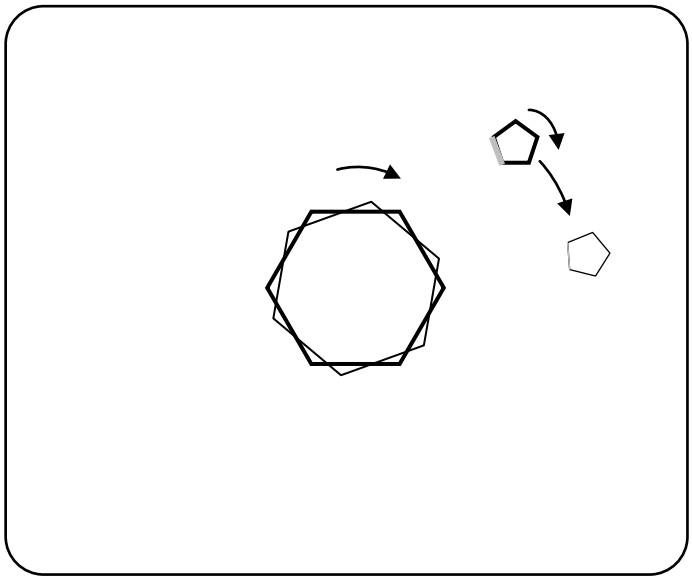


Рисунок 2.12. Земля и Луна.

Рисунок должен имитировать движение Луны (вторая фигура) вокруг Земли (первая фигура). Обратите внимание, что выделенная сторона будет всегда повернута в сторону Земли.

Лабораторная работа №3. Формирование параллельной и перспективной проекций трехмерной сцены

Цель: Научиться строить параметрически заданные каркасные модели простейших трехмерных объектов, и получать их перспективные и параллельные проекции на экране.

Задача: На основе исходных параметров построить каркасную модель трехмерного объекта, и визуализировать полученную сцену, используя параллельное и перспективное проецирования.

Результат: программа формирования визуального представления заданной трехмерной сцены. Отчет в печатном виде, оформленный в соответствии с приложением 2.

Системы координат, используемые в трехмерном моделировании

В приложениях трехмерной графики, в зависимости от решаемых задач, используются как минимум двумя системами координат. Для задания координат точек в трехмерном пространстве модели «мира» будем использовать общепринятую в векторной алгебре правую систему координат (рис. 3.1 а). При этом, если смотреть со стороны положительной полуоси в центр координат, то поворот на +90° (против часовой стрелке) переводит одну положительную ось в другую (направление движения расположенного вдоль оси и поворачивающегося против часовой стрелки правого винта и положительной полуоси совпадают).

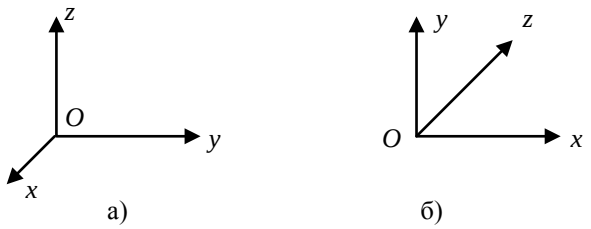


Рисунок 3.1. Правая (а) и левая (б) системы координат.

Для представления координат точек в системе связанной с плоскостью экрана, будет использоваться левая система координат (см. рис. 3.1 б). При

этом плоскость экрана совмещается с плоскостью XY, а ось z направлена в глубь экрана. В левой системе координат положительными будут повороты по часовой стрелке, если смотреть с положительного конца полуоси.

Геометрические преобразования в пространстве.

Как и на плоскости, так и в трехмерном пространстве любое преобразование координат точек или изменение системы координат можно представить в виде совокупности элементарных преобразований переноса, поворота относительно координатных осей и масштабирования. Матрицы для этих преобразований в системе однородных координат следующие:

1. Перенос точки на расстояния a, b, c, вдоль положительных направлений осей x, y, z соответственно, либо перенос начала системы координат на расстояния a, b, c, вдоль отрицательных направлений осей x, y, z соответственно:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a & b & c & 1 \end{bmatrix}.$$

2. Поворот точек плоскости на угол φ вокруг оси OZ, либо поворот осей системы координат вокруг оси OZ на угол $-\varphi$:

$$\mathbf{R_Z} = \begin{bmatrix} \cos\varphi & \sin\varphi & 0 & 0 \\ -\sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

3. Поворот точек плоскости на угол φ вокруг оси OX, либо поворот осей системы координат вокруг оси OX на угол $-\varphi$:

$$\mathbf{R_X} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & \sin\varphi & 0 \\ 0 & -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

4. Поворот точек плоскости на угол φ вокруг оси OY, либо поворот осей системы координат вокруг оси OY на угол $-\varphi$:

$$\mathbf{R_Y} = \begin{bmatrix} \cos\varphi & 0 & -\sin\varphi & 0 \\ 0 & 1 & 0 & 0 \\ \sin\varphi & 0 & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

5. Масштабирование координат точек плоскости относительно начала системы координат в S_X, S_Y, S_Z раз, по координатам x, y, z соответственно, либо базовых векторов $\bar{\mathbf{i}}, \bar{\mathbf{j}}, \bar{\mathbf{k}}$ в $1/S_X, 1/S_Y, 1/S_Z$ раз соответственно:

$$\mathbf{S} = \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Видовое преобразование.

В данной работе используется представление объектов сцены в виде каркасных (проволочных моделей). При этом каждый объект сцены задается наборами своих вершин и ребер. Вершины задаются своими трехмерными координатами в системе модели «мира». Будем называть такую систему **мировой**, а координаты - **мировыми**. Каждое ребро объекта будет определяться двумя вершинами, т.е. представлять собой отрезок прямой линии, соединяющий две вершины рис. 3.2.

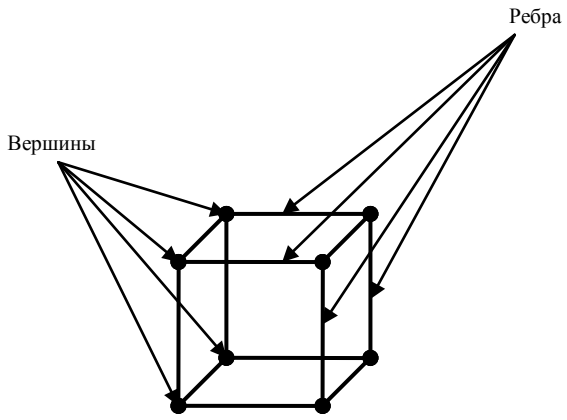


Рисунок 3.2. Вершины и ребра каркасной модели объекта

Для того чтобы получить изображение объекта, как он выглядит при наблюдении из некоторой точки, необходимо знать:

1. Координаты точки наблюдения в мировой системе координат.

2. Вектор, указывающий направление наблюдения.

3. Плоскость, на которой формируется изображение (плоскость экрана). Эту плоскость можно задать различными способами, например две пересекающихся прямых, три принадлежащие плоскости неколлинеарные точки, три взаимно перпендикулярных вектора и т. п.

В нашем случае для простоты примем следующее рис. 3.3:

1. Начало мировой системы координат находится в центре объекта.

2. Вектор наблюдения направлен из точки наблюдения E в начало мировой системы координат.

3. Плоскость экрана перпендикулярна вектору наблюдения, и этот вектор, пересекает ее в центре экрана.

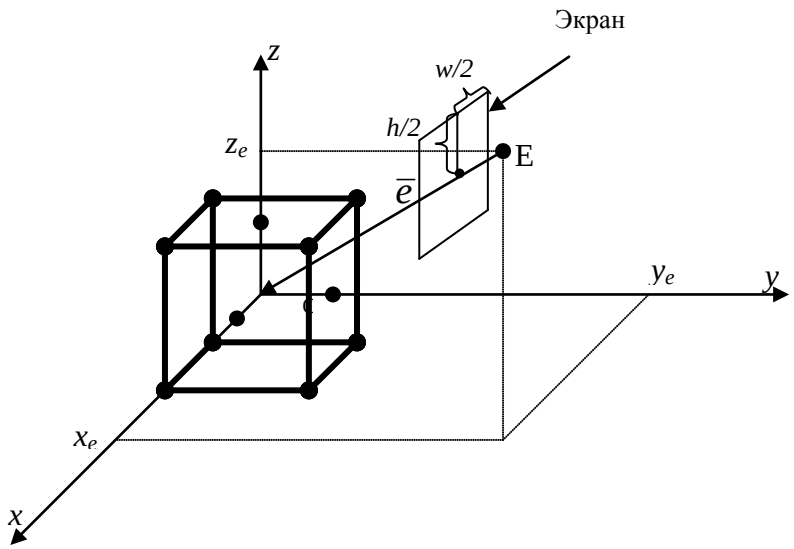


Рисунок 3.3. Расположение объектов сцены.

Чтобы получить изображение объекта выполним следующие действия:

1. Повернем мировую систему координат вокруг осей *z* и *y* так, чтобы ось *x* была направлена в точку E.

2. Перенесем начало этой системы координат в точку E.

3. Поменяем направление оси *x* на обратное.

4. Переименуем ось *x* в *z*, *y* в *x*, *z* в *y*.

5. Разместим экран на расстоянии *d* от точки наблюдения по новой оси *z*, так, чтобы ось *x* экрана была параллельна оси x новой системы координат, а ось *y* экрана – оси y новой системы координат.

Совокупность этих преобразований называется **видовым преобразованием**, а новая система координат – **видовой системой координат**.

Если теперь пересчитать координаты вершин объекта в видовой системе координат, и нарисовать на экране сцену используя видовые координаты *x* и *y*, игнорируя *z*, то мы получим изображение параллельной проекции сцены, как она выглядит при наблюдении из точки E.

Рассмотрим, каким образом можно выполнить видовое преобразование.

Будем задавать наше преобразование матрицей 4x4, которая получается путем перемножения следующих 4-х матриц элементарных преобразований:

Матрица $\mathbf{R}^{-1}{}_Z$ поворота системы координат на угол θ вокруг оси *z*

рис. 3.4.

Матрица $\mathbf{R}^{-1}{}_Y$ поворота системы координат на угол $\pi/2-\varphi$ вокруг оси *y*.

Матрица $\mathbf{T}^{-1}{}_E$ переноса начала системы координат в точку E.

Матрица **S** преобразования осей системы координат.

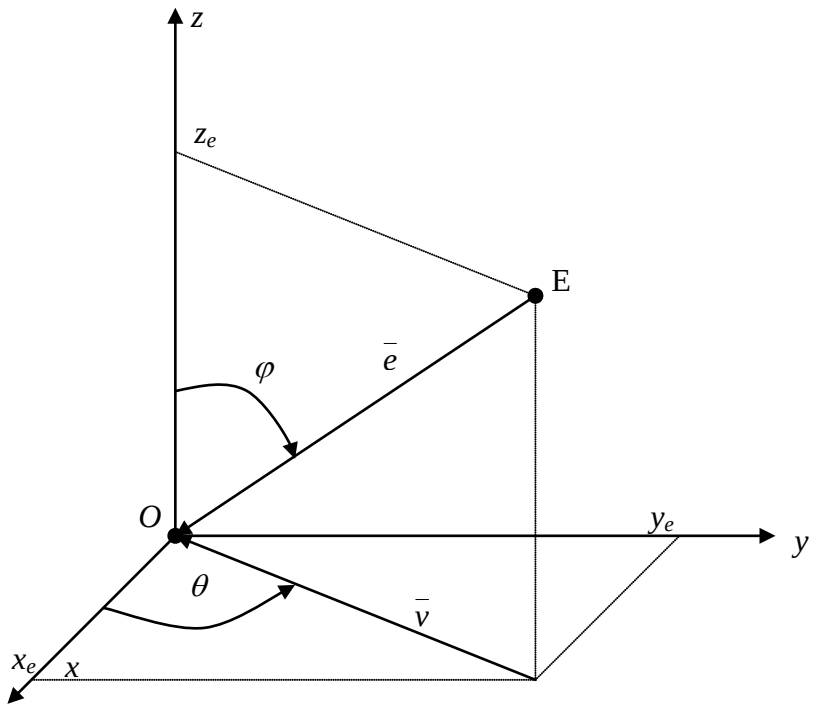


Рисунок 3.4. Система координат до видового преобразования.

Матрица поворота системы координат вокруг оси z на угол θ будет следующей:

$$\mathbf{R}_{\mathbf{Z}}^{-1} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Как видно из рис. 3: $\cos\theta = \frac{x_e}{|\bar{v}|}$, а, $\sin\theta = \frac{y_e}{|\bar{v}|}$, где $|\bar{v}| = \sqrt{x_e^2 + y_e^2}$

После этого поворота оси координат займут положение как на рис. 3.5.

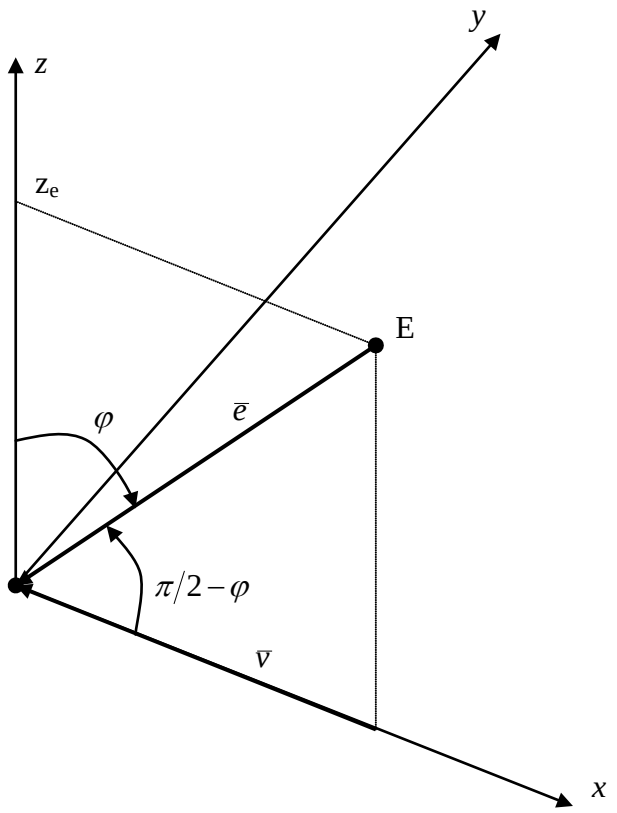


Рис 3.5. Система координат после поворота вокруг оси z.

Теперь видно, чтобы ось x была направлена в точку E необходимо повернуть систему координат вокруг оси y на угол $\frac{\pi}{2} - \varphi$. Матрица такого преобразования следующая:

$$\mathbf{R}_Y^{-1} = \begin{bmatrix} \cos\frac{\pi}{2}-\varphi & 0 & \sin\frac{\pi}{2}-\varphi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\frac{\pi}{2}-\varphi & 0 & \cos\frac{\pi}{2}-\varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \sin\varphi & 0 & \cos\varphi & 0 \\ 0 & 1 & 0 & 0 \\ -\cos\varphi & 0 & \sin\varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Как видно из рис. 3.5, $\cos\varphi = \frac{z_e}{|\bar{e}|}$, а $\sin\varphi = \frac{|\bar{v}|}{|\bar{e}|}$,

где $|\bar{v}| = \sqrt{x_e^2 + y_e^2}$, а $|\bar{e}| = \sqrt{x_e^2 + y_e^2 + z_e^2}$.

Сейчас наша система координат выглядит следующим образом рис. 3.6.

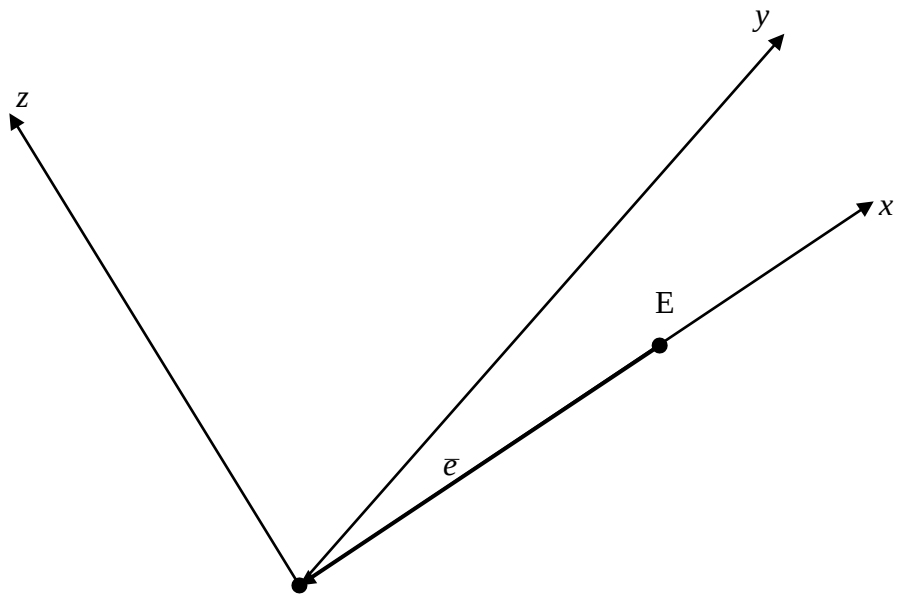


Рисунок 3.6. Система координат после поворота вокруг оси y.

Перенесем начало системы координат в точку E. Матрица для этого преобразования следующая:

$$\mathbf{T}_\mathbf{E}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -|\bar{e}| & 0 & 0 & 1 \end{bmatrix}, \text{ где } |\bar{e}| = \sqrt{x_e^2 + y_e^2 + z_e^2}.$$

После этого преобразования получим систему координат как на рис. 3.7.

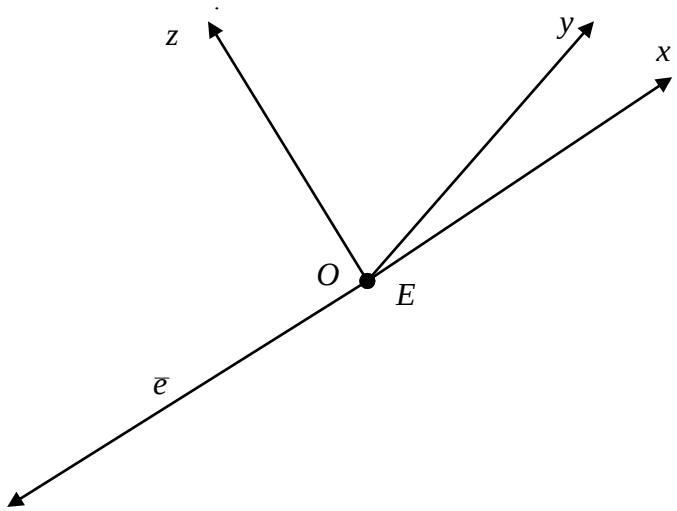


Рисунок 3.7. Положение координатных осей после переноса начала координат в точку E.

Сейчас необходимо изменить направление оси x на противоположное, и переименовать оси системы координат, для этого воспользуемся следующей матрицей:

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

Окончательное положение осей видовой системы координат получим следующее рис. 3.8.

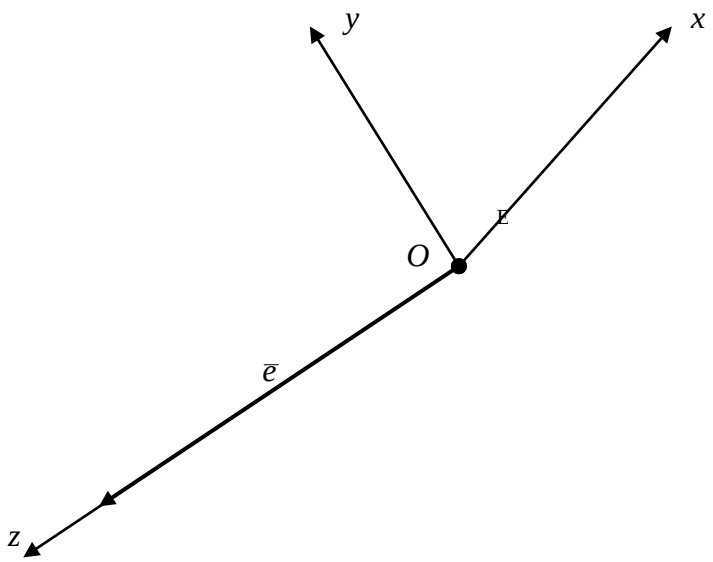


Рисунок 3.8. Окончательное положение осей видовой системы координат.

Перспективное преобразование.

Если применить полученную матрицу преобразования к координатам вершин объекта, то мы получим координаты вершин объекта в видовой системе координат. Если выполнить построение изображения объекта, используя только координаты *x* и *y*, мы получим параллельную проекцию нашего объекта на плоскости экрана. Для получения перспективной проекции объекта, мы должны рассчитать новые координаты *x* и *y* вершин объекта с учетом их расстояния до точки наблюдения. Эффект перспективы проявляется в кажущемся сокращении расстояний между точками при их движении от наблюдателя рис. 3.9.

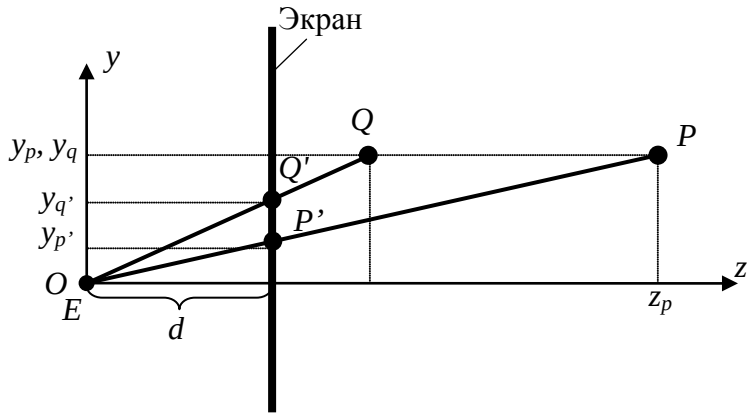


Рисунок 3.9. Получение перспективной проекции.

Рассмотрим треугольники *EPZp* и *EP'd*. Эти треугольники подобны, поэтому $\frac{d}{z_p} = \frac{y_{p'}}{y_p}$. Отсюда $y_{p'} = \frac{dy_p}{z_p}$.

Таким образом, если точка P задана своими видовыми координатами *x, y, z* , то координаты *x'* и *y'* ее центральной проекции на плоскость экрана вычисляются следующим образом:

$$\begin{cases} x' = \dfrac{d \bullet x}{z} \\ y_{p'} = \dfrac{d \bullet y}{z} \end{cases}$$

где *d* – расстояние от точки наблюдения до экрана.

Последнее, что нам необходимо сделать, это пересчитать полученные координаты проекций точек в системе экранных координат. Экранные

координаты *Sx* и *Sy* рассчитываются следующим образом:

$$\begin{cases} Sx = x' + \dfrac{W}{2} \\ Sy = \dfrac{H}{2} - y' \end{cases}$$, где W – ширина экрана, а H его высота.

Задания к лабораторной работе № 3

Перед выдачей заданий, студентов необходимо поделить на пары. Если количество студентов нечетное, то оставшемуся без пары студенту предлагается выполнять вариант №1.

В данной работе необходимо получить на экране компьютера параллельную и перспективную проекции трехмерной сцены. Объектом отображения является трехмерное выпуклое тело, задаваемое в виде каркасной модели, т.е. в виде совокупности координат вершин объекта в трехмерном пространстве, и отрезков прямых линий (ребер), соединяющих эти вершины рис. 3.10.

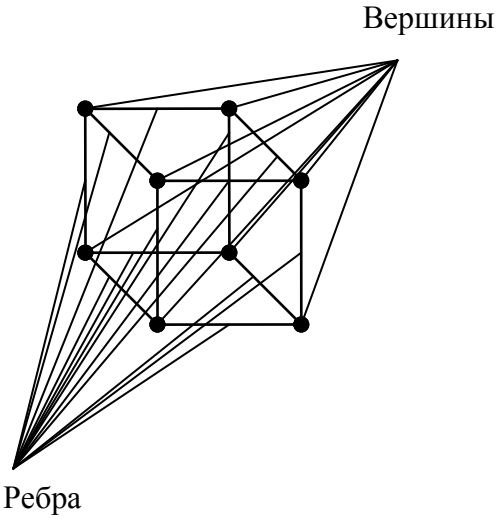


Рисунок 3.10. Каркасная модель куба.

Объекты задаются **параметрически**, т.е. вы должны в программе рассчитать координаты вершин объекта, и построить списки ребер, для заданных значений основных параметров объекта и его расположения относительно мировой системы координат. Например, объектом является куб, со стороной равной *a* (конкретное значение задается пользователем). Центр

симметрии куба находится в точке *O* начала системы координат. Ребра куба параллельны соответствующим осям координат. На основе этой информации можно построить объект, показанный на рис. 3.11.

Однако для построения проекции трехмерной сцены, этой информации недостаточно. Как минимум необходимо знать координаты точки наблюдения в мировой системе координат, вектор направления наблюдения, и расположение системы координат экрана компьютера относительно мировой системы координат. Все вышесказанное поясняет рис. 3.12. Здесь нужно отметить, что координаты точки наблюдения задает пользователь.

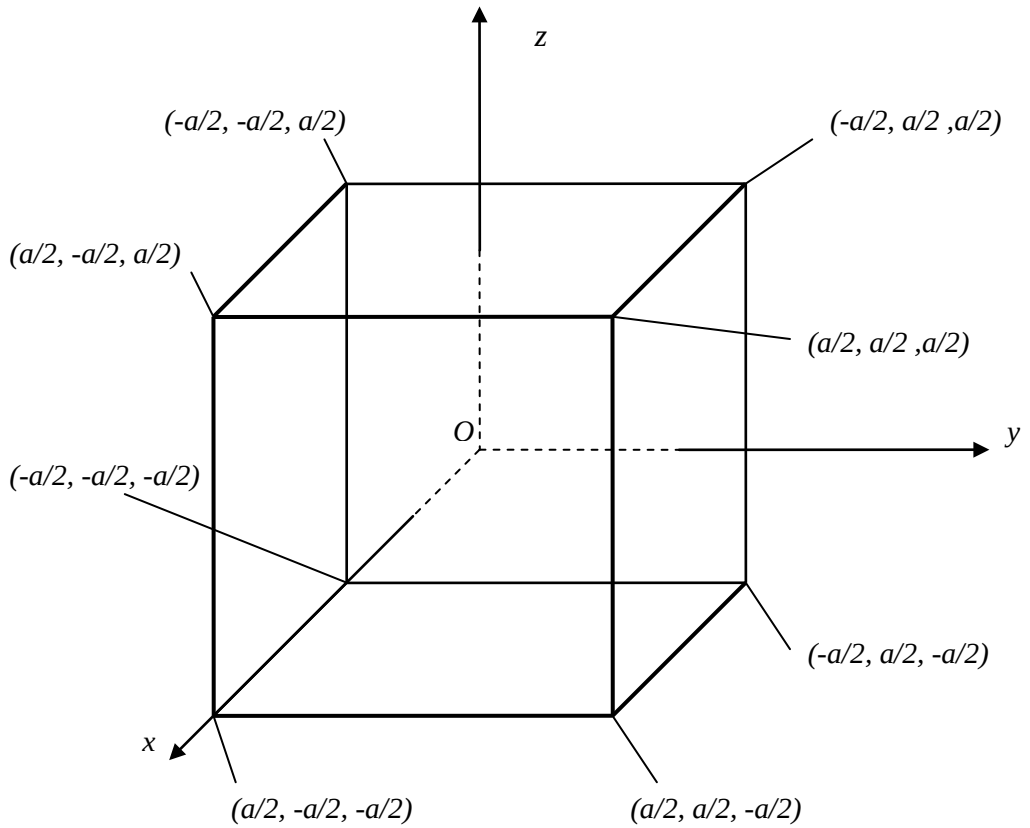


Рисунок 3.11. Параметрически заданный куб.

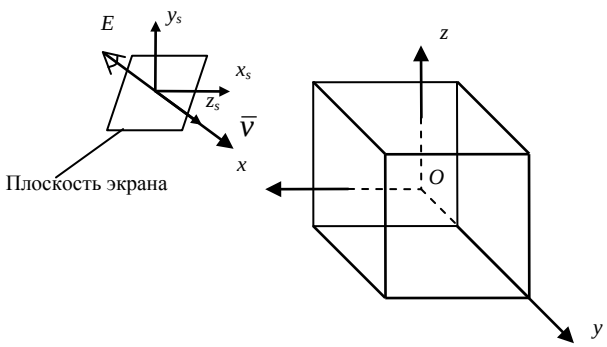


Рисунок 3.12. Получение проекции трехмерной сцены.

Вариант № 1.

Объектом является цилиндр, аппроксимируемый правильной *n*-гранной призмой рис.3.13. Основания цилиндра параллельны плоскости *xy*. Нижнее основание лежит в плоскости *xy*. Центр нижнего основания совпадает с точкой *O* начала мировой системы координат. Число граней призмы *n*, радиус описанной вокруг основания окружности *r*, высота призмы *h*, координаты точки наблюдения *E* и расстояние до экрана *d* задаются пользователем. Вектор наблюдения $\bar{v}$ направлен в точку *O*. Необходимо построить параллельную и перспективную проекции данной сцены.

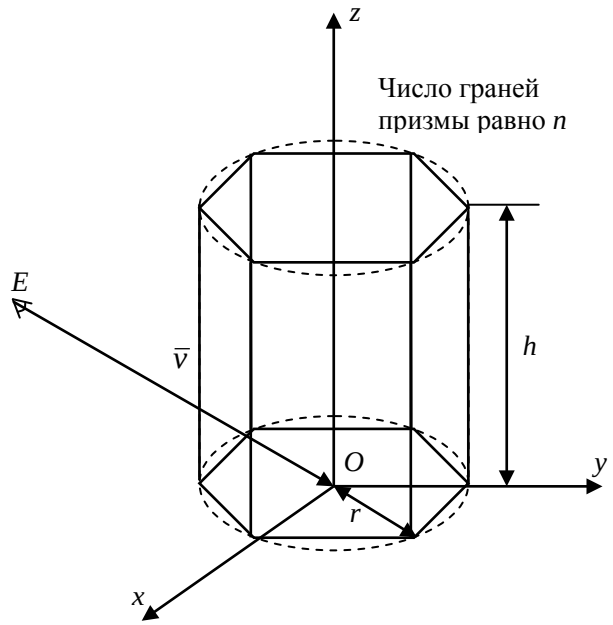


Рисунок 3.13. Цилиндр. Вариант №1.

Вариант № 2.

Объектом является цилиндр, аппроксимируемый правильной n-гранной призмой рис. 3.14. Основания цилиндра параллельны плоскости *xy*. Нижнее основание лежит в плоскости *xy*. Центр нижнего основания совпадает с точкой *O* начала мировой системы координат. Число граней призмы *n*, длина стороны основания призмы *a*, высота призмы *h*, координаты точки наблюдения *E* и расстояние до экрана *d* задаются пользователем. Вектор наблюдения $\bar{v}$ направлен в точку *O*. Необходимо построить параллельную и перспективную проекции данной сцены.

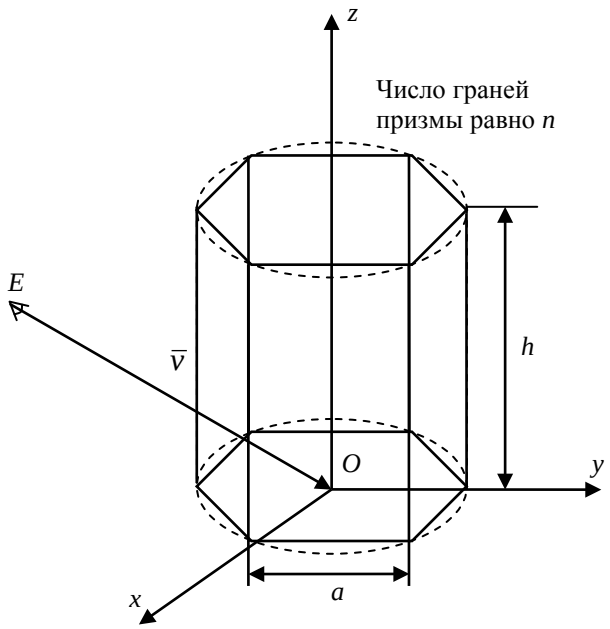


Рисунок 3.14. Цилиндр. Вариант №2.

Вариант № 3.

Объектом является конус, аппроксимируемый правильной *n*-гранной пирамидой рис. 4.15. Основание конуса лежит в плоскости *xy*. Центр основания совпадает с точкой *O* начала мировой системы координат. Число граней пирамиды *n*, радиус описанной вокруг основания окружности *r*, высота пирамиды *h*, координаты точки наблюдения *E* и расстояние до экрана *d* задаются пользователем. Вектор наблюдения $\bar{v}$ направлен в точку *O*. Необходимо построить параллельную и перспективную проекции данной сцены.

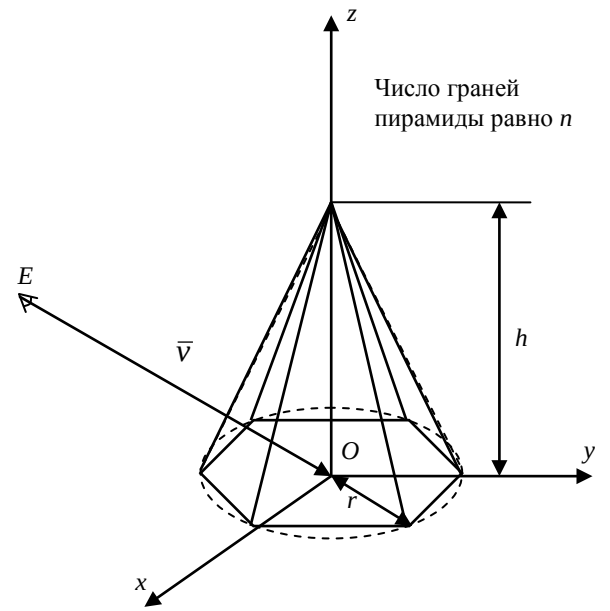


Рисунок 3.15. Конус. Вариант №3.

Вариант № 4.

Объектом является конус, аппроксимируемый правильной *n*-гранной пирамидой рис. 4.16. Основание пирамиды лежит в плоскости *xy*. Центр основания совпадает с точкой *O* начала мировой системы координат. Число граней пирамиды *n*, длина стороны основания пирамиды *a*, высота пирамиды *h*, координаты точки наблюдения *E* и расстояние до экрана *d* задаются пользователем. Вектор наблюдения $\bar{v}$ направлен в точку *O*. Необходимо построить параллельную и перспективную проекции данной сцены.

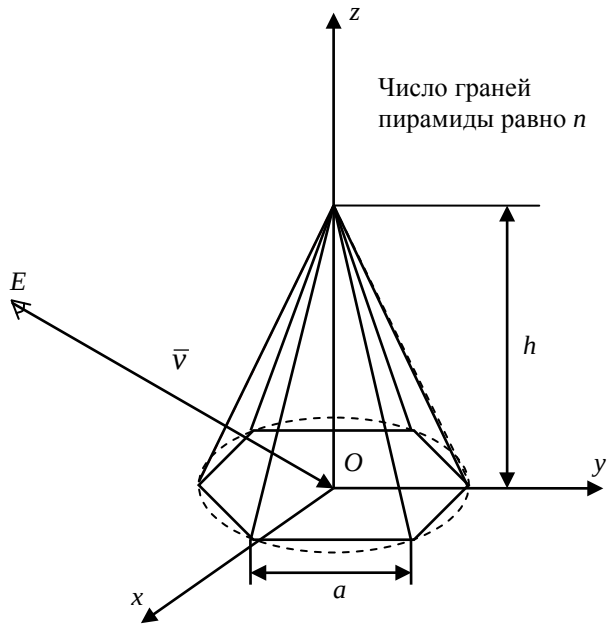


Рисунок 3.16. Конус. Вариант №4.

Вариант № 5.

Объектом является конус, аппроксимируемый правильной *n*-гранной пирамидой рис. 3.17. Основание конуса лежит в плоскости *xy*. Центр основания совпадает с точкой *O* начала мировой системы координат. Число граней пирамиды *n*, радиус описанной вокруг основания окружности *r*, угол раскрыва конуса β, координаты точки наблюдения *E* и расстояние до экрана *d* задаются пользователем. Вектор наблюдения $\bar{v}$ направлен в точку *O*. Необходимо построить параллельную и перспективную проекции данной сцены.

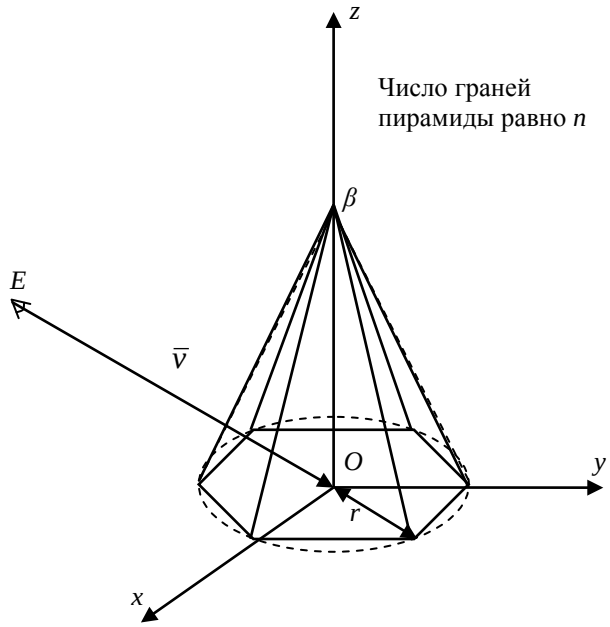


Рисунок 3.17. Конус. Вариант №5.

Вариант № 6.

Объектом является конус, аппроксимируемый правильной n-гранной пирамидой рис. 3.18. Основание пирамиды лежит в плоскости xy. Центр основания совпадает с точкой O начала мировой системы координат. Число граней пирамиды n, длина стороны основания пирамиды a, угол раскрыва конуса β, координаты точки наблюдения E и расстояние до экрана d задаются пользователем. Вектор наблюдения $\bar{v}$ направлен в точку O. Необходимо построить параллельную и перспективную проекции данной сцены.

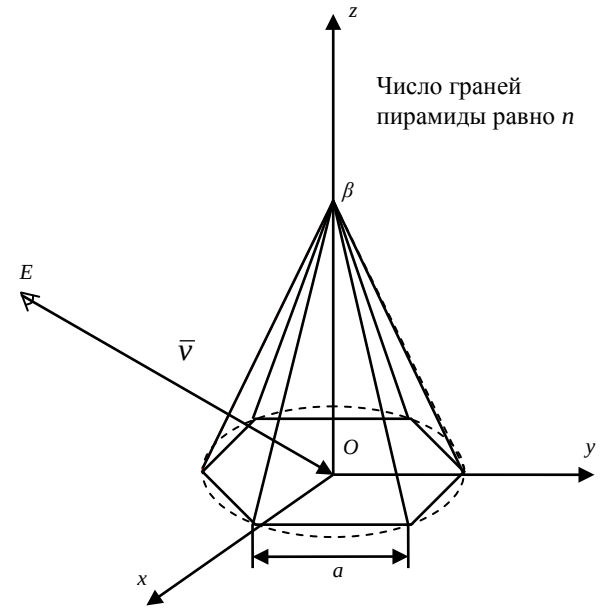


Рисунок 3.18. Конус. Вариант №6.

Вариант № 7.

Объектом является сфера, аппроксимируемая многогранником рис. 3.19. Центр сферы совпадает с точкой *O* начала мировой системы координат. Число параллелей *n*, число меридианов *m*, радиус сферы *r*, координаты точки наблюдения *E* и расстояние до экрана *d* задаются пользователем. Вектор наблюдения $\bar{v}$ направлен в точку *O*. Необходимо построить параллельную и перспективную проекции данной сцены.

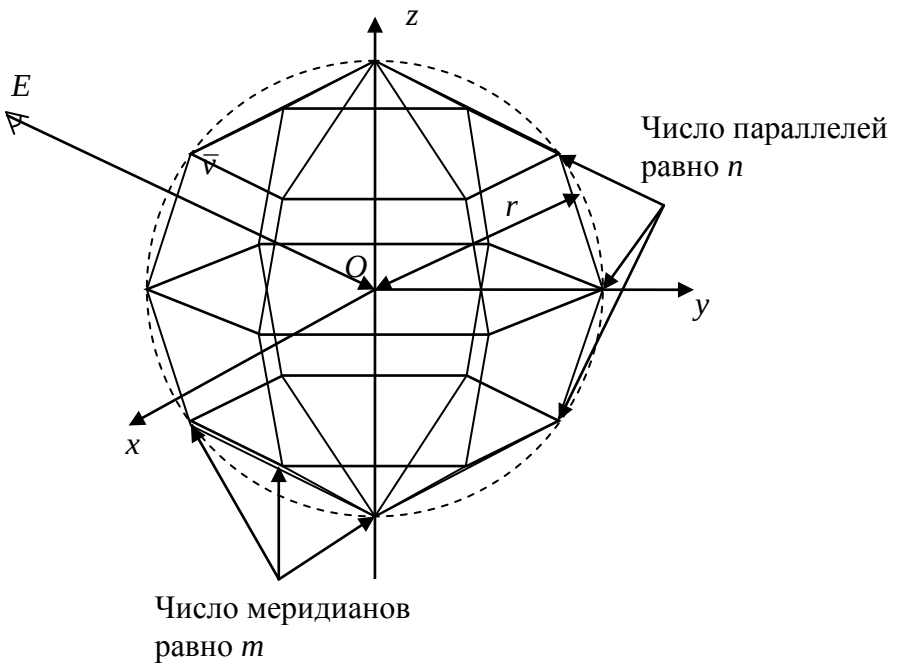


Рисунок3.19. Сфера. Вариант №7.

Вариант № 8.

Объектом является сфера, аппроксимируемая многогранником рис. 3.20. Центр сферы совпадает с точкой *O* начала мировой системы координат. Число параллелей *n*, число меридианов *m*, длина стороны меридиана *a*, координаты точки наблюдения *E* и расстояние до экрана *d* задаются пользователем. Вектор наблюдения $\bar{v}$ направлен в точку *O*. Необходимо построить параллельную и перспективную проекции данной сцены.

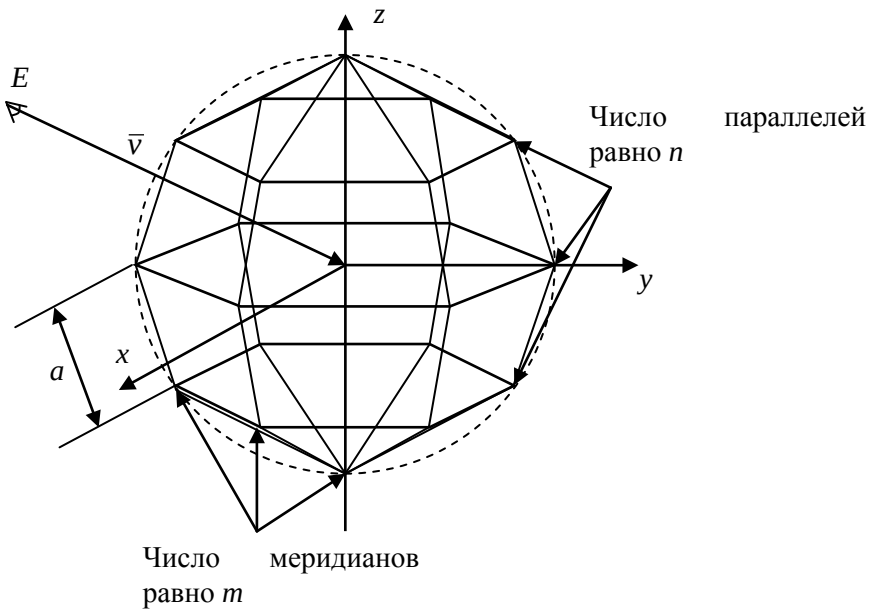


Рисунок 3.20. Сфера. Вариант №8.

Вариант № 9.

Объектом является полусфера, аппроксимируемая многогранником рис. 3.21. Основание полусферы лежит в плоскости *xy*. Центр основания совпадает с точкой *O* начала мировой системы координат. Число параллелей *n*, число меридианов *m*, радиус основания полусферы *r*, координаты точки наблюдения *E* и расстояние до экрана *d* задаются пользователем. Вектор наблюдения $\bar{v}$ направлен в точку *O*. Необходимо построить параллельную и перспективную проекции данной сцены.

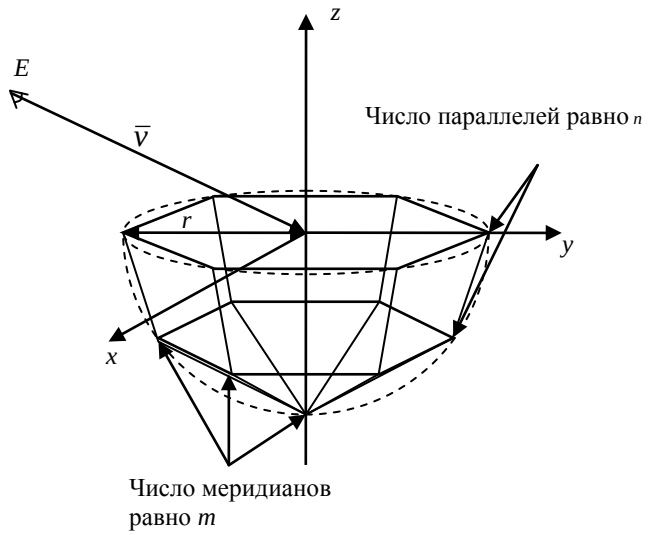


Рисунок 3.21. Полусфера. Вариант №9.

Вариант № 10.

Объектом является икосаэдр (рис. 3.22). Центр икосаэдра совпадает с точкой *O* начала мировой системы координат. Северный полюс икосаэдра находится на оси *Z*. Координаты точки наблюдения *E* и размер стороны икосаэдра *a* задаются пользователем. Вектор наблюдения $\bar{v}$ направлен в точку *O*. Необходимо построить параллельную и перспективную проекции данной сцены. Для расчета координат вершин икосаэдра прочитайте Приложение 1: «Построение правильных многогранников».

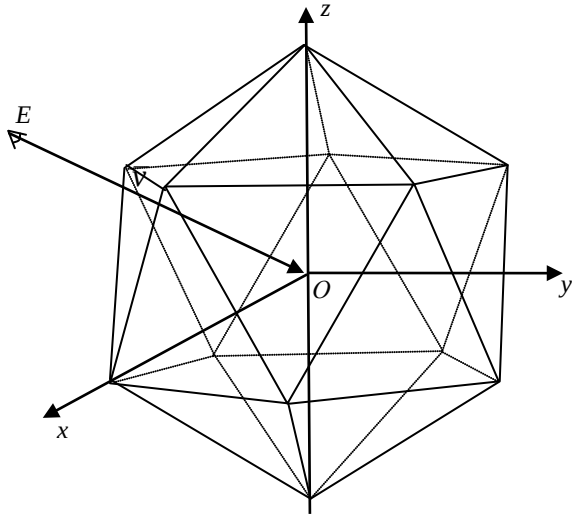


Рисунок 3.22. Икосаэдр. Вариант №10.

Лабораторная работа №4. Удаление невидимых поверхностей с помощью алгоритма Z-буфер

Цель: Научиться строить параметрически заданные полигональные поверхностные модели простейших трехмерных объектов, и осуществлять их визуализацию с удалением невидимых поверхностей с помощью алгоритма Z-буфер.

Задача: На основе исходных параметров построить полигональную поверхностную модель трехмерного объекта, и визуализировать полученную сцену, используя параллельное и перспективное проецирования и удаление невидимых поверхностей с помощью алгоритма Z-буфер.

Результат: программа формирования визуального представления заданной трехмерной сцены. Отчет в печатном виде, оформленный в соответствии с приложением 2.

Прежде чем рассматривать сам алгоритм Z-буфер, необходимо научиться рисовать на экране произвольные закрашенные треугольники, и при этом иметь возможность управлять цветом каждого пикселя закраски. Другими словами, необходим алгоритм попиксельной закраски произвольного треугольника. Рассмотрим два таких алгоритма. Первый алгоритм является очень простым для программной реализации, однако он использует арифметические операции с плавающей точкой, что сказывается на его быстродействии. Второй алгоритм более сложен для понимания и программной реализации, но в нем используются только целые числа и целочисленная арифметика без операций умножения и деления. За счет этого мы получаем значительный выигрыш в быстродействии этого алгоритма.

Закраска произвольного треугольника, (алгоритм 1)

Рассмотрим изображение треугольника на экране. Оно представляет собой множество точек, упорядоченных по вертикали и горизонтали рис. 4.1.

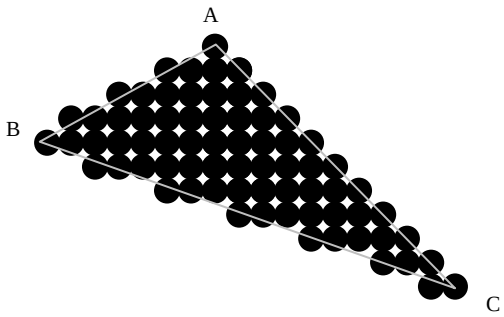


Рис 4.1. Попиксельное представление закрашенного треугольника

Будем отрисовывать треугольник последовательно, точка за точкой сверху вниз и слева направо. Для этого надо пройти по всем строкам экрана, которые пересекают треугольник, (т.е от минимального до максимального значения *y*-координат вершин треугольника) и нарисовать соответствующие горизонтальные отрезки (рис. 4.2).

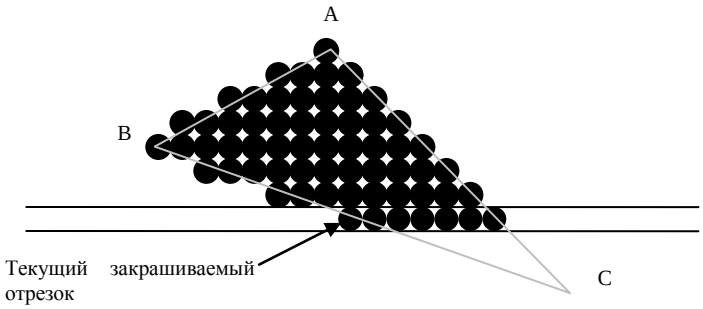


Рисунок 4.2. Последовательная закраска треугольника.

Для этого, сначала отсортируем вершины треугольника так, чтобы A была верхней а C – нижней. Тогда, $y_{\min} = Ay$, а $y_{\max} = Cy$, и нам нужно пройтись по всем строкам от Ay до Cy. Рассмотрим некоторую строку $sy, Ay \le sy \le Cy$. Если $sy < By,$ то эта строка пересекает стороны AB и AC. Если $By \le sy \le Cy,$ то строка пересекает BC и AC.

Поскольку нам известны координаты всех вершин треугольника, мы можем, написав уравнения прямых, содержащих стороны треугольника, найти точки пересечения прямой $y = sy$ со сторонами треугольника. Нарисуем горизонтальный отрезок, соединяющий эти точки, и повторим для всех строк от Ay до Cy. Так мы нарисуем весь треугольник.

Рассмотрим подробно нахождение точки пересечения прямой $y = sy$ и стороны треугольника, например AB. Для этого напишем уравнение прямой AB в виде $x = ky + b$:

$$x = \frac{Bx - Ax}{By - Ay}(y - Ay) + Ax$$

Подставим в него известное значение $y = sy$:

$$x = \frac{Bx - Ax}{By - Ay}(sy - Ay) + Ax$$

Для других сторон точки пересечения ищутся аналогично. Так для BC:

$$x = \frac{Cx - Bx}{Cy - By}(sy - By) + Bx$$

Для AC:

$$x = \frac{Cx - Ax}{Cy - Ay}(sy - Ay) + Ax$$

Необходимо отслеживать случаи $By = Cy$, $By = Ay$ и $Ay = Cy$, поскольку будет происходить деление на 0.

Закраска произвольного треугольника (алгоритм 2)

Рассмотренный ранее алгоритм позволяет закрасить любой треугольник, однако его быстродействие оставляет желать лучшего. В этом повинны как поточечное рисование на поверхности окна (поскольку оно чрезвычайно медленное), так и использование операций с плавающей точкой над вещественными числами. От поточечного рисования нам избавиться не удастся, поскольку необходимо иметь возможность задавать значение цвета для каждой точки треугольника. Однако API Windows предлагает метод, при использовании которого, поточечное рисование может быть ускорено в несколько раз.

Суть его заключается в том, чтобы создать в памяти битовую карту изображения, нарисовать треугольник в ней, и потом эту карту отобразить на поверхности окна. Поскольку рисование будет производиться в оперативной памяти, а не непосредственно на экране, то быстродействие алгоритма существенно возрастает. Рассмотрим, каким образом это реализуется в Borland C++ Builder.

Все необходимые средства для работы с битовой картой предоставляет класс TBitmap. Прежде чем начать работать с битовой картой, необходимо создать его экземпляр, и задать размеры битовой карты, и формат пикселей (т.е. глубину цвета). Для этого необходимо сделать следующие действия:

```
  Graphics::TBitmap *pBitmap=new Graphics::TBitmap();
//Создаем экземпляр TBitmap
  pBitmap->Width=MainForm->ClientWidth; //Задаем ширину
битовой карты
  pBitmap->Height=MainForm->ClientHeight; //Задаем высоту
битовой карты
  pBitmap->PixelFormat=pf32bit; //Устанавливаем глубину
цвета равную 32 бита.
```

Поточечное рисование на битовой карте осуществляется с помощью строк сканирования. Вся битовая карта представляется как одномерный массив горизонтальных строк, называемых строками сканирования. Каждая из этих строк представляет собой одномерный массив, каждый элемент которого для глубины цвета 32 бита, содержит значение цвета соответствующей точки. Это

значение представляет собой 32 разрядное целое число, в котором первый байт задает интенсивность синего цвета, второй байт – зеленого, третий байт – красного, а четвертый – значение прозрачности (канал альфа).

Чтобы работать со строкой сканирования с глубиной цвета 32 бита необходимо:

1. Завести переменную – указатель на целое число (`int *scl;`).
2. Получить необходимую строку сканирования (`scl=(int*)pBitmap->ScanLine[i];`
   где *i* – *y*-координата соответствующей строки);
3. Задать новые значения цветов точкам строки сканирования (`scl[j]=NewColor;`
   где *j* – *x*-координата необходимой точки, а *NewColor* – новое значение цвета этой точки типа **int**).

После того, как изображение будет полностью сформировано на битовой карте, его необходимо перенести на поверхность окна. Для этого необходимо использовать функцию **Draw** свойства **Canvas** окна.

```
Canvas->Draw(0,0,pBitmap);
```

У этой функции первые два параметра представляют собой координату верхней левой точки окна, с которой будет воспроизводиться изображение с битовой карты, указатель на которую передается третьим параметром. Эта операция приводит к практически мгновенному появлению изображения на поверхности окна.

После того, как необходимость в битовой карте пропадет, необходимо уничтожить ее экземпляр.

```
delete pBitmap;
```

Кроме использования битовых карт, можно дополнительно увеличить быстродействие алгоритма, отказавшись от использования вещественных чисел и операций с плавающей точкой.

Оказывается, можно разработать другой алгоритм закраски треугольника, в котором используются только целые числа, и отсутствует операция деления.

Сначала представим стороны треугольника в виде отрезков прямых линий (см. рис. 4.3.).

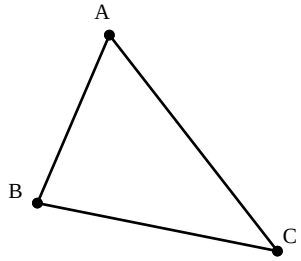


Рисунок 4.3. Представление сторон треугольника в виде набора отрезков.

Поскольку изображение на экране состоит из совокупности точек, то и изображения этих отрезков также будет состоять из точек.

Если рассмотреть с увеличением окрестность вершины A, можно увидеть примерно следующую картину (см. рис.4.4.).

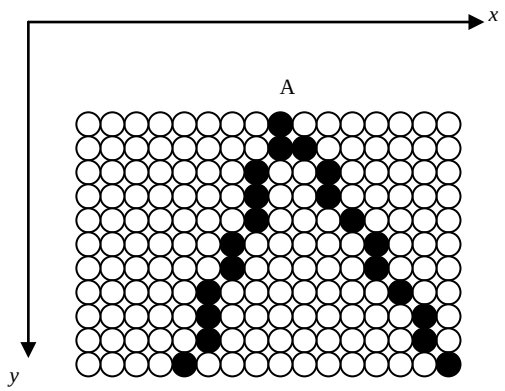


Рисунок 4.4. Вид окрестности точки A.

Значит, если мы сможем разработать алгоритм последовательного нахождения координат точек отрезков A B, A C, B C, то сможем нарисовать закрашенный треугольник просто рисуя горизонтальные отрезки соединяющие точки двух сторон с одинаковыми координатами *y* (см. рис. 4.5.).

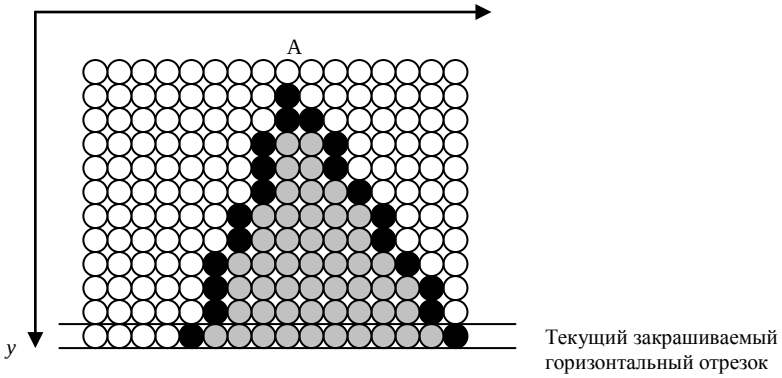


Рисунок 4.5. Попиксельная закраска треугольника.

Будем разрабатывать алгоритм нахождения координат точек между A и C использующий вещественные числа, но таким образом, что это ограничение потом можно будет легко устранить, и использовать только целочисленные переменные.

Вначале зададим *x=AX* и *y=AY*. В цикле будем задавать приращение 1 для переменной *y* и будем оставлять *x* либо неизменным, либо также увеличивать на 1. При этом последний выбор будем осуществлять так, чтобы новая точка сетки (*x*, *y*) располагалась как можно ближе к прямой линии, проходящей через точки A и C (см. рис.4.6.).

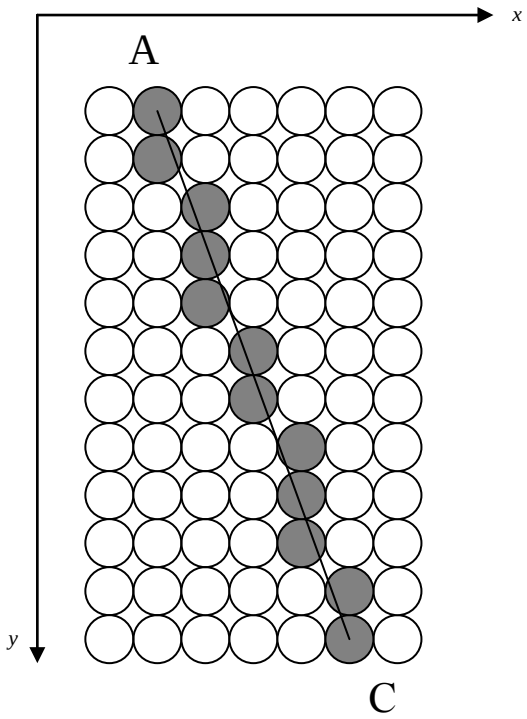


Рисунок 4.6. Выбор точек отрезка AC.

Это означает, что расстояние по горизонтали между новой выбранной точкой и этой линией не должно превышать значения 0.5.

Введем переменную *d* для обозначения этого расстояния. Потребуем, чтобы $-0.5 < d \le 0.5$.

Последнее неравенство обеспечивает условие для определения необходимости давать приращение переменной *x*.

Теперь можно написать первый вариант реализации функции нахождения координат точек между A и C:

```
{
  int x, y,x1;
  float t; //коэффициент наклона
  float d; //отклонение

  t=(float)(CX-AX)/(float)(CY-AY);
  d=0;
  x=AX;
```

```
    for (y=AY; y<=CY; y++)
    {
        x1=x; //координата x очередной точки на прямой
AC
        d+=t;
        if (d>0.5) {x++; d--;}
    }
  }
```

Отклонение, вначале устанавливается равным нулю и изменяется на каждом шаге цикла. Поскольку оно показывает, насколько левее точной прямой линии лежит вычисленная точка, то значение *d* увеличивается на значение наклона, если *y* увеличивается на 1 и *x* остается без изменения. Это условие не выполняется, если значение *d* превышает 0.5. В этот момент нужно увеличить значение *x* на 1. Соответственно, и отклонение *d* должно быть уменьшено на единицу.

Теперь посмотрим, как можно избавиться от вещественных значений переменных *t* и *d*.

Значение переменной *t* вычисляется следующим образом:

$$t = \frac{CX - AX}{CY - AY}$$

где числитель и знаменатель представляют собой целые числа.

Величина *d* вычисляется как конечная сумма элементов, каждый из которых равен либо *t*, либо -1. Поэтому *d* также можно записать в виде частного со знаменателем равным $CY - AY$.

Значит, можно перейти к целочисленным переменным *t* и *d*, путем умножения на значение знаменателя. Также просто избавиться от константы 0.5. Для этого нужно дополнительно умножить значение знаменателя на 2. Таким образом $t = 2lx$, где *lx=CX-AX*.

Теперь условный оператор в нашей функции можно заменить на:

```
if (d>ly) {x++; d-=ds;},
```

где *ds=2ly=2(CY-AY)*.

Таким образом, получаем новый вариант нашей функции:

```
{
   int x, y, t, d, lx, ly, ds;
   lx=CX-AX;
   ly=CY-AY;
   ds=ly << 1; //ds=2*ly;
   t=lx << 1; //t=2*lx;
   x=AX;
   for (y=AY; y<=CY; y++)
   {
       x1=x; //координата x очередной точки на прямой
AC
       if (d > ly) {x++;d -= ds;}
```

```
    }
}
```

Однако наша функция будет правильно работать только для случая, когда $lx \le ly$, и просто поменять местами *x* и *y* в противном случае нам не удастся (треугольник должен отрисовываться последовательно сверху – вниз).

Рассмотрим следующий рисунок:

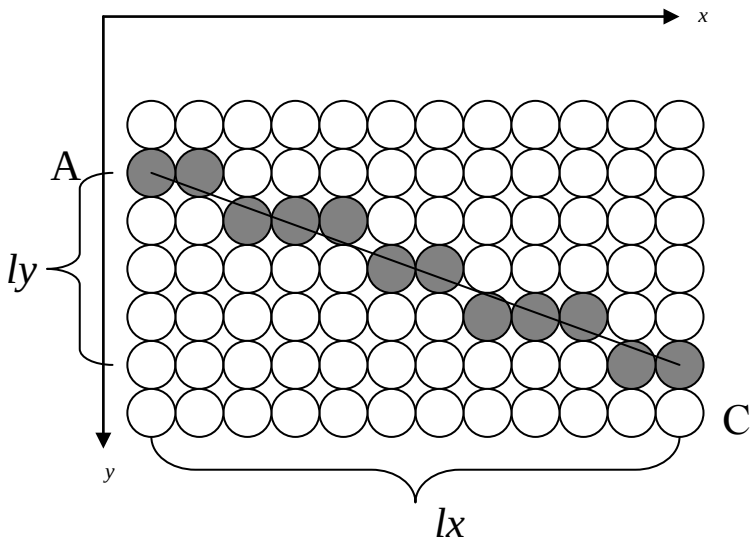


Рисунок 4.7. Случай, когда *lx>ly*.

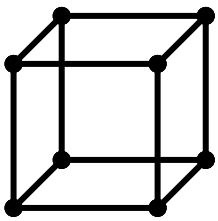
На нем хорошо видно, что в этом случае одному значению y может соответствовать несколько значений x. Но нас интересует из этих значений лишь максимальное. А это значит, что значение отклонения от прямой линии в этой точке будет максимальное, но не превысит 0.5. Применяя тот же подход что был рассмотрен выше, и эти соображения можно написать следующий вариант функции для данного случая:

```
{
   int x, y, t, d, lx, ly, ds;
   lx=CX-AX;
   ly=CY-AY;
   ds=ly << 1; //ds=2*ly;
   t=lx << 1; //t=2*lx;
   if (x!=CX)
       while (d > ly)
       {
           x++;
           d-=ds;
       }
   x1=x; // нашли координату x очередной точки
}
```

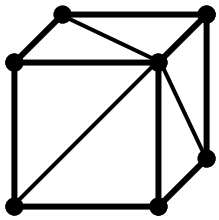
Для других сторон треугольника координаты точек прямых линий ищутся аналогично.

Поверхностная полигональная модель трехмерного объекта и удаление невидимых поверхностей

В данной лабораторной работе объект рассматривается как сплошная поверхность, состоящая из множества граней. Для простоты вычислений будем представлять каждую грань треугольником. Т.е. каждая грань задается координатами трех вершин треугольника. Треугольник выбран потому, что это выпуклая фигура, все точки которой всегда лежат в одной плоскости. Если нарисовать после видового и перспективного преобразования все грани объекта (все треугольники) по аналогии с рисованием каркасной модели, то результат окажется не тем, который ожидается. Вместо изображения объекта мы получим хаотическое нагромождение треугольников, в котором разглядеть объект будет крайне сложно. Это происходит потому, что теперь объект представляет собой сплошное тело, а это значит, что те грани, которые находятся ближе к наблюдателю, заслоняют собой грани находящиеся дальше от него рис. 4.8.б.



а) Каркасная модель



б) Поверхностная модель объекта с аппроксимацией поверхности треугольными гранями

Рисунок 4.8. Различные модели представления объекта.

Следовательно, нам необходимо в процессе рисования граней нарисовать только те точки поверхности граней, которые видны наблюдателю. Такая задача решается с помощью **алгоритмов удаления невидимых поверхностей.** В данной лабораторной работе необходимо использовать алгоритм под названием **Z-буфер.**

Алгоритм удаления невидимых поверхностей Z-буфер

Суть этого алгоритма заключается в построении рельефа по глубине наблюдаемой сцены, и использования этой информации для рисования только видимых точек поверхности объекта.

Информацию о глубине точек поверхности объекта несет координата *Z* этих точек, так как ось *Z* после преобразований у нас направлена в глубь экрана, и совпадает с направлением наблюдения.

Заведем двумерный массив (собственно Z-буфер) размером с экран, и заполним все его элементы каким-то большим числом, настолько большим, что координаты *z* для всех точек сцены заведомо меньше. Для каждой рисуемой точки подсчитаем значение ее координаты *z*. Если оно больше, чем значение в *Z*-буфере (точка закрыта какой-то другой точкой), или меньше, чем 0 (точка находится за камерой), то переходим к следующей точке. Если меньше, то рисуем точку на экране, а в *z*-буфер записываем значение координаты *z* текущей точки.

Имеет смысл считать значения не *z*, а $w = 1/z$. Тогда условия чуть изменяются – точка загорожена другой, если значение *w* меньше значения в *Z*-буфере; и точка находится за камерой, если w < 0. Буфер инициализируем нулями. Тогда не нужна проверка на положительность *w* - точка попадает в *Z*-буфер и на экран, только, если *w* больше текущего значения, и поэтому точки, для которых $w < 0$ в буфер никогда не попадут.

Для рисования граней объекта пригодны алгоритмы закрашивания по точкам многоугольников, рассмотренные выше. Единственное, что необходимо в них добавить, это работу с *Z*-буфером и расчет координаты *Z* для текущей точки поверхности треугольника по ее уже рассчитанным координатам *X* и *Y* и координатам трех вершин треугольника.

Расчет координаты Z текущей точки поверхности треугольника

Пусть треугольник задан координатами трех своих вершин *A*, *B* и *C*, причем:

$$\mathbf{A} = [x_1 \quad y_1 \quad z_1], \mathbf{B} = [x_2 \quad y_2 \quad z_2], \mathbf{C} = [x_3 \quad y_3 \quad z_3]/$$

Запишем уравнение плоскости проходящей через эти три точки в форме детерминанта

$$\begin{vmatrix} x & y & z & 1 \\ x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ x_3 & y_3 & z_3 & 1 \end{vmatrix} = 0$$

Здесь *x*, *y*, *z* – координаты текущей точки треугольника.

Отсюда:

$$x\begin{vmatrix} y1 & z1 & 1 \\ y2 & z2 & 1 \\ y3 & z3 & 1 \end{vmatrix} - y\begin{vmatrix} x1 & z1 & 1 \\ x2 & z2 & 1 \\ x3 & z3 & 1 \end{vmatrix} + z\begin{vmatrix} x1 & y1 & 1 \\ x2 & y2 & 1 \\ x3 & y3 & 1 \end{vmatrix} - \begin{vmatrix} x1 & y1 & z1 \\ x2 & y2 & z2 \\ x3 & y3 & z3 \end{vmatrix} = 0$$

Обозначим

$\begin{vmatrix} y1 & z1 & 1 \\ y2 & z2 & 1 \\ y3 & z3 & 1 \end{vmatrix}$ как a, $\begin{vmatrix} x1 & z1 & 1 \\ x2 & z2 & 1 \\ x3 & z3 & 1 \end{vmatrix}$ как b, $\begin{vmatrix} x1 & y1 & 1 \\ x2 & y2 & 1 \\ x3 & y3 & 1 \end{vmatrix}$ как c, $\begin{vmatrix} x1 & y1 & z1 \\ x2 & y2 & z2 \\ x3 & y3 & z3 \end{vmatrix}$ как d.

Тогда получим следующее уравнение

ax-by+cz-d=0

Отсюда находим координату z:

$$z = \frac{by - ax + d}{c},$$

или величину w:

$$w = \frac{1}{z} = \frac{c}{by - ax + d}$$

Задания к лабораторной работе №4

Перед выдачей заданий, студентов необходимо поделить на пары. Если количество студентов нечетное, то оставшемуся без пары студенту предлагается выполнять вариант №1. В данной работе необходимо получить на экране компьютера параллельную и перспективную проекции трехмерной сцены с удалением невидимых поверхностей с помощью алгоритма z-буфер.

Для каждого варианта задания необходимо взять трехмерную сцену из лабораторной работы №3, и визуализировать ее, используя алгоритм удаления невидимых поверхностей z-буфер.

Приложение 1. Построение правильных многогранников

Додекаэдр

Додекаэдр имеет 12 граней, 30 ребер и 20 вершин. Каждая из 12 граней является правильным

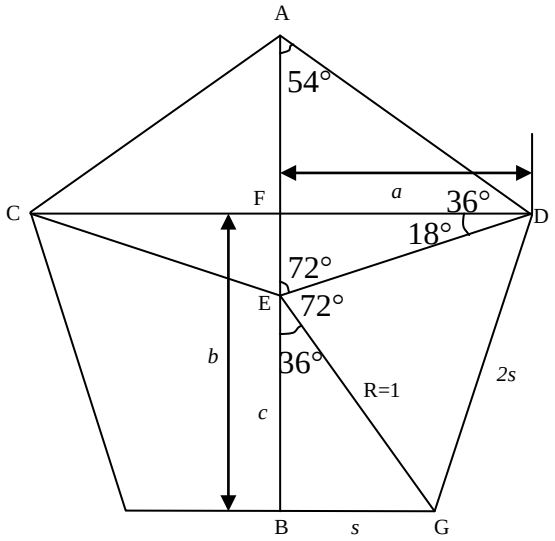


Рисунок 1. Пентагон

пентагоном, то есть пятиугольником с пятью равными сторонами. Додекаэдр вписывается в куб и это свойство можно использовать для его конструирования.

Перед использованием пятиугольника для построения додекаэдра рассмотрим сначала свойства самих пятиугольников. Один такой пятиугольник показан на рис. 1, его центр находится в точке E. Будем говорить, что все пять вершин пятиугольника лежат на окружности с единичным радиусом, центр которой расположен в точке E.

На рис. 1 показаны некоторые важные углы. Из них и из условия EG = R = 1 найдем

BE = cos 36° BG = sin 36°

Угол в 36° (=π/5 радиан) является одним из интереснейших, поскольку

2x36° = 72° 1/2x36° = 18° 72°+ 18° = 90°

Это позволяет записать значение cos 36° в виде очень простого выражения

$$\cos 36° = \left(1+\sqrt{5}\right)/4.$$

Доказать это выражение можно с помощью подстановки и = cos 72°, v = cos 36°.

Тогда будем иметь:

$$u = 2\cos^2 36 - 1 = 2v^2 - 1$$
$$v - 1 - 2\sin^2 18° = 1 - 2\cos^2 72° = 1 - 2u^2$$

Следовательно

$$u + v = 2(v^2 - u^2) = 2(v+u)(v-u)$$

Деление на $u + v$ дает

$$1 = 2(v-u) = 2(v-(2v^2-1)) = -4v^2 + 2v + 2$$

Нужно решить квадратное уравнение

$$4v^2 - 2v - 1 = 0$$

что и дает желаемый результат

$$v = (1+\sqrt{5})/4$$

На самом деле будем использовать еще более важную константу

$$\tau = (1+\sqrt{5})/2$$

то есть будем иметь

$$\cos 36° = \tau/2$$

Замечательное свойство параметра τ заключается в том, что возведение в квадрат дает точно такой же результат, как и увеличение на единицу:

$$\tau^2 = (1+\sqrt{5})^2/4 = (1+2\sqrt{5}+5)/4 = 1.5 + 0.5\sqrt{5} = 1 + (1+\sqrt{5})/2 = \tau + 1$$

так что имеем

$$\tau^2 = \tau + 1 \qquad (1)$$

Путем умножения обеих половин этого уравнения на степень переменной τ, найдем последовательность выражений

$$\tau^3 = \tau^2 + \tau$$
$$\tau^4 = \tau^3 + \tau^2$$
$$\tau^5 = \tau^4 + \tau^3$$

Аналогично, если поделим выражение (1) на степень τ, получим

$$\tau = 1 + \tau^{-1}$$
$$1 = \tau^{-1} + \tau^{-2}$$
$$\tau^{-1} = \tau^{-2} + \tau^{-3}$$
$$\tau^{-2} = \tau^{-2} + \tau^{-4}$$

Таким образом, каждый элемент последовательности

$$..., \tau^{-3}, \tau^{-2}, \tau^{-1}, 1, \tau, \tau^2, \tau^3, ...$$

является суммой двух предшествующих элементов.

Из всего этого следует, что если отрезок прямой линии разделить на две части так, что более длинная часть в τ раз длиннее короткой, тогда целый отрезок в τ раз длиннее длинного отрезка. Это соотношение известно как "золотое сечение".

Другое важное свойство константы τ заключается в том, что любая ее степень может быть записана в виде суммы $a\tau + b$, где a и b — целые числа:

$$\tau^2 = \tau + 1$$
$$\tau^3 = \tau^2 + \tau = 2\tau + 1$$
$$\tau^4 = \tau^3 + \tau^2 = 3\tau + 2$$
$$\tau^5 = \tau^4 + \tau^3 = 5\tau + 3$$
$$\tau^6 = \tau^5 + \tau^4 = 8\tau + 5$$

и так далее. Также имеем

$$\tau^{-1} = \tau - 1$$
$$\tau^{-2} = 1 - \tau^{-1} = 2 - \tau$$
$$\tau^{-3} = \tau^{-1} - \tau^{-2} = 2\tau - 3$$
$$\tau^{-4} = \tau^{-2} - \tau^{-3} = 3\tau - 5$$
$$\tau^{-5} = \tau^{-3} - \tau^{-4} = 5\tau - 8$$

и так далее.

На основании (1) можно легко проверить равенство

$(a\tau + b)(a\tau - b - a) = a^2 - ab - b^2$

где предполагается, что если a и b одновременно не равны нулю, то имеем:

$$\frac{1}{a\tau + b} = \frac{a\tau - b - a}{a^2 - ab - b^2}$$

Таким образом, любое частное $p(\tau)/q(\tau)$, где $P(\tau)$ и $q(\tau)$ — многочлены для

переменной τ с рациональными коэффициентами, может быть записано в линейной форме $a\tau + b$, причем a и b — рациональные числа. Это делает τ очень удобной константой для работы, поскольку есть хороший шанс упростить сложные выражения, если эта константа в них встречается.

Все это может показаться отклонением от темы исследования пятиугольника, но это не так. Вот один из примеров — любые две диагонали пятиугольника, пересекающиеся во внутренней точке делят одна другую в отношении золотого сечения. Мы этого использовать не будем, но, как следует из рис. 1, если в пятиугольнике провести некоторые дополнительные линии, то будет образовано много углов в 36°, 72°, 18° и 54° = 36° + 18°. Теперь можно записать

$c = \cos 36° = \tau/2$

$$s = \sin 36° = \sqrt{(1-c^2)} = \sqrt{(1-\tau^2/4)} = \sqrt{(1-(\tau+1)/4} = 1/2\sqrt{(3-\tau)}$$
$$\cos 72° = 2c^2 - 1 = 1/2\tau^2 - 1 = 1/2(\tau+1) - 1 = (\tau-1)/2$$

и, используя буквы с A до F, как показано на рис. 1, будем иметь:

$$AB = AE + EB = 1 + \cos 36° = 1 + \tau/2$$
$$EF = ED\cos 72° = (\tau-1)/2$$
$$b = BF = BE + EF = \cos 36° + \cos 72° = \tau/2 + 2(\tau-1) = \tau - 1/2$$
$$a = CF = FD = ED\sin 72° = 2\cos 36°\sin 36° = \tau s$$

(при этом будем иметь в виду, что в выражениях τs значение s можно заменить на $1/2\sqrt{(3-\tau)}$.

Теперь рассмотрим рис. 2, который представляет собой вид спереди на додекаэдр в кубе, который изображен на рис. 3 в перспективе. Здесь отрезок AB, отмеченный также на рис. 1, изображен в своей натуральной величине $1 + \tau/2$.

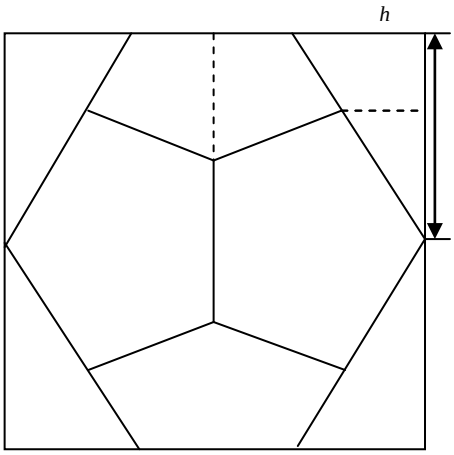


Рисунок 2. Додекаэдр в кубе (вид спереди).

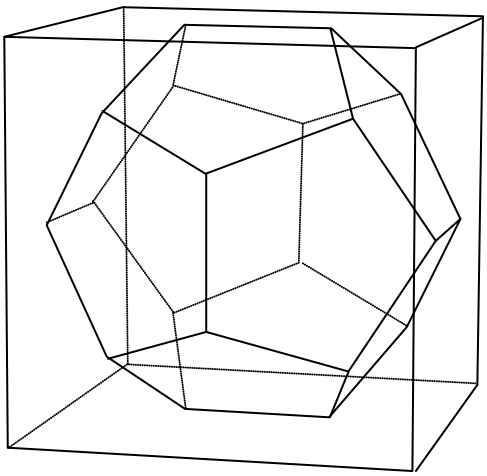


Рисунок 3. Додекаэдр в кубе (перспективная проекция).

Он представляет собой гипотенузу прямоугольного треугольника ABH, который можно будет использовать для вычисления длины половины ребра h куба. На основании теоремы Пифагора имеем:

$$AH^2 + BH^2 = AB^2$$
$$(h-s)^2 + h^2 = (1+\tau/2)^2$$
$$2h^2 - 2sh + s^2 = (1+\tau/2)^2$$
$$2h^2 - 2sh + (3-\tau)/4 = (1+\tau+\tau^2/4)$$
$$2h^2 - 2sh + (3-\tau)/4 = (1+\tau+(\tau+1)/4$$
$$4h^2 - 4sh - 3\tau - 1 = 0$$
$$h = (4s + \sqrt{(16s^2 + 16(3\tau+1))})/8$$
$$= (4s + \sqrt{(4(3-\tau) + 16(3\tau+1))})/8$$
$$= (2s + \sqrt{(11\tau+7)})/4$$

что можно упростить на основе соотношений

$$\tau^4 = 3\tau + 2$$
$$\tau^6 = 8\tau + 5$$

Тогда получим

$$\sqrt{(11\tau + 7)} = \sqrt{(\tau^6 + \tau^4)} = \tau^2\sqrt{(\tau^2 + 1)} = (\tau + 1)\sqrt{(\tau + 2)}$$

и окончательно

$$h = \frac{2s + (\tau + 1)\sqrt{(\tau + 2)}}{4}$$

Получив этот результат, мы теперь знаем размер стороны куба, и положение точек A и B на нем. Нам необходимо также знать позиции в трехмерном пространстве точек C и D, показанных на рис. 1. Они лежат на горизонтальной линии, проходящей через точку F, показанной как на рис. 1, так и на рис. 2, поэтому сначала найдем положение точки F. Из рис. 2 имеем

$$\frac{v}{b} = \frac{h}{\tau/2 + 1}$$

Следовательно

$$v = \frac{2bh}{\tau + 2} = \frac{2(\tau - 1/2)h}{\tau + 2} = (\tau - 1)h$$

Последнее равенство может быть проверено вычислением произведения τ + 2 и τ -1 с привлечением уравнения (3.1). Используя оба эти выражения для v и

$$\frac{d}{v} = \frac{h - s}{h}$$

(см. рис. 2), получим

$$d = (\tau - 1)(h - s)$$

Теперь легко найти положения точек C и D (см. рис. 1), поскольку они лежат на прямой линии, перпендикулярной плоскости рис. 2 и проходящей через точку F, а длина отрезков CF и DF была вычислена ранее как a = τs.

Благодаря симметрии позиции всех остальных точек вершин додекаэдра могут быть определены из только что найденных значений.

Икосаэдр

Как видно из рис. 4, икосаэдр можно разместить таким образом, что десять его вершин будут также вершинами двух горизонтальных пятиугольников. Две оставшиеся будем считать северным полюсом (наверху) и нижним полюсом (внизу), а прямую линию, проходящую через эти полюса, будем называть осью. Центр *O* икосаэдра будет находиться в точке начала системы координат, а ось икосаэдра совпадает с осью z. Два только что упомянутых пятиугольника будут иметь точно такие же размеры, как грани додекаэдра, описанного в предыдущем параграфе. То есть каждый из них, имеет единичный радиус описанной окружности, а стороны пятиугольника будут иметь длину 2*s*, где

$s = \sin 36°$

Каждая сторона этого пятиугольника в свою очередь является ребром икосаэдра, поэтому длина этих ребер также равна 2*s*. Из рис. 4 очевидно, что нижний горизонтальный пятиугольник может быть получен из верхнего путем поворота его на угол 36° вокруг оси и переноса по вертикали. Это означает, что остается только решить задачу нахождения координат по оси z для обоих пятиугольников и обоих полюсов. На рис. 5 изображен верхний пятиугольник из показанных на рис. 4; только для обозначения вершин вместо номеров 2, 3, 4, 5, 6 будем использовать буквы A, B, C, D, E. Точка F является центром пятиугольника ABCDE, а точка O — центр всего икосаэдра, буквой N обозначен северный полюс, точка M расположена в центре отрезка CN. Теперь нам нужно найти размеры h и r, обозначенные на рис. 5.

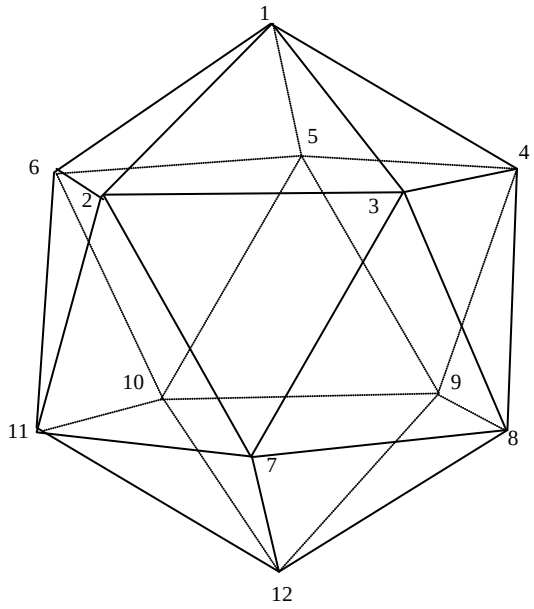


Рисунок 4. Икосаэдр (перспективная проекция)

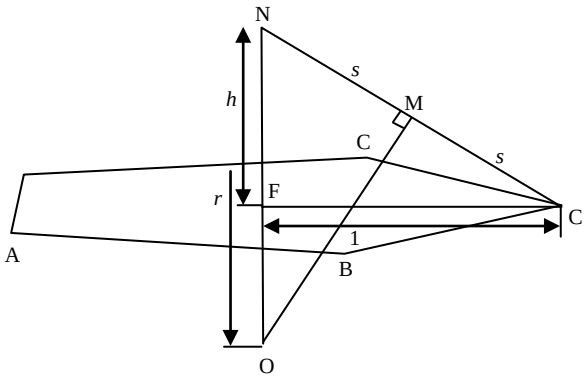


Рисунок 5. Пятиугольник и северный полюс икосаэдра

Поскольку треугольник NFC является прямоугольным, то можно применить теорему Пифагора для определения значения h = FN, выраженного через s = sin 36°:

$$FN^2 = CN^2 - CF^2$$
$$h^2 = (2s)^2 - 1$$
$$h = \sqrt{(4s^2 - 1)}$$

Снова подставляя $s = 1/2\sqrt{(3-\tau)}$

, а также используя $s^2 = (3-\tau)/4$

значение h можно также выразить через τ:

$$h = \sqrt{(2-\tau)} = \sqrt{(\tau^{-2})} = \tau^{-1} = \tau - 1$$

Чтобы найти r = ON, используем подобие треугольников NMO и NFC. Отсюда будем иметь

$$\frac{s}{r} = \frac{h}{2s}$$

что дает

$$r = \frac{2s^2}{h} = \frac{(3-\tau)/2}{\tau - 1} = \tau - 1/2$$

Координата z для всех точек в пятиугольнике ABCDE равна

$$r - h = (\tau - 1/2) - (\tau - 1) = \tau - 1/2 - \tau + 1 = 1/2$$

Это означает, что оба пятиугольника расположены на единичном расстоянии друг от друга или, другими словами, расстояние между ними равно радиусу описанной окружности для пятиугольника.

Возвращаясь к нумерации вершин в икосаэдре, показанном на рис. 4, обнаружим, что для вершин 2,3,4,5,6 координата z равна 0.5 (= r- h). Аналогично имеем z = -0.5 для вершин 7, 8, 9, 10, 11. Для обоих полюсов (точки 1 и 12) z-координаты будут равны $r = \tau$ - 1/2 и $-r = -(\tau$-1/2) соответственно и для этих точек $x = y=0$. Что касается координаты y для вершин 2,..., 11, то для их определения можно использовать тот факт, что все они лежат на горизонтальной окружности с радиусом 1. То есть при соответствующих углах a они равны cos a и sin a соответственно. Угол a может быть выбран следующим образом:

Вершина	**Угол *a***
2	-36°
3	36°
4	108°
5	180°
6	252°
7	0°
8	72°
9	144°
10	216°
11	288°

Приложение 2. Пример оформления отчета о работе

ФГОУ ВПО СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

ИНСТИТУТ КОСМИЧЕСКИХ И ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

КАФЕДРА «СИСТЕМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА»

Компьютерная геометрия и графика

Лабораторная работа №_
Геометрические преобразования на плоскости

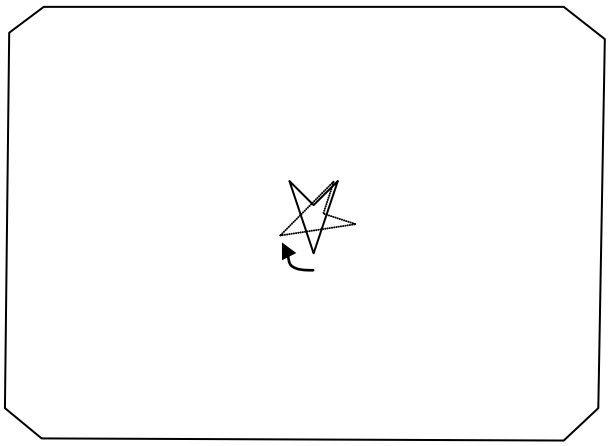
Выполнили:
студенты гр. …..
Петров Е.А.
Сидоров В.П.
Проверил:
Сиротин Э.Е.

г. Красноярск 2008 г.

1. Постановка задачи

Задание №18 (Вращающаяся стрелка)

Поворачивать любую фигуру, образованную замкнутой ломаной линией (стрелку), вокруг заданной точки на небольшой угол r по часовой или против часовой стрелки. Одну из линий стрелки выделите другим цветом. Совершить n поворотов.



Дополнение:

Добавьте в программу возможность менять направление вращения стрелки, и задавать новую точку, центр вращения, щелкая мышью на поверхности экрана.

2. Ход решения задачи.

Представим стрелку в виде четырехугольника следующей формы рис. 1.

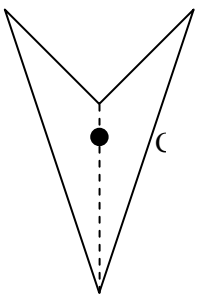


Рисунок 1. Вид стрелки и нумерация ее вершин.

Точку С – центр стрелки разместим в центре окна, и координаты вершин стрелки будем рассчитывать, используя координаты точки С как базисные. Примем, что координаты точки С представлены вектором $\overline{C} = [cx \quad cy]$. Тогда, координаты точки 0 зададим как вектор $[cx-20 \quad cy-30]$, точки 1 – $[cx \quad cy+30]$, точки 2 - $[cx+20 \quad cy-30]$, точки 3 - $[cx \quad cy-10]$. В однородных координатах, координаты вершин стрелки будут следующими:

$$0-[cx-20 \quad cy-30 \quad 1]$$
$$1-[cx \quad cy+30 \quad 1]$$
$$2-[cx+20 \quad cy-30 \quad 1]$$
$$3-[cx \quad cy-10 \quad 1]$$

Для имитации вращения стрелки вокруг своего центра, необходимо над координатами всех вершин выполнять преобразование поворота относительно заданной точки на заданный угол. Координаты точки - центра вращения, это координаты точки С. Преобразование можно представить как комбинацию трех элементарных преобразований, а именно:

1. Преобразование параллельного переноса начала системы координат в точку С.
2. Преобразование поворота точек плоскости на заданный угол относительно начала новой системы координат.
3. Преобразование параллельного переноса начала системы координат на старое место.

В матричной форме запишем наше преобразование в виде произведения матриц

$$M = T^{-1}RT,$$

где M – результирующая матрица преобразования,

$$T^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -cx & -cy & 1 \end{bmatrix}$$ - матрица переноса начала системы координат в точку С,

$$R = \begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$ - матрица поворота точек плоскости на угол φ относительно начала координат,

$$T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ cx & cy & 1 \end{bmatrix}$$ - матрица преобразования переноса начала системы координат на старое место.

Чтобы изобразить вращающуюся стрелку, применим следующий алгоритм:

1. Определим координаты центра окна *cx* и *cy*.
2. Рассчитаем на их основе координаты вершин стрелки, и занесем их в массив векторов.
3. Отобразим стрелку на поверхности окна.
4. Рассчитаем матрицу преобразования для поворота на угол 5° или -5° в зависимости от направления вращения стрелки, используя в качестве координат центра вращения, координаты *cx* и *cy*.
5. Применим полученную матрицу к векторам координат вершин стрелки.
6. Через небольшой промежуток времени, сотрем изображение стрелки на поверхности окна, и нарисуем ее заново, используя новые координаты вершин.
7. Будем повторять пункты 4 – 6 пока не будет получена команда на прекращение вращения.

8. Если пользователь произвел щелчок левой кнопкой мыши на поверхности окна, заменим координаты *cx* и *cy* на соответствующие координаты курсора мыши.

Примечание: пункт 8 выполняется параллельно с циклом 4 – 6.

3. Листинг программы.

Библиотека для работы с векторами и матрицами

Заголовочный файл Vectors2.h

```
//---------------------------------------------------------------------------
//  Определения классов для работы с векторами и матрицами
//  для двумерной графики
//---------------------------------------------------------------------------
#ifndef vectors2H
#define vectors2H
//---------------------------------------------------------------------------
#endif
//Предварительное объявление классов

class Vector2;
class Matrix2;

//=========================================================
// Класс для работы с двумерными векторами
// в однородных координатах
//=========================================================

class Vector2
{
   float X,Y,W;  //Поля для хранения координат вектора в системе X, Y, W
  public:
          Vector2(float x=0, float y=0, float w=1); //Конструктор со значениями по умолчанию
          Vector2(Vector2& vec); //Конструктор копирования
    Vector2  operator=(Vector2 a); //Определение операции присваивания для векторов
    Vector2  operator+(Vector2 a); //Определение операции сложения для векторов
    Vector2  operator-(Vector2 a); //                                    ...
вычитания...
    Vector2  operator*(float c);   //       ...         умножения вектора на скаляр
    Vector2  operator*(Matrix2 a); //       ...         умножения вектора на матрицу 3x3
    float&   operator[](int i);    //       ...         доступа к компонентам вектора
    void     clear(void);         //Функция делает вектор нулевым (0,0,1)
    Vector2   set(float x=0, float y=0, float w=1); // Функция для установки значений сразу нескольким компонентам вектора
};

Vector2 vector2(float x=0, float y=0, float w=1); // Функция создает вектор с заданными значениями компонентов

class Matrix2
{
   Vector2 M[3]; // Представляем матрицу 3x3 как совокупность 3-х векторов 3x1
  public:
  Matrix2(); // Конструктор без параметров (создает единичную матрицу)
  Matrix2(Matrix2& m);  // Конструктор копирования
```

```
  Matrix2 operator=(Matrix2 a);// Определение операции присваивания для матрицы
  Matrix2 operator+(Matrix2 a); // Определение операции сложения двух матриц
  Matrix2 operator-(Matrix2 a); // Определение операции вычитания двух матриц
  Matrix2 operator*(Matrix2 a); // Определение операции умножения двух матриц
  Vector2& operator[](int i);   // Определение операции доступа к элементам матрицы
  void identity(void);  // Функция делает единичную матрицу
};
```

Файл реализации Vectors2.cpp

```
//---------------------------------------------------------------------------
// Реализация классов для работы с векторами и матрицами
// для двумерной графики
//---------------------------------------------------------------------------
#pragma hdrstop

#include "vectors2.h"
#include <sysutils.hpp>
//---------------------------------------------------------------------------

#pragma package(smart_init)

//=========================================================
// Реализация класса Vector2
//=========================================================

//Конструктор со значениями по умолчанию

Vector2::Vector2(float x, float y, float w)
{
  X = x; Y = y; W = w;
}

//Конструктор копирования

Vector2::Vector2(Vector2& vec)
{
  X=vec[0]; Y=vec[1]; W=vec[2];
}

//---------------------------------------------------------

// Реализация операции присвавания векторов

Vector2 Vector2::operator=(Vector2 a)
{
  X = a[0]; Y = a[1];W = a[2];
  return *this ; //Возвращаем сам вектор
}

//---------------------------------------------------------

//Реализация операции сложения векторов

Vector2 Vector2::operator+(Vector2 a)
{
  Vector2 t;  //заводим новый вектор для результата,
                                                  //поскольку исходный изменяться не должен...

  //Вычисляем значения его компонентов...
  t[0] = X + a[0];
  t[1] = Y + a[1];
  t[2] = W + a[2];
```

```
 return t; // ... и возвращаем как результирующий вектор
}

//-------------------------------------------------------

//Реализация операции вычитания векторов

Vector2 Vector2::operator-(Vector2 a)
{
 Vector2 t;//заводим новый вектор для результата,
                                          //поскольку исходный изменяться не должен...

        //Вычисляем значения его компонентов...
 t[0] = X - a[0];
 t[1] = Y - a[1];
 t[2] = W - a[2];
 return t;  // ... и возвращаем как результирующий вектор
}

//-------------------------------------------------------

//Реализация операции умножения вектора на скаляр

Vector2 Vector2::operator*(float c)
{
 Vector2 t;  //заводим новый вектор для результата,
                                                  //поскольку исходный изменяться не должен...

         //Вычисляем значения его компонентов...

 t[0] = X * c;
 t[1] = Y * c;
 t[2] = W * c;
 return t;  // ... и возвращаем как результирующий вектор
}

//-------------------------------------------------------

//Реализация операции умножения вектора на матрицу 3x3

Vector2 Vector2::operator*(Matrix2 a)
{
 Vector2 t;  //заводим новый вектор для результата,
                                                  //поскольку исходный изменяться не должен...

         //Вычисляем значения его компонентов...
 t[0]=X * a[0][0] + Y * a[1][0] + W * a[2][0];
 t[1]=X * a[0][1] + Y * a[1][1] + W * a[2][1];
 t[2]=X * a[0][2] + Y * a[1][2] + W * a[2][2];
 return t;  // ... и возвращаем как результирующий вектор
}

//-------------------------------------------------------

//Реализация доступа к компонентам вектора

float& Vector2::operator[](int i)
{
 switch (i)
 {
   case 0: return X; //[0]
   case 1: return Y; //[1]
   case 2: return W; //[2]
```

```
    //Если индекс за границей массива, то формируем исключение
   default: throw ERangeError("Invalid index");
  }
}

//---------------------------------------------------------

// Функция делает вектор нулевым (0,0,1)

void Vector2::clear(void)
{
  X = 0; Y = 0; W = 1;
}

//---------------------------------------------------------

//Функция задает новые значения сразу нескольким
//компонентам вектора

Vector2 Vector2::set(float x, float y, float w)
{
  X=x; Y=y; W=w;
  return *this; //Возвращаем сам вектор
}

//=========================================================
// Реализация класса Matrix2
//=========================================================

// Конструктор по умолчанию...

Matrix2::Matrix2()
{
  identity();  //... создает единичную матрицу
}

//---------------------------------------------------------

//Конструктор копирования...

Matrix2::Matrix2(Matrix2& m)
{
        //... создает копию матрицы m
  M[0] = m[0];
  M[1] = m[1];
  M[2] = m[2];
}

//---------------------------------------------------------

//Реализация операции присваивания матриц

Matrix2 Matrix2::operator=(Matrix2 a)
{
        int i;

  for (i=0; i<3; i++) //Для всех строк матрицы...
        M[i] = a[i]; //... копируем значения строк матрицы a
  return *this;
}

//---------------------------------------------------------------------------
```

```
//Реализация операции сложения двух матриц

Matrix2 Matrix2::operator+(Matrix2 a)
{
  Matrix2 t; //заводим новую матрицу для результата,
                                                    //поскольку исходная изменяться не должна...

        //Вычисляем значения ее элементов...

  t[0] = a[0] + M[0];
  t[1] = a[1] + M[1];
  t[2] = a[2] + M[2];
  return t;  //... и возвращаем как результирующую
}

//-------------------------------------------------------

//Реализация операции вычитания двух матриц

Matrix2 Matrix2::operator-(Matrix2 a)
{
  Matrix2 t;  //заводим новую матрицу для результата,
                                                    //поскольку исходная изменяться не должна...

        //Вычисляем значения ее элементов...
  t[0] = M[0] - a[0];
  t[1] = M[1] - a[1];
  t[2] = M[2] - a[2];
  return t; //... и возвращаем как результирующую
}

//-------------------------------------------------------

//Реализация операции умножения двух матриц

Matrix2 Matrix2::operator*(Matrix2 a)
{
 int i; //Номер текущей строки
 Matrix2 t; //заводим новую матрицу для результата,
                                                    //поскольку исходная изменяться не должна...

        //Вычисляем значения ее элементов...
 for (i = 0; i < 3; i++)
   t[i] = M[i] * a;
 return t; //... и возвращаем как результирующую
}

//-------------------------------------------------------
//Реализация доступа к строкам матрицы
//Чтобы получить значение эл-та (i, j) используйте [i][j]

Vector2& Matrix2::operator[](int i)
{
 switch (i) //Проверяем границы индекса
 {
   case 0: return M[0]; //[0]
   case 1: return M[1]; //[1]
   case 2: return M[2]; //[2]
   //Если индекс за пределами, то формируем исключение
   default: throw ERangeError("Invalid index");
 }
}
```

```
//---------------------------------------------------------

//Функция создает единичную матрицу

void Matrix2::identity(void)
{
  int i,j;

  for (i = 0; i < 3; i++)  //Проходим по всем строкам...
    for (j = 0; j < 3; j++)//... и столбцам
      if (i==j)
        M[j][i] = 1.0; // Если диагональный элемент, то 1
      else
        M[i][j] = 0;   // Иначе - 0
}

//---------------------------------------------------------

//Функция создает вектор с заданными значениями компонентов

Vector2 vector2(float x, float y, float w)
{
  Vector2 vec(x,y,w); //Создаем новый вектор и задаем значения компонентам
  return vec; //Возвращаем его как результат
}
```

Основной модуль.

Заголовочный файл mainform.h

```
//---------------------------------------------------------------------------

#ifndef MainFormH
#define MainFormH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <ExtCtrls.hpp>
//---------------------------------------------------------------------------
class TfrMain : public TForm
{
__published:       // IDE-managed Components
  TTimer *Timer;
            TPanel *Panel1;
            TRadioGroup *rgDirect;
            TButton *Button1;
  void __fastcall FormCreate(TObject *Sender);
  void __fastcall TimerTimer(TObject *Sender);
  void __fastcall FormClose(TObject *Sender, TCloseAction &Action);
  void __fastcall FormShow(TObject *Sender);
  void __fastcall FormPaint(TObject *Sender);
  void __fastcall Button1Click(TObject *Sender);
            void __fastcall FormMouseDown(TObject *Sender, TMouseButton Button,
          TShiftState Shift, int X, int Y);
private: // User declarations
public:               // User declarations
  __fastcall TfrMain(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TfrMain *frMain;
//---------------------------------------------------------------------------
```

```
#endif
```

Файл реализации mainform.cpp

```
//---------------------------------------------------------------------------

#include <vcl.h>
#pragma hdrstop

#include "MainForm.h"
#include "vectors2.h" //Подключаем библиотеку для работы с векторами и матрицами
#include <math.h>  //Подключаем библиотеку математических ф-ций
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"

class Square   //Класс для представления прямоугольника
{
  Vector2 sq[4];  //Массив координат вершин
  public:
    Square();     //Конструктор по умолчанию
    Vector2& operator[](int i);   //Операция доступа к координатам вершин
          //Функция рисует прямоугольник на поверхности рисования canvas
      //три стороны одним цветом (c1), одну - другим (c2)
    void    draw(TCanvas *canvas, TColor c1, TColor c2);
};

Square::Square() //Конструктор по умолчанию
{ //Создает прямоугольник, у к-рого все вершины
          //расположены в точке (0,0,1)

  int i;
  Vector2 vec(0,0,1); //Вспомогательный вектор
  for (i=0; i<4; i++) //Проходим по всем вершинам...
    sq[i] = vec; //... и задаем им координаты вектора vec
}

Vector2& Square::operator[](int i)
{
  if (i<0 || i>3) //Проверяем границы индекса
    //Если индекс за пределами, то формируем исключение
    throw ERangeError("Invalid Side Index");
  else
          //Иначе - возвращаем соответсвующую вершину
    return sq[i];
}

//Функция рисует прямоугольник на поверхности рисования canvas
//три стороны одним цветом (c1), одну - другим (c2)

void Square::draw(TCanvas *canvas, TColor c1, TColor c2)
{
  if (canvas) //Рисуем, только если canvas существует
  {
    canvas->Pen->Color=c1; //Сначала установим цвет c1
    canvas->Pen->Style=psSolid;  //Устанавливаем сплошную линию
    int i;
    for (i=0; i<4; i++)
    {
      canvas->MoveTo(sq[i][0], sq[i][1]);//Перемещаем перо в новую точку
      if (i!=3)
        canvas->LineTo(sq[i+1][0], sq[i+1][1]); //Рисуем три стороны прямоугольника цветом c1
      else
```

```
      {
        canvas->Pen->Color=c2; //Устанавливаем цвет c2
        canvas->LineTo(sq[0][0], sq[0][1]); //Рисуем четвертую сторону цветом c2
      }
     }
   }
}


//Функция выполняет поворот четырехугольника sq
//на угол degree, вокруг точки cp
void RotateSquare(Square& sq, float degree, Vector2& cp)
{
  Matrix2 mf,  //матрица поворота относительно точки (0,0)
                                        mt,  //матрица переноса точки cp в точку (0,0)
         mtc, //матрица переноса точки (0,0) в точку cp
         m; //результирующая матрица преобразования
  float sinf, cosf;
  int i;
  sinf = sin(M_PI * degree / 180); //sin угла поворота
  cosf = cos(M_PI * degree / 180); //cos угла поворота
  //Сначала делаем все матрицы единичными
  mf.identity();
  mt.identity();
  mtc.identity();
  //Формируем матрицу переноса точки (0,0) в точку cp
  mtc[2][0] = -cp[0];
  mtc[2][1] = -cp[1];
  //Формируем матрица обратного переноса начала координат
  //из точки cp в точку (0,0)
  mt[2][0] = cp[0];
  mt[2][1] = cp[1];
  //Формируем матрицу поворота на угол degree   cosf  sinf 0
  //                                    -sinf cosf 0
  //                                      0     0  1
  mf[0][0] = mf[1][1]= cosf;
  mf[0][1] = sinf;
  mf[1][0] = -sinf;

  m = mtc * mf * mt; //Вычисляем матрицу общего преобразования...

  for (i=0; i<4; i++) //... и применяем ее ко всем вершинам четырехугольника
    sq[i] = sq[i] * m;
}

//---------------------------------------------------------------------------

//Определение глобальных переменных

Square square;  //Экземпляр четырехугольника
Graphics::TBitmap *bitmap=NULL;  //Указатель на битовую карту изображения
int cx, cy; //Координаты центра окна
Vector2 cpoint; //Центр вращения

TfrMain *frMain;
//---------------------------------------------------------------------------
__fastcall TfrMain::TfrMain(TComponent* Owner)
  : TForm(Owner)
{
}
//---------------------------------------------------------------------------
//Когда создаем окно в первый раз
void __fastcall TfrMain::FormCreate(TObject *Sender)
```

```
{
  cx=ClientWidth / 2; //Задаем координаты
  cy=ClientHeight /2; //центра окна...
  cpoint[0]=cx, cpoint[1] = cy; //и центра вращения
  //Формируем четырехугольник
  square[0]=vector2(cx-20,cy-30);
  square[1]=vector2(cx,cy+30);
  square[2]=vector2(cx+20,cy-30);
  square[3]=vector2(cx,cy-10);
  try    //Пробуем создать битовую карту,
                                  // если возникнет исключение выполнится секция catch
  {
    bitmap = new Graphics::TBitmap();
    bitmap->Width = ClientWidth;
    bitmap->Height = ClientHeight;
    bitmap->PixelFormat = pf32bit;
    if (bitmap->PixelFormat != pf32bit) //проверяем установился ли нужный формат пикселей...
      throw ""; //нет, формируем исключение
  }
  catch (...)
  {
          //Если были исключения,
    //то выдаем сообщение...
    Application->MessageBoxA("Невозможно создать битмап", "Ошибка", MB_ICONHAND | MB_OK);
    Application->Terminate();//... и завершаем работу программы
  }
  //Если все прошло удачно...
    bitmap->Canvas->Brush->Style=bsSolid;
  bitmap->Canvas->Brush->Color=clBtnFace;
  //Чистим битмап
  bitmap->Canvas->FillRect(Rect(0,0,bitmap->Width-1,bitmap->Height-1));
  //Рисуем четырехугольник в битмапе
  square.draw(bitmap->Canvas, clRed, clGreen);
}
//---------------------------------------------------------------------------
//Обработчик события таймера

void __fastcall TfrMain::TimerTimer(TObject *Sender)
{
  int i;
  float degree; //Значение угла поворота

  //Стираем в битмапе старое изображение...
  bitmap->Canvas->Brush->Color=clBtnFace;
  //... рисуя закрашеный прямоугольник во весь битмап
  bitmap->Canvas->FillRect(Rect(0,0,bitmap->Width-1, bitmap->Height-1));
  //Определяем в какую сторону вращать четырехугольник
  if (rgDirect->ItemIndex)
          degree = -5;  //Против часовой
  else
          degree = 5;  //По часовой
  RotateSquare(square, degree ,cpoint);  //Поворачиваем четырехугольник...
  //... и рисуем его в битмапе
  square.draw(bitmap->Canvas, clRed, clGreen);
  //Рисуем точку вращения...
  bitmap->Canvas->Pen->Color = clRed;
  bitmap->Canvas->Brush->Color = clRed;
  //... кружочком красного цвета
  bitmap->Canvas->Ellipse(cpoint[0]-5,cpoint[1]-5,cpoint[0]+5,cpoint[1]+5);
  //Отображаем битмап на окне
  Canvas->Draw(0, 0, bitmap);
}
//---------------------------------------------------------------------------
```

```
//Когда окно закрывается...
void __fastcall TfrMain::FormClose(TObject *Sender, TCloseAction &Action)
{
 Timer->Enabled = False; //Останавливаем таймер
 if (bitmap)  //Если битмап существует,
 delete bitmap; //то уничтожаем его экземпляр
 bitmap = NULL;

}
//---------------------------------------------------------------------------
//Когда окно рисуется в первый раз...
void __fastcall TfrMain::FormShow(TObject *Sender)
{
        //... нужно отобразить битмап на окне
 Canvas->Draw(0,0,bitmap);

}
//---------------------------------------------------------------------------
//Если окно нужно перерисовать,...
void __fastcall TfrMain::FormPaint(TObject *Sender)
{
 Canvas->Draw(0,0,bitmap); //... то отобразим битмап на окне
}
//---------------------------------------------------------------------------
//Если нажали кнопку на окне,...
void __fastcall TfrMain::Button1Click(TObject *Sender)
{
 Timer->Enabled = !(Timer->Enabled); //Переключили состояние таймера в противоположное
 if (Timer->Enabled) //Если таймер включился,...
   Button1->Caption = "Стоп"; //То кнопка теперь будет "Стоп"...
 else
   Button1->Caption = "Пуск"; //... Иначе - "Пуск"
}
//---------------------------------------------------------------------------
//Если нажали кнопку мыши в окне
void __fastcall TfrMain::FormMouseDown(TObject *Sender,
     TMouseButton Button, TShiftState Shift, int X, int Y)
{
        if (Button == mbLeft) //Если нажали левую кнопку мыши,
                                                                                //то
задаем новые координаты центра вращения
 {
        cpoint[0] = X;
  cpoint[1] = Y;
 }
}
//---------------------------------------------------------------------------
```

4. Скриншоты работы программы

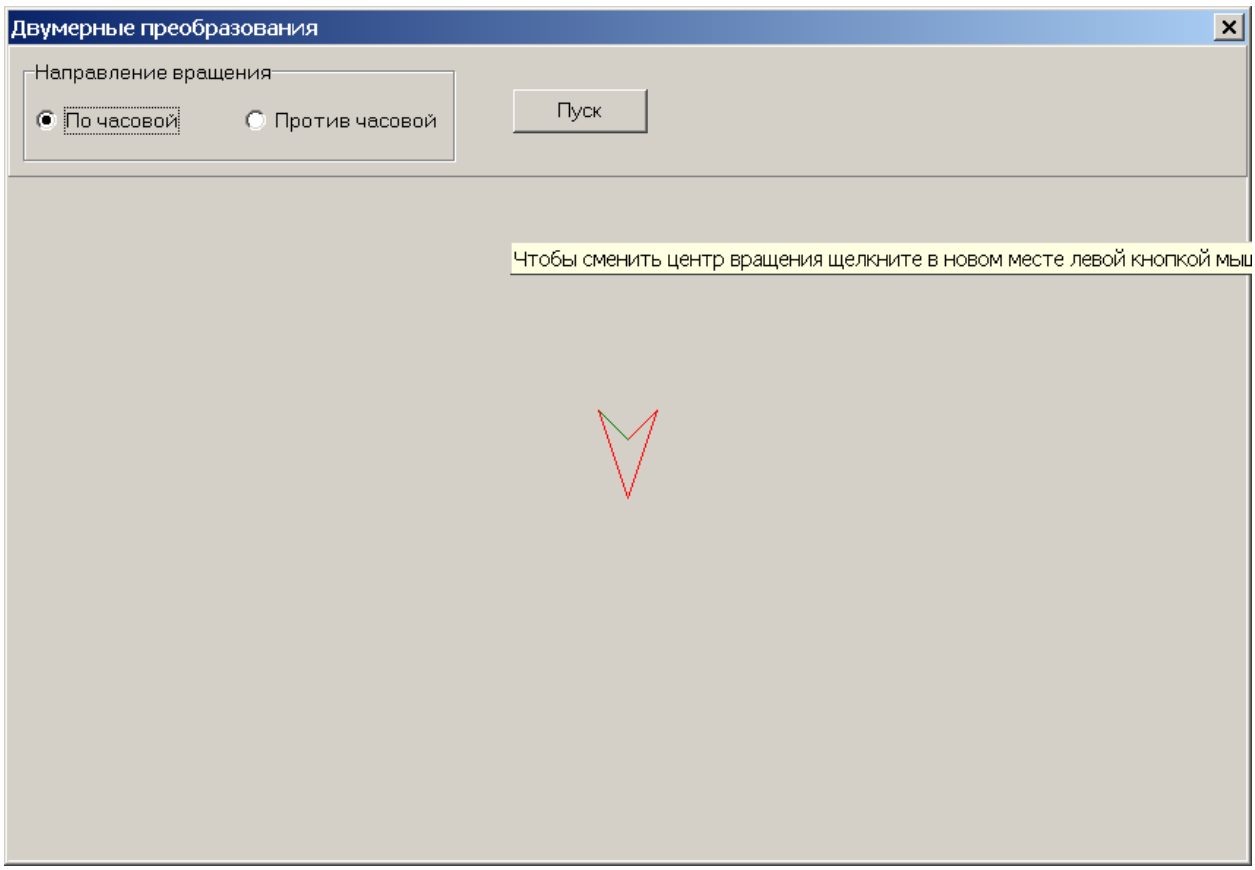


Рис 2. Начальный вид окна программы.

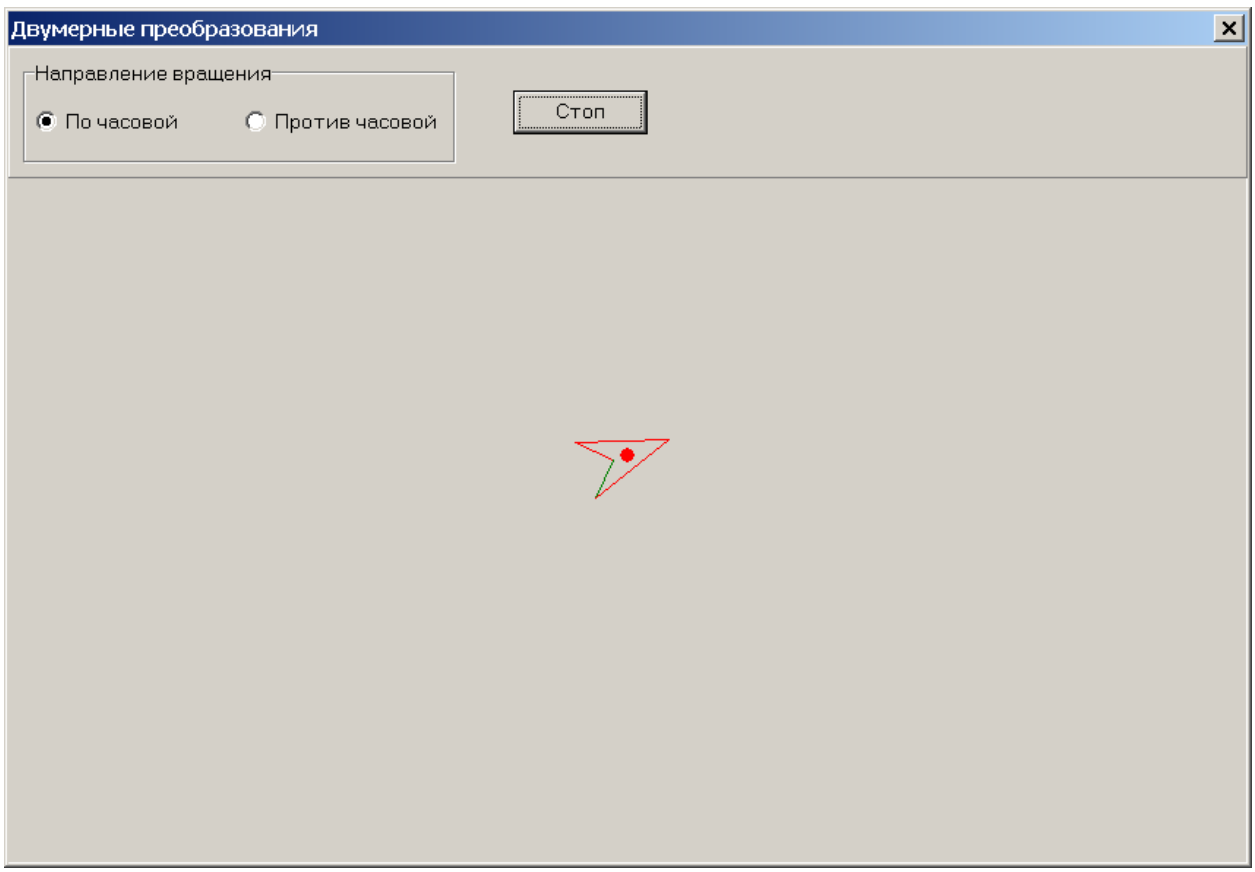


Рисунок 3. Вращение стрелки.

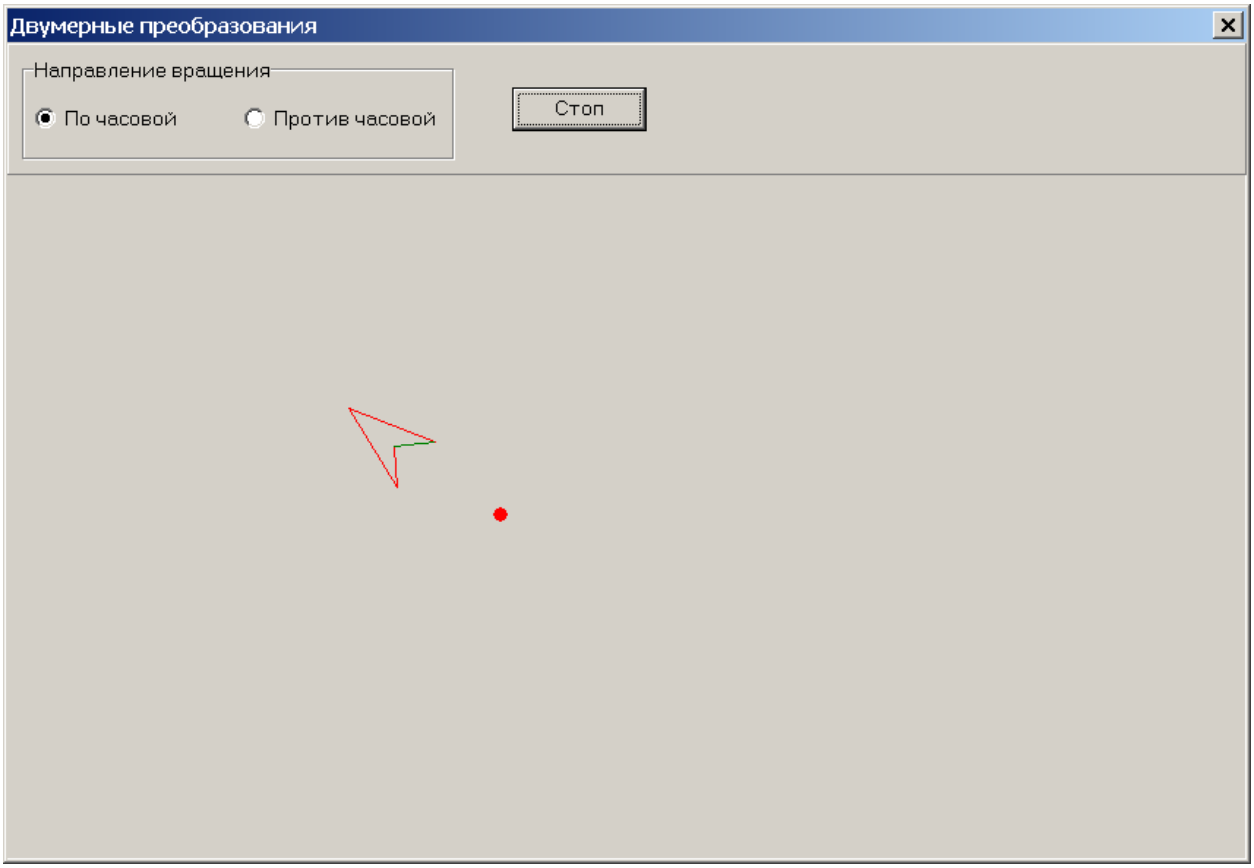


Рисунок 4. Новый центр вращения.

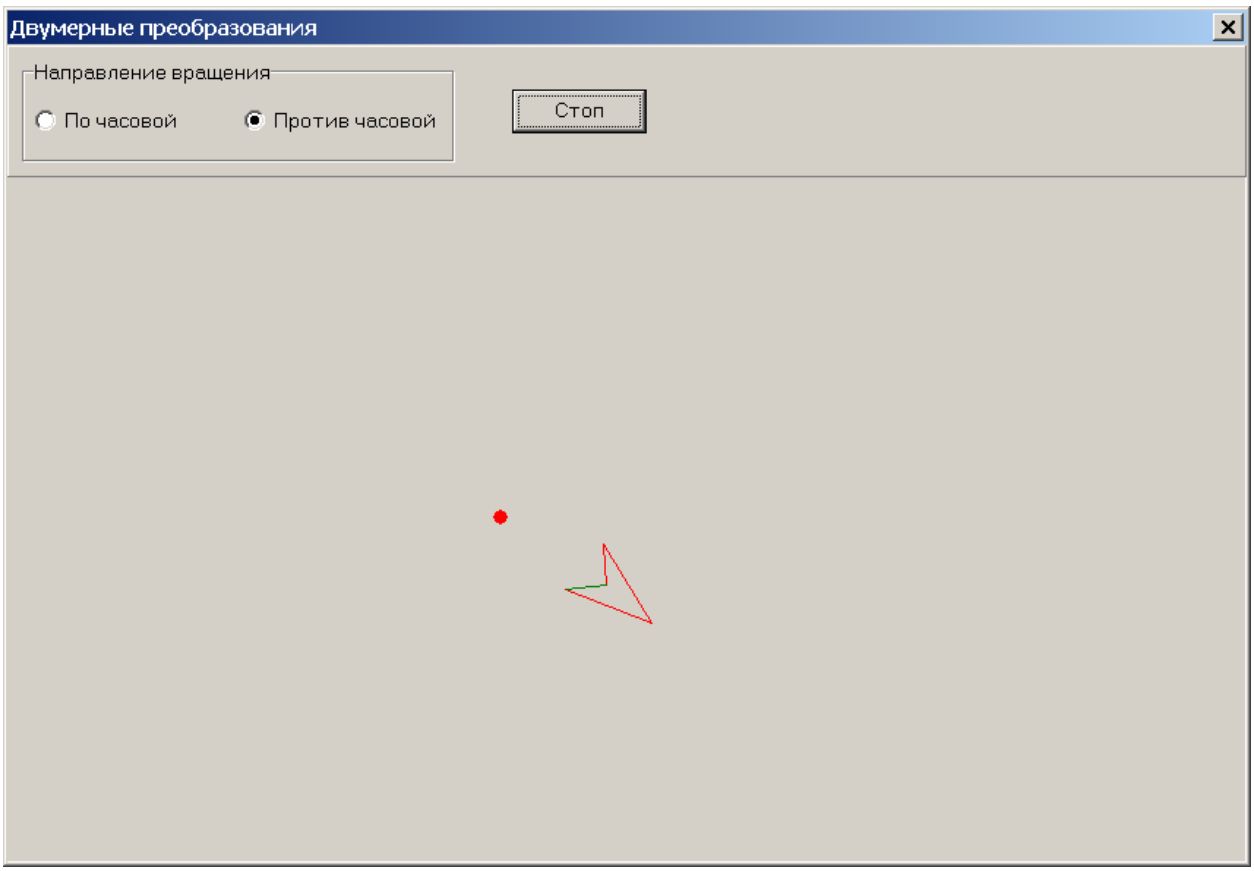


Рисунок 5. Сменили направление вращения.

Литература

Основная:

1. Дергач В.В. Компьютерная графика: Учебное пособие. ИПЦ КГТУ, 2003 г.
2. Петров М.Н. Компьютерная графика: Учебное пособие. СПб.: Питер, 2002 г.
3. Никулин Е.А. Компьютерная геометрия и алгоритмы машинной графики: Учебное пособие. СПб.: БХВ-Петербург, 2003 г.
4. Шикин Е.В. Компьютерная графика. Полигональные модели. М.: Диалог-МИФИ, 2001 г.
5. Тихомиров Ю. OpenGL. Программирование трехмерной графики – 2-е изд. - СПб.: «БХВ – Санкт-Петербург», 2002 г.

Дополнительная:

1. Иванов В.П. Трехмерная компьютерная графика. М.: Радио и связь, 1999 г.
2. Л.С. Рубан, Н.В. Соснин. Аффинные преобразования. Методические указания. КГТУ, 1999 г.
3. Федорова Н.А. Математические основы компьютерной графики. Методические указания. Красноярск, 1999 г.
4. Павлидис Т. Алгоритмы машинной графики и обработки изображений. Пер. с англ. М.: Радио и связь, 1986 г.
5. Роджерс Д., Адамс Дж. Математические основы машинной графики: пер. с англ. – М.: Машиностроение, 1980 г.