МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«Национальный исследовательский ядерный университет «МИФИ»
(НИЯУ МИФИ)
КАФЕДРА ИНФОРМАЦИОННЫХ СИСТЕМ И ТЕХНОЛОГИЙ

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»
на тему «Расширение языков стекового калькулятора и интерпретатора арифметических выражений операцией побитового сдвига. Вычисление количества острых углов выпуклой оболочки. Вычисление расстояния от выпуклой оболочки до стандартного прямоугольника. Нахождение суммы периметров частично видимых граней полиэдра»

Группа	К04-361
Студент	А.Ю. Шедько
Руководитель работы к.ф.-м.н., доцент	Е.А. Роганов

Москва 2016

Аннотация

Работа посвящена модификации проектов «Компилятор формул», «Интерпретатор арифметических выражений», «Выпуклая оболочка» и «Изображение проекции полиэдра». В первом из этих проектов решалась задача расширения языка стекового калькулятора выражений операцией побитового сдвига. Модификация второго пректа требовала решения задачи расширения итерпретатора арифметических выражений, расширенного операциями побитового сдвига со специальным приоретом. В проекте «Выпуклая оболочка» вычислялось расстояние от выпуклой оболочки до стандартного прямоугольника и количество острых углов в выпуклой оболочке. В последнем из проектов определялась сумма периметров частично видимых граней полиэдра.

Содержание

1. Введение

В проектах «Компилятор формул» и «Интерпретатор арифметических выражений» были решены задачи расширения языков стекового калькулятора и интерпретатора арифметических выражений операциями побитового сдвига. Применена структура данных «хэш-таблица».Решение задачи требовало представления о формальных грамматиках, основы ООП и знания языка Ruby.

Проект «Выпуклая оболочка»[1] решает задачу индуктивного перевычисления выпуклой оболочки последовательно поступающих точек плоскости и таких её характеристик, как периметр и площадь. Целью данной работы является индуктивное вычисление расстояние до заранее заданного стандартного прямоугольника и количества острых улов выпуклой оболочки. Применено два специфических алгоритма, значительно ускоряющих вычисление расстояние от прямоугольника до отрезка [2] и проверку пересечения прямоугольника и отрезка (алгоритм Лианга-Барски) [3]. Решение этой задачи требует знания теории индуктивных функций, основ аналитической геометрии и векторной алгебры и языка Ruby [4].

Проект «Изображение проекции полиэдра» [5] — пример классической задачи, для успешного решения которой необходимо знакомство с основами вычислительной геометрии. Задачей, решаемой в данной работе, является модификация эталонного проекта с целью определения суммы периметров частично видимых граней заданного полиэдра. Для этого необходимы хорошее понимание ряда разделов аналитической геометрии и векторной алгебры, основ объектно-ориентированного программирования и языка Ruby.

Для подготовки пояснительной записки необходимо знакомство с программой компьютерной вёрстки LaTeX [6], умение набирать математические формулы [7] и включать в документ графические изображения и исходные коды программ.

Общее количество строк в рассмотренных проектах составляет около 1476, из которых 634 были изменены или добавлены автором в процессе работы над задачами модификации.

2. Модификация проекта «Компилятор формул»

Постановка задачи

В предположении, что язык стекового калькулятора расширен операциями L (left) и R (right), реализующими побитовый сдвиг влево и вправо соответственно, компилировать формулы, содержащие операции << и >>.

Теоретические аспекты

С формальной точки зрения компилятор представляет собой программную реализацию некоторой функции $\tau\colon L_1 \to L_2$, действующей из множества цепочек одного языка L_1 в множество цепочек другого L_2 таким образом, что $\forall\omega \in L_1$ *семантика цепочек* ω *и* $\tau(\omega) \in L_2$ *совпадает.*

Для решения задачи необходимо задать грамматики, описывающие языки стекового калькулятора и компилятора, соответственно G_0 и G_s:

G_0:

```
S → F  | SRF | SLF
F → T  | F+T | F-T
T → M  | T*M | T/M
M → (S) | V
V → a  | b  | ... | z
```

G_s:

```
e → ee+ | ee-| ee* | ee/ | ee>>| ee<<|
    | a | b | ··· | z
```

где L, R соответствуют <<, >>, т. е. сдвигу влево и вправо, а S — классу сдвигаемых выражений (ассоциативность аналогична + и -).

Нужный нам компилятор τ представляет собой программную реализацию отображения из множества цепочек языка $L(G_0)$ в множество цепочек языка $L(G_s)$ По этой причине его можно рассматривать, как функцию на пространстве последовательностей. Легко понять, что эта функция не индуктивна.

Построим индуктивное расширение функции, чтобы с его помощью реализовать однопроходный алгоритм, осуществляющий нужный нам перевод. Заметим, что любую правильную формулу можно откомпилировать с соблюдением следующих двух условий:

- переменные в выходной цепочке (программе для стекового калькулятора) будут идти в том же порядке, что и переменные в исходной формуле;
- все операции в выходной цепочке будут расположены позже соответствующих им операций в исходной формуле.

Любую формулу можно компилировать так: встретив имя переменной, немедленно записывать его в массив, где мы будем накапливать результат компиляции, а встретив знак операции или скобку, записывать в этот массив те из предыдущих, но ещё не обработанных операций (будем называть их *отложенными*), которые выполнимы в данный момент, после чего «откладывать» и новый знак.

В качестве контейнера для хранения отложенных операций можно использовать стек. Этот стек и будет содержать ту дополнительную информацию, которая необходима для индуктивного перевычисления функции T осуществляющей компиляцию исходной формулы. Основная проблема — понять, что надо делать, когда в исходной формуле встречается очередная операция или скобка.

Таким образом, встретив в исходной формуле очередной знак операции или скобку, нужно иногда просто положить её в стек отложенных операций, а иногда — извлечь предварительно одну или несколько ранее отложенных операций и добавить их в массив результата. С точки зрения теории индуктивных функций необходимо построить индуктивное расширение функции T осуществляющей компиляцию исходной формулы, и найти для неё отображение G обеспечивающее её перевычисление при удлинении входной формулы.

Для построения индуктивной функции компиляции цепочки ω необходимо разделить символы на категории: `SYM_LEFT, SYM_RIGHT, SYM_OPER` и `SYM_OTHER` соответственно. правила построения таковы:

- открывающую скобку всегда помещать в стек, имя переменной — всегда сразу добавлять в массив результата компиляции;
- когда в формуле встречается закрывающая скобка нужно все операции, появившиеся в ней *после соответствующей ей открывающей скобки*, извлечь из

стека отложенных операций и добавить в массив результата (не забыть также извлечь открывающую скобку);
- операции и скобки помещаются в стек в соответствии с приоритетом соответствующей операции или скобке. Открывающие скобки имеют наименьший приоритет, закрывающие — наибольший; для операций он будет описан в дальнейшем.

Более подробно см. [8]

Применяемые структуры данных

Стек, используемый в эталонном проекте, не требует пояснений. Помимо стека используется константная хэш-таблица (в качестве отображения между множествами операций двух грамматик). Подробно с работой хеш-таблицы в языке Ruby можно ознакомиться в документации к языку [9] и на соответствующей странице Википедии.

Детали реализации

Отображение между множествами операций языков калькулятора и компилятора описывается данной таблицей.

```
CONV_TABLE = {">>" => "R",
              "<<" => "L",
               "+" => "+",
               "-" => "-",
               "/" => "/",
               "*" => "*"}
```

Приоритет операций задаётся следующим образом:

```
def priority(c)
  (c == '+' or c == '-') ? 1 : ( c=='L' or c=='R' )? 0 : 2
end
```

Этот фрагмент кода устанавливает наименьший приоритет для сдвигов (0) и наивысший для деления и умножения (2).

Наиболее значительным является изменение, касающееся обработки символов операций, а именно обработка символов `>>` и `<<`, так как они занимают два строковых символа, являясь одним логическим символом. Идея проста: хранить первый символ в обрабатываемой последовательности, соответствующий одной из операций, в специальной переменной `@op`, а при поступлении следующего символа, если последовательность составлена верно и этот символ совпадает с `<` или `>`, записывать его в `@op`. После этого, вне зависимости от содержимого `@op`, происходит стандартная процедура обработки отложенных операций с использованием значения `CONV_TABLE[@op]`, которое затем помещается в стек. После успешной обработки символа `@op` очищается.

Код, реализующий обработку символов операций:

```
def process_symbol(c)
  ...
  when SYM_OPER
    if ((c=='>'||c=='<')&&@op=="")
      @op=c
    else
      @op+=c
      process_suspended_operators(CONV_TABLE[@op])
      push(CONV_TABLE[@op])
      @op=""
  end
  ...
end
```

Таким образом в язык стекового компилятора добавлена операция побитового сдвига.

Возможные обобщения

Имеет смысл ввести в язык стекового компилятора также остальные побитовые операции (`&`, `&`, `| ^,~`), их логические аналоги (`&&`,`||`).

3. Модификация проекта «Интерпретатор арифметических выражений»

Постановка задачи

Вычисляются значения выражений, содержащих битовые операции << и >>, приоритет первой из которых является минимальным, а второй — максимальным.

Теоретические аспекты

Если компилятор осуществляет перевод с одного языка на другой, то интерпретатор *вычисляет* значение арифметической формулы, в которой вместо имён переменных содержатся записанные тем или иным способом числа.

Грамматика G_0 из предыдущего пункта должна быть изменена для учёта максимального приоритета операции сдвига вправо:

G_0:

S_0 →	F	|	S_0LF			
F →	T	|	F+T	|	F-T	
T →	M	|	T*M	|	T/M	
S_1 →	M	|	S_1RF			
M →	(S_0)	|	V			
V →	a	|	b	|	...	| z

`S_0` — класс сдвигаемых влево выражений;
`S_1` — класс сдвигаемых вправо выражений.

G_s можно оставить без изменений.

Класс `Calc`, реализующий алгоритм интерпретации арифметических выражений может быть построен при помощи уже реализованного класса `Compf`.

Интерпретатор выражений отличается от компилятора тем, что вместо идентификаторов переменных во входной формуле стоят *числа* и получаемую выходную формулу необходимо *выполнять* (вместо её печати). Для того чтобы реализовать стековый калькулятор, естественно, необходим стек.

При появлении на входе цифры, будем помещать её в стек калькулятора `@s`, а при появлении операции доставать из стека её аргументы и помещать туда результат её выполнения. На конечном этапе на вершине стека будет находиться результат выполнения арифметического выражения.

Детали реализации

Здесь также как и в первом проекте применена хеш-таблица для отображения множества операций стекового компилятора в множество операций языка Ruby.

```
CONV_TABLE = {"R" => ">>",
              "L" => "<<",
              "+" => "+",
              "-" => "-",
              "/" => "/",
              "*" => "*"}
```

В соответствии с задачей, приоритет операций был изменён следующим образом:

```
def priority(c)
  (c == '+' or c == '-') ? 1 : c=='L' ? 0 : (c=='R') ? 3 : 2
end
```

Таблица 1. Приоритет операций

Операция	Приоритет
L	0
+, -	1
*, /	2
R	3

4. Модификация проекта «Выпуклая оболочка»

Постановка задачи

Требуется модифицировать эталонный проект:

1. для индуктивного вычисления количества всех острых внутренних углов выпуклой оболочки (задача 40).
2. для индуктивного вычисления расстояния от выпуклой оболочки до заданного стандартного прямоугольника (Задача 52).

Теоретические аспекты

Задача построения выпуклой оболочки множества точек может быть сформулирована следующим образом: для множества точек M необходимо найти наименьшее *выпуклое* множество, включающее M. *Выпуклым* будем называть любое множество M, удовлетворяющее условию: $\forall x_1, x_2 \in M\ [x_1, x_2] \in M$.

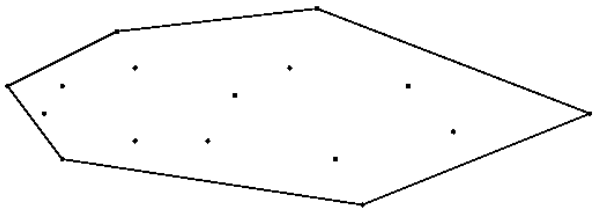

Рис. 1. Выпуклая оболочка множества точек

Пусть X —множество точек на плоскости $\mathbb{R}^2$, $\mathcal{P}$ — множество всех выпуклых фигур на плоскости. Тогда тройка (f, g, h), где $f\colon X^* \to \mathcal{P}$ — *выпуклая оболочка последовательности точек*, $g\colon X^* \to \mathbb{N}$ — *количество острых углов в ней*, $h\colon X^* \to \mathbb{R}$ — *расстояние от неё до стандартного прямоугольника*, задаёт индуктивную функцию

$$F\colon X^* \to \mathcal{P} \times \mathbb{N} \times \mathbb{R}, F = \begin{pmatrix} f \\ g \\ h \end{pmatrix}.$$

Функция индуктивного перевычисления G представляет собой реализацию следующей идеи: пусть для некоторой последовательности точек X^* значения f, g, h уже известны. Тогда при поступлении новой точки x возможны два случая: либо точка лежит внутри выпуклой оболочки, либо снаружи неё. Если x внутри, то её можно игнорировать так как она не изменит характеристик выпуклой оболочки. Рассмотрим случай когда она находится снаружи. Хорошим представлением удаляемых рёбер будет критерий *освещённости* рёбер оболочки из этой точки2 .

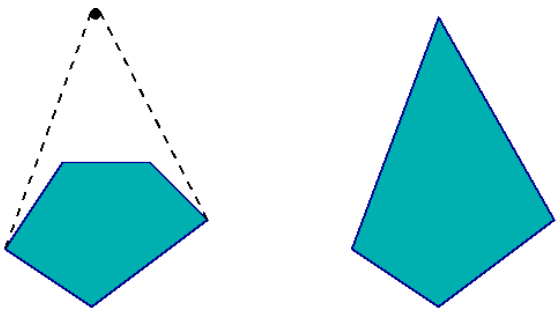

Рис. 2. Изменение выпуклой оболочки при добавлении точки

Для получения новой оболочки необходимо удалить все освещённые рёбра, а концы оставшейся ломаной соединить двумя новыми рёбрами с добавляемой точкой x. Если добавляемая точка лежит на продолжении одного из рёбер, то оболочка должна измениться, поэтому ребро, на продолжении которого лежит точка x, мы также будем считать освещённым.

Для вычисления функции g —количества острых углов в выпуклой оболочке нам необходимо хранить *множество* точек, углы при которых острые (`Set::@@angles` будем обозначать A). Случаи точки и двуугольника тривиальны (0 и 2 острых угла соответственно). Для треугольника достаточно напрямую проверить, являются ли углы острыми при помощи сравнения скалярного произведения с нулём (метод точки `ac_angle?(a,b)` — двухместный предикат "угол образуемый точкой-вершиной и точками `a`, `b` острый") и если углы острые, добавить точки в множество A.

Для выпуклой оболочки из более чем 3 точек алгоритм вычисления количества острых углов такой:

- Если точка удаляется при добавлении новой вершины в выпуклую оболочку, удалить её из множества A.
- После удаления всех освещённых из добавляемой вершин следует добавить в A вершины, смежные с добавленной и обладающие острым углом и добавляемую вершину, если угол при ней острый. Естественно также нужно исключить из A те из смежных с добавляемой точек, углы при которых не острые. (см. рис. 2)

Далее рассмотрим задачу вычисления функции h — расстояния от выпуклой оболочки до стандартного прямоугольника.[1]

Расчёт расстояния от точки до прямоугольника и от отрезка до прямоугольника как подзадача входит в расчёт для выпуклой оболочки, поэтому здесь рассмотрен не будет. Рассмотрение метода вычисления расстояния до отрезка рассмотрено в разделе *Используемые алгоритмы и структуры данных*.

В случае треугольника достаточно проверить расстояние от каждой стороны, если центр прямоугольника не лежит внутри треугольника, если центр внутри, то g автоматически становится 0, так как выпуклая оболочка включает в себя свою внутренность.

Рассмотрим случай общей выпуклой оболочки: учитывая что расстояние до выпуклой оболочки может только уменьшаться, нам нужно лишь проверить, не попадает ли прямоугольник в получаемую при добавлении вершины область, и вычислить расстояние до добавляемых отрезков.

Таким образом мы знаем как индуктивно вычислять расстояние от выпуклой оболочки до стандартного прямоугольника.

[1]Подробный код методов реализующих эту задачу доступен в приложении А.

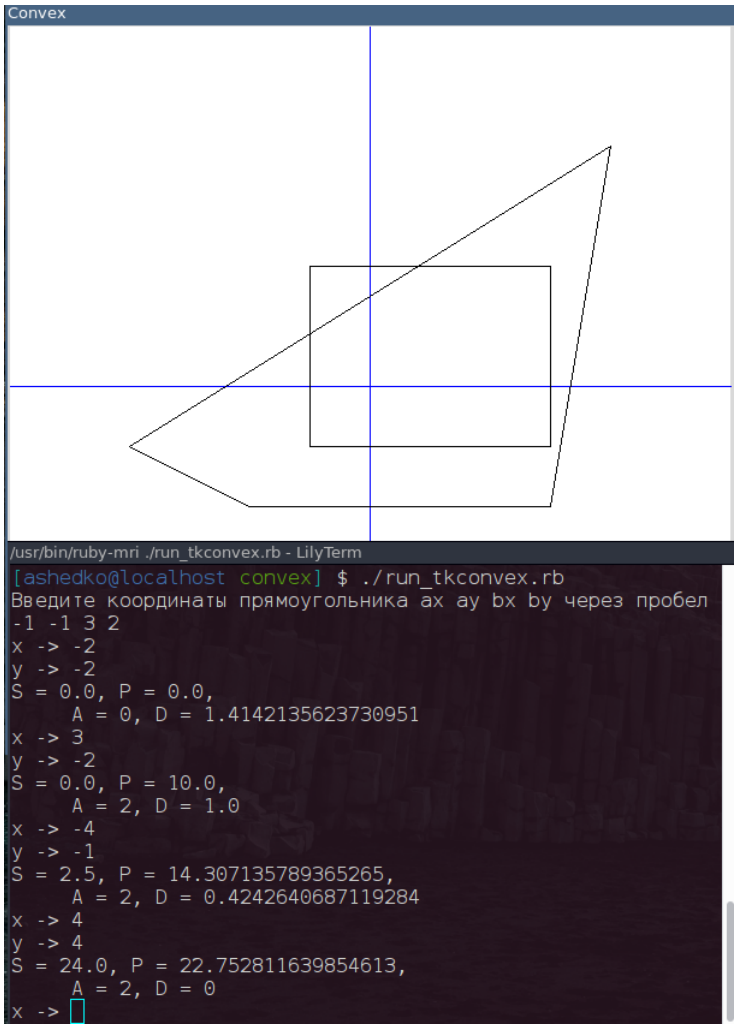

Рис. 3. Иллюстрация работы программы

Используемые алгоритмы и структуры данных

Структуры данных

Дек (двухсвязный список), используемый в эталонном проекте не нуждается в пояснениях.

При вычислении количества острых углов используется структура данных *множество*. По своей сути она представляет собой хэш, в котором ключи и есть данные. В работе она используется из-за простоты добавления и удаления элементов, так как `Set` в языке Ruby позволяет выполнять над собой стандартные теоретико-множественные операции.

Вычисление расстояния от отрезка до прямоугольника

Прежде чем начать работу с отрезком проверим его на пересечение с прямоугольником. (алгоритм описан ниже) В начале вычислим положение концов отрезка относительно прямоугольника, воспользовавшись таблицей,

Характеристика	Номер четверти
$(\mathbf{0}, \mathbf{0})$	0
$(\mathbf{0}, \mathbf{1})$	1
$(\mathbf{1}, \mathbf{0})$	2
$(\mathbf{1}, \mathbf{1})$	3

где характеристика — результат сравнения координат точки с координатами центра прямоугольника (первая координата — горизонтальная, вторая — вертикальная; см. рисунок 4).

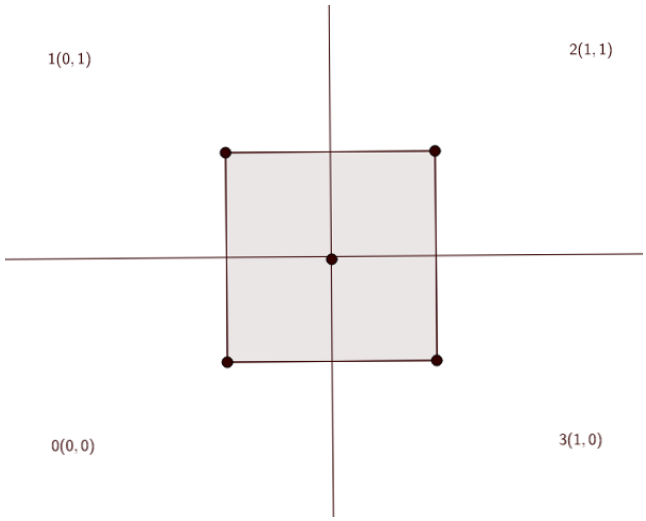

Рис. 4. Разделение пространства вокруг прямоугольника

Затем проверим не является ли отрезок точкой, и если является, остаётся только проверить расстояние до ближайших двух сторон прямоугольника при помощи метода вычисления расстояния от точки до отрезка.

Если же отрезок невырожденный, возможно 3 случая его положения относительно прямоугольника (Рис. 5)

a) оба конца лежат в одной четверти;

b) концы в соседних четвертях;

c) концы в противоположных четвертях.

В случае **a** необходимо вычислить: расстояние от отрезка до ближайшей вершины прямоугольника и расстояния от концов отрезка до соответствующих им сторон прямоугольника (см. приложение А) после чего взять минимальное из них В случае **b** необходимо вычислить: расстояние от отрезка до вершин прямоугольника в соответствующих четвертях и расстояния от концов отрезка до стороны прямоугольника. Далее аналогично **a** взять минимальное расстояние. В случае **c** нужно всего лишь взять меньшее из расстояний от вершин, не лежащих в одной четверти с концами отрезка, до отрезка.

Таким образом можно быстро находить расстояние от отрезка до прямоугольника. В данном случае задача упрощается тем, что если отрезок целиком внутри прямоугольника, то расстояние до него равно 0.

Выгода по сравнению с «лобовым» методом методом проверки алгоритм работает примерно в 3 раза быстрее (в «лобовом» методе производится 12 операций вычис-

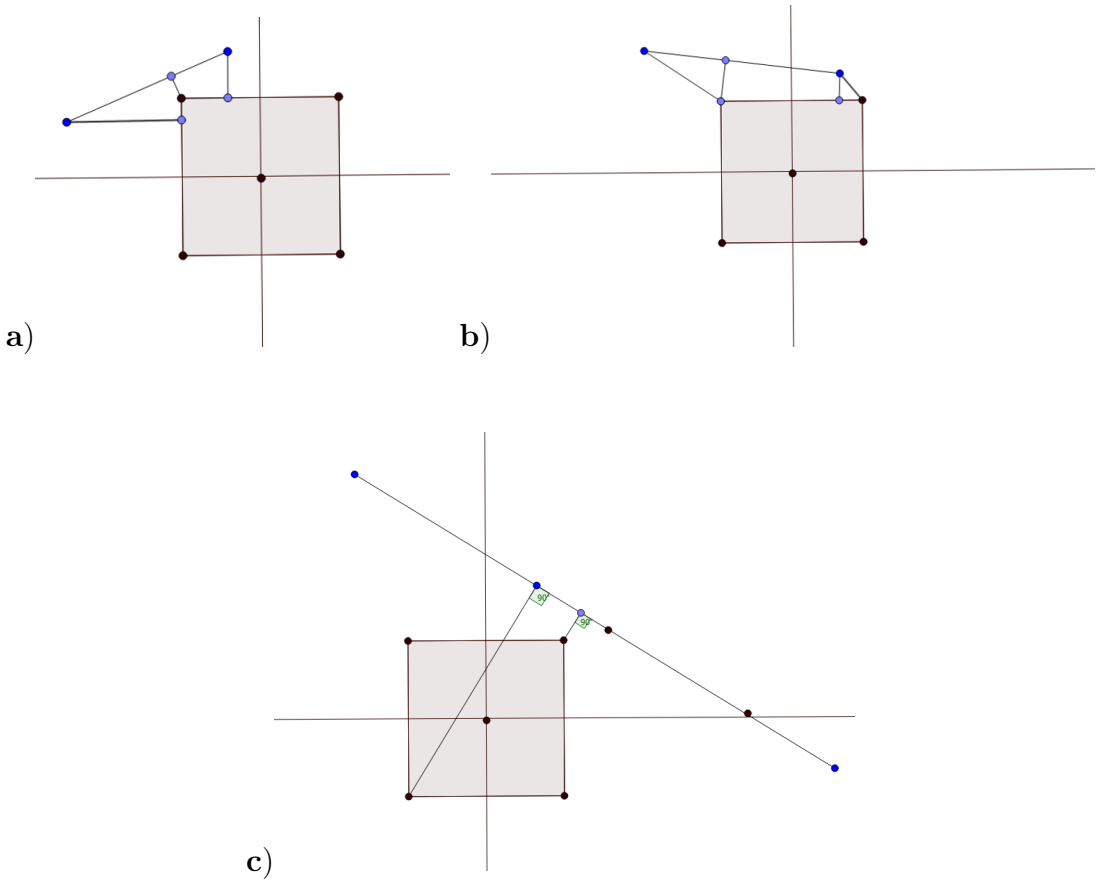

Рис. 5. Различные варианты расположения точек

ления длины, этот алгоритм позволяет ограничиться 4) Идея алгоритма автором найдена в заметках конференции, посвящённой базам данных [2]

Вычисление пересечения отрезка и прямоугольника (*segment clipping*)

Алгоритм Лианга-Барского [3] использует параметрическое уравнение прямой и неравенства, описывающие прямоугольник (clipping window), для вычисления пересечений между прямой и прямоугольником. С помощью этих пересечений можно выяснить какая часть отрезка пересекает прямоугольник.

Параметрическое уравнение прямой:

$$x = x_0 + u(x_1 - x_0) = x_0 + t\Delta x$$

$$y = y_0 + u(y_1 - y_0) = y_0 + t\Delta y$$

Точка находится внутри прямоугольника, если: $x_{\min} \leqslant x_0 + t\Delta x \leqslant x_{\max}$ и $y_{\min} \leqslant y_0 + t\Delta y \leqslant y_{\max}$, где:

$p_1 = -\Delta x, q_1 = x_0 - x_{\min}$

$p_2 = \Delta x, q_2 = x_{\max} - x_0$

$p_3 = -\Delta y, q_3 = y_0 - y_{\min}$

$p_4 = \Delta y, q_4 = y_{\max} - y_0$

Чтобы получить пересечение, нужно учесть следующее.

1. Если отрезок параллелен i-тому ребру $p_i = 0$ для этого ребра.
2. Если для этого $i, q_i < 0$, отрезок полностью находится вне прямоугольника.
3. Когда $p_i < 0$ отрезок направлен внутрь через это ребро, а если $p_i > 0$ —наружу.

4. Для ненулевого p_k, $u = \frac{q_i}{p_i}$ даёт точку пересечения.
5. Вычислить u_1, u_2. Для u_1, посмотреть на рёбра, где $p_i < 0$. u_1 — максимум из $\{0, q_i/p_i\}$.Для u_2, посмотреть на $p_i > 0$. u_2 —минимум из $\{1, q_i/p_i\}$. Если $u_1 > u_2$, то отрезок находится снаружи и потому также не пересекает прямоугольник.

Возможные обобщения

В качестве более общей задачи можно рассматривать:
- задачу в пространстве произвольной размерности;
- прямоугольник общего положения (решается при помощи преобразования поворота и сдвига для координат точек);
- поиск расстояния до других фигур (круг, треугольник, произвольная область ограниченная конечным числом ломанных). Возможное решение — конформное отображение.

5. Модификация проекта «Изображение проекции полиэдра»

Постановка задачи

Все рёбра делятся на три класса: *полностью видимые*, *видимые частично* и *полностью невидимые*. Назовём грань «гранью с полностью видимыми рёбрами», если все образующие её рёбра полностью видимы. Если все образующие грань рёбра полностью невидимы, то грань будем называть «гранью с полностью невидимыми рёбрами». Все остальные грани будем называть «гранями с частично видимыми рёбрами». Модифицируйте эталонный проект таким образом, чтобы определялась и печаталась следующая характеристика полиэдра: *сумма периметров проекций «граней с частично видимыми рёбрами», проекция центра которых находится строго вне квадрата единичной площади с центром в начале координат и сторонами, параллельными координатным осям.* (28).

Теоретические аспекты

Задача изображения проекции полиэдра без невидимых рёбер может быть решена следующим образом: будем отображать отрезки — проекции рёбер полиэдра, причём части рёбер, *затенённые* гранями полиэдра, учитывать не нужно. Полиэдр задаётся тремя наборами: вершин, рёбер, граней, соответствием между ними и вращением трёхмерного пространства. Проектирование производится при помощи бесконечной перспективы (Физическая интерпретация: свет падает на объект из бесконечно удалённой точки параллельным потоком и мы видим тень объекта на плоскости, перпендикулярной вектору направления света). Далее вектор направления света будем называть *вектором проектирования*.

Механизм проектирования предельно прост: после преобразования координат (пространственный поворот на соответствующие углы Эйлера и гомотетия) для отображения точки достаточно лишь *забыть* её z-координату.

Вкратце опишем алгоритм *затенения* ребра гранью: Рассмотрим произвольное ребро. Каждая грань разбивает его на *затенённые* и *освещённые* участки. Несложно

показать что в нашем случае (грани-выпуклые многоугольники) освещённая часть представляет собой либо ребро, либо Ø. (см. рис.6)

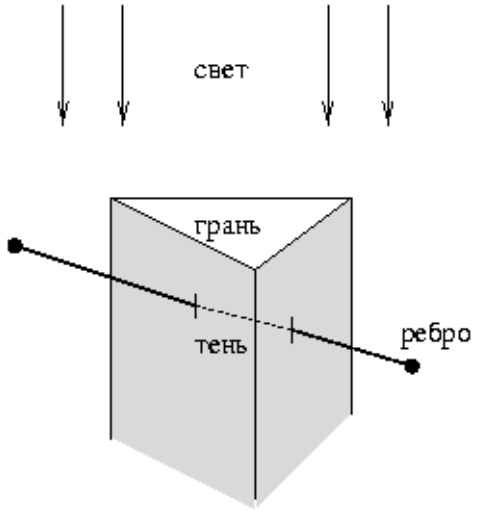

Рис. 6. Затенение ребра гранью

Для получения конечного результата необходимо учесть тени от всех граней (Утверждение открывает широкий простор для оптимизации — нахождения достаточного условия).

Воспользовавшись тем фактом что ребро — одномерный объект, задачу нахождения пересечения освещённых областей от всех граней можно свести к задаче поиска дополнения покрытия интервала $(0, 1)$ набором других интервалов. Тень от грани — пересечение ребра и открытой бесконечной призмы, сверху ограниченной гранью, а с «боков» — плоскостями, содержащими рёбра грани и вектор проектирования, нормали которых направлены внутрь призмы. Процесс построения призмы показан на рисунке 7.

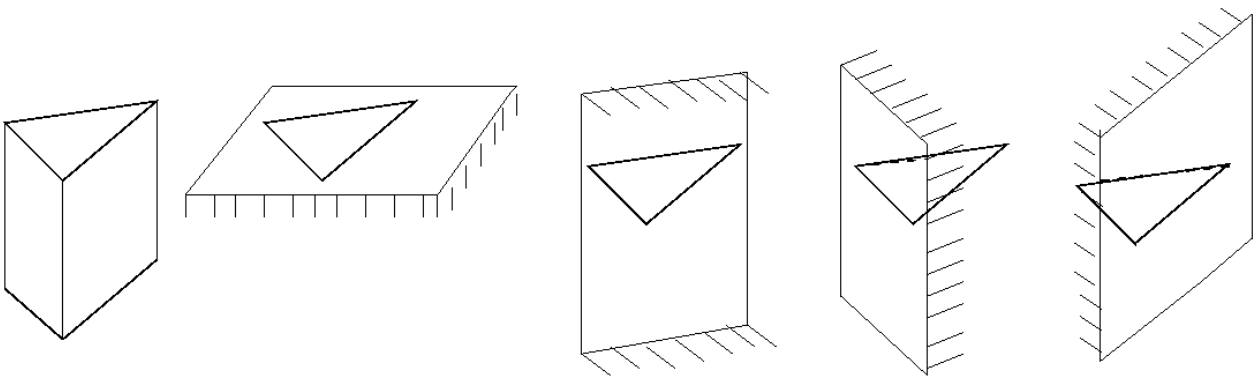

Рис. 7. Построение призмы затенения

Детали реализации

Таким образом мы можем сказать какие рёбра будут *видимыми*, а какие *невидимыми*, что соответственно позволяет нам определить какие гани являются *частично видимыми*.

Метод грани, решающий эту задачу:

```
class Facet

  ...

  def part_vis?
    return false if !@edges
    pr1,pr2 = true, true
    trtab = @edges.map{|edge| [edge.compl_visible?,edge.invisible?]}
    trtab.each{|x|pr1&&=x[0];pr2&&=x[1]}
    !(pr1||pr2)
  end
end
```

Естественно для этого требуется знать какие рёбра принадлежат грани. Воспользуемся структурой данных «хэш» для удаления дубликатов рёбер и затем поставим каждой грани в соответствие набор рёбер, ей принадлежащих. Это можно сделать при помощи простой проверки что множество вершин грани содержит концы ребра.

```
def edges_uniq
  edges = {}
  @edges.each do |e|
    if edges[[e.beg, e.fin]].nil? && edges[[e.fin, e.beg]].nil?
      edges[[e.beg, e.fin]] = e
    end
  end
  @edges = edges.values
end
```

Наконец, для определения положения центра грани необходимо проверить что его координаты в изначальной системе отсчёта соответствуют неравенствам: $|x_c| > 0,5$ и $|y_c| > 0,5$. Откуда следует что параметры проектирования должны быть известны в классе `Polyedr`. Код метода, решающего эту задачу:

```
def outside_sqr?(alpha,beta,gamma,c)
  cent = center
  cent = cent.rz(-gamma).ry(-beta).rz(-alpha)*(1/c)
  (cent.x>0.5||cent.x<-0.5)&&(cent.y>0.5||cent.y<-0.5)
end
```

В итоге все методы, требующиеся для решения поставленной задачи описаны и остаётся только проверить все грани на соответствие условиям задачи и сложить периметры граней им удовлетворяющих. На рисунке 8 представлен результат работы программы, где точками отмечены центры граней, удовлетворяющих условиям задачи.

Возможные обобщения

Задачу можно обобщить, например, потребовав покрывать грани какой-либо текстурой и описывать условия задачи с учётом текстуры. Другой задачей может быть поиск суммы расстояний между правильными пирамидами. построенными на гранях полиэдра, как на основаниях.

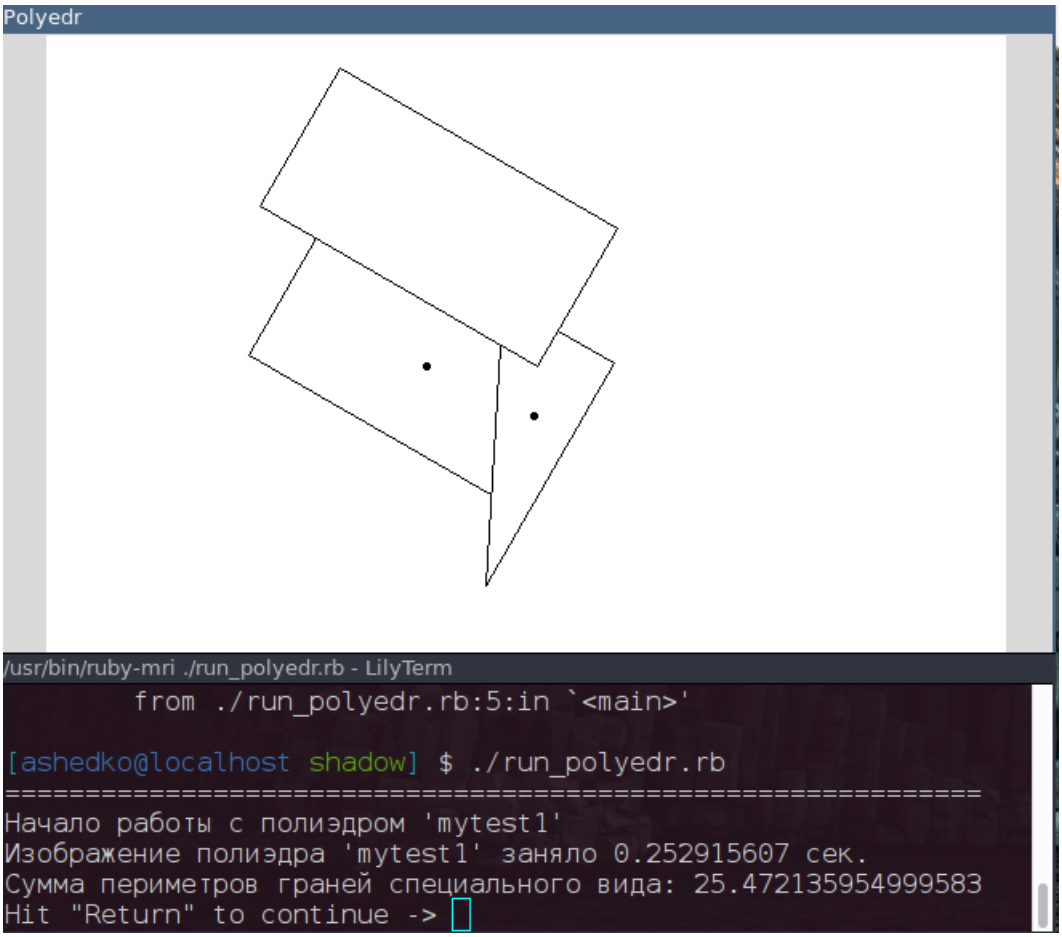

Рис. 8. Иллюстрация работы программы

Список литературы и интернет-ресурсов

[1] `https://home.mephi.ru/files/2373/material_ici_toc.zip/index.html` — Описание проекта «Выпуклая оболочка».

[2] *Advances in Spatial and Temporal Databases: 9th International Symposium.* — SSTD 2005, Angra Dos Reis Brazil, August 22-24, 2005, Proceedings, Стр. 333.

[3] Liang, Y.D., and Barsky, B., *A New Concept and Method for Line Clipping.* — ACM Transactions on Graphics, 3(1):1-22, January 1984.

[4] `http://ru.wikipedia.org/wiki/Ruby` — Википедия (свободная энциклопедия) о языке Ruby.

[5] `https://home.mephi.ru/files/3214/material_ici_toc.zip/index.html` — Описание проекта «Изображение проекции полиэдра».

[6] С.М. Львовский. *Набор и вёрстка в системе LaTeX, 3-е изд., испр. и доп.* — М., МЦНМО, 2003. Доступны исходные тексты этой книги.

[7] D. E. Knuth. *The TeXbook.* — Addison-Wesley, 1984. Русский перевод: Дональд Е. Кнут. *Все про TeX.* — Протвино, РДТеХ, 1993.

[8] `https://home.mephi.ru/files/2077/material_ici_toc.zip/index.html` — Описание проекта *«Стековый компилятор формул»* Е.А. Роганов

[9] `http://ruby-doc.org/` — Документация языка Ruby

6. Приложение А

Проект «Выпуклая оболочка». Методы, отвечающие за вычисление расстояния от отрезка до прямоугольника.

```
class R2Point
 ...

 # Расстояние от точки до отрезка
 def dist_segm(a,b)
   l = a.dist(b)
   return self.dist(a) if l==0
   t = [0, [1, (self+(a*(-1))).dot(b+(a*(-1)))/(l*l)].min].max
   proj = a + (b + (a * (-1))) * t
   # p proj
   self.dist(proj)
 end

 def distance_rect(endp,a,b)
# Алгоритм описан на странице 333 журнала
# Advances in Spatial and Temporal Databases:
# 9th International Symposium, SSTD 2005, Angra Dos Reis
# Brazil, August 22-24, 2005, Proceedings
   a,b = R2Point.new([a.x,b.x].min,[a.y,b.y].min)
                    ,R2Point.new([a.x,b.x].max,[a.y,b.y].max)
   rect = [a,R2Point.new(a.x,b.y),b,R2Point.new(b.x,a.y)]
   return self.pdist(a,b,rect) if endp == self
   return 0 if self.intersect_rect(endp,a,b)
   # Вычисление положения конца отрезка
   t1=self.x<=>(a.x+b.x)/2
   t2=self.y<=>(a.y+b.y)/2
   p1 = D[[(t1+t1.abs)/2, (t2+t2.abs)/2]]
   u1=endp.x<=>(a.x+b.x)/2
   u2=endp.y<=>(a.y+b.y)/2
   p2 = D[[(u1+u1.abs)/2, (u2+u2.abs)/2]]
   # Соотношение между концами отрезка
   case (p1 - p2)%4
     # Концы отрезка в одной четверти
   when 0
     # Необходимо отсортировать концы
     # отрезка для определения их взаимного
     # расположения относительно вершины прямоугольника
     self.x<=endp.x? l,r = self,endp : l,r = endp,self
     if p1 == 1 || p1 == 2
       return [l.dist_segm(rect[(p1-1)%4],rect[p1]),
               r.dist_segm(rect[(p1+1)%4],rect[p1]),
               rect[p1].dist_segm(self,endp)].min
     else
       return [r.dist_segm(rect[(p1-1)%4],rect[p1]),
               l.dist_segm(rect[(p1+1)%4],rect[p1]),
```

```
                rect[p1].dist_segm(self,endp)].min
      end
    when 1, 3 #Концы отрезка в соседних четвертях
      return [self.dist(rect[p1]),endp.dist(rect[p2]),
              rect[p1].dist_segm(self,endp),
              rect[p2].dist_segm(self,endp),
              endp.dist_segm(rect[p1],rect[p2])
             ].min
    # Концы отрезка в противоположных четвертях
    when 2
      return [rect[p1-1].dist_segm(self, endp),
              rect[p2-1].dist_segm(self, endp)
             ].min
    end
  end


  def pdist(a,b,rect)
    t1=@x<=>(a.x+b.x)/2
    t2=@y<=>(a.y+b.y)/2
    p1 = D[[(t1+t1.abs)/2, (t2+t2.abs)/2]]
    return 0 if self.inside?(a,b)
    [self.dist_segm(rect[(p1-1)%4],rect[p1]),
    self.dist_segm(rect[(p1+1)%4],rect[p1]),
    self.dist(rect[p1])].min
  end
#вычисление пересечения отрезка
  def intersect_rect?(endp,a,b)
    # Liang, Y.D., and Barsky, B.,
    # "A New Concept and Method for Line Clipping",
    # ACM Transactions on Graphics, 3(1):1-22, January 1984.
    l,r = endp,self
    # p l,r
    dx, dy = r.x - l.x, r.y - l.y
    q = [l.x-a.x,b.x-l.x,l.y-a.y,b.y-l.y]
    p = [-dx,dx,-dy,dy]
    d1,d2=[],[]
    (0..3).each {|i|
      if p[i]==0
        return false if q[i]<0
      else
        p[i]<0 ? d1<<q[i]/p[i] : d2<<q[i]/p[i]
      end
    }
    !((d1<<0).max>(d2<<1).min)
  end


# точка внутри треугольника
  def in_triangle?(p1,p2,p3)
```

```
    d=((p2.y-p3.y)*(p1.x-p3.x)+(p3.x-p2.x)*(p1.y-p3.y))
    lambd1 = ((p2.y-p3.y)*(@x-p3.x)+(p3.x-p2.x)*(@y-p3.y))/d
    lambd2 = ((p3.y-p1.y)*(@x-p3.x)+(p1.x-p3.x)*(@y-p3.y))/d
    lambd3 = 1 - lambd1 - lambd2
    eps = (10**-15).to_f
    lambd1>eps&&lambd2>eps&&lambd3>eps
  end

# Скалярное произведение
  def dot(other)
    @x*other.x+@y*other.y
  end

# принадлежность точки интервалу
def between?(x,a,b)
  x<b&&x>a||x<a&&x>b

# Сумма векторов
  def +(other)
    R2Point.new(@x + other.x, @y + other.y)
  end

# Произведение вектора на скаляр
  def *(scalar)
    R2Point.new(@x * scalar, @y * scalar)
  end

  ...

end
```