|  |  |
| --- | --- |
| **Gerb-BMSTU_01** | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

**Лабораторная работа №9**

**по дисциплине «Компьютерная графика»**

|  |  |
| --- | --- |
| **Тема: Посторение трехмерных поверхностей алгоритмом плавающего горизонта**  **Студент:** Пересторонин Павел  **Группа:** ИУ7-43Б  **Оценка (баллы) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**  **Преподаватель:** Куров А. В. |  |

Москва.

2020 г.

Цель работы: изучение и программная реализация алгоритма Плавающего горизонта построения трехмерных поверхностей.

Результат: должна быть разработана программа, позволяющая осуществлять ввод пределов и шага изменения координат x, z, выбора уравнения поверхности из заранее сформированного списка, построение поверхности. Должен быть обеспечен поворот изображения (поверхности) вокруг каждой из трех координатных осей. Система координат должна быт неподвижной. Выполнить масштабирование для обеспечения размещения исходного изображения целиком в пределах поля вывода

Список уравнений поверхностей задается в отдельном модуле.

**Теоретический материал**

Алгоритм плавающего горизонта – построение поверхностей заданных неявным уравнением вида F(x, y, z) = 0.

Идея данного алгоритма состоит в том, что мы заданную поверхность отсекаем плоскостями, перпендикулярными оси Z (с каким-то определенным нами шагом). Наблюдатель распологается на положительной координате оси Z (0, 0, number), number > 0 и смотрит в сторону начала координат. Уравнение плоскостей –

Z = const. Следует заметить, что в данном случае задача решается в пространстве изображений.

Основные этапы алгоритма:

1. Рассматриваемая поверхность рассекается плоскостями, перпендикулярными плоскостями, перпендикулярными оси Z. В каждом сечении получается кривая, описываемая уравнением y = f(x, z=const) или x = q(y, z=const).

2. Полученные кривые можно спроецировать на плоскость Z = 0 и изобразить видимые части каждой кривой. Изображение надо строить начиная с кривой, полученной в ближайшем к наблюдателю сечении. Кривая, полученная в сечении ближайшей плоскостью является видимой. То же самое можно сказать и про 2 кривую (тоже будет видима; вторая кривая расположена либо выше первой, либо ниже первой (в любом случае видим; в частном случае совпадают, но тогда тоже будем видеть)). Начиная с 3 кривой надо решать задачу определения видимости точек кривой.

Решение задачи определения видимости кривой:

1. Определение границ построения поверхности: сначала определяем границы по х (границы кривых), затем – по Z (кол-во секущих плоскостей = границы по Z / шаг по Z). Шаг по Х в лучшем случае = 1 пиксел.

2. Изображаем первые 2 кривые, полученные сечением (по уравнению кривой находим кооридинаты У и чертим кривые).

3. Изображаем остальные кривые.

Участок кривой будет видимым, если он располагается выше верхней кривой или ниже нижней кривой. Если мы можем вычислить значение функции, вычисляя задачу для каждого пикселя, то задача решается легко: мы должны определить значение функции в очередной точки на кривой y = f(x, z=const), и сравнить с максимальным (или минимальным) значением: если полученное значение больше (меньше) максимального (минимального) значения. Высвечиваем все видимые точки.

Почему алгоритм называется плаващим горизонтом?

Во-первых следует сказать, что рассматирвается 2 горизонта: верхний и нижний. Верхний горизонт – видимые участки кривых с наибольшим значением ординат.

Нижний горизонт – видимые участки кривых с наименьшим значением ординат.

Ситуация похожа на алгоритм Кируса-Бека: мы держим максимальные значения и минимальные значения и сравниваем каждое очередное значение с максимальным и минимальным и если новое значение превосходит максимальное или меньше минимального, то точка отображается, а значения обновляются.

С одной стороны можно называть горизонтом то, что мы уже назвали, а можно называть массивы, в которых мы для координаты Х храним максимальную (верхний горизонт) и минимальную (нижний горизонт) ординаты.

Работа алгоритма сводится к вычислению значения у(х, z=const) и сравнению y(x, z=const) > Ymax(x), если y(x, z=const) > Ymax, то Ymax = y(x, z=const). В противном случае надо проверить y(x) < Ymin, если условие выполняется, то Ymin(x) = y(x, z=const) – процедура поддержания горизонта (проверка экстремумов для очередной точки)

Точка текущей кривой невидима, если она находится между 2 горизонтами.

Если по каким-либо причинам мы не можем (не хотим) вычислять значение функции, то алгоритм требуется усовершенствовать:

Если видимость точек одинакова, то ситуация простая: кривая между ними либо видима полностью, либо полностью невидима. Однако если точки имеют разные видимости, то можно решить простым способом: чертить участок кривой исходя из видимости первой точки, однако у такого решения есть большой минус: где-то мы будем изображать в действительности невидимые участки, или же не изображать видимые участки. Эту проблему надо решить и решение следующее:

Находится точка пересечения горизонта и текущей кривой и изображаем видимый участок. С учетом того, что dx – мало и функция гладкая, то есть не имеет резких изменений, то мы аппроксимируем функцию отрезком.

Распишем вычисления для каждой прямой в общем виде:

m = (yn – yk) / (xn – xk), где n = k + 1 – соседние точки

y = m(x – xk) + yk

mпред(x – xk) + ykпред = mтек(x – xk) + yтек

x(mпред – mтек) = xk(mпред – mтек) + (yтек – yпред)

x = xk – (утек – упред) / (mтек – mпред)

x = xk – (утек – упред) / (dyтек – dyпред) / dx

x = xk – (dx(утек – упред) / (dyтек – dyпред)

Если текущая точка видима, то изобразить от точки пересечения до текущей точки, если видима предыдущая точка, то изобразить от предыдущей точки до точки пересечения.

При повороте можно столкнуться с небольшой проблемой: отдельные прямые не будут восприниматься как поверхность. Качество изображения улучшится, если начальные точки кривых соединить отрезками (это придает ощущение того, что кривые не являются чем-то отдельным, а составляют единое целое). Это называется левым и правым боковыми ребрами.

Алгоритм:

1. Обработать левое боковое ребро.

Если точка является первой точкой кривой и лежит на первой кривой, то запомнить ее (P). Если точка принадлежит не первой кривой, то соединить eе с точкой P и запомнить в качестве точки p.

2. Для каждой точки кривой сделать следующие действия:

\* Определить видимость точки (если y(x) > Ymax(x) или y(x) < Ymin(x), то точка видима, иначе – невидима).

\* Если видимость кривой изменилась, то найти точку пересечения кривой с горизонтом (с верхним или нижним, выяснить).

\* Если текущий сегмент кривой видим, то изобразить его полностью.

\* Если видимость изменилась, то

если текущая точка невидима, то изобразить участок кривой от предыдущей точки до точки пересечия

если текущая точка видима, то изобразить участой кривой от точки пересечения до текущей точки.

\* Заполнить массивы верхнего и нижнего горизонта

3. Обработать правое боковое ребро.

**Исходный код программы.**

# Функция, которая удаляет ложные ребра

**def** **make\_uniq**(sections):

**for** section **in** sections:

section.sort()

**return** list(filter(**lambda** x: (sections.count(x) % **2**) == **1**, sections))

# Функция проверки, принадлежит ли точка point отрезку section

**def** **point\_in\_section**(point, section):

**if** abs(vect\_mul(get\_vect(point, section[**0**]), get\_vect(\*section))) <= **1e-6**:

**if** (section[**0**] < point < section[**1**] **or** section[**1**] < point < section[**0**]):

**return** True

**return** False

# Функция получения "элементарных" отрезков многоугольника

# (см. мой алгоритм)

**def** **get\_sections**(section, rest\_points):

points\_list = [section[**0**], section[**1**]]

**for** p **in** rest\_points:

**if** point\_in\_section(p, section):

points\_list.append(p)

points\_list.sort()

sections\_list = list()

**for** i **in** range(len(points\_list) - **1**):

sections\_list.append([points\_list[i], points\_list[i + **1**]])

**return** sections\_list

# Функция выброса ложных ребер из результирующего многоугольника

**def** **get\_uniq\_sections**(figure):

all\_sections = list()

rest\_points = figure[**2**:]

**for** i **in** range(len(figure)):

cur\_section = [figure[i], figure[(i + **1**) % len(figure)]]

all\_sections.extend(get\_sections(cur\_section, rest\_points))

rest\_points.pop(**0**)

rest\_points.append(figure[i])

**return** make\_uniq(all\_sections)

# Функция рисования результата (многоугольника)

**def** **draw\_figure**(figure):

**for** section **in** get\_uniq\_sections(figure):

draw\_section(round(section[**0**][**0**]), round(section[**0**][**1**]),

round(section[**1**][**0**]), round(section[**1**][**1**]), res\_color)

# Функция вычисления векторного произведения

**def** **vect\_mul**(v1, v2):

**return** v1[**0**] \* v2[**1**] - v1[**1**] \* v2[**0**]

# Функция вычисления скалярного произведения

**def** **scalar\_mul**(v1, v2):

**return** v1[**0**] \* v2[**0**] + v1[**1**] \* v2[**1**]

# Функция проверки многоугольника на выпуклость

**def** **check\_polygon**(verteces):

**if** len(verteces) < **3**:

**return** False

sign = **1** **if** vect\_mul(get\_vect(verteces[**1**], verteces[**2**]),

get\_vect(verteces[**0**], verteces[**1**])) > **0** **else** -**1**

**for** i **in** range(**3**, len(verteces)):

**if** sign \* vect\_mul(get\_vect(verteces[i - **1**], verteces[i]),

get\_vect(verteces[i - **2**], verteces[i - **1**])) < **0**:

**return** False

**if** sign < **0**:

verteces.reverse()

**return** True

# Функция получения внутренней нормали к грани

# p1, p2 - вершины грани, cp - одна из точек многоугольника для проверки

# внутренняя нормаль или внешняя нашлась

**def** **get\_normal**(p1, p2, cp):

vect = get\_vect(p1, p2)

norm = [**1**, **0**] **if** vect[**0**] == **0** **else** [-vect[**1**] / vect[**0**], **1**]

**if** scalar\_mul(get\_vect(p2, cp), norm) < **0**:

**for** i **in** range(len(norm)):

norm[i] = -norm[i]

**return** norm

# Функция нахождения нормалей ко всем граням многоугольника (отсекателя)

**def** **get\_normals\_list**(verteces):

length = len(verteces\_list)

normal\_list = list()

**for** i **in** range(length):

normal\_list.append(get\_normal(verteces[i], verteces[(i + **1**) % length],

verteces[(i + **2**) % length]))

**return** normal\_list

# Функция проверки принадлежности точки point отсекателю относительно

# грани [p1, p2]

**def** **check\_point**(point, p1, p2):

**return** True **if** vect\_mul(get\_vect(p1, p2), get\_vect(p1, point)) <= **0** **else** False

**def** **find\_intersection**(section, edge, normal):

wi = get\_vect(edge[**0**], section[**0**])

d = get\_vect(section[**0**], section[**1**])

Wck = scalar\_mul(wi, normal)

Dck = scalar\_mul(d, normal)

diff = [section[**1**][**0**] - section[**0**][**0**], section[**1**][**1**] - section[**0**][**1**]]

t = -Wck / Dck

**return** [section[**0**][**0**] + diff[**0**] \* t, section[**0**][**1**] + diff[**1**] \* t]

# Функция отсечения многоугольника относительно одной грани отсекателя

**def** **edgecut\_figure**(figure, edge, normal):

res\_figure = list()

**if** len(figure) < **3**:

**return** []

prev\_check = check\_point(figure[**0**], \*edge)

**for** i **in** range(**1**, len(figure) + **1**):

cur\_check = check\_point(figure[i % len(figure)], \*edge)

**if** prev\_check:

**if** cur\_check:

res\_figure.append(figure[i % len(figure)])

**else**:

res\_figure.append(find\_intersection([figure[i - **1**],

figure[i % len(figure)]], edge, normal))

**else**:

**if** cur\_check:

res\_figure.append(find\_intersection([figure[i - **1**],

figure[i % len(figure)]], edge, normal))

res\_figure.append(figure[i % len(figure)])

prev\_check = cur\_check

**return** res\_figure

# Функция отсечения фигуры

**def** **cut\_figure**(figure, cutter\_verteces, normals\_list):

res\_figure = figure

**for** i **in** range(len(cutter\_verteces)):

cur\_edge = [cutter\_verteces[i],

cutter\_verteces[(i + **1**) % len(cutter\_verteces)]]

res\_figure = edgecut\_figure(res\_figure, cur\_edge,

normals\_list[i])

**if** len(res\_figure) < **3**:

**return** []

**return** res\_figure

# Функция-решение задачи (проверка выпуклости и затем отсечение)

**def** **solve**():

**if** **not** check\_polygon(verteces\_list):

mb.showerror("Невыпуклый многоугольник")

**return**

normals\_list = get\_normals\_list(verteces\_list)

cutted\_figure = cut\_figure(figure\_list, verteces\_list, normals\_list)

draw\_figure(cutted\_figure)

Конец кода.

**Интерфейс и примеры работы.**

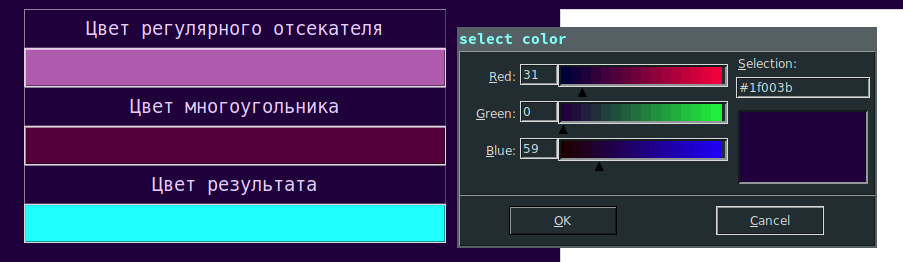
Интерфейс:



Интерфейс предусматривает 2 формата ввода: через поля (координаты) и через выбор мышью точки на плоскости.

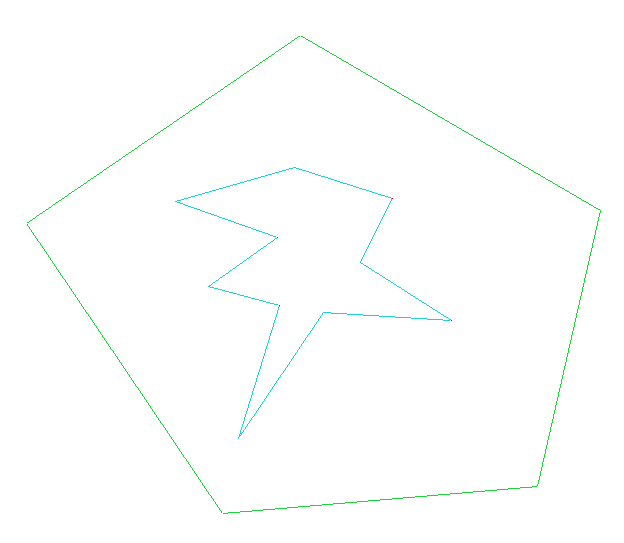
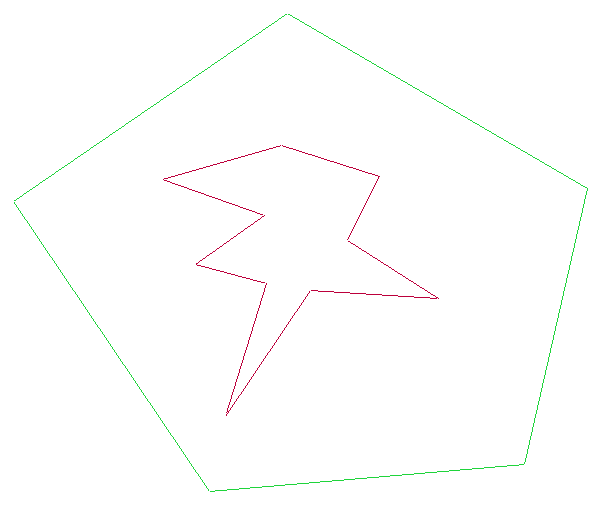
При вводе с помощью мышки границ отсекателя вершины вводятся нажатием правой кнопки мыши. Для замыкания следует нажать кнопку Enter.

При вводе с помощью мышки границ многоугольника вершины вводятся нажатием левой кнопки мыши. Для замыкания следует нажать кнопку **с**. Цвета могут быть любые, используется палитра:

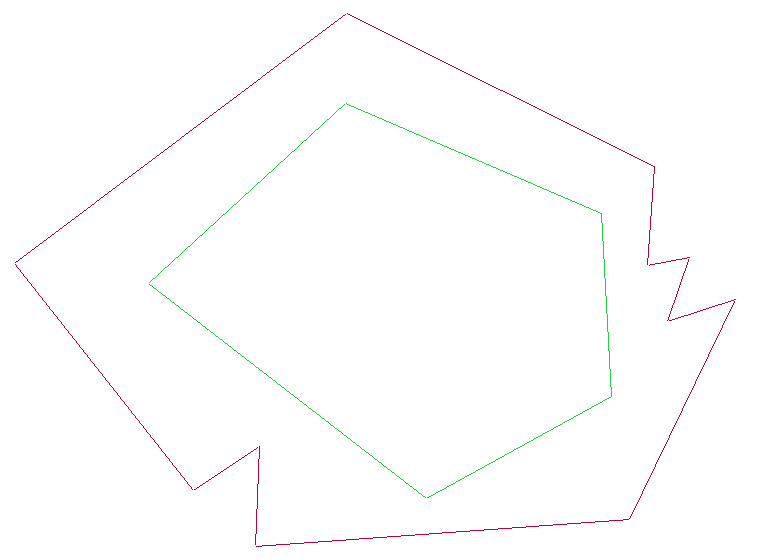


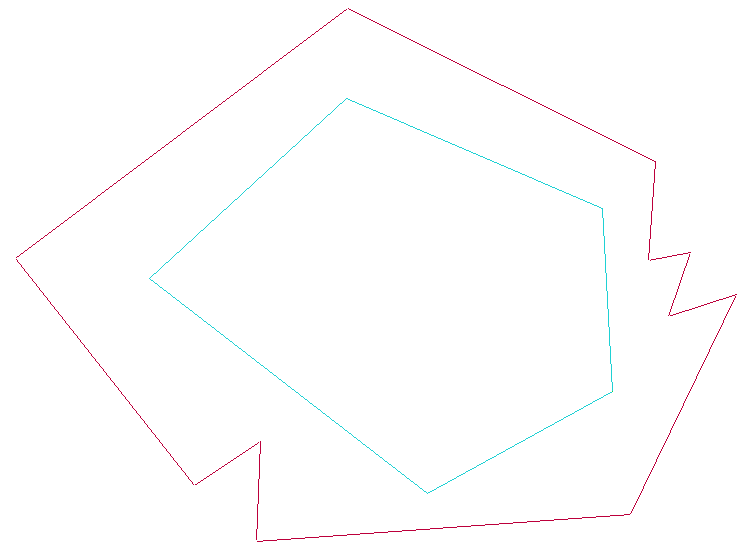
**Примеры работы**

Многоугольник полностью внутри отсекателя:

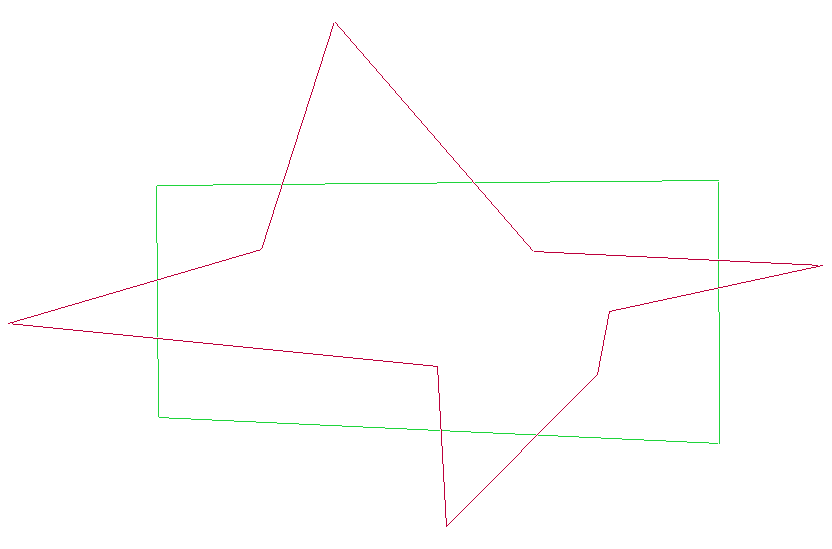


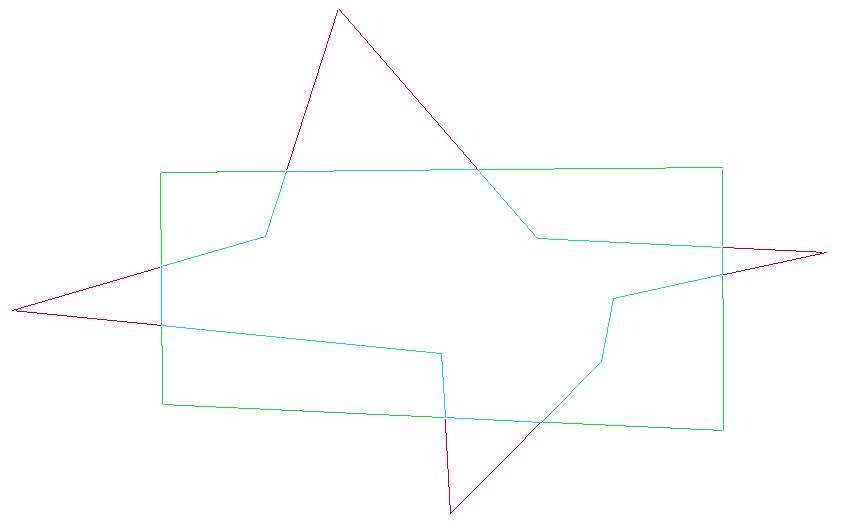
Многоугольник вне отсекателя:



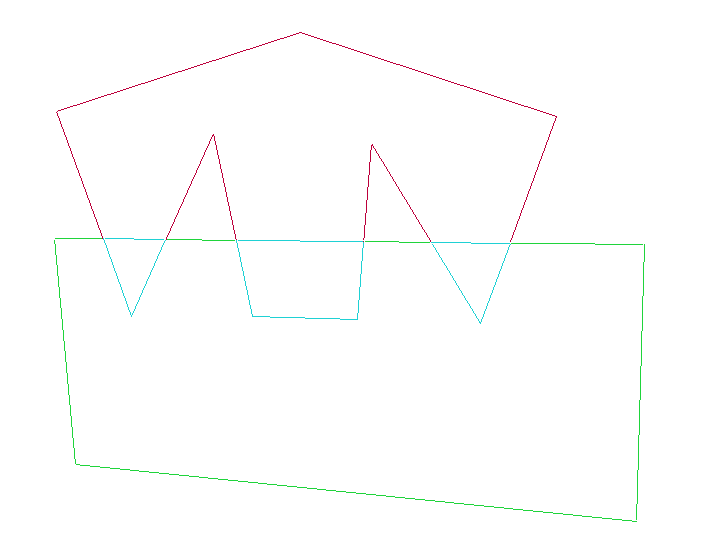
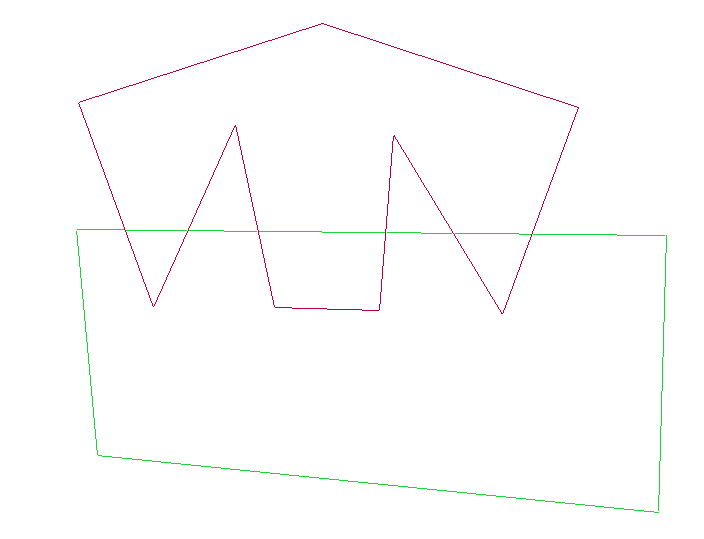


Многоугольник внутри отсекателя (пример похожий на пример с лекции):





Исключительная ситуация с лекции:



Пример, похожий на исключительную ситуацию (однако невидимая часть многоугольника теперь с другой стороны):

