МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ «МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ им. Н. Э. Баумана»

Факультет: Информатика и системы управления (ИУ)

Кафедра: Теоретическая информатика и компьютерные технологии (ИУ9)

КУРСОВАЯ РАБОТА

По теме: Визуализация технологического процесса станков с ЧПУ

Студента очной формы обучения по направлению 010400 кафедры ИУ9

группы ИУ9-51

Нестерчука Данилы Андреевича

Научный руководитель:
доцент Каганов Ю. Т.

МОСКВА 2017

Содержание

Введение 2

1. Постановка задачи 3

2. Общие представление об этапах техпроцесса фрезеровки детали 4

3. G-код 6

4. Выбор формата представления 3-х мерных объектов 7

5. Алгоритмы отсечения 10

5.1 Хранение данных 10

5.2 Общая схема алгоритма 3-х мерного отсечения 11

5.3 Алгоритм Сазерленда — Ходжмана 12

5.4 Функция clipByPlane 14

5.5 Заполнение «дырок» в полигональной сетке 16

5.6 Инверсия отсеченных граней 17

5.7 Выпуклость полученных полигонов 19

6. Заключение 20

7. Список литературы 21

Введение

В современном мире все шире и шире применяется производство деталей на фрезерных, токарных и токарно-фрезерных станках. И это не удивительно, так как цена на такую обработку стремительно падает – цена на литье тех же деталей уже отличается на порядок. Одна из главных причин удешевления работы станков: снабжение их ЧПУ модулями – это сильно экономит человеко-часы, т.к. автоматика берет на себя большую часть работы.

Конечно же, перед отправкой реального материала в станок, хочется проверить корректность программы обработки. Именно в написании такой программы и состоит задача этого курсового проекта. Будущая программа позволит симулировать и визуализировать, с помощью средств компьютерной графики, процесс обработки детали под управлением CAM-программы.

Постановка задачи

Основная задача это работы – создание программы визуализации движения рабочего инструмента и отрезания материала в процессе. Движение фрезы должно выполняться по управляющей программе в формате G-кода для фрезерного станка с 3 степенями свободы. Срезание материала должно выполняться с высокой точностью, что позволит технологу определить ошибки управляющей программы. Технолог должен иметь возможность визуально определить столкновения рабочего инструмента и частей станка, не стоит задачи автоматически определять столкновения. В программу должны будут загружаться следующие компоненты: модели заготовки и фрезы, ограничивающие объекты, управляющая программа и модуль станка (опционально). Так же необходимо реализовать управление камерой и светом. Возможность применения программы к симуляции токарной обработки не рассматривается, т.к. потребуются совершенно другие алгоритмы оптимизации и отсечения.

Общие представления об этапах техпроцесса фрезеровки детали

Перед тем как готовое изделие выйдет со станка, оно должно пройти множество этапов разработки, тестирования и расчетов.

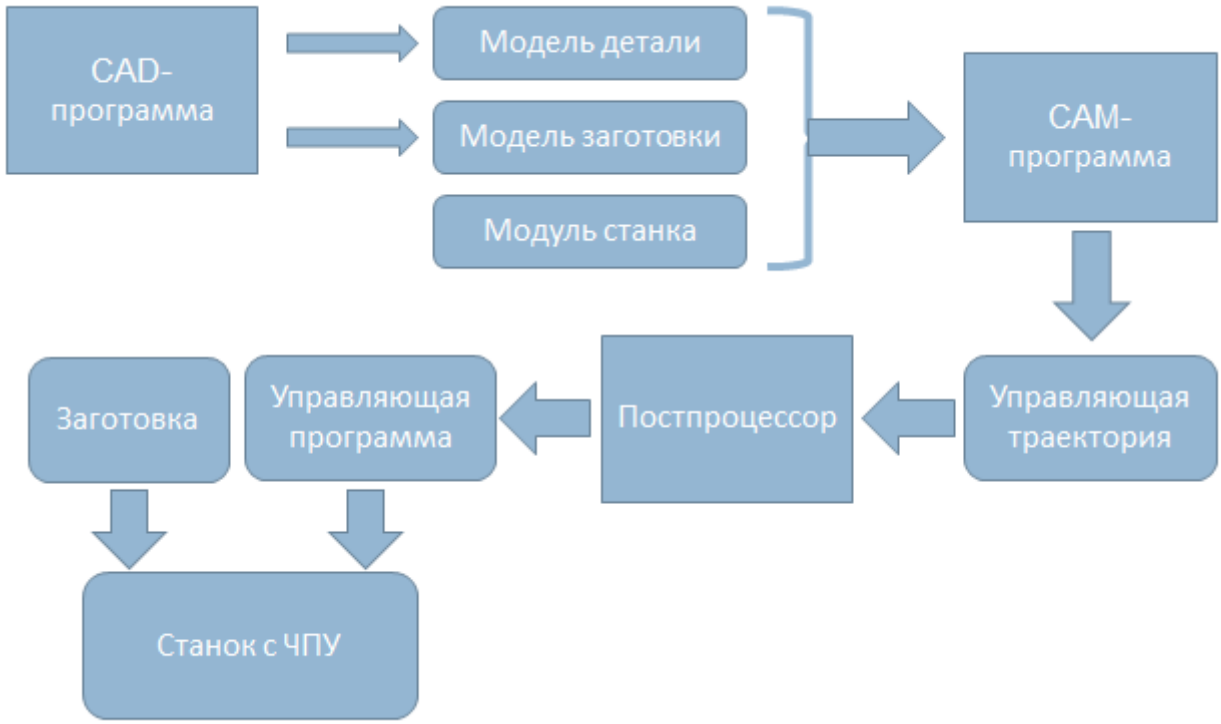

Рисунок 1. Общая схема разработки и производства детали на станке с ЧПУ

Во-первых, в CAD (САПР) программе разрабатывается модель будущего изделия, она может быть представлена как в полигональном формате, так и в формате конструктивной геометрии. Во-вторых, в CAM-программе, которая принимает на вход модуль станка и модели заготовки и детали, создается управляющая траектория для инструмента. Управляющая траектория представлена во внутреннем формате CAM программы и не универсальна для множества всех станков, поэтому появляется необходимость ее дополнительно обработать. Постпроцессор преобразует управляющую траекторию в

управляющую программу на языке G-кода, которая действительна только для определенного множества станков, которые способны принимать всё множество команд данной программы.

На данном этапе в работу включается программа визуализации техпроцесса. Обычно это еще происходит еще до постпроцессора, сразу после генерации управляющей траектории, но у меня нет доступа к закрытому формату управляющей траектории. Поэтому, мою программу можно воспринимать как станок, который принимает на вход управляющую программу, и модель заготовки вместо реального материала. Для моей программы (назовем ее CamVis) нужен и постпроцессор – его я возьму из списка стандартной библиотеки программы go2cam [4].

G-код

G-код – это примитивный язык программирования, который скорее можно назвать языком команд, использующийся в подавляющем большинстве современных станков с ЧПУ [10]. Каждая строка начинается с новой команды и буквы G или M, выполняющая перемещение инструмента и системные вызовы соответственно (вроде подпрограмм, конца/начала программы и т.п.).

К примеру, команда G01 X20 Y10 Z10 F200, переместит инструмент в точку (20,10,10) со скоростью подачи 200 мм/мин.

G02 X35 Y25 I20 J-5 – команда круговой интерполяции.

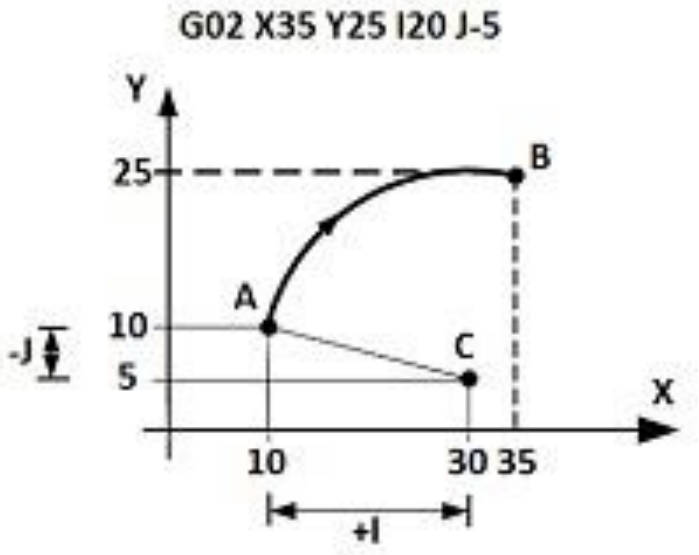

Рисунок 2. Команда круговой интерполяции G02 X35 Y25 I20 J-5

Выбор формата представления 3-х мерных объектов

Для загрузки, обработки и выведения на экран необходимо выбрать формат представления всех 3-ех мерных объектов в сцене.

Вариант 1. Воксельное представление.

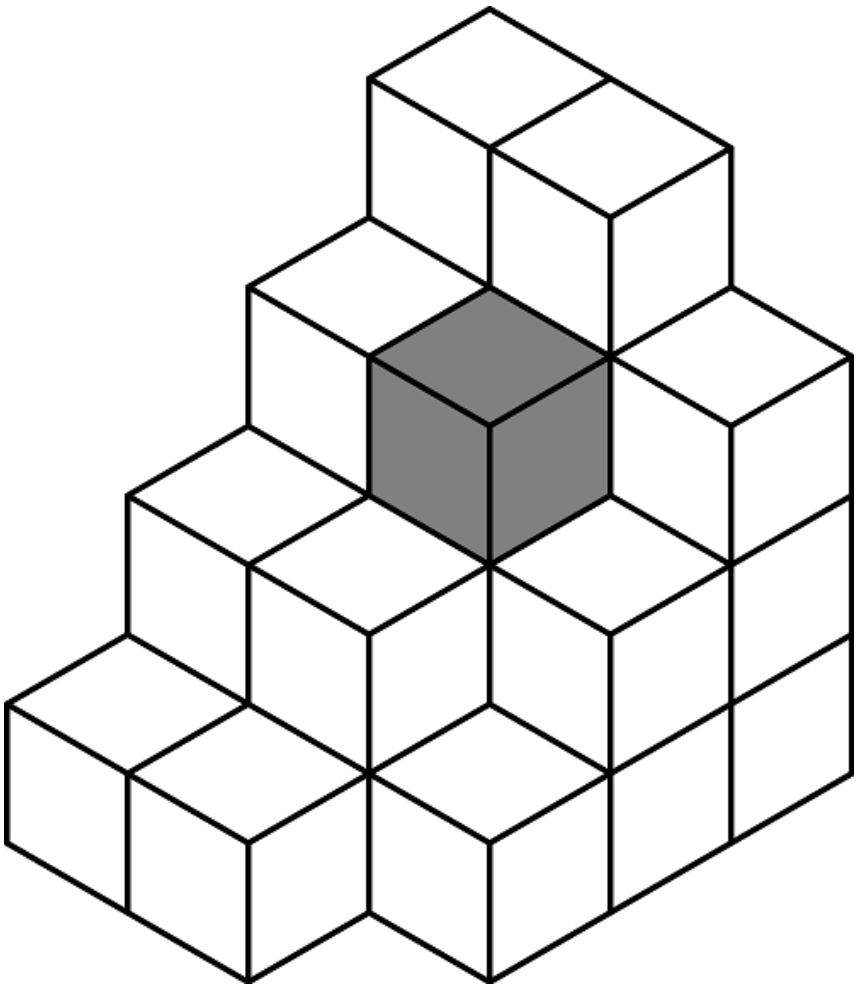

Рисунок 3. Воксельная модель

+ Простота алгоритмов отсечения

+ Высокая точность

- Сложные алгоритмы перевода с полигональной сетки в воксельный вид

- Занимает большие объемы памяти, даже в виде окто-деревьев

- Низкая производительность

Исходя из приведенных выше преимуществ и недостатков, было решено не использовать этот формат.

Вариант 2. Конструктивная геометрия

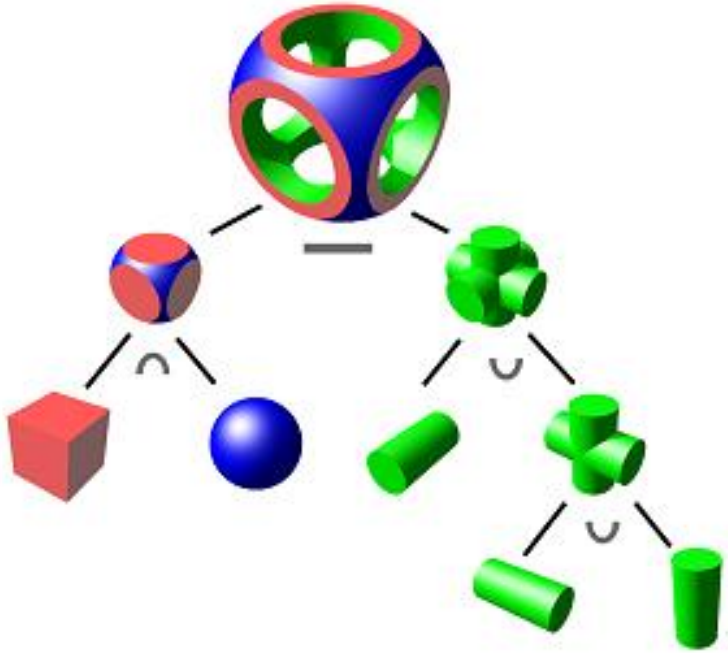

Рисунок 4. Модель в формате конструктивной геометрии

Данный формат отлично приспособлен для задач 3-ех мерного отсечения и чаще всего используется в CAD/CAM системах. Он обладает высокой производительностью, занимает минимум памяти и алгоритмы отсечения тривиальны. Однако сам формат крайне сложный в интерпретации и необходимо писать сложные алгоритмы для чтения файлов этого формата и отображения на экране. Ввиду высокой сложности реализации [1], было решено отказаться от этого формата.

Вариант 3. Полигональная сетка

+ Простой формат

+ Высокая производительность

- Сложность алгоритмов отсечения

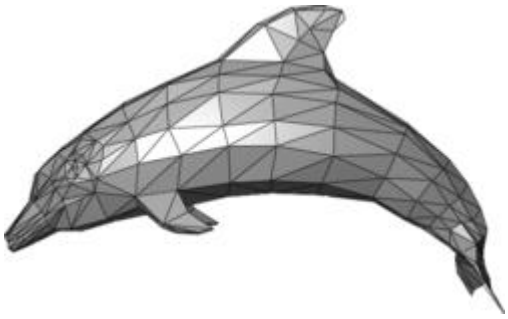

Рисунок 5. Полигональная сетка

Данный формат отличается удобством интерпретации, в частности расширение .obj, которое я и буду использовать в своей программе. Анализируя совокупность этих факторов, я решил использовать именно этот формат, как основной. Сетка, подающаяся на вход программы должна быть замкнутой, как для базы (заготовки), так и для отсекателя (фрезы).

Алгоритмы отсечения

1. Хранение данных

Данные об 3-ех мерных объектах в программе хранятся в виде 3 основных классов:

- AMesh – Класс, хранящих полигональную сетку в виде массива точек (AVertex) и массива граней (AFace). Помимо этих основных, структура так же содержит временные массивы для временных точек, граней и списков вершин обрабатываемых граней.
- AFace – Состоит из списка вершин (в виде кольцевого буфера), нормали к поверхности, коэффициента d (для составление уравнения плоскости) и специальной марки (метки), которая меняется в процессе обработки, в зависимости от статуса этого полигона.
- AVertex – Состоит из 3 координат, метки (аналогично, как для полигона) и коэффициента d, равняющегося расстоянию от точки до текущей обрабатываемой плоскости.

В качестве вспомогательных структур я использую:

- SparseTable – разреженная таблица, при обращении по (x,y) и (y, x) выдает один и тот же элемент.
- MapOfList – Ассоциативный массив списков (при нормальной работе программы все списки должны быть длинны 2)

Общая схема алгоритма 3-ех мерного отсечения

Весь алгоритм отсечения, придуманный для этой работы, базируется на алгоритме Сазерленда-Ходжмана [9], адаптированного для 3-ех мерного случая. Весь алгоритм можно расписать примерно так:

1) Применяем к заготовке алгоритм Сазерленда-Ходжмана на каждую грань отсекателя, используя ее как отсекающую плоскость.
2) После каждого отсечения замываем полученную полигональную сетку (используется функция closeMesh) и триангулируем полученные полигоны.
3) После получения модели пересечения заготовки и отсекателя, мы:
   a) Инвертируем все оригинальные грани, попавшие под отсечение (метка CLIPPED). Триангулируем их.
   b) Удаляем все грани, оказавшиеся внутри отсекателя (метка INSIDE)
4) Переносим все временные грани, отмеченные как CLIPPED, CLOSED и ONTO_C (частный случай) в основной массив. Временные массивы обнуляются.

Алгоритм Сазерленда-Ходжмана

Сначала речь пойдет о стандартном алгоритме для 2-ух мерного случая. На вход алгоритму подается список вершин полигона, выходной список (пустой) и линия отсечения. Начинается последовательная обработка каждой точки: если она лежит с внутренней стороны отсекателя, то добавляется в выходной список. При этом проверяется еще и положение следующей точки – если она имеет другое положение (допустим, первая точка была внутри, а следующая снаружи), то считается, что это ребро пересекает линию отсечения. Вычисляется точка пересечения и вносится в выходной список. В случае, если отсекатель – не просто прямая, а полигон (обязательно выпуклый), тогда алгоритм применяется поочередно для каждого ребра отсекателя.

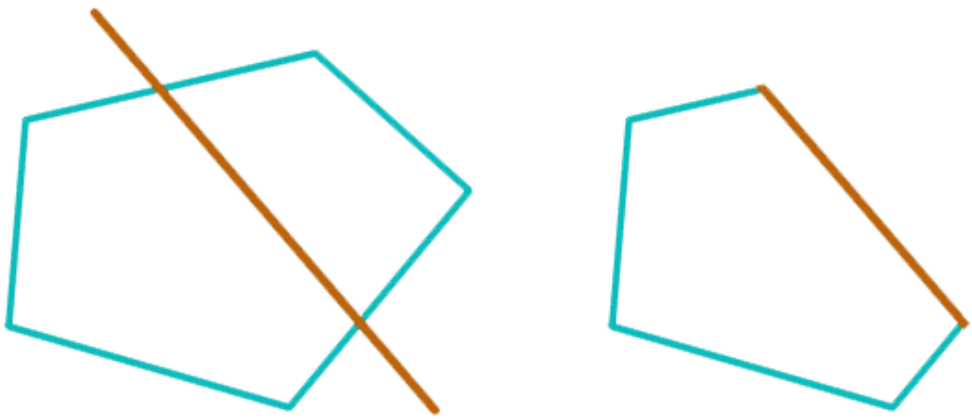

Рисунок 6. Отсечение полигона прямой по алгоритму Сазерленда-Ходжмана.[9]

Вычисление положения точки вычисляется по формуле:

$$A = A - AB \times \frac{\langle N, A \rangle - d}{\langle N, AB \rangle}$$

Формула 1.[2] Вычисление положения точки относительно плоскости, где A – обрабатываемая точка, B – следующая на ней, N – нормаль к отсекающей плоскости, d – свободный коэффициент в общем уравнении плоскости вида Ax + By + Cz + d = 0.

Адаптация этого алгоритма к 3-ех мерному случаю весьма проста [2][3]: вместо отсекающих прямых будут отсекающие плоскости. Накладывается ограничение на отсекатель – он должен быть выпуклым многогранником. Но при этом возникает проблема не замкнутости полигональной сетки. Для этого будет использоваться функция closeMesh.

Функция cliByPlane

В данном разделе будет более подробно рассмотрена функция clipByPlane, как из класса AMesh, так и из класса AFace.

Первая, AMesh::clipByPlane, принимая на вход только данные о плоскости отсечения, сначала вычисляет расстояния до точек и маркирует их: INSIDE (только в случае, если точка обрабатывается первый раз), OUTSIDE (не имеет ограничений) и ONTO (только если точка не OUTSIDE). Потом она создает структуры данных ComputedEdges – разреженная таблица, созданная для хранения уже вычисленных точек пересечений ребер, т.к. каждое ребро будет обрабатываться дважды и OpenEdges – хранит контур отсечения. Далее, для каждой грани, не помеченной OUTSIDE, вызывается функция AFace::clipByPlane.

AFace::clipByPlane принимает на вход данные о плоскости отсечения и обрабатываемой грани. Мы идем по списку точек в виде пары: f (текущая точка) и s (следующая за ней). Если текущая точка имеет метку INSIDE или ONTO, то она заносится в новый список. Далее, если точка была INSIDE или OUTSIDE идет проверка: если следующая точка имеет отличную от текущей метку, то вычисляется точка пересечения. При вычислении проверяется наличие этой в разреженной таблице, куда заносятся все вычисленные пересечения для данной отсекающей плоскости. После прохода по списку точек, можно определить новую метку для полигона: он может быть помечен OUTSIDE, если все точки были OUTSIDE или все кроме одной – ее метка

должна быть ONTO (т.е. полигон касается отсекающей плоскости одной точкой). Может быть помечен INSIDE, при тех же аналогичных условиях. Если была хоть одна пара точек INSIDE-OUTSIDE, то полигон будет помечен CLIPPED. Если грань была помечена как CLOSED, т.е. грань полученная в результате работы функции closeMesh, то она может перейти только в состояние OUTSIDE. Т.к. все обрабатываемые полигоны являются выпуклыми, можно точно сказать, что линия пересечения будет состоять из ровно 2 точек. Заносим все точки отмеченные CLIPPED и ONTO в контур отсечения (с ссылками друг на друга), для последующего использования в функции closeMesh. Если полигональная сетка замкнута – контур тоже получится замкнутый.

Заполнение «дырок» в полигональной сетке

При таком алгоритме, необходимо заполнять пробел в полигональной сетке, образованный отсчением.

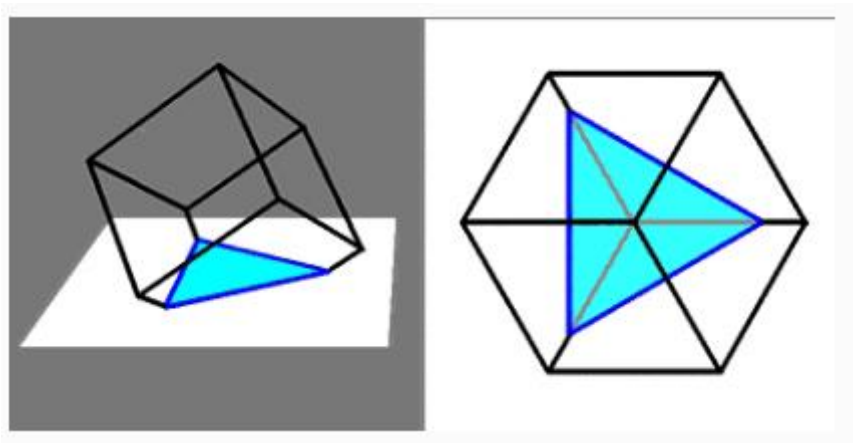

Рисунок 7. Сечение куба. На месте этого сечения и будет образовываться «дырка» в сетке.

Для этого применяется функция closeMesh, делающая это за линейное время. Еще на этапе отсечения (функция AFace::clipByPlane) каждая точка пересечения и другие точки, находящиеся в плоскости отсекателя (метка ONTO), заносятся в специальную структуру OpenEdges, представляющую из себя ассоциативный массив списков. Ключом каждого элемента является номер точки пересечения, а значение – список, состоящий из 2 элементов (при нормальной работе программы): предыдущей и последующей точки пересечения в контуре, опивающем «дырку» в полигональной сетке. Функция closeMesh, вытаскивает из этого массива точки, пока не кончатся, и переходит по любой из списка, если та не указывает назад, занося ее в список вершин нового полигона.

Инверсия отсеченных граней

После выполнения алгоритма Сазерленда-Ходжмана мы получаем пересечение отсекателя и заготовки, а нужна разница. Для этого применяется функция инверсии отсечения.

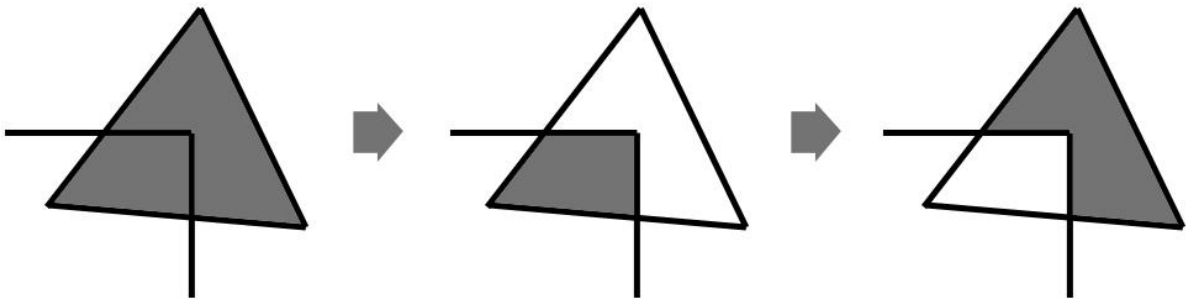

Рисунок 8. Работа алгоритма инверсии, темным показан получаемый полигон.

На этапе отсечения мы помечали все точки пересечения CLIPPED или ONTO, так же, для условия работы алгоритма отсечения необходимо, чтобы хоть одна точка было помечена OUTSIDE и хоть одна INSIDE. В том время как из состояния INSIDE точка перейти в другое при следующих отсечениях другими плоскостями, то состояние OUTSIDE с ней останется до конца работы алгоритма. Таким образом, можно гарантировать, что в изначальном списке полигона будет хоть одна точка OUTSIDE и хоть одна помеченная CLIPPED или ONTO. Работа алгоритма инверсии заключается в следующем: Находим первую попавшуюся точку с меткой OUTSIDE (ищем в изначальном списке!), и идем от нее в обе стороны, добавляя все точки в новый список, пока не встретим точку с меткой CLIPPED или ONTO. Как только мы их встретили, это значит, что мы нашли последовательность ребер пересечения, а т.к. полигон

выпуклый, эта последовательность будет единственной. Добавляем эту последовательность (из списка отсеченного полигона!) в список и полигон готов. Таким образом, алгоритм работает за линейное время, но требует для своей работы сохранения списка и изначального полигона, и отсеченного.

Выпуклость полученных полигонов

Так как применение алгоритма Сазерленда-Ходжмана требует от нас выпуклости полигонов в модели заготовки (но не требует выпуклости самой модели), необходимо было реализовать алгоритм, который разбивал бы невыпуклые многоугольники на выпуклые. Был реализован рекурсивный алгоритм: Находя в полигоне угол больше 180° (определяется с помощью векторного произведения, т.к. проверки на правую тройку), мы соединяем его в любой подходящей точкой, делим полигон по этой линии на 2 и вызываем эту же функцию для полученных новых полигонов.

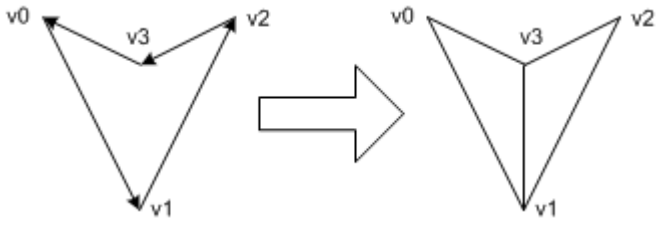

Рисунок 9. Разбиение невыпуклого полигона на выпуклые.

Заключение

Полученная программа имеет ценность не только как демонстрация курсовой работы, но и практическое применение, пусть и ограниченное. При расширении принимаемых команд и оптимизации под 5-ти осевые станки, появится возможность более широкого применения этой программы.

Алгоритмы, представленные в данной программе, могут быть хорошо распараллелены, особенно на ЭВМ с большим числом маломощных вычислительных узлов (к примеру, видеокарте). Однако в коде имеются места, где нельзя допускать одновременный доступ нескольких потоков.

В будущем, для улучшения качества работы программы, можно переписать функцию makeConvex, для получения более аккуратной полигональной сетки при разбиении невыпуклых полигонов. Текущая версия использует более быстрый вариант, но при триангуляции отдает предпочтение первой подходящей точке, что проводит к постепенной деградации полигональной сетки при множественных отсечениях сложными фигурами.

Список литературы

1. Parasolid XT Format Reference, April 2008
2. David Eberly, Clipping Mesh Against A Plane, Geometric Tools, LLC, 2008
3. Line and Polygon Clipping, Foley & Van Dam, Chapter 3
4. Go2cam User Manual, 2016, V6.03
5. www.ecse.rpi.edu/Homepages/wrf/Research/Short_Notes/pnpoly.html 25.01.1017
6. algolist.manual.ru/maths/geom/intersect/linefacet3d.php 25.01.2017
7. compgraphics.info/OpenGL/lighting/light_sources.php 25.01.2017
8. www.songho.ca/opengl/gl_transform.html 25.01.2017
9. gamedevelopment.tutsplus.com/tutorials/understanding-sutherland-hodgman-clipping-for-physics-engines--gamedev-11917 25.01.2017
10. http://www.intuwiz.ru/articles/g-code.html 25.01.2017