

ОГЛАВЛЕНИЕ

ПОСТАНОВКА ЗАДАЧИ	3
ВВЕДЕНИЕ	4
ПРОЕКТИРОВАНИЕ	5
КОНСТРУИРОВАНИЕ АРХИТЕКТУРЫ	7
КОНСТРУИРОВАНИЕ КЛАССОВ	9
КОНСТРУИРОВАНИЕ МЕТОДОВ ЯДРА	13
ТЕСТИРОВАНИЕ	17
РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ	18
ИСТОЧНИКИ	20

ПОСТАНОВКА ЗАДАЧИ

Задача состоит в создании графического редактора со следующими функциями:

1. Построение векторных объектов: отрезок, полилиния (состоящая из чередующихся отрезков и дуг, построенных по трем точкам), окружность, дуга.
2. Удаление выбранных объектов. Выбор и изменение в диалоговом режиме типов и цветов заливки выбранных замкнутых контуров. Выбор объектов производить тремя способами: 1) выбор объекта (группы объектов) мышью; 2) выбор с использованием прямоугольной рамки: 2.1) выбранными считаются те, которые полностью попали внутрь прямоугольной области; 2.2) выбранными считаются и те, части которых попали внутрь прямоугольной области.
3. Организация диалога с помощью меню, причем построение и удаление объектов должно быть представлено в виде режимов работы, а не в виде отдельных операций. Ввод объектов производить в режиме “тянущаяся линия”.
4. Сохранение и считывание результатов работы в файлах.

ВВЕДЕНИЕ

Поскольку данная работа пишется уже по факту завершения разработки, рассмотрение программы будет осуществляться в обратном порядке. Сначала будет рассмотрено проектирование, т.е. рассмотрение задачи, её разложение на подзадачи, после чего будет произведено рассмотрение каждого из программных компонентов – начиная от классов, ответственных за графический интерфейс, продолжая классами, реализующими высокоуровневые функции, и далее спускаясь до базовых математических функций (т.н. функций ядра).

ПРОЕКТИРОВАНИЕ

Проектирование - сложный творческий процесс, являющийся неотъемлемой составной частью инженерной деятельности, он не сводится к разработке чертежей, а рассматривается как начальный этап создания нового изделия.

- В.В.Курицына «САПР. Конспект лекций»^[1], МАТИ, 2011

Первая подзадача – построение векторных объектов. Фактически, векторный объект в нашем случае представляет собой набор точек, который соотносятся между собой определенным образом. При помещении на растр эти точки приобретают конкретные абсолютные координаты. В лабораторных работах данная подзадача уже решена. Единственное, для чего придется дописать алгоритм – полилиния, так как в ней нужно строить окружность (дугу) по 3-м точкам.

Вторая подзадача – удаление объектов, также не представляет сложности на конечном этапе. Но перед ним объекты нужно выделить. Выделение точки при попадании нажатия мыши в её окрестности также уже было реализовано. Выделение объектов с помощью прямоугольной рамки является не столь тривиальным. Представим, что у нас есть линия и пользователь выбрал режим захвата при частичном попадании – вполне возможно построить прямоугольную рамку, которая, де-факто, захватывает часть линии, некоторый её отрезок, но не затрагивает конечные точки, который представляют линию в памяти. В этом случае необходимо построить уравнение прямой, функция чего уже была реализована в лабораторных работах, и проверить его на пересечение с прямоугольной рамкой дважды, ибо проверка единственного пересечения будет перехватывать и случаи касания, а не только включения, что, впрочем, едва-ли может омрачить пользовательский опыт. Существует и обратная задача при выборе режима захвата при полном попадании – окружность представлена в памяти координатами центра и радиусом – как фиксировать её попадание? Необходимо проверять попадание центра, после чего проверять, можно ли, добавив или вычтя из центра окружности радиус попасть за пределы рамки. Но это простой случай. Что насчет частичной окружности полилинии? Если дуга окружности представлена таким образом, что выходит за границы рамки отсутствующая её часть. В этом случае нужно, применяя уравнение окружности, вычислять максимальный угол отклонения по окружности, после чего осуществлять проверку для него отдельно, помимо проверки по доступным направлениям на осях, количество коих может и вовсе быть равно нулю, если дуга, например, существует как часть окружности в пределах от $\pi/6$ до $\pi/3$ и т.п. Впрочем, я сомневаюсь, что такие «точные» алгоритмы действительно востребованы и, скорее всего, никто из прочих решающих аналогичную задачу не будет заниматься их разработкой.

Третья подзадача – реализация режимов работы и ввод в режиме тянущейся линии. Данная задача также уже была полностью реализована в

лабораторных работах. Режимы работы представляют собой не более чем хранимые в памяти состояния программы, которые могут быть реализованы в виде стека, что обеспечивало-бы их наложение.

Четвертая подзадача – сохранение в файл. Как для классического веб-разработчика, вопрос выбора формата для меня не стоит – им будет JSON. Сама запись данных в файл не представляет сложности, ибо является простой записью коллекции объектов. Архитектурно код, написанный еще для лабораторных работ, хранит каждую операцию над холстом как объект, будь то даже заливка – представляет собой точку начала, тип перехода и цвет.

Программа разрабатывается на современной знакомой автору технологии WPF – дальнейшем развитии WindowsForms, поддержка которой прекращена. Панель управления реализуем вверху и слева окна, редактируемую область – в нижнем правом углу.

КОНСТРУИРОВАНИЕ АРХИТЕКТУРЫ

Конструирование опирается на результаты проектирования и уточняет все инженерные решения, принятые при проектировании.

- В.В.Курицына «САПР. Конспект лекций»^[1], МАТИ, 2011

Выводимое изображение представляет текущее внутреннее состояние системы. Написанный ранее класс `BitmapDrawer` обеспечивает хранение фигур разного рода и их отображение реализует паттерн «Фасад» по отношению к классу `DirectBitmap`^[2], который, в свою очередь, обеспечивает исключение сборщика мусора (метод `LockBits`) из цепочки доступа к массиву пикселей класса `Bitmap`, тем самым реализует паттерн «Прокси». Поскольку класс `BitmapDrawer` несет уже достаточно большой, но всё ещё базовый функционал, реализуем поверх него еще один прокси, назовем его `BitmapEditor`. Поверх него реализуем «Фасад» – `StatefulEditor`, реализующий хранение состояния взаимодействия с пользователем в стеке состояния, что, как было указано ранее, обеспечивает их наложение. С целью сохранения работы в файл определим модель данных `Saved`, она содержит сведения о текущих фигурах на поле, методы `ToSaved` в классе `BitmapEditor` и конструктор, принимающий данный тип в качестве аргумента, восстанавливающий из него состояние класса. Как ни странно, данные методы размещаются именно в данном классе, а не `StatefulEditor`, поскольку последний – отвечает за состояние взаимодействия с пользователем, которое не нужно сохранять. Получим следующую структуру: таблица 1.

<p>(1) public class DirectBitmap : IDisposable { public Bitmap Bitmap { get; private set; } public int[] Bits { get; private set; } public bool Disposed { get; private set; } public int Height { get; private set; } public int Width { get; private set; } private GCHandle BitsHandle { get; private set; } public DirectBitmap(int width, int height) { BitsHandle = GCHandle.Alloc(...);... } }</p>	<p>(2) public class BitmapDrawer { public DirectBitmap CurrentFrame { get; init; } public List<Dot> Dots { get; private set; } public List<Line> Lines { get; private set; } public List<Circle> Circles { get; private set; } public List<Arc> Arcs { get; private set; } public List<Rectangle> Rectangles { get; private set; } public List<AreaFiller> AreaFillers { get; private set; } public List<InterpolatedPoints> InterpolatedPoints { get; private set; } public List<InterpolatedPoints> LagrangePolys { get; private set; } public List<InterpolatedPoints> Besie2Polys { get; private set; } public List<IGraphicalElement> ConstantObjects { get; private set; } }</p>
<p>(3) class BitmapEditor { protected readonly BitmapDrawer drawer; public int BackgroundColorArgb { get; set; } public List<PolyLineF> PolyLines { get; init; } public List<CircleF> Circles { get; init; } public List<ArcF> Arcs { get; init; } public List<FillerF> Fillers { get; init; } public BitmapEditor(int width, int height) { ... drawer = new BitmapDrawer(width, height); } public BitmapEditor(Saved saved) { ... drawer = new(saved.Width, saved.Height); } }</p>	<p>(4) class StatefulEditor : BitmapEditor { public enum States { ... } public Stack<States> State { get; init; } public Stack<PointF> CachedPoints { get; init; } public List<AEditorElement> CurrentlySelectedObjects { get; init; } public PointF RelativenessPoint { get; set; } public int FillingColorArgb { get; set; } public int DrawingColorArgb { get; set; } public string DrawingPattern { get; set; } public StatefulEditor(int width, int height) : base(width, height) { ... } public StatefulEditor(Saved saved) : base(saved) { ... } }</p>

Таблица 1 – принципиальная схема классов проекта.

Как можно заметить, в классе `StatefulEditor` определено перечисление состояний `States`, члены которого записываются в стек. Данный стек определяет текущее и последующие (включенные ранее) состояния редактора. Коллекция

CachedPoints описывает точки, кешированные в текущий момент, что обеспечивает динамическое отображение фигуры, рисуемой пользователем, которая, по завершению рисования, будет перенесена в коллекцию постоянного хранения. Коллекция CurrentlySelectedObjects обеспечивает хранение выделенных объектов.

Теперь опишем интерфейсы и абстрактные классы для графических объектов, существующих в нашем редакторе. Названия сами говорят за функциональность. Имеем следующую структуру: таблица 2. Коллекция

Имя интерфейса	Поля, свойства, методы
IMoveable	public void Move(PointF diff);
IScaleable	public void Scale(float scale, PointF relativeTo);
IRotateable	public void Rotate(float angleR, PointF relativeTo);
IColored	public int ColorArgb { get; set; }
IPatterned	public string Pattern { get; set; } public IEnumerator<bool> PatternResolver { get; }
IToRussianString	public string ToRussianString { get; }
AEditorElement <i>наследуется от всех вышеназванных интерфейсов, кроме IPatterned</i>	protected const int LightGreenArgb = -7278960; protected const int LightCoralArgb = -1015680; protected const string DefaultPattern = "+";

Таблица 2 – принципиальная схема интерфейсов проекта.

КОНСТРУИРОВАНИЕ КЛАССОВ

Начнем с основы – пользовательского интерфейса. Он представляет собой набор кнопок и изображение, предусматривающее обратку нажатий и перемещений мыши. В целом, на этом его функция заканчивается – ему просто нужно передать событие далее по коду. Интерфейс разработан на языке XAML с широким применением компонентов StackPanel, Button, TextBox и т.д. Каждая кнопка переводит StatefulEditor в определенное состояние. После каждого состояния, кроме нулевого, может следовать только определенный набор других состояний, что задает сценарии использования программы – таблица 3.

Название состояния	Описание
None, ClickCapture = None	Выделение объектов кликом мышки.
PartialSelection_NotStarted	Выделение частичным включением, первая точка рамки не поставлена.
PartialSelection_FirstPointAdded	Выделение частичным включением, первая точка рамки поставлена.
FullSelection_NotStarted	Выделение полным включением, первая точка рамки не поставлена.
FullSelection_FirstPointAdded	Выделение полным включением, первая точка рамки поставлена.
PolylineDrawing_NotStarted	Рисование полилинии, первая точка не поставлена.
PolylineDrawing_LastIsLine	Рисование полилинии, последняя из точек является точкой отрезка.
PolylineDrawing_LastIsCircle	Рисование полилинии, последняя из точек является средней точкой дуги.
CircleDrawing_NotStarted	Рисование окружности, первая точка не поставлена.
CircleDrawing_CenterSelected	Рисование окружности, точка центра поставлена.
ArcDrawing_NotStarted	Рисование дуги, первая точка не поставлена.
ArcDrawing_FirstPointAdded	Рисование дуги, поставлена первая точка.
ArcDrawing_SecondPointAdded	Рисование дуги, поставлена вторая точка, задающая направление дуги относительно первой точки.
Filling	Режим заливки.
RelativePointSelection	Режим выбора точки относительности.
CapturedObjectsEdition	Режим редактирования выделенных сущностей.

Таблица 3 – описание состояний класса StatefulEditor.

Из конечных состояний можно вернуться в либо в состояние None, либо в первичное состояние данного режима. Так, например, после состояния, CircleDrawing_CenterSelected, если пользователь поставил еще одну точку, а равно указал точку на окружности однозначно задав геометрический объект – мы добавляем полученную фигуру в список постоянных объектов в базовом классе и возвращаемся в состояние CircleDrawing_NotStarted, что позволяет рисовать следующую окружность.

Поскольку пользователю может требоваться прервать текущее состояние (например – он передумал рисовать окружность!), мы добавляем в класс метод TryExit, который осуществляет переход в состояние None с очисткой кешированных точек.

Рассмотрим следующий метод – RenderCurrentState. Данный метод является перезаписанным (override). Базовый класс ничего не знает о неких состояниях. Поэтому в первую очередь наш метод подделывает текущий объект, который рисуется динамически, следуя движениям мыши, под постоянный, который уже нарисован. После чего следует вызов метода базового класса RenderCurrentState. Это обеспечивает динамическое рисование – рисунок 1.

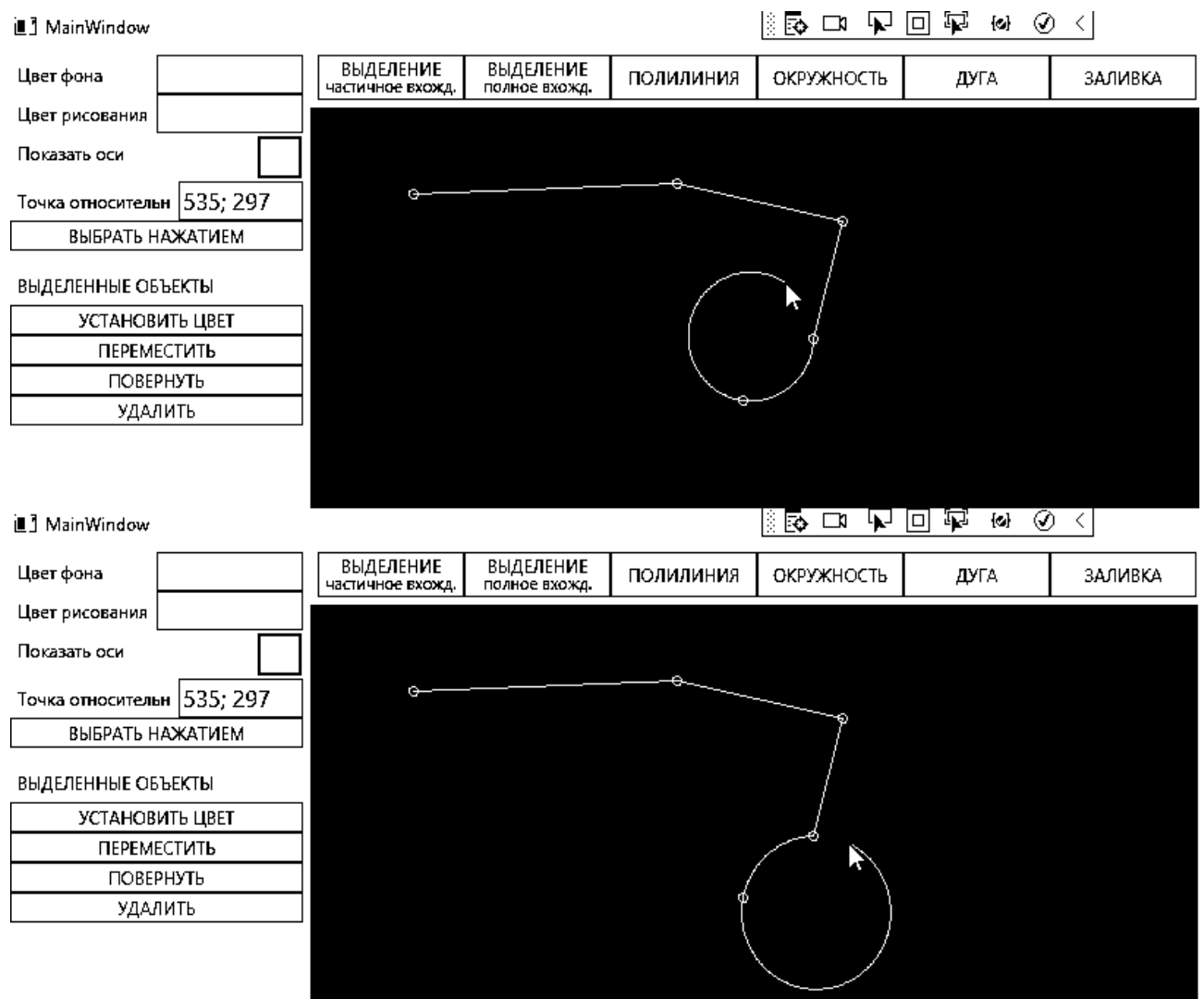


Рисунок 1 – возможности динамического отображения рисуемого объекта.

Спускаемся дальше по иерархии классов. Класс `BitmapEditor` представляет собой хранилище объектов на текущем холсте. Но в нем реализованы и некоторые функции, в частности – функция выделения объектов. Рассмотрим некоторую алгоритмизацию выделения при частичном включении объекта в рамку, ибо обнаружение полного включения, как правильно, является более тривиальным. В случае с выделением отрезка все просто – находим уравнение прямой и проверяем пересечение с рамкой выделения в рамках координат двух точек отрезка. Выделение заливки осуществляется либо по базовой точке (точке начала), либо по всем точкам заливки, алгоритм чего почти не отличается от самой заливки. Интереснее обстоят дела с выделением окружности. Глобальность есть два варианта: первый – построить уравнение перпендикуляра от центра окружности до сторон рамки выделения и проверить, меньше ли расстояние между точками на данной прямой с координатой X центра окружности о координатой X точки пересечения данной прямой со сторонами рамки, при условии, что это расстояние меньше, чем до ближайшего к центру угла рамки, лежащего на данной стороне. Но это порождает серьезное усложнение с обнаружением включения дуг. Поэтому было принято решение написать хоть и менее изящный, но зато значительно более простой и понятный алгоритм – цикл проходит по окружности, размещая на ней точки попадания с шагом, например $\pi/36$ – после чего проверяется попадание каждой из точек в рамку. В этом случае всё существенно упрощается с другой, ведь она представляет собой, как структура данных, - центр окружности, угол начала дуги, угол конца дуги и направление движение (увеличение или уменьшение угла). Следующим методом является перевод полилинии в примитивы. Поскольку в данном классе мы храним полилинию целиком, что обеспечивает её выделение при захвате лишь одного из её примитивов, то вот класс `BitmapDrawer`, для которого настоящий класс реализует паттерн Прокси, ничего не знает о составных объектах. В целом и окружность, и дуга, и заливка являются примитивами и их передача ниже по иерархии требует лишь трансляции из одной модели данных в другую без реальных преобразований значений. Полилинию же необходимо разбить на отрезки и дуги. Это не очень сложно, по двум точкам строим отрезок, если встречаем точку дуги – то берем еще одну (следующую) точку и строим дугу – программный код 1.

```
private void AddPolylineObjectToDrawer(PolyLineF polyLine) {
    var pts = polyLine.Points; var clr = polyLine.ColorArgb; var ptrn = polyLine.PatternResolver;

    for(var i = 1; i < pts.Count; i++) {
        var pp = pts.At(i - 1); var pc = pts.At(i); var pn = pts.At(i + 1);
        if(pc.IsCirclePoint) {
            if(i++ == PolyLines.Count - 1) break;
            drawer.Arcs.Add((Arc)
                (new ArcF(pp, pc, pn, polyLine.ColorArgb, polyLine.Pattern)));
        } else {
            drawer.Lines.Add(new(pp, pc, clr, ptrn));
        }
    }
}
```

Программный код 1 – приведение полилинии к 2D примитивам.

Спускаемся дальше по иерархии классов. Класс `BitmapDrawer` представляет собой класс из библиотеки, которая используется во всех лабораторных и курсовой работе, поэтому многие его классы были продублированы в настраиваемых проектах, после чего было реализовано `explicit`-преобразование, так, например, в программном коде 1, представленном выше - из `coursework.Models.ArcF` в `GraphicLibrary.Models.Arc`, что позволяет не менять код данной библиотеки, дабы не ставить под угрозу работоспособность зависимых проектов. Сам же класс `BitmapDrawer` реализует две основные функции – рендер текущих объектов, добавленных в его коллекции, и их преобразование. Рассмотрим некоторую алгоритмизацию. Рисование линии осуществляется классическим алгоритмом – итерацией по переменной, обеспечивающей неперепуск пикселей, т.е. по той, разность по которой между первой и второй точками больше, чем по другой. Рисование окружности осуществляется несколько более изящно. Сначала вычисляется длина окружности по известной формуле, используя радиус, что однозначно задаёт её длину в пикселях, после чего вычисляется угловой шаг путем деления 2π на полученную длину, далее начинается цикл от 0 до 2π с полученным шагом, где для каждого угла вычисляется точка на окружности, куда ставится пиксель. Преимуществом данного подхода является полная готовность алгоритма к рисованию дуги и возможности правильного отображения линии любого паттерна. Заливка осуществляется методом 4-х или 8-ми пикселей, путем использования стека, а не рекурсии, ибо при заливке больших площадей легко превысить размер стека в C#, размер которого зафиксирован на 1 MiB. Да, некоторым способом увеличение данного стека возможно, однако, при всём желании, это не является легитимным решением проблемы, а, по своей сути, является попыткой потратить на исполнение больше ресурсов вместо оптимизации. Рисование прямоугольной рамки (для выделения объектов, в том числе) состоит не более, чем в рисовании 4-х линий после вычисления левой верхней и нижней правой точек. Дополнительными методами данного класса являются интерполяция ломаной, полином Лагранжа, кривые Безье 2-й степени, однако они не подлежат рассмотрению в данной работе и, в целом, используют класс, который будет рассмотрен далее.

КОНСТРУИРОВАНИЕ МЕТОДОВ ЯДРА

Если вы хотите, чтобы отдельные части вашего алгоритма можно было применять в дальнейшем при построении новых программ, то единственный реальный путь к этому – вычленив претендующие на многократное использование функциональные компоненты вашей программы и оформить их в виде модулей. В результате такой деятельности формируется библиотека функций.

- Викиверситет, «Программное ядро»^[4].

Рассмотрим некоторые базовые математические функции, которые применяются перекрестно по всем проектам решения, включающего данную работу.

Функция `GetCirleRadius(PointF center, PointF onCirle)` позволяет получить радиус окружности имея координаты её центра и точку на окружности. Получая разницу по двум координатам, вычисляется корень из суммы их квадратов, используя обычную теорему Пифагора. Возвращаемым значением является число с плавающей точкой.

Функция `FindPointOnCircle(PointF center, float radius, float angleR)` позволяет получить координаты точки на окружности, имея координаты её центра, радиус и угол отклонения, на котором располагается искомая точка. Возвращаемым значением являются две координаты – числа с плавающей точкой.

Функция `FindAngleOfPointOnCircle(PointF target, PointF center)` решает обратную задачу к названной выше – она получает угол отклонения, на котором находится точка на окружности с заданным центром. Данная функция находит четверть окружности, в которой находится искомая точка, после чего применяет следующую формулу – программный код 2. Возвращаемое значение – число.

```
// К углу относительно ближайшей оси добавить остальную часть,  
// где (PI/2) – число радиан в четверти  
return (PI / 2 * (part - 1)) + (part == 1 || part == 3 ?  
    Atan(Abs(ry) / Abs(rx)) : Atan(Abs(rx) / Abs(ry)));
```

Программный код 2 – вычисления угла точки на окружности.

Функция `RotatePoint(PointF target, PointF relativeTo, float angleR)` решает задачу поворота точки. Очевидно, что поворот может осуществляться только относительно чего-либо. Точка относительно является, де-факто, центром окружности, на которой мы вращаем точку. С учетом вышеописанных функций мы просто получаем изначальный угол, добавляем к нему требуемый, после чего находим точку с полученным углом. Данный метод предусматривает вырожденные случаи, вроде поворота на 0 или же поворота точки относительно самой себя, в результате чего возвращается исходная точка. Возвращаемым значением являются две координаты.

Функция `MirrorPoint(PointF target, PointF relativeTo)` решает задачу отражения точки относительно другой точки. Несмотря на то, что данную функцию можно было бы реализовать через нахождение разницы в координатах, после чего смещения дважды на эту разницу в сторону точки относительности, она реализована как обычный поворот точки на 180 градусов относительно точки относительности, поскольку вышеописанные функции поворота уже протестированы и отлажены. Возвращаемым значением являются две координаты.

Функция `FindLinearEquation(PointF p1, PointF p2)` позволяет найти уравнение прямой по заданным точкам, де-факто, находя коэффициенты k и b в уравнении прямой $y=kx+b$ – программный код 3. Возвращаемым значением являются два числа.

```
/* Построим уравнение прямой, задаваемой отрезком из двух точек, формула для
 * которого известна. Имеем каноническое уравнение (x-x1)/(x2-x1) == (y-y1)/(y2-y1).
 * Пусть mx=x2-x1, my=y2-y1. Имеем (x-x1)/mx = (y-y1)/my. Домножим на my.
 * Имеем (x-x1)my/mx=y-y1. Перенесем. Имеем (x-x1)my/mx - y1 = y. Раскроем.
 * Пусть k=my/mx. Имеем kx - kx1 - y1. Получим уравнение вида kx+b=y.
 * Упростим вышесказанное до k=(y2-y1)/(x2-x1), b=(y-kx);
 */
public static (float k, float b) FindLinearEquation(PointF p1, PointF p2)
{
    float
        x1 = p1.X, y1 = p1.Y,
        x2 = p2.X, y2 = p2.Y,
        k = (y2 - y1) / (x2 - x1),
        b = p1.Y - (k * p1.X);

    return (k, b);
}
```

Программный код 3 – функция нахождения уравнения прямой.

Функция `MirrorPoint(PointF target, LineF relativeTo)` решает задачу отражения точки относительно отрезка и опирается на две вышеописанные функции. Задача отражения точки относительно линии является надзадачей к задаче отражения относительно точки. Понятно, что любое отражение является, в конечном итоге, отражением относительно точки, в данном случае нам нужно найти таковую точку на прямой, задаваемой отрезком `relativeTo` и отразить `target` относительно неё. Точка на прямой должна определяться перпендикуляром. Найти таковой перпендикуляр не очень сложно - сначала построим уравнение прямой, задаваемой отрезком из двух точек, формула для которого известна. Теперь необходимо решить следующее уравнение - находящее точку на данной прямой $y=kx+b$, ближайшей к данной точке. Из аналитической геометрии мы имеем возможность найти уравнение прямой, перпендикулярной данной и проходящую через заданную точку `target`. Поскольку мы уже знаем угловой коэффициент прямой `relativeTo`, мы, де-факто, можем сразу же взять таковой коэффициент для искомой прямой, в виде $-1/k$. Следовательно мы можем

записать уравнение искомой перпендикулярной прямой, проходящей через точку $\text{target}(x_1, y_1)$ в виде $y - y_1 = -1/k(x - x_1)$, а равно $y = -1/k(x - x_1) + y_1$. Теперь остается решить уравнение пересечения. Имеем систему $kx + b - y = 0$, $kNx + bN - y = 0$. Вычтем уравнения. Имеем $(k - kN)x + (b - bN) = 0$; Имеем $X = -(b - bN)/(k - kN)$. Далее находим Y подстановкой. Таким образом получим точку отражения данной точки относительно данной прямой. Отметим, что в данном случае имеется вырожденный случай, если relativeTo представляет собой вертикальную линию - для неё невозможно построить уравнение. В этом случае k и b будут представлять собой бесконечности. Этот случай следует рассмотреть отдельно. По сути, в данном случае необходимо просто переместить точку в сторону прямой на 2 (разницу по X). Функция представлена в программном коде 4.

```
public static PointF MirrorPoint(PointF target, LineF relativeTo)
{
    if(relativeTo.Start.X == relativeTo.End.X) {
        return MirrorPoint(target, new PointF(relativeTo.Start.X, target.Y));
    }

    var (k, b) = FindLinearEquation(relativeTo);
    var (kNormal, bNormal) = FindLinearEquation(-1 / k, target);

    var xMirror = -(b - bNormal) / (k - kNormal);
    var yMirror = (kNormal * xMirror) + bNormal; // y = kx + b

    return MirrorPoint(target, new PointF(xMirror, yMirror));
}
```

Программный код 4 – функция отражения точки относительно линии.

Функция $\text{FindCenter}(\text{PointF } p_1, \text{PointF } p_2, \text{PointF } p_3)$ решает задачу нахождения центра окружности по 3-м точкам на ней – рисунок 2. Данная функция требуется для рисования дуги по 3-м точкам. В целом, рисунок справа полностью описывает алгоритм нахождения. Требуется построить две прямые p_1p_2 , p_3p_2 , после чего построить к ним перпендикуляры, проходящие через точку центра, что, в целом, было косвенно описано выше. Возвращаемым значением являются две координаты.

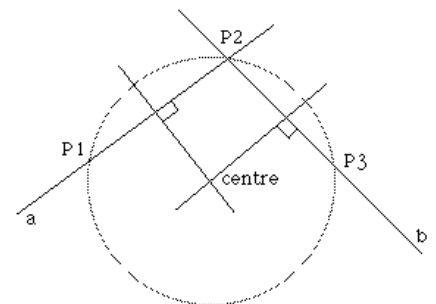


Рисунок 2.

Функция $\text{MinimalAngleBetweenAngles}(\text{float } a_1, \text{float } a_2)$ решает задачу нахождения минимального угла между двумя углами на окружности. Например, если один угол равен 0.1 , а второй – $2\pi - 0.1$, то в положительных числах разница между ними составляет $2\pi - 0.2$, но вот в геометрическом смысле разница между ними составляет 0.2 . Данное вычисление, в целом, является простым, однако решает важную задачу для построения дуги – помогает определить направление дуги. Направление дуги задается таким образом, что она рисуется в сторону второй точки относительно первой. Алгоритм написан таким образом, чтобы

программа в целом позволяла пользователю построить любую дугу, а результат был интуитивно ожидаем.

Функция `FindPerimeter(IEnumerable<LineF> lines)` находит периметр переданных линий. Она рассматривает начало каждой линии как центр окружности, а конец – как точку на ней, вызывая функцию `GetCircleRadius`, далее производя суммирование.

Функция `FindArea(IEnumerable<LineF> lines)` вычисляет площадь замкнутого контура, используя метод трапеций^[5] – рисунок 3. Смысл метода состоит в том, что площадь полигона S можно представить трапециями, у которых абсциссы являются основаниями, а разности ординат соседних точек высотами. Аналогичное преобразование можно проделать для любого другого многоугольника.

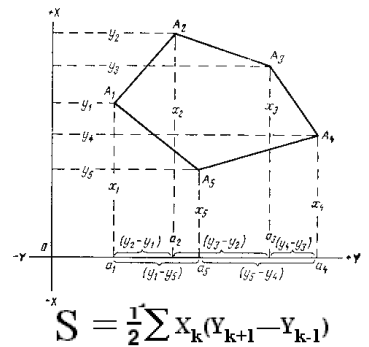
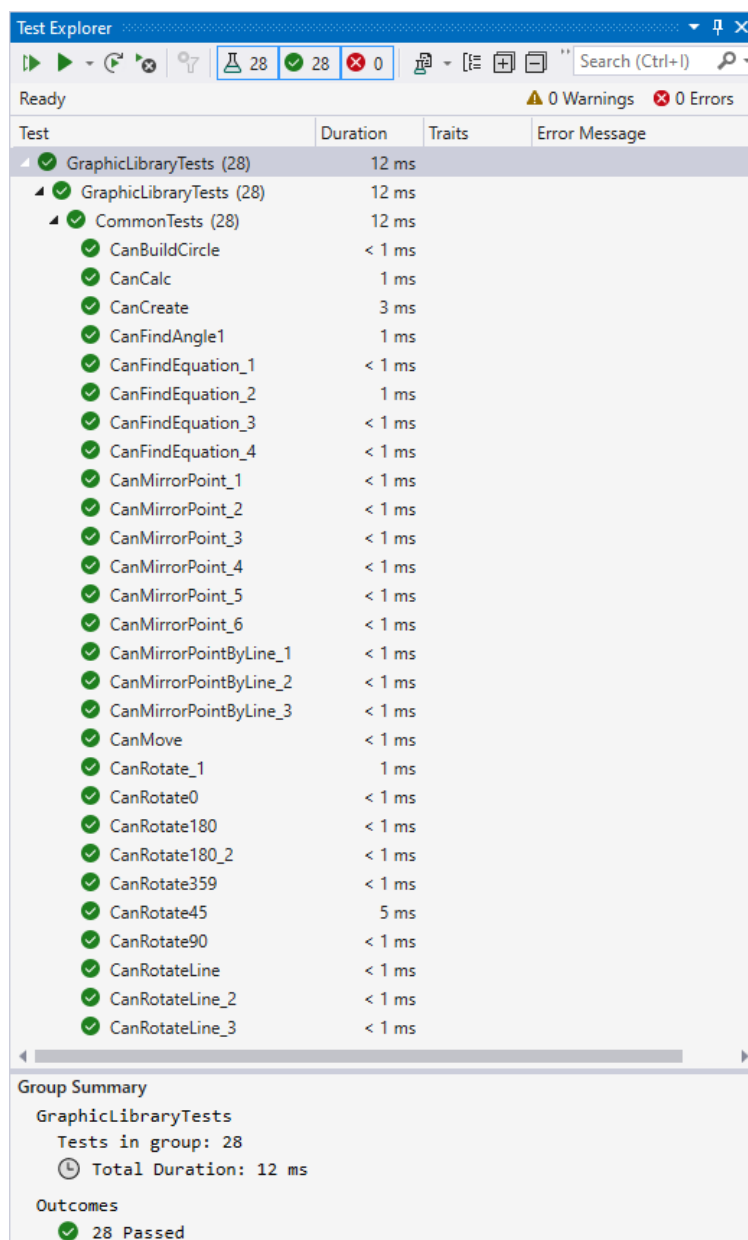


Рисунок 3.

ТЕСТИРОВАНИЕ

С целью тестирования методов ядра был добавлен проект xUnit – фреймворка для тестирования .net приложений. Суммарно было добавлено 28 тестов с простыми сценариями использования – рисунок 4.



The screenshot shows the Test Explorer window with the following data:

Test	Duration	Traits	Error Message
✓ GraphicLibraryTests (28)	12 ms		
✓ GraphicLibraryTests (28)	12 ms		
✓ CommonTests (28)	12 ms		
✓ CanBuildCircle	< 1 ms		
✓ CanCalc	1 ms		
✓ CanCreate	3 ms		
✓ CanFindAngle1	1 ms		
✓ CanFindEquation_1	< 1 ms		
✓ CanFindEquation_2	1 ms		
✓ CanFindEquation_3	< 1 ms		
✓ CanFindEquation_4	< 1 ms		
✓ CanMirrorPoint_1	< 1 ms		
✓ CanMirrorPoint_2	< 1 ms		
✓ CanMirrorPoint_3	< 1 ms		
✓ CanMirrorPoint_4	< 1 ms		
✓ CanMirrorPoint_5	< 1 ms		
✓ CanMirrorPoint_6	< 1 ms		
✓ CanMirrorPointByLine_1	< 1 ms		
✓ CanMirrorPointByLine_2	< 1 ms		
✓ CanMirrorPointByLine_3	< 1 ms		
✓ CanMove	< 1 ms		
✓ CanRotate_1	1 ms		
✓ CanRotate0	< 1 ms		
✓ CanRotate180	< 1 ms		
✓ CanRotate180_2	< 1 ms		
✓ CanRotate359	< 1 ms		
✓ CanRotate45	5 ms		
✓ CanRotate90	< 1 ms		
✓ CanRotateLine	< 1 ms		
✓ CanRotateLine_2	< 1 ms		
✓ CanRotateLine_3	< 1 ms		

Group Summary
GraphicLibraryTests
Tests in group: 28
⌚ Total Duration: 12 ms
Outcomes
✓ 28 Passed

Рисунок 4.

РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

Рисование любого объекта состоит в целом в выборе режима и дальнейшем рисовании левой кнопкой мыши. Выход из режима (отмену операции) можно осуществить клавишей Esc. Особенно имеет только рисование полилинии. Для неё имеется рисование точки конца отрезка – левой кнопкой мыши и рисование средней точки дуги – правой кнопкой мыши, что позволяет рисовать линии любого вида – рисунок 4. Завершение объекта путем постановки последней точки осуществляется нажатием средней кнопки мыши. После выделения объектов с помощью соответствующего инструмента, на рисунке 5 – выделения частичным включением, можно менять свойства объектов: перемещать их, вращать, устанавливать цвет, удалять. Выделенные объекты отображаются в списке слева снизу.

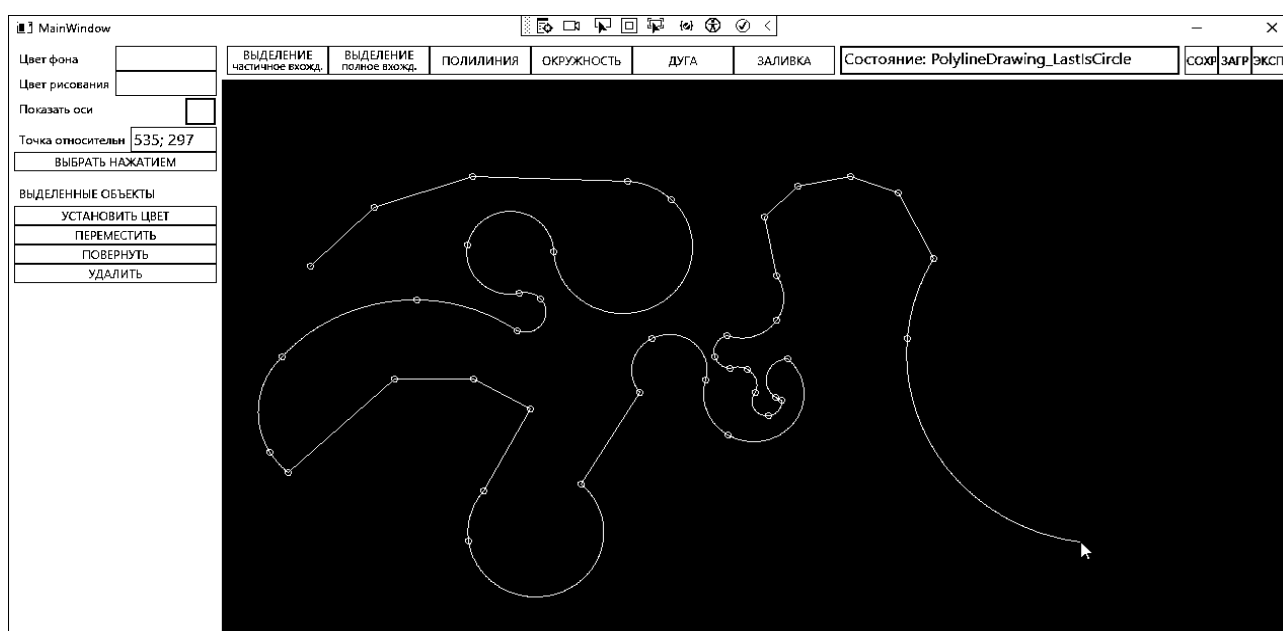


Рисунок 5.

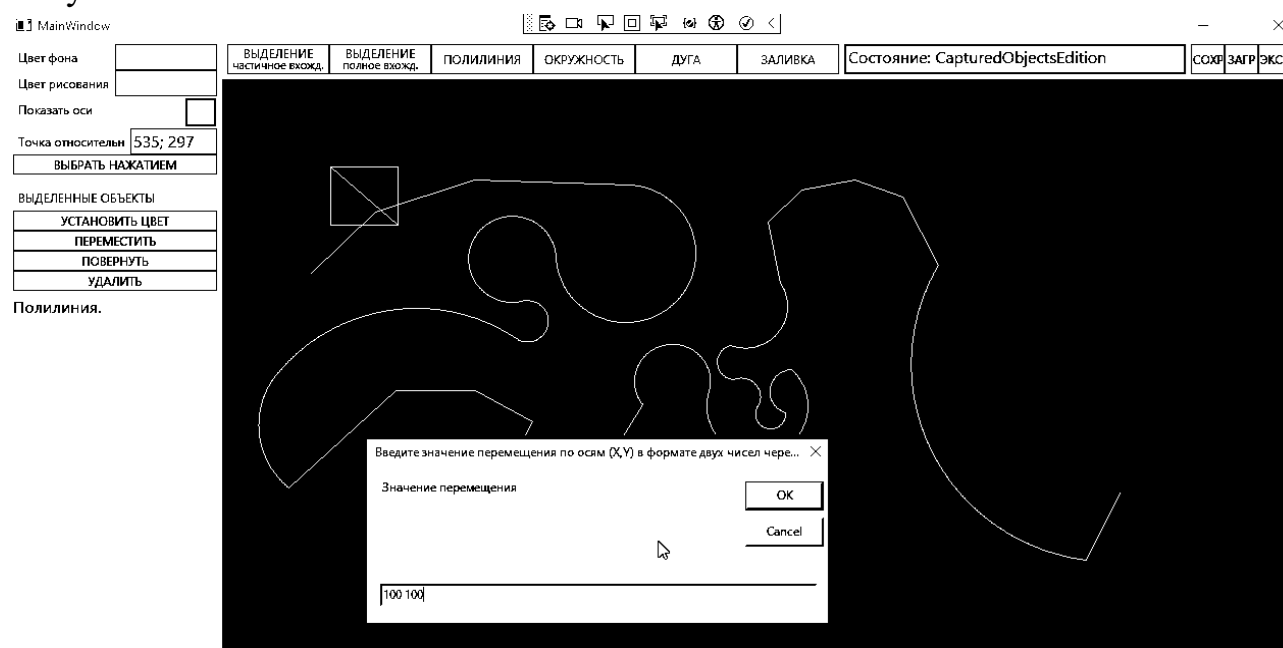


Рисунок 6.

В правом верхнем углу находятся 3 кнопки – сохранить, загрузить, экспортировать. Первые 2 работают с файлами формата `vcdproj`, представляющими, по-сути, JSON-сериализованное состояние холста, которое можно восстановить путем загрузки файла. Экспорт служит для выгрузки изображения как растрового.

ИСТОЧНИКИ

1. Системы автоматизированного проектирования. Конспект лекций: Учебное пособие. – М.: МАТИ, кафедра ТПДЛА, 2011. – 72 с.
2. Сайт о программировании [Электронный ресурс] - stackoverflow.com/a/34801225/11325184 (дата обращения: 14.04.2023)
3. Сайт об алгоритмах [Электронный ресурс] - algolist.manual.ru/maths/geom/equation/circle.php (дата обращения: 14.04.2023)
4. Викиверситет. Проект Фонда Викимедиа. [Электронный ресурс] - ru.wikiversity.org/wiki/Ядро_архитектуры (дата обращения: 19.04.2023)
5. Сайт об алгоритмах [Электронный ресурс] - algolist.manual.ru/maths/geom/polygon/area.php (дата обращения: 14.04.2023)