Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Институт информационных технологий

Факультет компьютерных технологий

Кафедра информационных систем и технологий

Дисциплина: Основы алгоритмизации и программирования (ОАиП)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовой работе

на тему

«Создание графского редактора на C#»

Студентка: гр. 181073 Лазарева Ю.А.

Руководитель: ассистент кафедры ИСиТ

Сицко Владимир Александрович

Минск 2022

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 3](#_Toc103717559)

[1. Анализ прототипов, литературных источников и формирование требований к проектируемому программному средству 5](#_Toc103717560)

[2. Анализ требований к программному средству и разработка технического задания (функциональных требований) 9](#_Toc103717561)

[3. Проектирование программного средства 10](#_Toc103717562)

[4. Создание (конструирование) программного средства 19](#_Toc103717563)

[5. Оценка работы программного средства и анализ полученных результатов 22](#_Toc103717564)

[6. Руководство пользователя (по установке и эксплуатации) 26](#_Toc103717565)

[ЗАКЛЮЧЕНИЕ 36](#_Toc103717566)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 37](#_Toc103717567)

[ПРИЛОЖЕНИЕ А 38](#_Toc103717568)

# ВВЕДЕНИЕ

Представление данных на мониторе компьютера в графическом виде впервые было реализовано в середине 50-х годов для больших ЭВМ, применявшихся в научных и военных исследованиях. С тех пор графический способ отображения данных стал неотъемлемой частью подавляющего числа компьютерных систем, в особенности персональных.

Компьютерная графика – это специальная область информатики, изучающая методы и средства создания и обработки изображений с помощью программно-аппаратных вычислительных комплексов. Она охватывает все виды и формы представления изображений, доступных для восприятия человеком либо на экране монитора, либо в виде копии на внешнем носителе (бумага, киноплёнка, ткань и прочее).

Без компьютерной графики невозможно представить себе не только компьютерный, но и обычный, вполне материальный мир. На сегодняшний день компьютеры и компьютерная графика неотъемлемая часть жизни современного общества. Для примера назовём медицину (компьютерная томография), научные исследования (визуализация строения вещества, векторных полей и других данных), моделирование тканей и одежды, опытно-конструкторские разработки, рекламные щиты, цветные журналы, спецэффекты в фильмах - всё это в той или иной мере имеет отношение к компьютерной графике. Поэтому созданы программы для создания и редактирования изображений, то есть графические редакторы.

Компьютерной графикой в последнее время занимаются многие, что обусловлено высокими темпами развития вычислительной техники. Более 90% информации здоровый человек получает через зрение или ассоциирует с геометрическими пространственными представлениями. Компьютерная графика имеет огромный потенциал для облегчения  
процесса познания и творчества.

В настоящее время рынок программного обеспечения переполнен различными программами и редакторами, позволяющими обрабатывать и редактировать цифровые фото. Человеку, не слишком хорошо понимающему особенности тех или иных программных средств, порой очень сложно разобраться в этом многообразии софта. Однако, правильный выбор программных средств для решения конкретной задачи по обработке фотоснимков является одним из залогов успеха получения законченных фотографий. Говоря о графических редакторах, прежде всего, необходимо отметить, что все цифровые изображения подразделяются на векторные и точечные. В первом случае изображения построены из различных геометрических элементов или примитивов (отрезков, треугольников, прямоугольников или окружностей). Поэтому векторная графика позволяет легко манипулировать масштабом изображения без каких бы то ни было геометрических искажений, а потому широко используется для построения шрифтов, рисованных изображений, в оформительской и издательской работе. Например, очень популярным редактором для обработки векторных изображений является программа Adobe Illustrator, которая часто применяется в издательском деле для подготовки к печати рисованных иллюстраций для газет, журналов и книг. Но я не ограничусь обзором всего одной программы и опишу несколько их типов.

Графический редактор – программа (или пакет программ), позволяющая создавать и редактировать двумерные изображения с помощью компьютера

Актуальность курсовой работы на тему «Графический редактор» обусловлена тем, что любой школьник, студент и преподаватель в настоящее время должен владеть компьютерными технологиями на высоком уровне.

Объектом курсовой работы служат современные компьютерные технологии с их аппаратным и программные обеспечением.   
Предметом курсового исследования являются функциональные возможности системы графических редакторов, используемые в науке и образовании.

Целью курсовой работы является создание графического редактора, используя объектно-ориентированный язык программирования, описание его функциональных возможностей.

Гипотеза курсовой работы состоит в следующем: систематическое и регулятивное освоение и использование функциональных возможностей графического редактора способствует повышению уровня графической подготовки школьников и студентов, решающих учебные и научные задачи.

Задачами курсовой работы выделяется:

* создание графического редактора, используя объектно-ориентированный язык программирования, описание его функциональных возможностей;
* возможность выбора цвета, использования ластика, кисти;
* создание фигур, заливка фигуры.

# 1. Анализ прототипов, литературных источников и формирование требований к проектируемому программному средству

Работа над любой иллюстрацией заключается в создании объектов, их редактировании и задании их правильного взаимного расположения. Исходя из этого, основными принципами работы в CorelDRAW можно назвать следующие:

* создание простых геометрических фигур или произвольных кривых и ломаных, как замкнутых, так и разомкнутых. Вставка и форматирование текста;
* редактирование любого объекта, изменение цвета контура и заливки, изменение формы объекта;
* вставка готовых картинок или ранее созданных иллюстраций в документ;
* применение разнообразных художественных эффектов;
* точное позиционирование объектов относительно страницы и друг друга.

Элементы рабочего окна редактора. При первом запуске программы CorelDRAW на экране появится диалоговое окно, в котором можно выбрать режим работы. Если флажок Show this Welcome Screen at startup (Показывать это окно при запуске) оставить установленным, то это диалоговое появится и при следующем запуске программы. Чтобы начать работу над новым рисунком, необходимо выбрать вариант New Graphic (Новый рисунок).

Интерфейс рабочего окна программы, как и интерфейс большинства приложений, предназначенных для работы в ОС Windows, содержит в первую очередь стандартные элементы: заголовок, системное меню, главное меню и стандартную панель инструментов.

Главное меню обеспечивает доступ к командам CorelDRAW, объединенным в следующие группы:

* **File** (Файл) – команды, позволяющие открывать, сохранять, импортировать, экспортировать файлы, распечатывать рисунки, публиковать их в Internet; в нижней части списка команды расположен список последних открывавшихся файлов;
* **Edit** (Правка) – команды, позволяющие отменять и повторять последние действия, копировать объекты, настраивать их свойства;
* **View** (Вид) – команды, позволяющие настраивать окно отображения объектов, устанавливать и настраивать вспомогательные элементы окна;
* **Layout** (Макет) – команды, позволяющие добавлять, удалять и переиме­новывать страницы графического документа и настраивать их свойства;
* **Arrange** (Упорядочить) – команды, позволяющие активизировать инстру­менты преобразования и выравнивания объектов;
* **Effects**(Эффекты) – команды, позволяющие добавить различные специальные эффекты к объектам (например, эффекты выдавливания, тени, прозрачности и т.п.);
* **Bitmaps** (Растры) – команды, позволяющие преобразовывать векторные изображения в растровые и наоборот, редактировать растровые изображения и добавлять к ним различные эффекты;
* **Text** (Текст) – команды для работы с текстовыми объектами;
* Tools (Инструменты) – команды, позволяющие настраивать интерфейс программы, включать и отключать панели инструментов, а также настраивать их;
* **Window** (Окно) – команды, позволяющие переключаться между окнами разных файлов, одновременно открытых в CorelDRAW, а также вызывать допол­нительные окна палитр, докеров и панелей инструментов;
* **Help** (Помощь) – команды, позволяющие обратиться к встроенной справочной системе CorelDRAW.



Рисунок 1.1 – заставка при загрузке CorelDRAW

Далее рассмотрим такой аналог как Adobe Illustrator.

Adobe Illustrator – это программа, созданная для работы с векторной графикой. С помощью Adobe Illustrator дизайнеры создают красочные иллюстрации, иконки, паттерны, логотипы, различные макеты для печати и многое другое. Начинающему дизайнеру будет полезно освоить главные функции редактора, ведь без него как без рук.

Основные функции:

* работа с геометрическими фигурами. В Иллюстраторе можно ловко работать с фигурами объединять их, вычитать, дополнять другими контурами и так далее. Геометрические формы часто являются основами для иконок и логотипов, поэтому работа с ними – вещь фундаментальная;
* создание иллюстраций. Делайте яркие иллюстрации из фигур, линий и контуров, применяйте различные эффекты и текстуры. В Adobe Illustrator можно создавать любые формы, накладывать их друг на друга с помощью слоев и применять различные эффекты: градиенты, тени, текстуры и прочее;
* трассировка растровой графики. Иллюстратор дает возможность переводить растровое изображение в векторное за пару кликов;
* редактирование контуров. Рисуйте самостоятельно инструментом «Перо» или редактируйте чужие иллюстрации. Вы можете скачивать проекты в Ai-формате со стоков и редактировать каждый элемент;
* применение заливок. В Adobe Illustrator можно с легкостью применять различные заливки к объектам и контурам;
* подготовка к печати. Визитки, плакаты или огромные рекламные щиты. Так как векторная графика масштабируется до любых размеров, в Иллюстраторе можно создавать рекламные щиты размером с дом. Параметры экспорта файла позволяют настроить вылеты, цветовой профиль для печати, а также выбрать формат;
* работа с монтажными областями. С помощью монтажных областей удобно, к примеру, делать баннеры для сайта в различных размерах;
* преобразование текста в кривые. Любой текст в Adobe Illustrator можно перевести в кривые, то есть сделать векторным. Это позволит работать с ним так, будто это векторная форма – можно изменять форму букв, сделать логотип или красивую надпись;
* создание кистей. Сделав собственные кисти, вы сможете добавить еще больше уникальности своим иллюстрациям. Если на создание своих кистей времени нет, их можно загружать со сторонних ресурсов.



Рисунок 1.2 – Adobe Illustrator

Литература для создания данного графического редактора бралась источников, таких как:

* metanit.com;
* question-it.com;
* stackowerflow.ru.

Требования к курсовому проекту формировались из концепта самого программного средства. Как было описано в введении, что графический редактор представляет собой программу или пакет программ, позволяющая создавать, просматривать, обрабатывать и редактировать цифровые изображения (рисунки, картинки, фотографии) на компьютере. Для этого должны быть основные функции программы, такие как:

* выбор цвета;
* использование ластика;
* использование кисти;
* создание фигур;
* заливка фигур;
* очистка листа;
* и другие действия пользователя.

# 2. Анализ требований к программному средству и разработка технического задания (функциональных требований)

Анализ требований – часть процесса разработки программного обеспечения, включающая сбор требований к программному обеспечению, выявление взаимосвязей, их систематизацию, а также документирование. Являясь частью общеинженерной дисциплины «инженерия требований». Необходимость объясняется принятием во внимание возможных противоречий в требованиях различных заинтересованных лиц, таких как заказчики, разработчики или пользователи. Полнота и качество анализа требований играют ключевую роль в успехе всего проекта. Требования к программному обеспечению должны быть документируемые, выполнимые, тестируемые, с уровнем детализации, достаточным для проектирования системы.

Требования подразделяются на:

* нефункциональные – требования, определяющие свойства, которые система должна демонстрировать и ограничения, которые должна соблюдать, не относящиеся в свою очередь к поведению системы;
* функциональные – требования объясняют, что должно быть сделано. Они идентифицируют задачи или действия, которые должны быть выполнены, данные требования определяют действия, которые также система должна выполнить. С помощью письменной установки требований можно добиться полного выполнения поставленных задач и приступить к переходу к следующему циклу постановки дополнительных требований в процессе развития программы, что сделает систему развивающейся.

В ходе определения необходимости требований перейдем непосредственно к ним.

Рассмотрев аналоги, нужно сформировать требования к программному продукту:

* возможность выбора цвета;
* возможность заливки;
* возможность создания фигур;
* возможность заливки;
* возможность очистить лист.

Таким образом, сформированные функциональные требования расширяют своей конкретикой нефункциональные требования из предыдущего раздела, объясняя ключевые моменты проекта на данном этапе.

# 3. Проектирование программного средства

Разработка курсового проекта имеет распространенные требования, которые ставятся похожим графическим редакторам. В связи с этим для разработки ПО были выбраны соответствующие инструменты:

* Visual Studio – интегрированная среда разработки, редактор кода;
* C# – компилируемый статически типизированный язык

программирования общего назначения;

* WPF (Windows Presentation Foundation) – это .NET платформа для создания клиентских приложений для настольных систем.

Далее подробно рассмотрим каждый инструмент.

Microsoft Visual Studio – линейка продуктов компании Microsoft, включающих интегрированную среду разработки программного обеспечения и ряд других инструментов. Данные продукты позволяют разрабатывать как консольные приложения, так и игры, и приложения с графическим интерфейсом, в том числе с поддержкой технологии Windows Forms, UWP а также веб-сайты, веб-приложения, веб-службы как в родном, так и в управляемом кодах для всех платформ, поддерживаемых Windows, Windows Mobile, Windows CE, .NET Framework, .NET Core, .NET, MAUI, Xbox, Windows Phone .NET Compact Framework и Silverlight. После покупки компании Xamarin корпорацией Microsoft появилась возможность разработки IOS и Android программ. Visual Studio включает в себя редактор исходного кода с поддержкой технологии IntelliSense и возможностью простейшего рефакторинга кода. Встроенный отладчик может работать как отладчик уровня исходного кода, так и отладчик машинного уровня. Распространяется бесплатно, интерфейс изображен на рисунке 3.1.

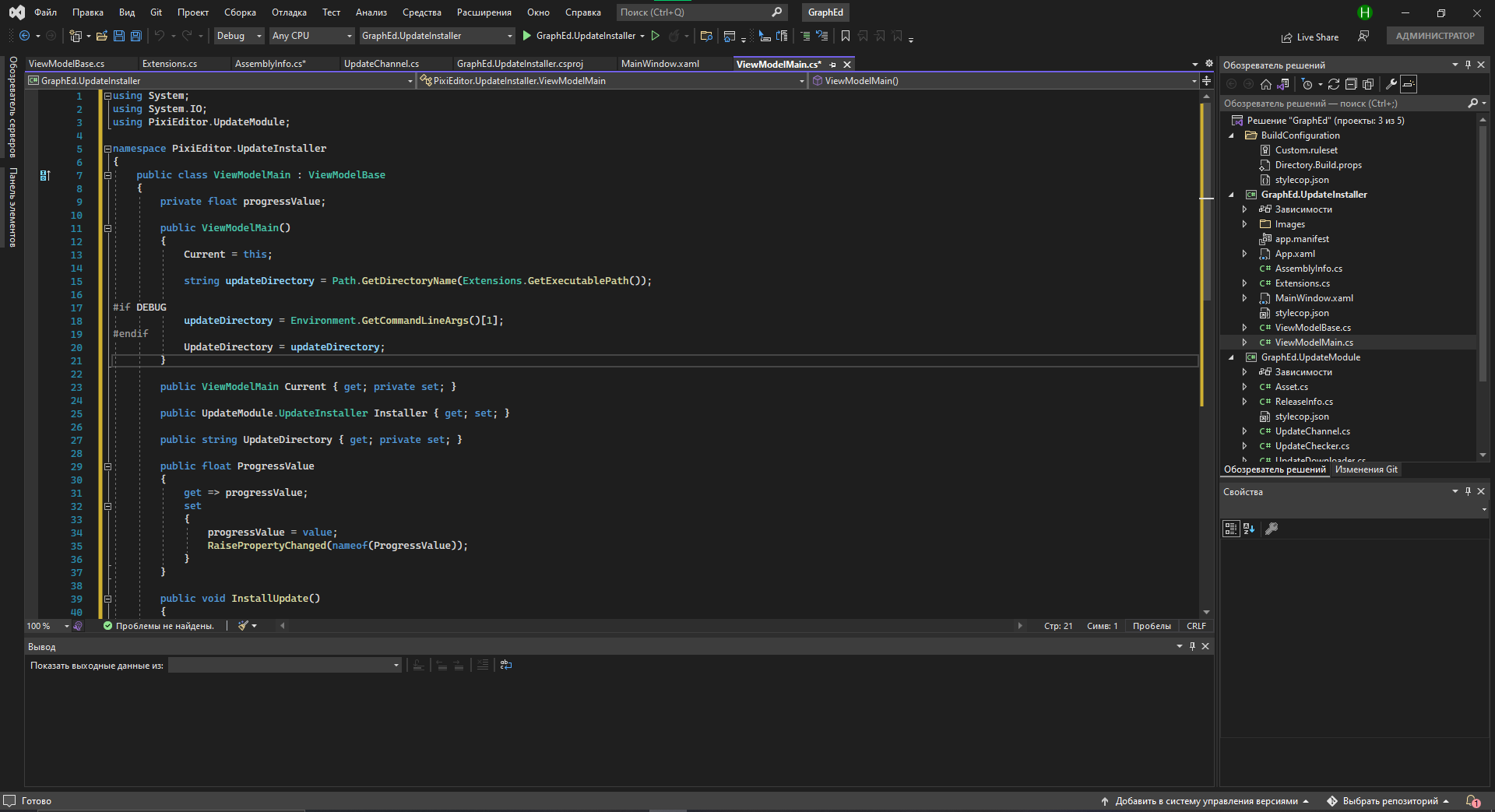


Рисунок 3.1 – Редактор кода Visual Studio 2022

Информационная база состоит из входной и выходной информации.

Входными данным будут те данные, которые пользователь будет вводить/изменять. Программой будут обрабатываться такие события, как:

* нажатия комбинации клавиш Ctrl+N, Ctrl+O, Ctrl+S, Ctrl+Shift+S, Ctrl+Shift+Alt+S, Ctrl+Z, Ctrl+Y, Ctrl+C, Ctrl+X, Ctrl+V, Ctrl+J, Ctrl+A, Ctrl+D, Ctrl+Shift+I, Ctrl+Shift+C, H, V, B, M, W, G, L, C, R, E, O, U, Z;
* нажатие ПКМ;
* нажатия на UI кнопки в программе.

В дальнейшем список входных параметров может изменяться в связи с совершенствованием программы.

Выходная информация представляет собой результат обработки входной информации для решения поставленной задачи.

Выходной информацией разрабатываемого приложения является:

* изображение квадрата на холсте;
* изображение круга на холсте;
* заливка фигуры на холсте;
* использование пипетки;
* сохранение файла;
* открытие файла.

При разработке данного проекта использовался подход модульного программирования. Он в наиболее полной мере позволяет поддерживать и развивать данный продукт в дальнейшем.

Модульное программирование – это организация программы, как совокупности небольших независимых блоков, называемых модулями, структура и поведение которых подчиняются определённым правилам. Использование модульного программирования позволяет упростить тестирование программы и обнаружение ошибок. Аппаратно-зависимые подзадачи могут быть строго отделены от других подзадач, что улучшает мобильность создаваемых программ.

Модуль – функционально законченный фрагмент программы. Во многих языках оформляется в виде отдельного файла с исходным кодом или поименованной непрерывной её части. Некоторые языки предусматривают объединение модулей в пакеты.

Термин «модуль» в программировании начал использоваться в связи с внедрением модульных принципов при создании программ. В 1970-х годах под модулем понимали какую-либо процедуру или функцию, написанную в соответствии с определёнными правилами. Например: «модуль должен быть простым, замкнутым (независимым), обозримым (от 50 до 100 строк), реализующим только одну функцию задачи, имеющим одну входную и одну выходную точку».

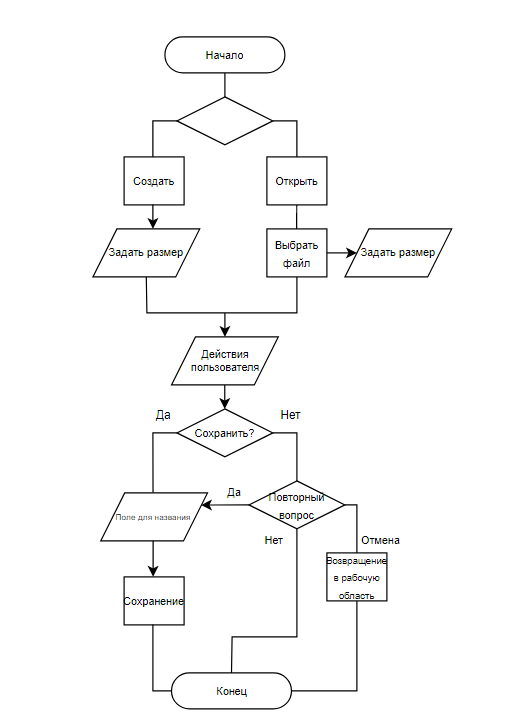
Первым основные свойства программного модуля более-менее чётко сформулировал Д. Парнас (David Parnas) в 1972 году: «Для написания одного модуля должно быть достаточно минимальных знаний о тексте другого». Таким образом, в соответствии с определением, модулем могла быть любая отдельная процедура (функция) как самого нижнего уровня иерархии (уровня реализации), так и самого верхнего уровня, на котором происходят только вызовы других процедур-модулей.

При создании проекта будет применен этот подход, а также будет создано множество скриптов, которые будут придерживаться принципов ООП и SOLID. В частности, особое внимание уделяется принципу единственной ответственности и абстракции. Это будет видно на структуре скриптов.

Также в процессе проектирования были разработаны 5 блок-схем, описывающую логику тех или иных процессов.

Блок-схема – это схематичное представление процесса, системы или компьютерного алгоритма. Блок-схемы часто применяются в разных сферах деятельности, чтобы документировать, изучать, планировать, совершенствовать и объяснять сложные процессы с помощью простых логичных диаграмм. Для построения блок-схем применяются прямоугольники, овалы, ромбы и некоторые другие фигуры (для обозначения конкретных операций), а также соединительные стрелки, которые указывают последовательность шагов или направление процесса. Блок-схемы варьируются от незамысловатых, нарисованных вручную до подробных, составленных на компьютере диаграмм со множеством шагов и процессов. Если учесть все возможные вариации, блок-схемы можно признать одним из самых распространенных видов схем во всем мире. Они широко используются в разных сферах как технической, так и нетехнической направленности. Иногда блок-схемы получают более узкоспециальные названия, например, схема процесса, схема рабочего процесса, функциональная блок-схема, моделирование бизнес-процессов, модель и нотация бизнес-процессов (BPMN) или схема технологического процесса (PFD). Они тесно связаны с другими распространенными видами схем, такими как диаграммы DFD и диаграммы активности на унифицированном языке моделирования (UML). Рассмотрим разработанные схемы.

На рисунке 3.2 изображена блок-схема, описывающая логику работы программы.



На рисунке 3.2 изображена блок-схема, описывающая логику работы программы.

Логика данного программного средства следующая: пользователь входит в графический редактор и ему предлагается выбор – создать новый файл, либо открыть. Разница только в том, что при создании нового файла программа дает задать размер изображения, с выбором файла и его последующим открытием такого нет. После, каких-ибо действий пользователя по изменениям файла ему будет предложено сохранить результат, либо же не сохранять, но при этом программа спросит еще раз, точно ли пользователь хочет выйти и не сохранится, предоставив три варианта:

* да – сохранить;
* нет – не сохранять;
* отмена – возвращение в рабочую область.

И непосредственно выход из программы.

На рисунке 3.3 изображена блок-схема, описывающая логику создания проекта.

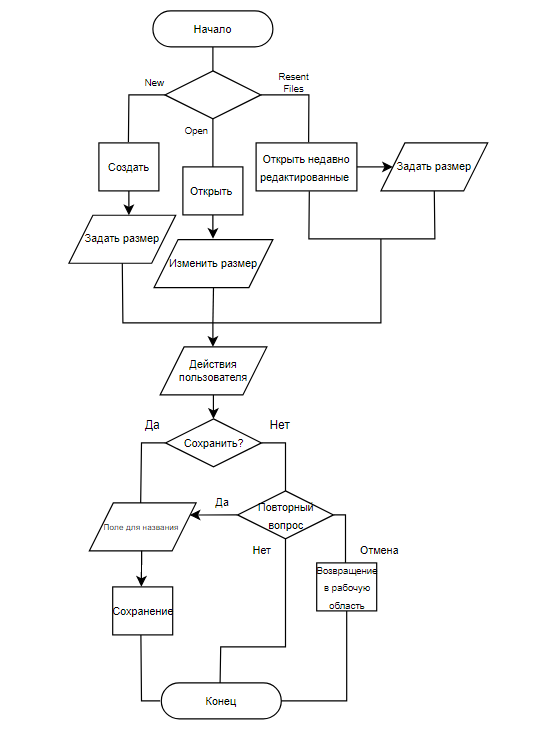


Рисунок 3.3 – Создание проекта

Логика практически такая же, как и в предыдущей блок схеме, только в блок-схеме о логике работы программы показан основной концепт, в случае с созданием проекта прописан только один дополнительный момент: использование ранее редактируемого изображения. Логика в том, чтобы создать, открыть или использовать недавно редактируемый файл и произвести те или иные действия с изображением.

Теперь рассмотрим алгоритм создания фигур на рисунке 3.4

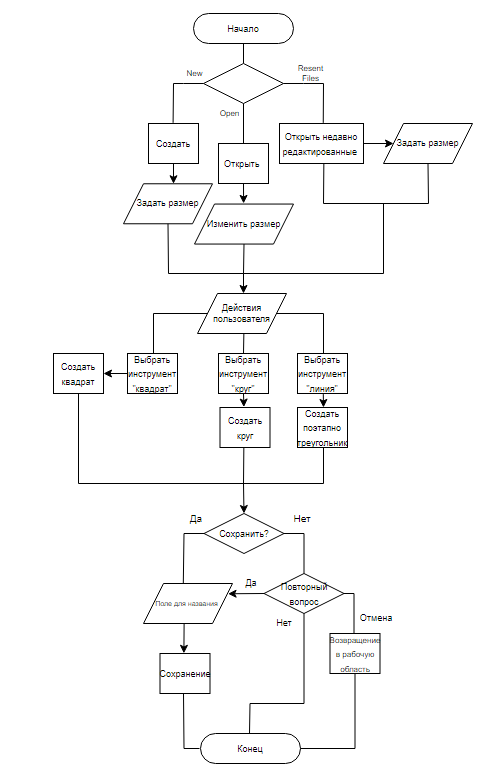


Рисунок 3.4 – Алгоритм создания фигур

Данный алгоритм плавно вытекает из двух предыдущих. Для того, чтобы создать фигуру нужно создать, открыть, либо выбрать недавно использованный файл, по желание изменить размер изображения и перейти к непосредственному созданию фигур.

Следующий алгоритм – изменение размеров фигур, представленный на рисунке 3.5.

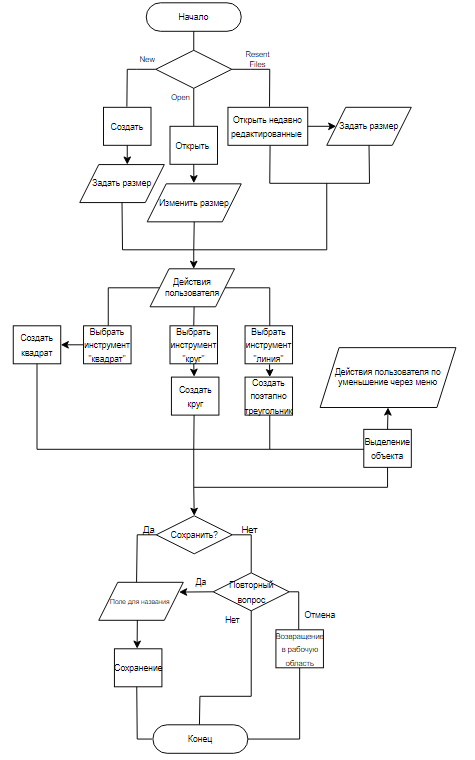


Рисунок 3.5 – Алгоритм изменения размера фигур

Алгоритм показывает, как будет проходить изменение размера, путем выделения пользователем объекта и изменения его размера.

Последняя схема алгоритма – это схема алгоритма сохранения изображания, представленная на рисунке 3.6.

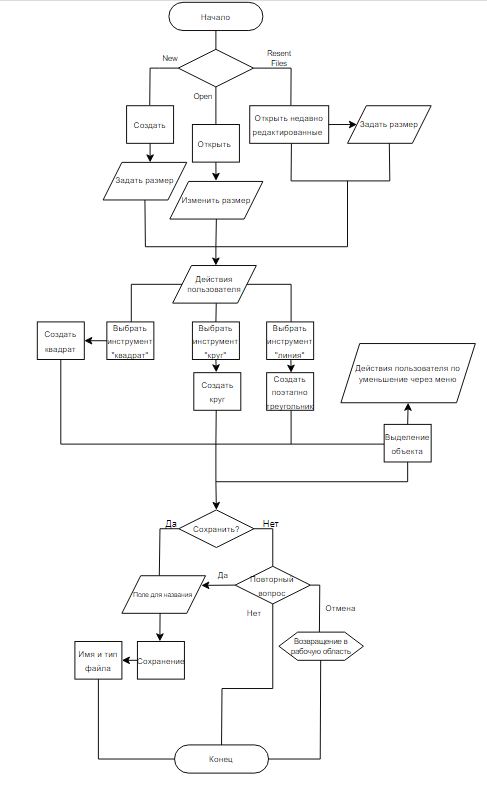


Рисунок 3.6 – Алгоритм сохранения

По алгоритму видно, что пользователь может сохранить файл придумав название и выбрав тип, так же может не сохранять и при этом утерять весь процесс работы.

Пользовательский интерфейс меню с сохранением представлен на рисунке 3.7.

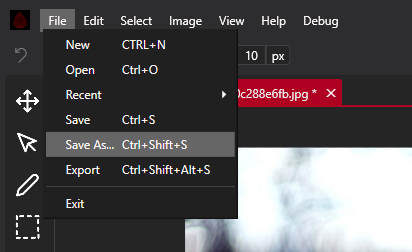


Рисунок 3.7 – Меню с сохранением

# 4. Создание (конструирование) программного средства

В ходе работы были использованы \*.cs, \*.xaml файлы. Их использование в программном средстве будет представлено скриншотом ниже.

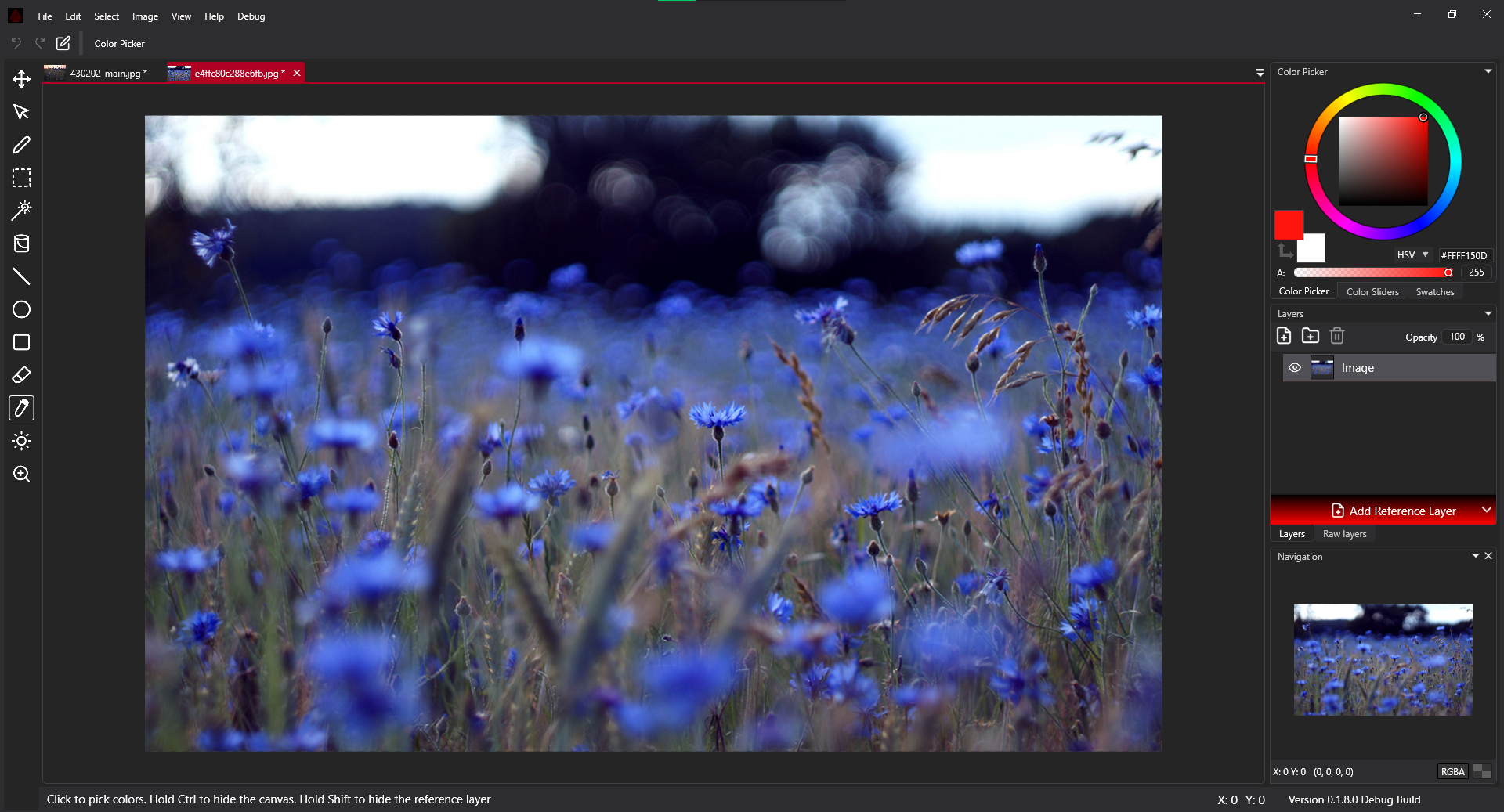


Рисунок 4.1 – Открытый файл

В данном окне есть все необходимое для создания и изменения графического изображения.

Рассмотрим созданные файлы в процессе разработки данного программного продукта, представленные на рисунке 4.2.

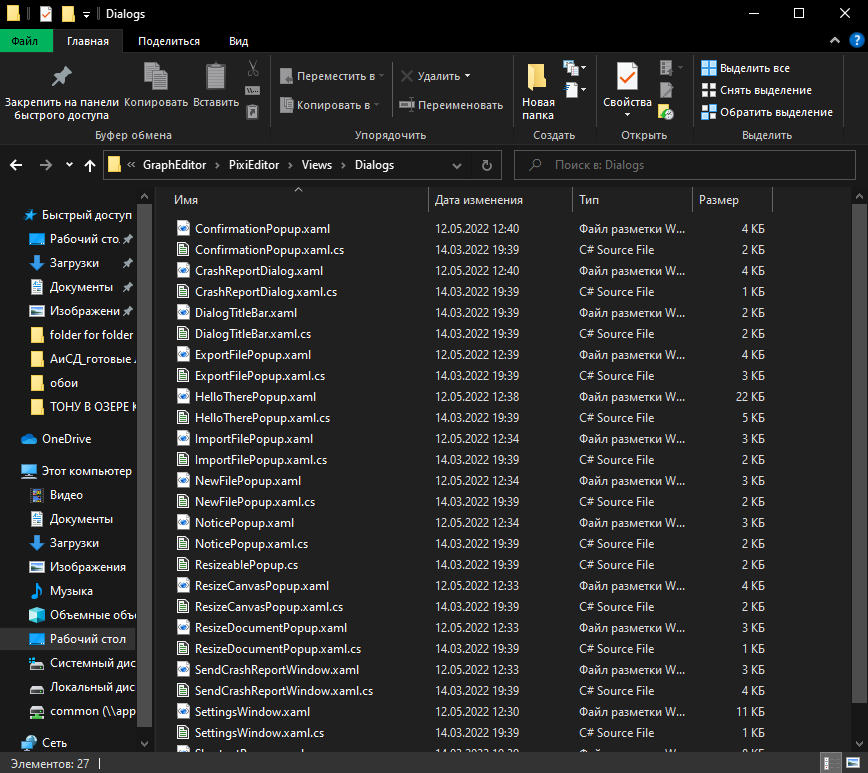


Рисунок 4.2 – Разработанные файлы

Файл ConfirmationPopup предназначен для подтверждения тех или иных действий в всплывающем окне.

Файл CrashReportDialog отвечает за создание диалогового окна, сообщающего об ошибке.

Файл SavetFilePopup предназначен для сохранения файла.

Файл ExportFilePopup предназначен для экспорта файла.

Файл HelloTherePopup предназначен для окна приветствия.

Файл ImportFilePopup предназначен для импорта файла.

Файл NewFilePopup предназначен для создания нового файла.

Файл ResizeCanvasPopup предназначен для изменения размера холста.

В ходе работы было разработано множество файлов для удобства работы с данным ПО.

Предназначение файлов сводится к созданию и управлению формами, сохранением, созданием, импортом и экспортом, созданием фигур, заливки и других действий. Разделение логики на множество небольших файлов позволяет локализировать проблемы при их возникновении и упрощении процесса тестирования и отладки.

Таким образом, видно, что были учтены все тонкости, описанные в разделе проектирования программного средства, а также использованы практики для разработки надежного, высокопроизводительного и расширяемого кода.

# 5. Оценка работы программного средства и анализ полученных результатов

Для работы с программой необходим файл «PixiEditor.exe», который является набором WPF-форм, которые можно с легкостью объединить с другими формами WPF, включая формы из других проектов. Этот файл является конечным результатом компиляции проекта.

Окно при начале работы с графическим редактором.

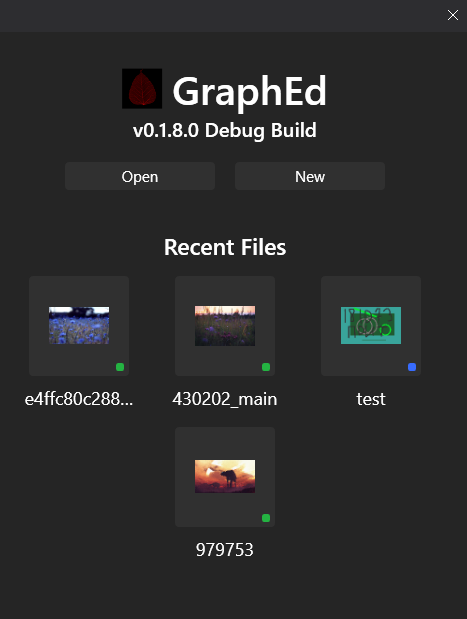


Рисунок 5.1 – Главное меню

На данном скриншоте представлено создание нового изображения с возможность запомнить введенные размеры.

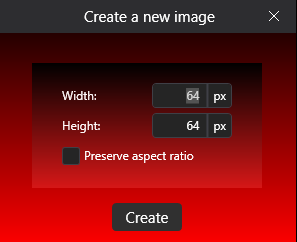


Рисунок 5.2 – Создание нового изображения

Программа проектировалась таким образом, чтобы интерфейс был эргономичным и по пирамиде Люшера гармонировал с цветовой гаммой, что не мало важно для удобства пользователя

Также было уделено внимание тестированию данного продукта, поскольку имелась необходимость провести проверку соответствия между реальным поведением программы и её ожидаемым поведением на конечном наборе тестов, выбранных определенным образом.

Цели тестирования для данного продукта были следующие:

* повысить вероятность того, что приложение, предназначенное для

тестирования, будет работать правильно при любых обстоятельствах;

* повысить вероятность того, что приложение, предназначенное для

тестирования, будет соответствовать всем описанным требованиям;

* предоставление актуальной информации о состоянии продукта на

данный момент.

Тестовый сценарий (Test Case) – это артефакт, описывающий совокупность шагов, конкретных условий и параметров, необходимых для проверки реализации тестируемой функции или её части. Далее будут приведены созданные тестовые сценарии для программы.

Чем позже дефект был обнаружен, тем дороже обойдется его исправление и тем больше усилий для этого потребуется. Кроме того, дефекты, заложенные в систему на ранних уровнях проектирования особенно коварны – их трудно отследить и правильно интерпретировать. Вывод напрашивается сам собой: чем раньше в жизненном цикле программы начнется тестирование, тем в большей степени мы можем быть уверены в ее качестве.

Также стоит учитывать, что невозможно провести исчерпывающее тестирование, которое бы покрывало все комбинации пользовательского ввода и состояний системы, за исключениями примитивных случаев. Вместо этого необходимо использовать анализ рисков и расстановку приоритетов, что позволит более эффективно распределять усилия по обеспечению качества ПО. Даже если тестирование не обнаружило дефектов, еще не значит, что программа готова к релизу. Нахождение и исправление дефектов будут не важны, если система окажется неудобной в использовании, и не будет удовлетворять ожиданиям и потребностям пользователя.

Таблица 5.1 Тест-кейсы

|  |  |  |
| --- | --- | --- |
| Действие | Ожидаемый результат | Результат |
| *1* | *2* | *3* |
| Использование кнопки Open | Кнопка будет активна, откроется проводник | Ожидаемый результат достигнут |
| Изменение размера изображения | Можно будет изменить ширину и высоту | Ожидаемый результат достигнут |
| Выбор инструмента «Квадрат» | Пользователь нарисует квадрат | Ожидаемый результат достигнут |
| Заливка фигуры | Пользователь выбирает инструмент для заливки, затем цвет в палитре, нажимает на область, которую хочет закрасить | Ожидаемый результат достигнут |
| Выбор инструмента «Пипетка» | Пользователь использует данный инструмент и выбирает необходимый ему цвет с изображения | Ожидаемый результат достигнут |
| Выбор инструмента «Ластик» | Пользователь использует данный инструмент, стирая не нужный элемент | Ожидаемый результат достигнут |
| Сохранение файла | Пользователь войдет в вкладку меню File и выберет Save As…  Далее появится проводник с полем для имени и типа файла, заполняя верно все поля | Ожидаемый результат достигнут |
| Выход из приложения | Приложение предложит сохранится перед выходом, пользователь может согласиться, может выйти и в этом случае потеряет все изменения | Ожидаемый результат достигнут |

Продолжение таблицы 5.1

Таким образом, был проверен основной функционал разрабатываемого приложения, а также достигнуты все цели, которые были поставлены перед началом тестирования. Приложение выполняет все основные поставленные перед ним задачи корректно.

Минимальные системные требования:

* ОЗУ 1Гб и более;
* свободное место на устройстве: 400 Мб;
* процессор тактовой частотой от 1 000 МГц;
* операционная система Windows (32-bit или 64-bit): Windows 7 /

Windows 8 / Windows 10.

Область применения разрабатываемого проекта достаточно широка. Приложение не привязано к региону и может использоваться пользователями со всего мира. Относительно невысокие системные требования позволяют пользоваться продуктом даже на низко производительных конфигурациях.

# 6. Руководство пользователя (по установке и эксплуатации)

Для использования данного продукта необходимо установить его на персональный компьютер. После установки необходимо запустить ярлык «PixiEditor.exe». Программное средство проектировалось для использования в решение не сильно сложных задач, связанных с векторной графикой.

Рассмотрим интерфейс программы. После запуска на экране появится главное меню, представленное на рисунке 6.1 из которого можно начать работу.

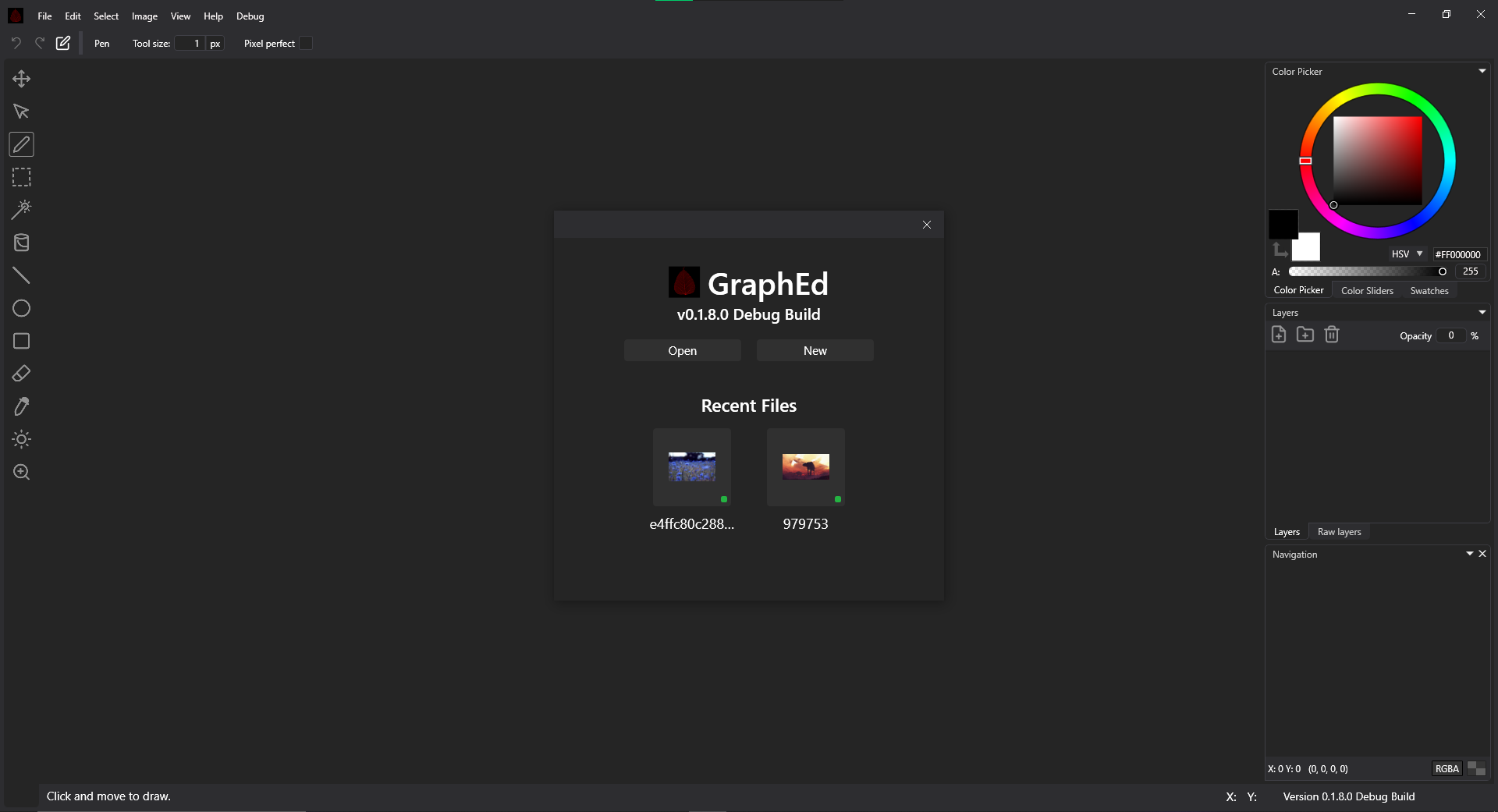


Рисунок 6.1 – Главное меню

Далее можно выбрать открытие файла, создание нового или редактирование недавно использованного. Цель графического редактора – Создание и изменение графического изображения.

Здесь будет продемонстрирован выбор каждого из пунктов меню.

Пункт Open (открыть), есть возможность использовать комбинацию клавиш Ctrl+O, представлено на рисунке 6.2. Так же при открытии файла можно изменить размер изображения, скриншот представлен на рисунке 6.3.

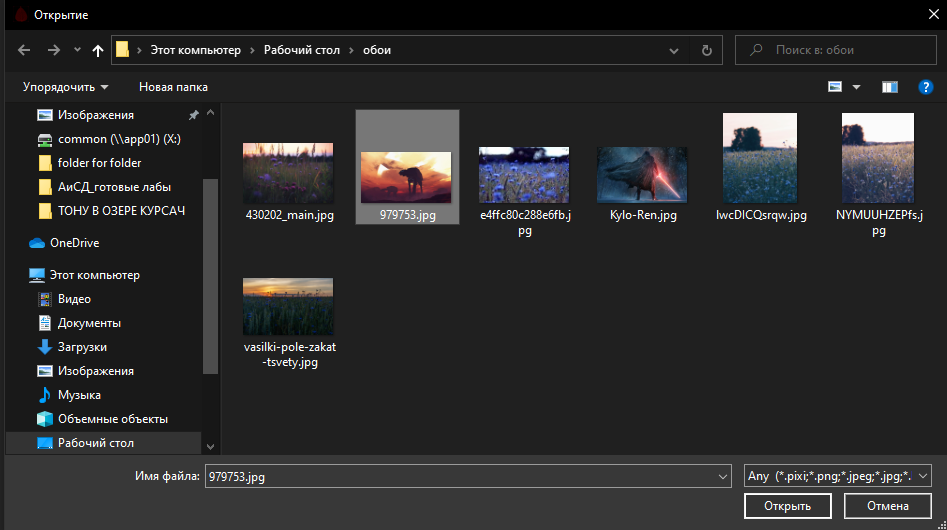
 Рисунок 6.2 – Пункт меню – Open



Рисунок 6.3 – Изменение размеров изображения в пункте открытия файла

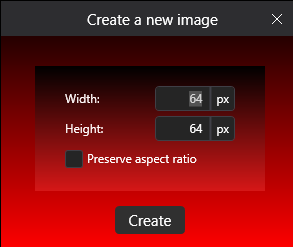
Пункт New (новый файл/создать), есть возможность использовать комбинацию клавиш Ctrl+N, представлено на рисунке 6.4.

Рисунок 6.4 – Пункт меню – New

Пункт Resent Files (повторно открыть), данный пункт появляется, если через приложение на данном устройстве открывались и изменялись какие-либо графические файлы, представлено на рисунке 6.4.

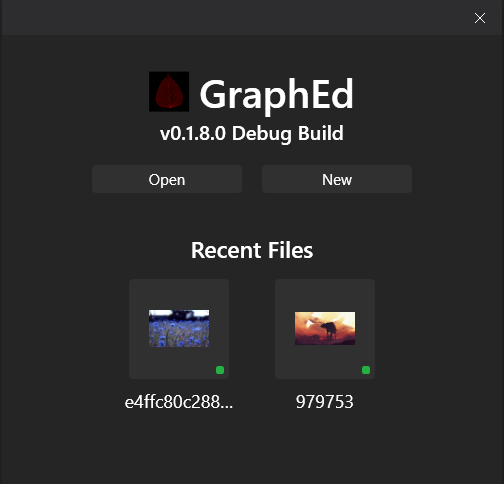


Рисунок 6.5 – Пункт меню – Resent Files

Так же при открытии недавно использованного файла можно изменить размер изображения, скриншоты представлены на рисунках 6.6, 6.7.

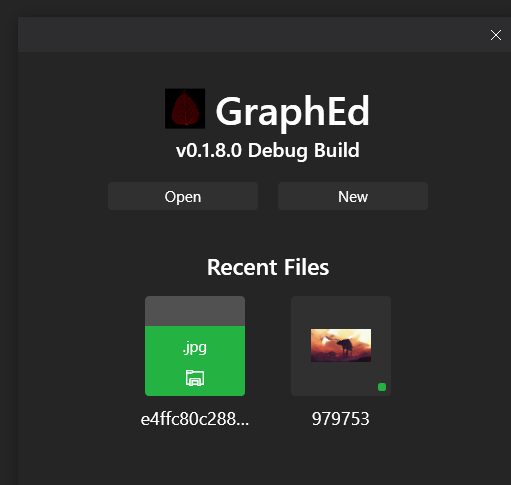


Рисунок 6.6 – Пункт меню – Resent Files

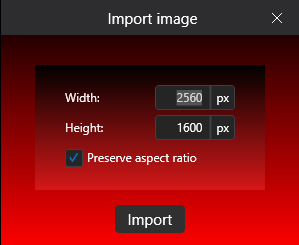


Рисунок 6.7 – Пункт меню – Resent Files

Непосредственно работа графического редактора будет продемонстрирована при создании нового документа. Скриншоты будут представлены на рисунках 6.8 -

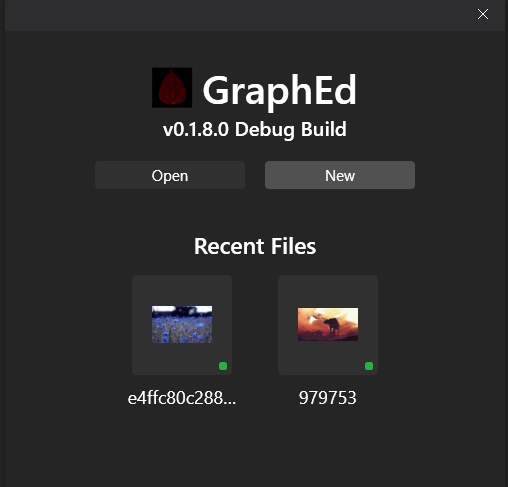


Рисунок 6.8 – Выбор пункта New

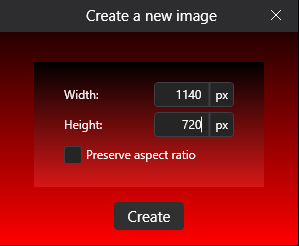
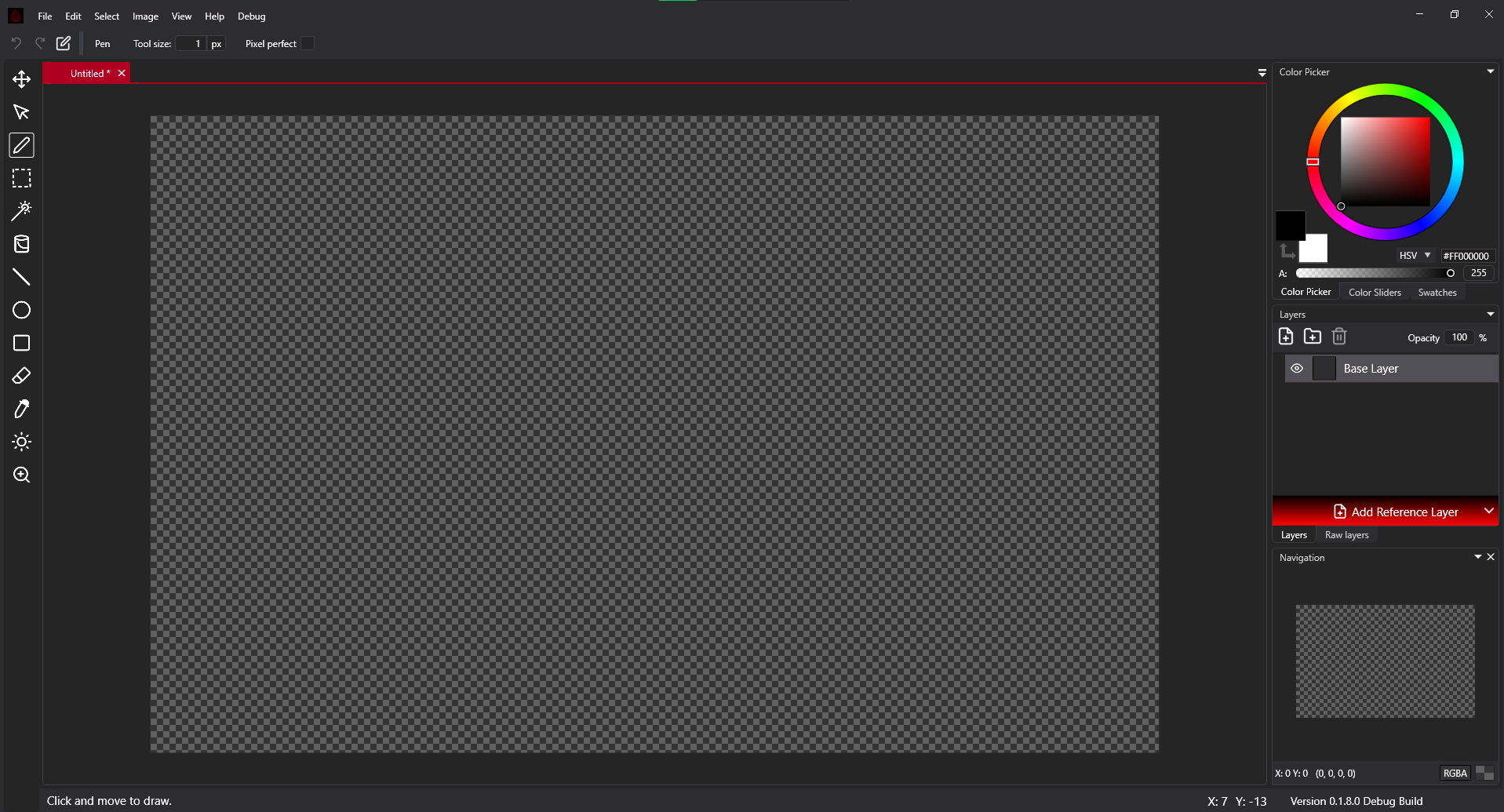


Рисунок 6.9 – Выбор размера изображения

Рисунок 6.10 – Создание файла

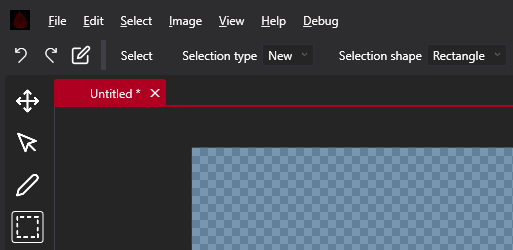


Рисунок 6.11 – Выбор инструмента Выделение области

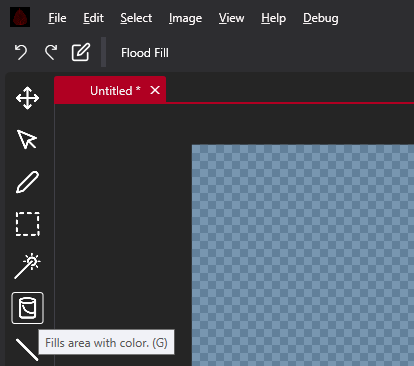


Рисунок 6.12 – Выбор инструмента Заливка

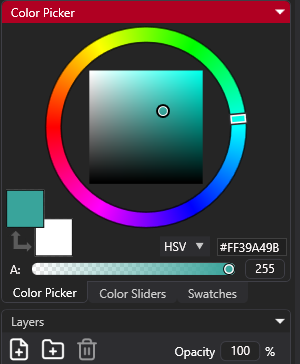


Рисунок 6.13 – Выбор цвета заливки

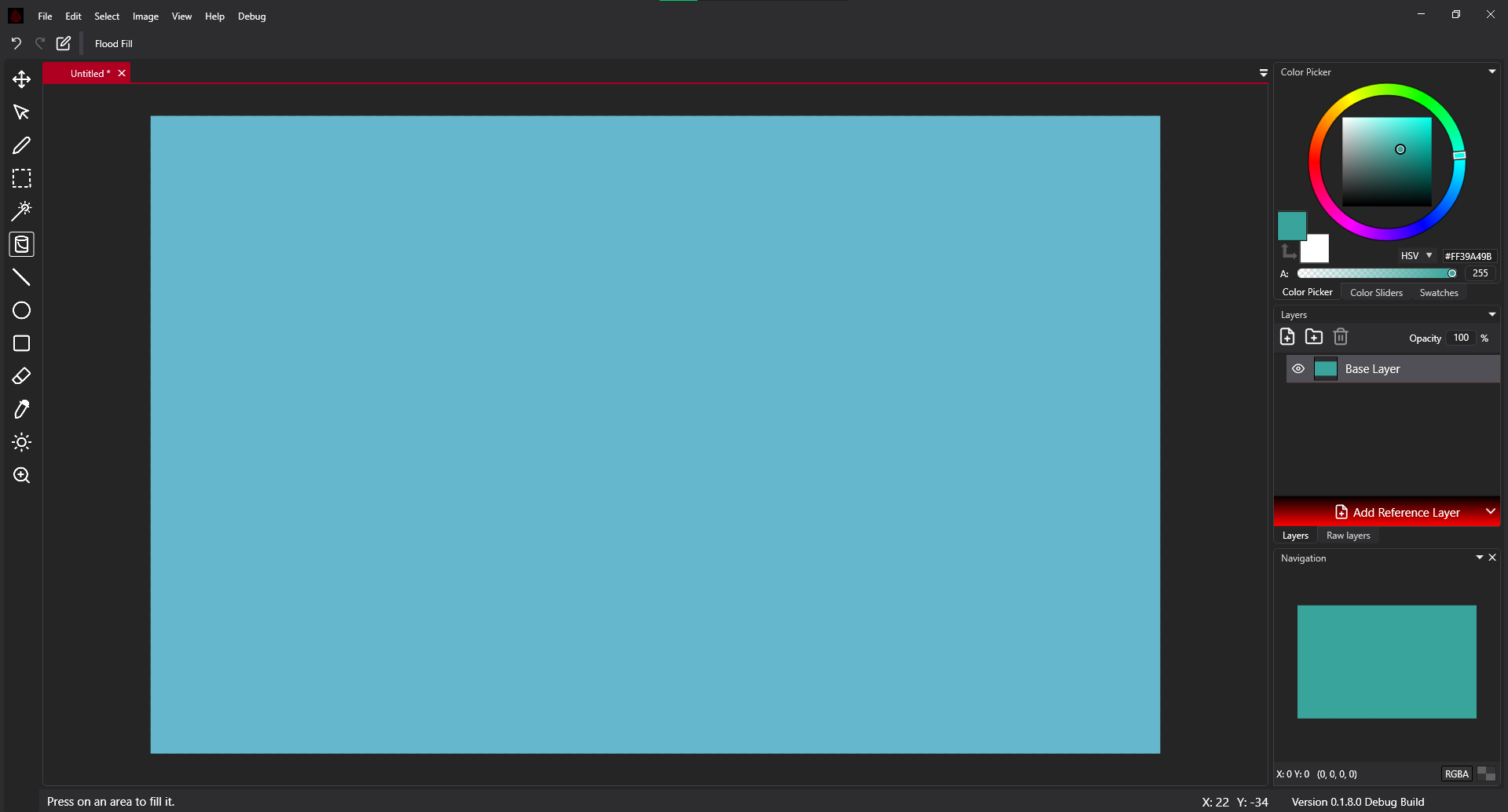
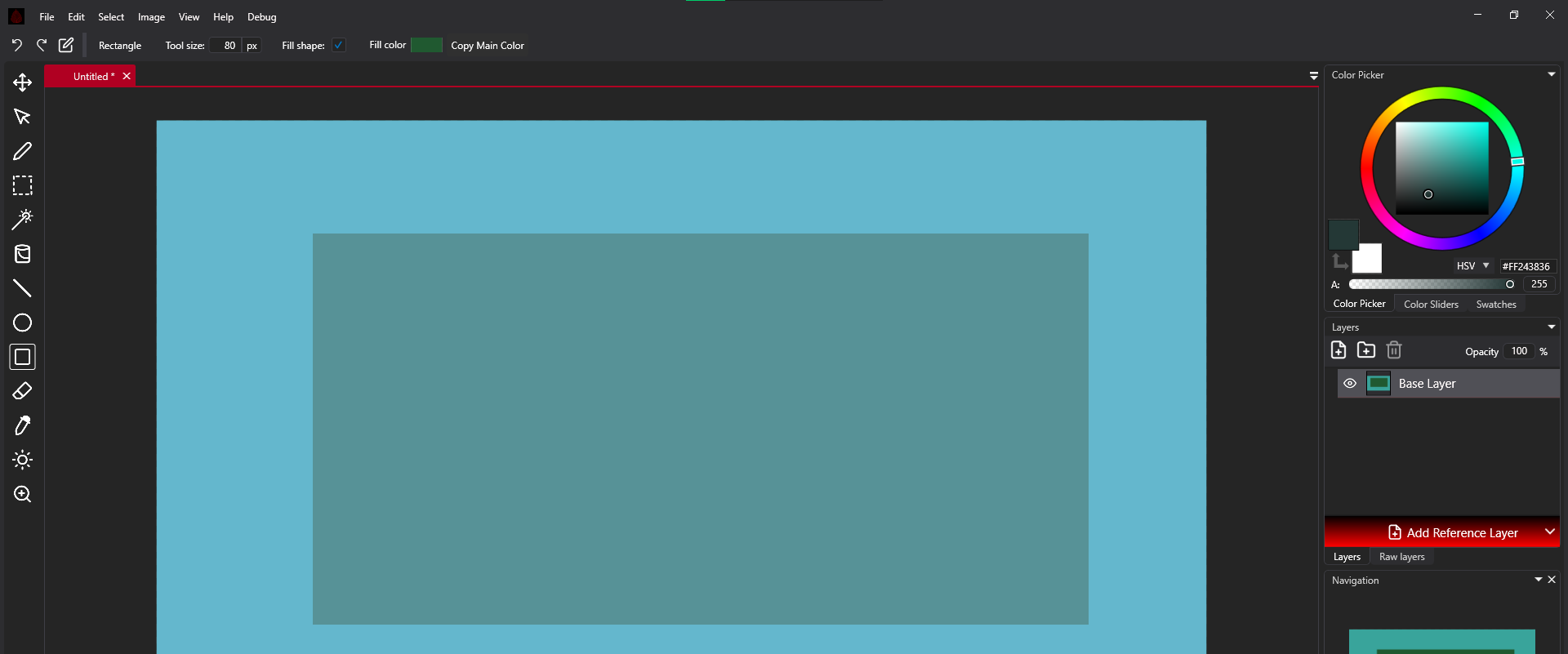
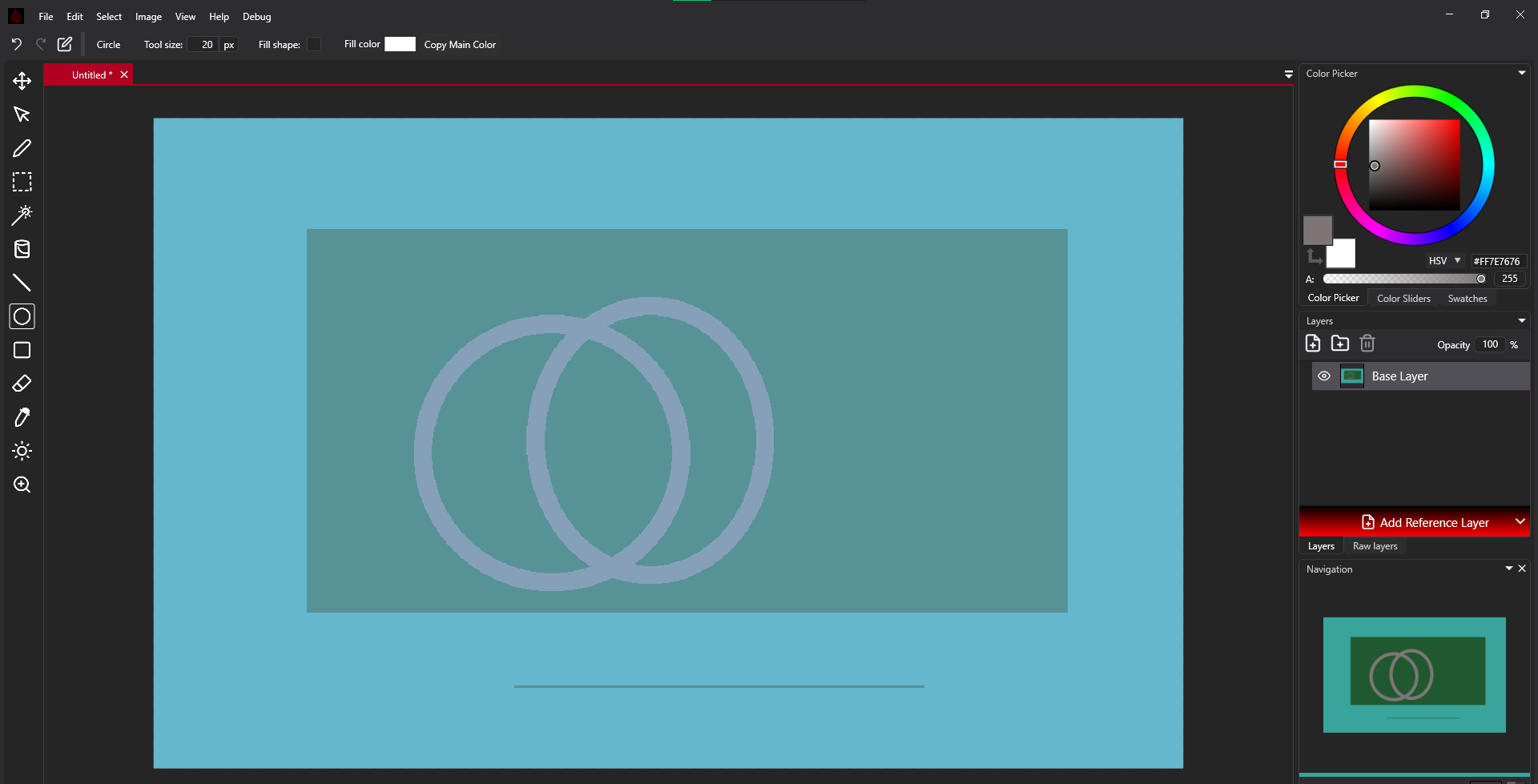
 Рисунок 6.14 – Заливка выделенной области

Рисунок 6.15 – Использование инструмента Квадрат



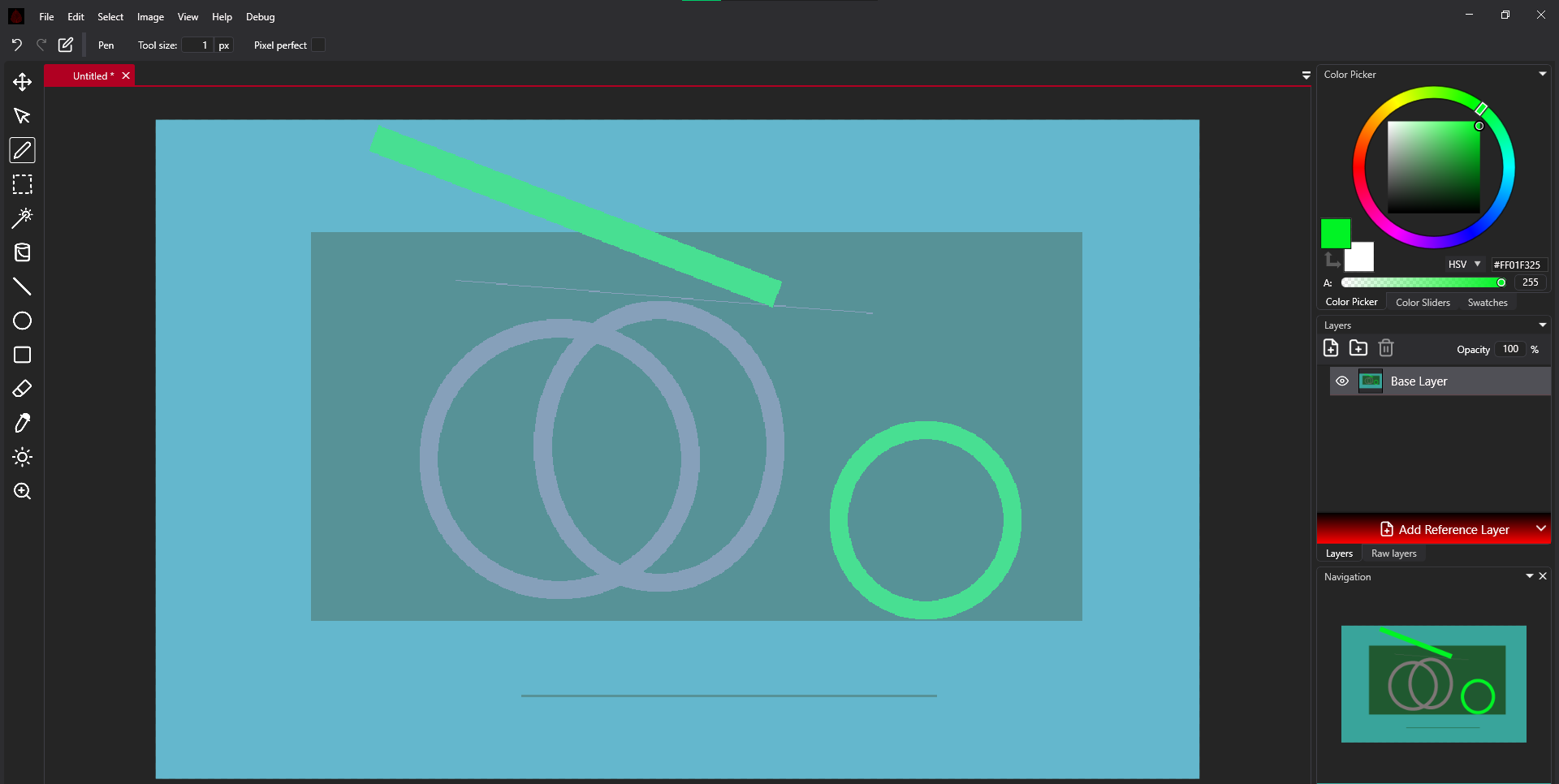
Рисунок 6.16 – Использование инструмента Круг

Рисунок 6.17 – Использование инструмента Пипетка

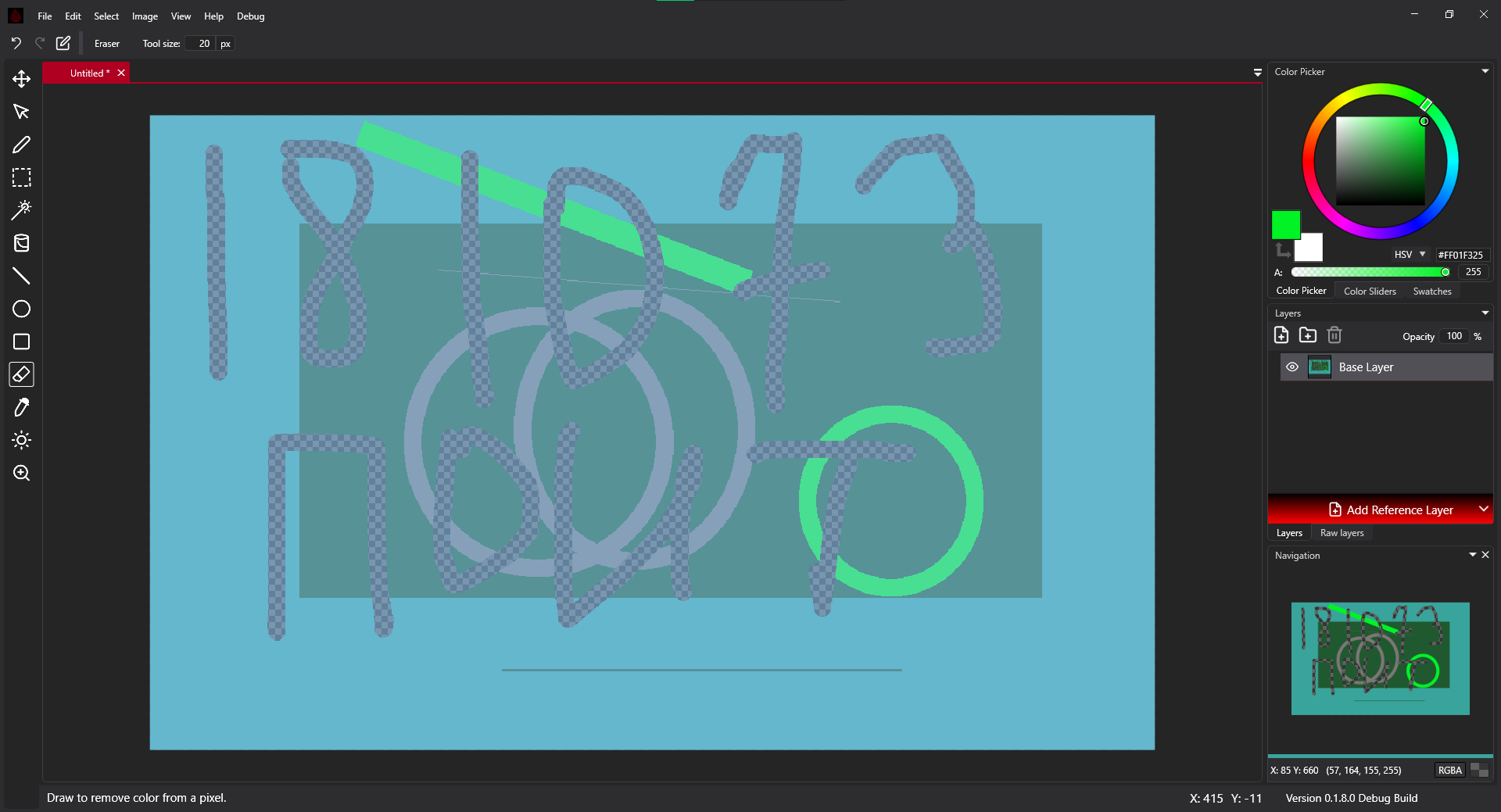
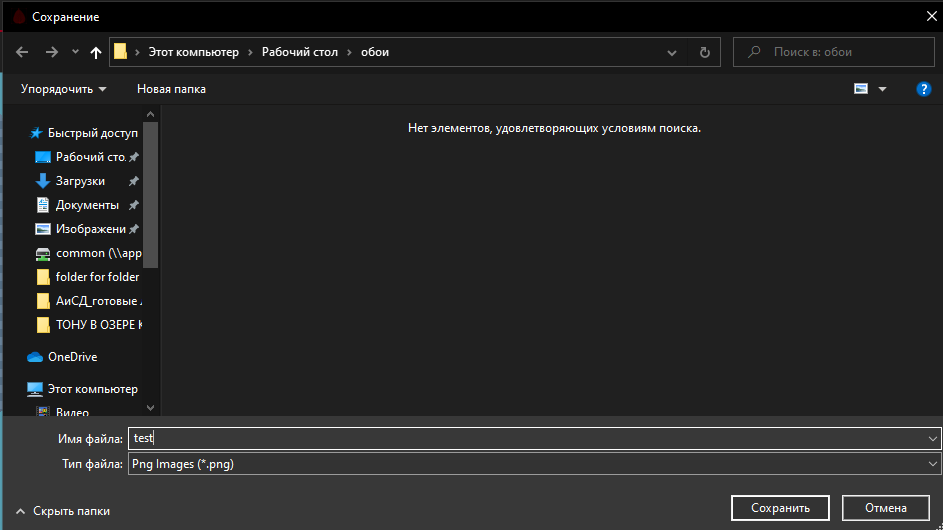
Рисунок 6.18 – Использование инструмента Ластик

Рисунок 6.19 – Сохранение изображения

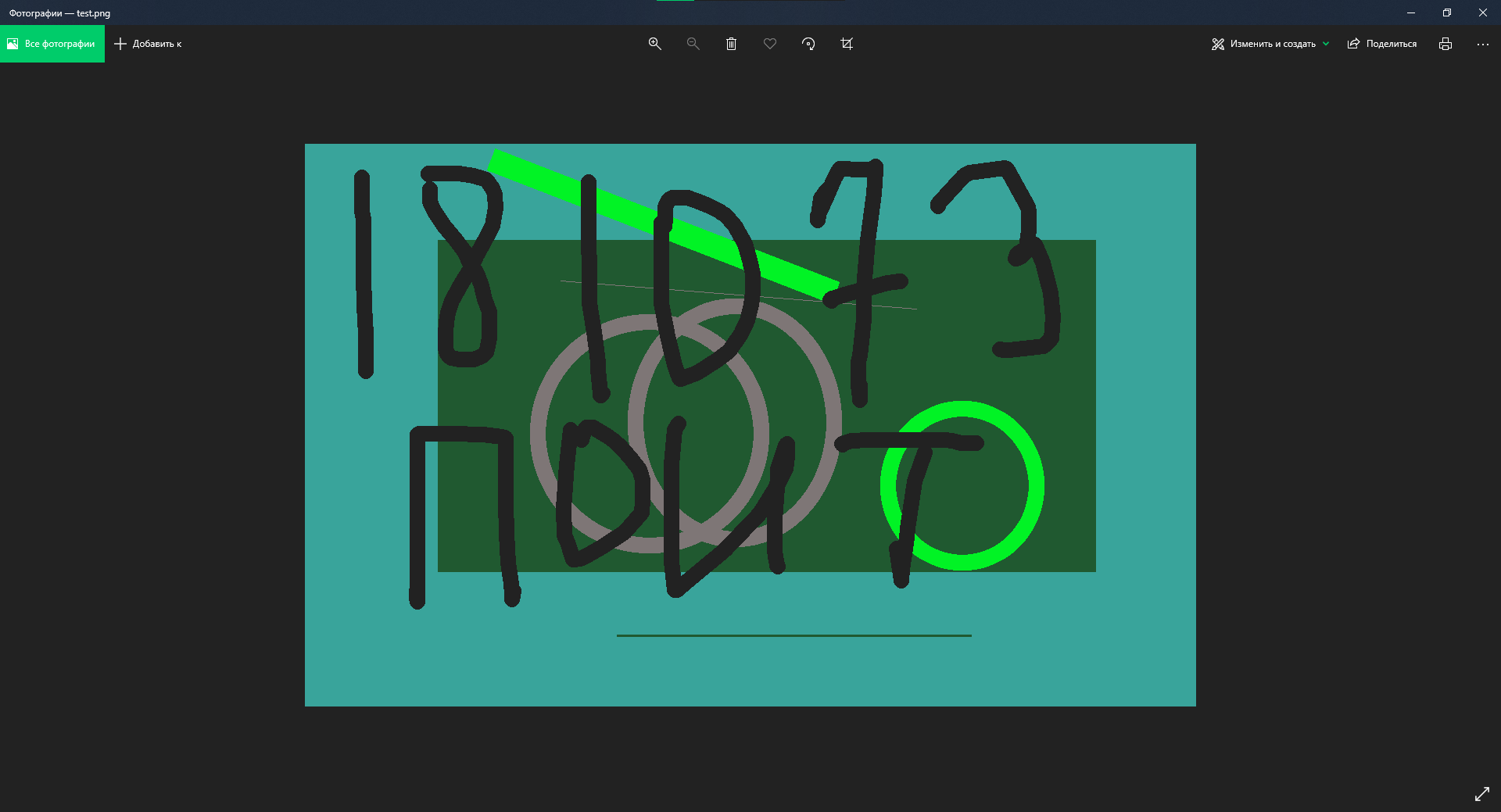


Рисунок 6.20 – Проверка файла

# ЗАКЛЮЧЕНИЕ

В ходе создания данного курсового проекта был разработан графический редактор, демонстрирующий потенциальные возможности языка С# и WPF.

При разработке программного продукта поставленная задача была выполнена полностью.

В работе:

* представлена информация о средствах и среде разработки;
* представлен алгоритм работы программы;
* приведен теоретический материал о компонентах,

используемых в программе.

Разработка курсового проекта расширила знания, необходимые для создания более сложных проектов. Были закреплены теоретические и практические знания, умение самостоятельно анализировать поставленную задачу и разбивать ее на просты подзадачи.

Программный продукт получился достаточно конкурентоспособным, благодаря перенятую лучших черт из рассматриваемых аналогов. Данный проект может развиваться дальше, так как в современном мире многим необходим простой и интуитивно понятный графический редактор, для выполнения несложных задач.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Кудрина, Е. В. Основы алгоритмизации и программирования на языке С#:учеб. пособие для СПО / Е. В. Кудрина, М. В. Огнева. — М. : Издательство Юрайт, 2019. — 322 с.
2. Кудрина, Е. В. Основы алгоритмизации и программирования на языке С#:учеб. пособие для бакалавриата и специалитета / Е. В. Кудрина, М. В. Огнева. — М. : Издательство Юрайт, 2019. — 322 с.
3. Metanit – Сайт о программировании [Электронный ресурс]. – Режим доступа: https://metanit.com – Дата доступа: 23.04.2022

# ПРИЛОЖЕНИЕ А

(обязательное)

Код программы

using PixiEditor.Models.DataHolders;

using System;

using System.Collections.Generic;

using System.Collections.ObjectModel;

using System.Linq;

namespace PixiEditor.Models.Layers

{

// Notice for further developemnt. Remember to expose only GroupData classes if you want to modify this LayerStructure groups

// LayerStructure should figure out the GuidStructureItem from its internal data. This will ensure that data will

// modify correctly.

// You should pass GuidStructureItem for operating on protected and private methods for faster data manipulation.

/// <summary>

/// Class containing layer groups structure and methods to operate on it.

/// </summary>

public class LayerStructure

{

public event EventHandler<LayerStructureChangedEventArgs> LayerStructureChanged;

public WpfObservableRangeCollection<GuidStructureItem> Groups { get; set; }

private Document Owner { get; }

/// <summary>

/// Checks whenever group contains only single layer and none subgroups.

/// </summary>

/// <param name="layerGuid">Guid of layer to check.</param>

/// <param name="layerGroup">Group to check.</param>

/// <returns>True if group contains single layer (EndLayerGuid and StartLayerGuid == layerGuid) and none subgroups.</returns>

public static bool GroupContainsOnlyLayer(Guid layerGuid, GuidStructureItem layerGroup)

{

return layerGroup != null && layerGroup.Subgroups.Count == 0 && layerGroup.StartLayerGuid == layerGuid && layerGroup.EndLayerGuid == layerGuid;

}

/// <summary>

/// Deep clones groups.

/// </summary>

/// <param name="groups">Groups to clone.</param>

/// <returns>ObservableCollection with cloned groups.</returns>

public static WpfObservableRangeCollection<GuidStructureItem> CloneGroups(WpfObservableRangeCollection<GuidStructureItem> groups)

{

WpfObservableRangeCollection<GuidStructureItem> outputGroups = new();

foreach (var group in groups.ToArray())

{

outputGroups.Add(group.CloneGroup());

}

return outputGroups;

}

/// <summary>

/// Finds parent group by layer guid.

/// </summary>

/// <param name="layerGuid">Guid of group to check.</param>

/// <returns><see cref="GuidStructureItem"/>if parent group was found or null if not.</returns>

public GuidStructureItem GetGroupByLayer(Guid layerGuid)

{

return GetGroupByLayer(layerGuid, Groups);

}

/// <summary>

/// Finds <see cref="GuidStructureItem"/> (Group) by it's guid.

/// </summary>

/// <param name="groupGuid">Guid of group.</param>

/// <returns><see cref="GuidStructureItem"/> if group was found or null if not.</returns>

public GuidStructureItem GetGroupByGuid(Guid? groupGuid)

{

return GetGroupByGuid(groupGuid, Groups);

}

public WpfObservableRangeCollection<GuidStructureItem> CloneGroups()

{

return CloneGroups(Groups);

}

// This will allow to add new group with multiple layers and groups at once. Not working well, todo fix

/\*public GuidStructureItem AddNewGroup(string groupName, IEnumerable<Layer> layers, Guid activeLayer)

{

var activeLayerParent = GetGroupByLayer(activeLayer);

List<GuidStructureItem> sameLevelGroups = new List<GuidStructureItem>();

var group = AddNewGroup(groupName, activeLayer);

if (activeLayerParent == null)

{

sameLevelGroups.AddRange(Groups);

}

else

{

sameLevelGroups.AddRange(activeLayerParent.Subgroups);

}

sameLevelGroups.Remove(group);

group.Subgroups = new ObservableCollection<GuidStructureItem>(sameLevelGroups);

sameLevelGroups = new(sameLevelGroups.Where(x => IsChildOf(activeLayer, x)));

Guid lastLayer = activeLayer;

foreach (var layer in layers)

{

if (layer.LayerGuid == activeLayer)

{

continue;

}

Owner.MoveLayerInStructure(layer.LayerGuid, lastLayer, false);

lastLayer = layer.LayerGuid;

}

return group;

}\*/

/// <summary>

/// Adds a new group to layer structure taking into consideration nesting. Invokes LayerStructureChanged event.

/// </summary>

/// <param name="groupName">Name of a group.</param>

/// <param name="childLayer">Child layer of a new group.</param>

/// <returns>Newly created group (<see cref="GuidStructureItem"/>).</returns>

public GuidStructureItem AddNewGroup(string groupName, Guid childLayer)

{

var parent = GetGroupByLayer(childLayer);

GuidStructureItem group = new(groupName, childLayer) { IsExpanded = true };

if (parent == null)

{

Groups.Add(group);

}

else

{

group.Parent = parent;

parent.Subgroups.Add(group);

}

group.GroupsChanged += Group\_GroupsChanged;

LayerStructureChanged?.Invoke(this, new LayerStructureChangedEventArgs(childLayer));

return group;

}

public GuidStructureItem AddNewGroup(string groupName, GuidStructureItem childGroup)

{

if (childGroup == null)

{

throw new ArgumentException("Child group can't be null.");

}

GuidStructureItem group = new(groupName, childGroup.StartLayerGuid, childGroup.EndLayerGuid, new[] { childGroup }, childGroup.Parent) { IsExpanded = true };

if (childGroup.Parent == null)

{

Groups.Add(group);

Groups.Remove(childGroup);

}

else

{

childGroup.Parent.Subgroups.Add(group);

childGroup.Parent.Subgroups.Remove(childGroup);

}

childGroup.Parent = group;

group.GroupsChanged += Group\_GroupsChanged;

LayerStructureChanged?.Invoke(this, new LayerStructureChangedEventArgs(GetGroupLayerGuids(group)));

return group;

}

/// <summary>

/// Moves group and it's children from one index to another. This method makes changes in <see cref="Document"/> Layers.

/// </summary>

/// <param name="groupGuid">Group guid to move.</param>

/// <param name="newIndex">New group index, relative to <see cref="Document"/> Layers.</param>

public void MoveGroup(Guid groupGuid, int newIndex)

{

var group = GetGroupByGuid(groupGuid);

var parentGroup = group.Parent;

bool reverseOrder = true;

int groupTopIndex = Owner.Layers.IndexOf(Owner.Layers.First(x => x.GuidValue == group.EndLayerGuid));

int groupBottomIndex = Owner.Layers.IndexOf(Owner.Layers.First(x => x.GuidValue == group.StartLayerGuid));

int difference = newIndex - groupTopIndex;

if (newIndex < groupTopIndex)

{

reverseOrder = false;

difference = newIndex - groupBottomIndex;

}

if (difference == 0)

{

return;

}

Unassign(parentGroup, group);

List<Guid> layersInOrder = GetLayersInOrder(new GroupData(groupTopIndex, groupBottomIndex));

MoveLayersInGroup(layersInOrder, difference, reverseOrder);

LayerStructureChanged?.Invoke(this, new LayerStructureChangedEventArgs(layersInOrder));

}

/// <summary>

/// Checks if group is nested inside parent group.

/// </summary>

/// <param name="group">Group to check.</param>

/// <param name="parent">Parent of that group.</param>

/// <returns>True if group is nested inside parent, false if not.</returns>

public bool IsChildOf(GuidStructureItem group, GuidStructureItem parent)

{

if (group == null)

{

return false;

}

foreach (var subgroup in parent.Subgroups)

{

if (subgroup == group)

{

return true;

}

if (subgroup.Subgroups.Count > 0)

{

if (IsChildOf(group, subgroup))

{

return true;

}

}

}

return false;

}

/// <summary>

/// Checks if layer is nested inside parent group.

/// </summary>

/// <param name="layerGuid">Layer GUID to check.</param>

/// <param name="parent">Parent of that group.</param>

/// <returns>True if layer is nested inside parent, false if not.</returns>

public bool IsChildOf(Guid layerGuid, GuidStructureItem parent)

{

var layerParent = GetGroupByLayer(layerGuid);

return layerParent == parent ? true : IsChildOf(layerParent, parent);

}

/// <summary>

/// Reassigns (removes) group data from parent group.

/// </summary>

/// <param name="parentGroup">Parent group to reassign data in.</param>

/// <param name="group">Group which data should be reassigned.</param>

public void Unassign(GroupData parentGroup, GroupData group)

{

Unassign(GetGroupByGuid(parentGroup.GroupGuid), GetGroupByGuid(group.GroupGuid));

}

/// <summary>

/// Reassigns (removes) layer data from parent group.

/// </summary>

/// <param name="parentGroup">Parent group to reassign data in.</param>

/// <param name="layer">Layer which data should be reassigned.</param>

public void Unassign(GroupData parentGroup, Guid layer)

{

PreMoveReassignBounds(GetGroupByGuid(parentGroup.GroupGuid), layer);

}

/// <summary>

/// Reassigns (adds) layer data to parent group.

/// </summary>

/// <param name="parentGroup">Parent group to reassign data in.</param>

/// <param name="layerGuid">Group which data should be reassigned.</param>

public void Assign(GroupData parentGroup, Guid layerGuid)

{

Assign(GetGroupByGuid(parentGroup.GroupGuid), layerGuid);

}

/// <summary>

/// Reassigns (adds) group data to parent group.

/// </summary>

/// <param name="parentGroup">Parent group to reassign data in.</param>

/// <param name="group">Group which data should be reassigned.</param>

public void Assign(GroupData parentGroup, GroupData group)

{

Assign(GetGroupByGuid(parentGroup?.GroupGuid), GetGroupByGuid(group.GroupGuid));

}

/// <summary>

/// Assigns parent to a layer.

/// </summary>

/// <param name="layer">Layer to assign parent to.</param>

/// <param name="parent">Parent which should be assigned. Null indicates no parent.</param>

public void AssignParent(Guid layer, Guid? parent)

{

AssignParent(layer, parent.HasValue ? GetGroupByGuid(parent) : null);

}

/// <summary>

/// Assigns group new parent.

/// </summary>

/// <param name="group">Group to assign parent</param>

/// <param name="referenceLayerGroup">Parent of group.</param>

public void ReassignParent(GuidStructureItem group, GuidStructureItem referenceLayerGroup)

{

group.Parent?.Subgroups.Remove(group);

if (Groups.Contains(group))

{

Groups.Remove(group);

}

if (referenceLayerGroup == null)

{

if (!Groups.Contains(group))

{

Groups.Add(group);

group.Parent = null;

}

}

else

{

referenceLayerGroup.Subgroups.Add(group);

group.Parent = referenceLayerGroup;

}

LayerStructureChanged?.Invoke(this, new LayerStructureChangedEventArgs(GetGroupLayerGuids(group)));

}

/// <summary>

/// Gets all layers inside group, including nested groups.

/// </summary>

/// <param name="group">Group to get layers from.</param>

/// <returns>List of layers.</returns>

public List<Layer> GetGroupLayers(GuidStructureItem group)

{

List<Layer> layers = new();

var layerGuids = GetGroupLayerGuids(group);

foreach (var layerGuid in layerGuids)

{

layers.Add(Owner.Layers.First(x => x.GuidValue == layerGuid));

}

return layers;

}

/// <summary>

/// Sets parent groups IsExpanded to true.

/// </summary>

/// <param name="layerGuid">Guid of layer which parents will be affected.</param>

public void ExpandParentGroups(Guid layerGuid)

{

var group = GetGroupByLayer(layerGuid);

while (group != null)

{

group.IsExpanded = true;

group = group.Parent;

}

}

/// <summary>

/// Sets parent groups IsExpanded to true.

/// </summary>

/// <param name="group">Group which parents will be affected.</param>

public void ExpandParentGroups(GuidStructureItem group)

{

GuidStructureItem currentGroup = group;

while (currentGroup != null)

{

currentGroup.IsExpanded = true;

currentGroup = currentGroup.Parent;

}

}

private static Guid GetNextLayerGuid(Guid layer, List<Guid> allLayers, bool above)

{

int indexOfLayer = allLayers.IndexOf(layer);

int modifier = above ? 1 : -1;

int newIndex = Math.Clamp(indexOfLayer + modifier, 0, allLayers.Count - 1);

return allLayers[newIndex];

}

/// <summary>

/// Gets all layers inside group, including nested groups.

/// </summary>

/// <param name="group">Group to get layers from.</param>

/// <returns>List of layer guids.</returns>

private List<Guid> GetGroupLayerGuids(GuidStructureItem group)

{

Layer layerTop = Owner.Layers.FirstOrDefault(x => x.GuidValue == group.EndLayerGuid);

Layer layerBottom = Owner.Layers.FirstOrDefault(x => x.GuidValue == group.StartLayerGuid);

if (layerTop == null || layerBottom == null)

{

return new List<Guid>();

}

int indexTop = Owner.Layers.IndexOf(layerTop);

int indexBottom = Owner.Layers.IndexOf(layerBottom);

return GetLayersInOrder(new GroupData(indexTop, indexBottom));

}

private List<Guid> GetLayersInOrder(GroupData group)

{

List<Guid> layerGuids = new();

int minIndex = group.BottomIndex;

int maxIndex = group.TopIndex;

for (int i = minIndex; i <= maxIndex; i++)

{

layerGuids.Add(Owner.Layers[i].GuidValue);

}

return layerGuids;

}

private void PreMoveReassignBounds(GuidStructureItem parentGroup, Guid layer)

{

if (parentGroup != null)

{

Guid oldStart = parentGroup.StartLayerGuid;

Guid oldEnd = parentGroup.EndLayerGuid;

GuidStructureItem parentOfParent = parentGroup.Parent;

if (parentGroup.Subgroups.Count == 0 && parentGroup.StartLayerGuid == layer && parentGroup.EndLayerGuid == layer)

{

RemoveGroup(parentGroup);

}

else

{

if (parentGroup.EndLayerGuid == layer)

{

parentGroup.EndLayerGuid = FindBoundLayer(parentGroup, layer, false);

}

if (parentGroup.StartLayerGuid == layer)

{

parentGroup.StartLayerGuid = FindBoundLayer(parentGroup, layer, true);

}

}

if (parentOfParent != null)

{

ApplyBoundsToParents(parentOfParent, parentGroup, oldStart, oldEnd);

}

LayerStructureChanged?.Invoke(this, new LayerStructureChangedEventArgs(layer));

}

}

private void Unassign(GuidStructureItem parentGroup, GuidStructureItem group)

{

if (parentGroup != null)

{

Guid oldStart = parentGroup.StartLayerGuid;

Guid oldEnd = parentGroup.EndLayerGuid;

if (parentGroup.Subgroups.Count == 1 && parentGroup.StartLayerGuid == group.StartLayerGuid && parentGroup.EndLayerGuid == group.EndLayerGuid)

{

RemoveGroup(parentGroup);

}

else

{

if (group.EndLayerGuid == parentGroup.EndLayerGuid)

{

parentGroup.EndLayerGuid = FindBoundLayer(parentGroup, group.StartLayerGuid, false);

}

if (group.StartLayerGuid == parentGroup.StartLayerGuid)

{

parentGroup.StartLayerGuid = FindBoundLayer(parentGroup, group.EndLayerGuid, true);

}

}

if (parentGroup.Parent != null)

{

ApplyBoundsToParents(parentGroup.Parent, parentGroup, oldStart, oldEnd);

}

}

}

private void Assign(GuidStructureItem parentGroup, Guid layerGuid)

{

if (parentGroup != null)

{

Guid? oldStart = parentGroup.StartLayerGuid;

Guid? oldEnd = parentGroup.EndLayerGuid;

int layerIndex = Owner.Layers.IndexOf(Owner.Layers.First(x => x.GuidValue == layerGuid));

int folderTopIndex = Owner.Layers.IndexOf(Owner.Layers.First(x => x.GuidValue == parentGroup.EndLayerGuid));

int folderBottomIndex = Owner.Layers.IndexOf(Owner.Layers.First(x => x.GuidValue == parentGroup.StartLayerGuid));

int finalTopIndex = Math.Max(folderTopIndex, layerIndex);

int finalBottomIndex = Math.Min(folderBottomIndex, layerIndex);

Guid? topBoundLayer = FindBoundLayer(layerGuid, finalTopIndex, finalBottomIndex, false);

Guid? bottomBoundLayer = FindBoundLayer(layerGuid, finalTopIndex, finalBottomIndex, true);

if (topBoundLayer == parentGroup.EndLayerGuid)

{

parentGroup.EndLayerGuid = layerGuid;

}

if (bottomBoundLayer == parentGroup.StartLayerGuid)

{

parentGroup.StartLayerGuid = layerGuid;

}

if (parentGroup.Parent != null)

{

ApplyBoundsToParents(parentGroup.Parent, parentGroup, oldStart, oldEnd);

}

var args = new LayerStructureChangedEventArgs(layerGuid);

if (topBoundLayer.HasValue)

{

args.AffectedLayerGuids.Add(topBoundLayer.Value);

}

if (bottomBoundLayer.HasValue)

{

args.AffectedLayerGuids.Add(bottomBoundLayer.Value);

}

LayerStructureChanged?.Invoke(this, args);

}

}

private void Assign(GuidStructureItem parentGroup, GuidStructureItem group)

{

if (parentGroup != null)

{

Guid oldStart = parentGroup.StartLayerGuid;

Guid oldEnd = parentGroup.EndLayerGuid;

int folderTopIndex = Owner.Layers.IndexOf(Owner.Layers.First(x => x.GuidValue == group.EndLayerGuid));

int folderBottomIndex = Owner.Layers.IndexOf(Owner.Layers.First(x => x.GuidValue == group.StartLayerGuid));

int parentFolderTopIndex = Owner.Layers.IndexOf(Owner.Layers.First(x => x.GuidValue == parentGroup.EndLayerGuid));

int parentFolderBottomIndex = Owner.Layers.IndexOf(Owner.Layers.First(x => x.GuidValue == parentGroup.StartLayerGuid));

int finalTopIndex = Math.Max(folderTopIndex, parentFolderTopIndex);

int finalBottomIndex = Math.Min(folderBottomIndex, parentFolderBottomIndex);

Guid topBoundLayer = FindBoundLayer(group.StartLayerGuid, finalTopIndex, finalBottomIndex, false);

Guid bottomBoundLayer = FindBoundLayer(group.EndLayerGuid, finalTopIndex, finalBottomIndex, true);

if (topBoundLayer == parentGroup.EndLayerGuid)

{

parentGroup.EndLayerGuid = group.EndLayerGuid;

}

if (bottomBoundLayer == parentGroup.StartLayerGuid)

{

parentGroup.StartLayerGuid = group.StartLayerGuid;

}

if (parentGroup.Parent != null)

{

ApplyBoundsToParents(parentGroup.Parent, parentGroup, oldStart, oldEnd);

}

}

}

private void AssignParent(Guid layer, GuidStructureItem parent)

{

var currentParent = GetGroupByLayer(layer);

if (currentParent != null)

{

PreMoveReassignBounds(currentParent, layer);

}

Assign(parent, layer);

LayerStructureChanged?.Invoke(this, new LayerStructureChangedEventArgs(layer));

}

private void Group\_GroupsChanged(object sender, GroupChangedEventArgs e)

{

List<Guid> layersAffected = new List<Guid>();

e.GroupsAffected.ForEach(x => layersAffected.AddRange(GetGroupLayerGuids(x)));

LayerStructureChanged?.Invoke(this, new LayerStructureChangedEventArgs(layersAffected));

}

private void RemoveGroup(GuidStructureItem parentFolder)

{

parentFolder.GroupsChanged -= Group\_GroupsChanged;

var layerGuids = GetGroupLayerGuids(parentFolder);

if (parentFolder.Parent == null)

{

Groups.Remove(parentFolder);

}

else

{

parentFolder.Parent.Subgroups.Remove(parentFolder);

}

LayerStructureChanged?.Invoke(this, new LayerStructureChangedEventArgs(layerGuids));

}

private void ApplyBoundsToParents(GuidStructureItem parent, GuidStructureItem group, Guid? oldStart, Guid? oldEnd)

{

Guid parentOldStart = parent.StartLayerGuid;

Guid parentOldEnd = parent.EndLayerGuid;

if (parent.Subgroups.Count == 0)

{

RemoveGroup(parent);

}

if (parent.StartLayerGuid == oldStart)

{

parent.StartLayerGuid = group.StartLayerGuid;

}

if (parent.EndLayerGuid == oldEnd)

{

parent.EndLayerGuid = group.EndLayerGuid;

}

if (parent.Parent != null)

{

ApplyBoundsToParents(parent.Parent, parent, parentOldStart, parentOldEnd);

}

}

private Guid FindBoundLayer(Guid layerGuid, int parentFolderTopIndex, int parentFolderBottomIndex, bool above)

{

return GetNextLayerGuid(

layerGuid,

GetLayersInOrder(new GroupData(parentFolderTopIndex, parentFolderBottomIndex)),

above);

}

private Guid FindBoundLayer(GuidStructureItem parentFolder, Guid layerGuid, bool above)

{

int parentFolderTopIndex = Owner.Layers.IndexOf(Owner.Layers.First(x => x.GuidValue == parentFolder.EndLayerGuid));

int parentFolderBottomIndex = Owner.Layers.IndexOf(Owner.Layers.First(x => x.GuidValue == parentFolder.StartLayerGuid));

return FindBoundLayer(layerGuid, parentFolderTopIndex, parentFolderBottomIndex, above);

}

private void MoveLayersInGroup(List<Guid> layers, int moveBy, bool reverseOrder)

{

List<Guid> layerGuids = reverseOrder ? layers.Reverse<Guid>().ToList() : layers;

for (int i = 0; i < layers.Count; i++)

{

Guid layerGuid = layerGuids[i];

var layer = Owner.Layers.First(x => x.GuidValue == layerGuid);

int layerIndex = Owner.Layers.IndexOf(layer);

Owner.Layers.Move(layerIndex, layerIndex + moveBy);

}

}

private GuidStructureItem GetGroupByLayer(Guid layerGuid, IEnumerable<GuidStructureItem> groups)

{

foreach (var currentGroup in groups)

{

var endLayer = Owner.Layers.First(x => x.GuidValue == currentGroup.EndLayerGuid);

var startLayer = Owner.Layers.First(x => x.GuidValue == currentGroup.StartLayerGuid);

int topIndex = Owner.Layers.IndexOf(endLayer);

int bottomIndex = Owner.Layers.IndexOf(startLayer);

var layers = GetLayersInOrder(new GroupData(topIndex, bottomIndex));

if (currentGroup.Subgroups.Count > 0)

{

var group = GetGroupByLayer(layerGuid, currentGroup.Subgroups);

if (group != null)

{

return group;

}

}

if (layers.Contains(layerGuid))

{

return currentGroup;

}

}

return null;

}

private GuidStructureItem GetGroupByGuid(Guid? groupGuid, IEnumerable<GuidStructureItem> groups)

{

foreach (var group in groups)

{

if (group.GroupGuid == groupGuid)

{

return group;

}

if (group.Subgroups.Count > 0)

{

var guid = GetGroupByGuid(groupGuid, group.Subgroups);

if (guid != null)

{

return guid;

}

}

}

return null;

}

public LayerStructure(WpfObservableRangeCollection<GuidStructureItem> items, Document owner)

{

Groups = items;

Owner = owner;

}

public LayerStructure(Document owner)

{

Groups = new WpfObservableRangeCollection<GuidStructureItem>();

Owner = owner;

}

}

}