

Contents

Лабораторная работа 2	2
Способы и средства хранения и обработки графических данных	2
Вариант ИВ1: разработка растрового редактора	2
1. Цель и практический результат	2
2. Соответствие варианту ИВ1 и замечание по структуре контейнера	3
3. Выполнение основных требований ИВ1	3
3.1 Создание, загрузка и сохранение контейнера	3
3.2 Редактирование единичных пикселей	3
3.3 Непрерывная отрисовка “кистью”	4
3.4 Закраска области (“заливка”)	4
3.5 Выделение, копирование/вырезание и вставка фрагмента	4
4. Структура контейнера и пикселя	4
4.1 Структура контейнера .minint	4
4.2 Структура пикселя	4
5. Основные алгоритмы	4
6. UML-диаграммы (PlantUML)	5
6.1 Основной рабочий цикл редактора	5
6.2 Формат контейнера и сериализация	6
6.3 Инструменты рисования и заливки	7
6.4 Выделение и буфер обмена фрагментов	8
7. Проверка работоспособности	9
8. Вывод	10
Приложение А. Исходные тексты	10
A.1. Minint.Core/Models/MinintContainer.cs	10
A.2. Minint.Core/Models/MinintDocument.cs	10
A.3. Minint.Core/Models/MinintLayer.cs	12
A.4. Minint.Core/Models/RgbaColor.cs	13
A.5. Minint.Core/Services/IBmpExporter.cs	14
A.6. Minint.Core/Services/ICompositor.cs	14
A.7. Minint.Core/Services/IDrawingService.cs	14
A.8. Minint.Core/Services/IFloodFillService.cs	15
A.9. Minint.Core/Services/IFragmentService.cs	15
A.10. Minint.Core/Services/IGifExporter.cs	15
A.11. Minint.Core/Services/IImageEffectService.cs	16
A.12. Minint.Core/Services/IImageEffectsService.cs	16
A.13. Minint.Core/Services/IMinintSerializer.cs	16
A.14. Minint.Core/Services/IPaletteService.cs	17
A.15. Minint.Core/Services/IPatternGenerator.cs	17
A.16. Minint.Core/Services/Impl/Compositor.cs	18
A.17. Minint.Core/Services/Impl/DrawingService.cs	19
A.18. Minint.Core/Services/Impl/FloodFillService.cs	19
A.19. Minint.Core/Services/Impl/FragmentService.cs	20
A.20. Minint.Core/Services/Impl/ImageEffectsService.cs	21
A.21. Minint.Core/Services/Impl/PaletteService.cs	22
A.22. Minint.Core/Services/Impl/PatternGenerator.cs	23
A.23. Minint.Infrastructure/Export/BmpExporter.cs	24
A.24. Minint.Infrastructure/Export/GifExporter.cs	26
A.25. Minint.Infrastructure/Serialization/MinintSerializer.cs	31
A.26. Minint.Tests/CompositorTests.cs	35
A.27. Minint.Tests/DrawingTests.cs	36
A.28. Minint.Tests/ExportTests.cs	37
A.29. Minint.Tests/FloodFillTests.cs	38
A.30. Minint.Tests/FragmentServiceTests.cs	39
A.31. Minint.Tests/ImageEffectsTests.cs	40
A.32. Minint.Tests/PatternGeneratorTests.cs	41
A.33. Minint.Tests/SerializerTests.cs	42
A.34. Minint/App.axaml.cs	44

A.35. Minint/Controls/EditableTextBlock.cs	45
A.36. Minint/Controls/PixelCanvas.cs	47
A.37. Minint/Controls/Viewport.cs	55
A.38. Minint/Program.cs	57
A.39. Minint/ViewLocator.cs	61
A.40. Minint/ViewModels/EditorViewModel.cs	62
A.41. Minint/ViewModels/MainWindowViewModel.cs	73
A.42. Minint/ViewModels/ToolType.cs	78
A.43. Minint/ViewModels/ToolTypeConverters.cs	78
A.44. Minint/ViewModels/ViewModelBase.cs	78
A.45. Minint/Views/ContrastDialog.axaml.cs	79
A.46. Minint/Views/MainWindow.axaml.cs	79
A.47. Minint/Views/NewContainerDialog.axaml.cs	80
A.48. Minint/Views/PatternDialog.axaml.cs	80

Лабораторная работа 2

Способы и средства хранения и обработки графических данных

Вариант ИВ1: разработка растрового редактора

1. Цель и практический результат

Цель работы - разработать растровый редактор, выполняющий создание, загрузку, редактирование и сохранение графического контейнера с пиксельными данными, а также реализовать базовые инструменты рисования и редактирования фрагментов.

Практический результат:

- разработано настольное приложение Minint на C# + Avalonia;
- реализован собственный бинарный формат контейнера .minint с чтением/записью;
- реализованы инструменты Brush, Eraser, Fill, Select, Copy/Cut/Paste;
- подготовлена документация с UML-диаграммами и приложением исходного кода.

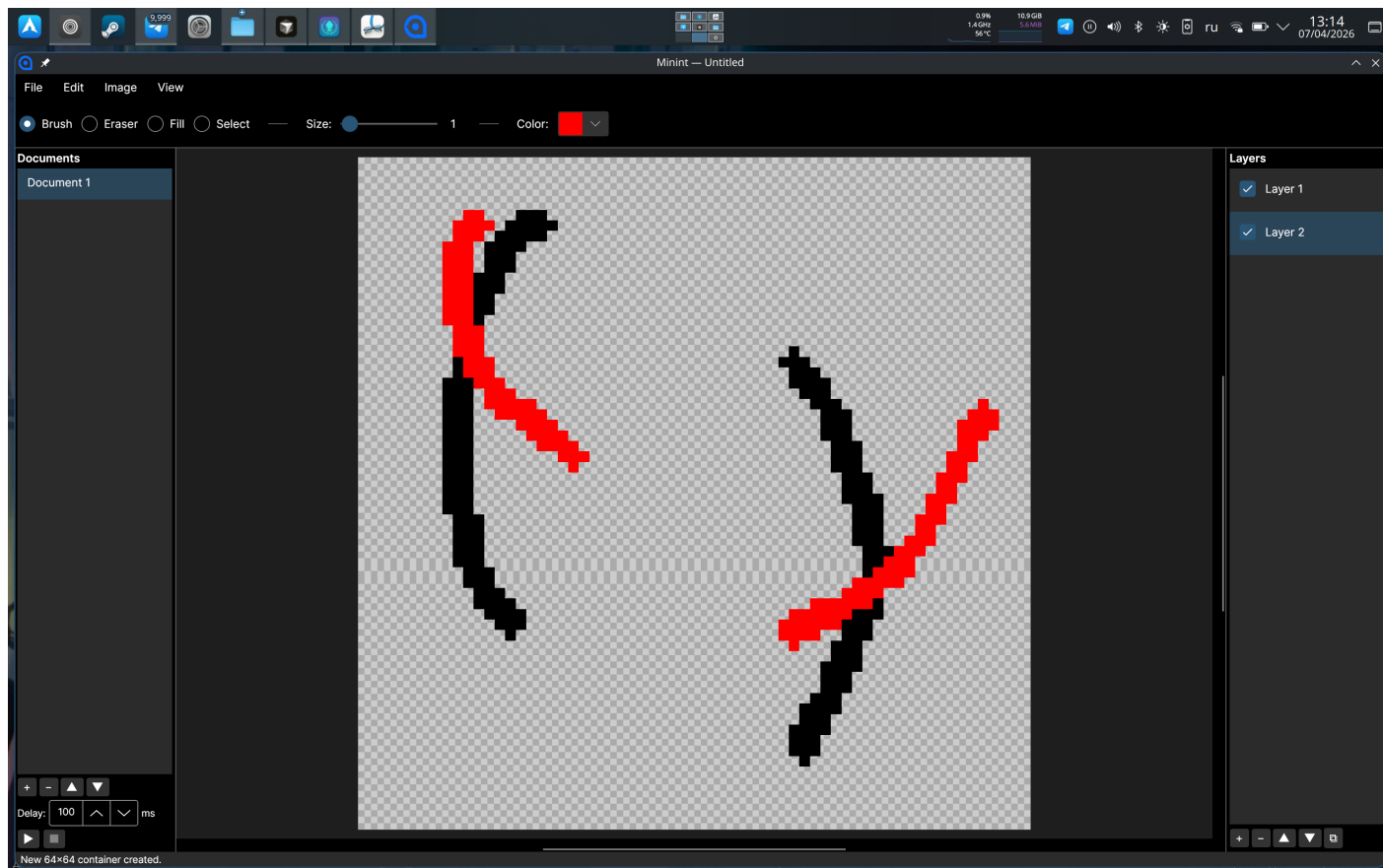


Figure 1: Скриншот ПО

2. Соответствие варианту ИВ1 и замечание по структуре контейнера

По методическим указаниям ИВ1 требуется использовать структуру пикселя и контейнера из одного из вариантов KB1–KB4.

В текущем проекте фактически реализован палитровый контейнер с RGBA-палитрой и индексами пикселей (MinintContainer/MinintDocument/MinintLayer), что не совпадает буквально с описаниями KB1–KB4, но полностью закрывает функциональные требования ИВ1 (создание, редактирование, загрузка, сохранение, инструменты рисования, работа с фрагментами).

Это ограничение фиксируется в отчёте явно, чтобы не было расхождения между кодом и документацией.

3. Выполнение основных требований ИВ1

3.1 Создание, загрузка и сохранение контейнера

- создание нового контейнера выполняется через `EditorViewModel.NewContainer(...)`;
- загрузка/сохранение выполняется через `MinintSerializer` (собственная реализация чтения/записи);
- контейнер хранит общие размеры, набор документов (кадров), палитры и слои.

Реализация: `Minint/ViewModels/EditorViewModel.cs`, `Minint.Infrastructure/Serialization/MinintSerializer`, `Minint.Core/Models/*`.

3.2 Редактирование единичных пикселей

- инструмент `Brush` изменяет значения пикселей маской радиуса;
- инструмент `Eraser` записывает индекс прозрачного цвета (0);
- выбор цвета выполняется через текущий `SelectedColor` и палитру документа.

Реализация: Minint.Core/Services/Impl/DrawingService.cs, Minint/ViewModels/EditorViewModel.cs.

3.3 Непрерывная отрисовка “кистью”

- при перемещении мыши по зажатой кнопке вызывается последовательная обработка точек;
- маска кисти вычисляется как круг по радиусу;
- поддерживается визуальный preview маски инструмента.

Реализация: Minint/Controls/PixelCanvas.cs, Minint/Core/Services/Impl/DrawingService.cs, Minint/ViewModels/EditorViewModel.cs.

3.4 Закраска области (“заливка”)

- реализован алгоритм flood fill (4-связность);
- заливка ограничена областью одинакового исходного индекса;
- алгоритм работает в границах изображения.

Реализация: Minint.Core/Services/Impl/FloodFillService.cs.

3.5 Выделение, копирование/вырезание и вставка фрагмента

- реализована рамка выделения (SelectionRect);
- данные буфера обмена хранятся в palette-independent виде (ClipboardFragment, RGBA);
- вставка поддерживает предпросмотр и подтверждение позиции;
- прозрачные пиксели фрагмента при вставке пропускаются.

Реализация: Minint/ViewModels/EditorViewModel.cs, Minint/Controls/PixelCanvas.cs, Minint.Core/Services

4. Структура контейнера и пикселя

4.1 Структура контейнера .minint

Контейнер состоит из:

1. Заголовка файла:
 - сигнатура MININT;
 - версия формата;
 - ширина и высота;
 - количество документов;
 - резерв.
2. Набора документов:
 - имя документа;
 - задержка кадра;
 - палитра RGBA;
 - набор слоёв.
3. Набора слоёв:
 - имя слоя;
 - признак видимости;
 - непрозрачность;
 - массив индексов пикселей.

4.2 Структура пикселя

Логически пиксель хранится как индекс в палитре документа (int в ОЗУ, переменная ширина 1..4 байта в файле), а итоговый цвет формируется по таблице RgbColor.

5. Основные алгоритмы

1. Запись контейнера в бинарный поток (WriteHeader, WriteDocument, WriteLayer).
2. Чтение и валидация контейнера (ReadHeader, ReadDocument, ReadLayer).
3. Круглая кисть по маске радиуса.

4. Очистка пикселей ластиком.
5. Flood fill по очереди.
6. Копирование/вставка фрагмента с отсечением по границам.
7. Композиция слоёв при обновлении холста.

6. UML-диаграммы (PlantUML)

6.1 Основной рабочий цикл редактора

Report/lab2/uml/lr2-editor-workflow.puml

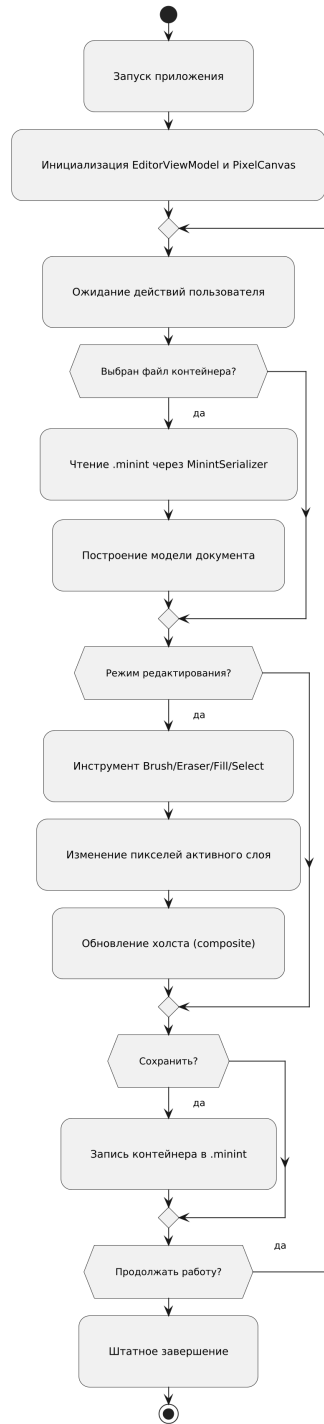


Figure 2: Основной рабочий цикл

6.2 Формат контейнера и сериализация

Report/lab2/uml/lr2-container-serialization.puml

ЛР2 ИВ1: структура контейнера и сериализация

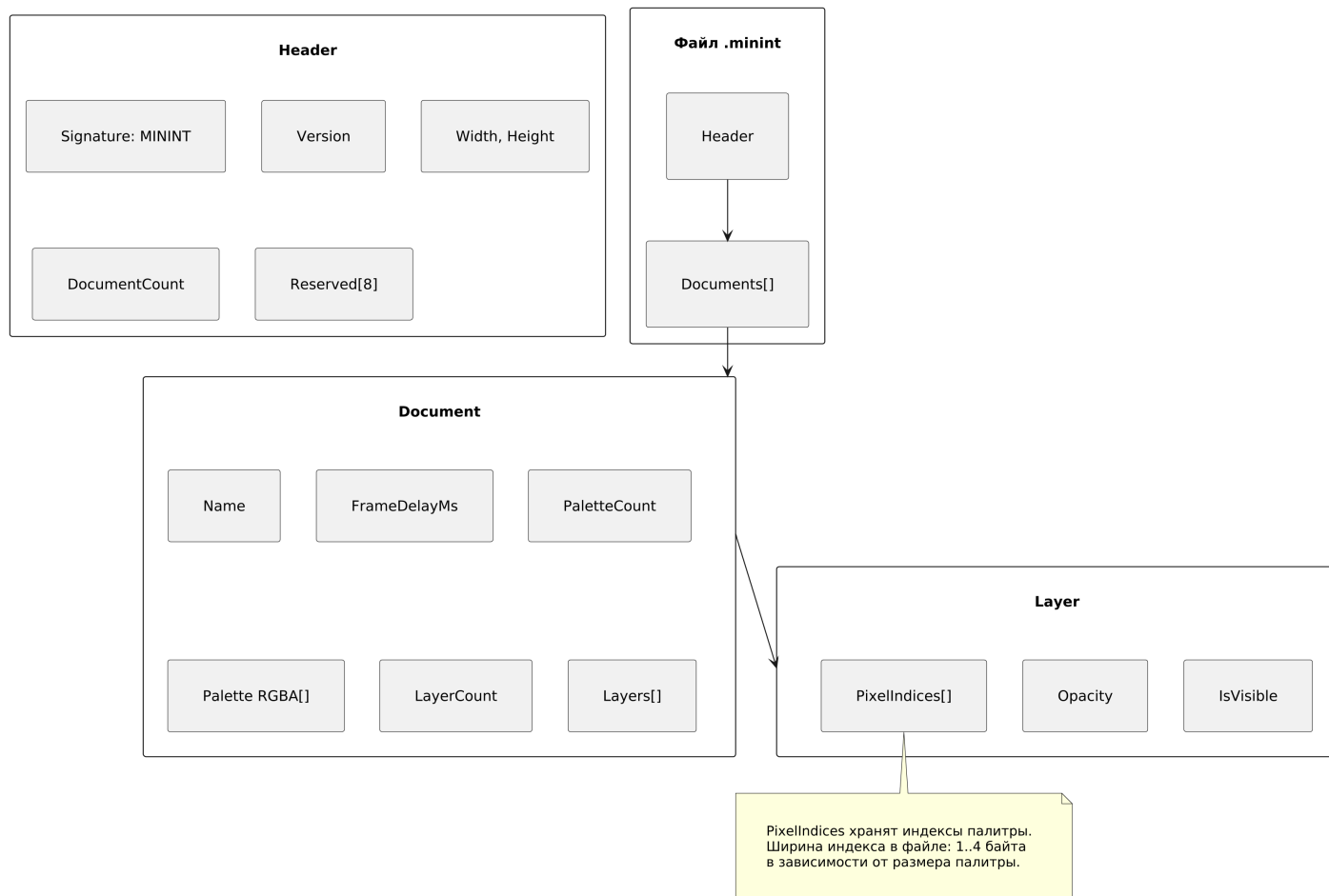


Figure 3: Контейнер и сериализация

6.3 Инструменты рисования и заливки

Report/lab2/uml/lr2-tools-and-fill.puml

ЛР2 ИВ1: обработка инструментов рисования

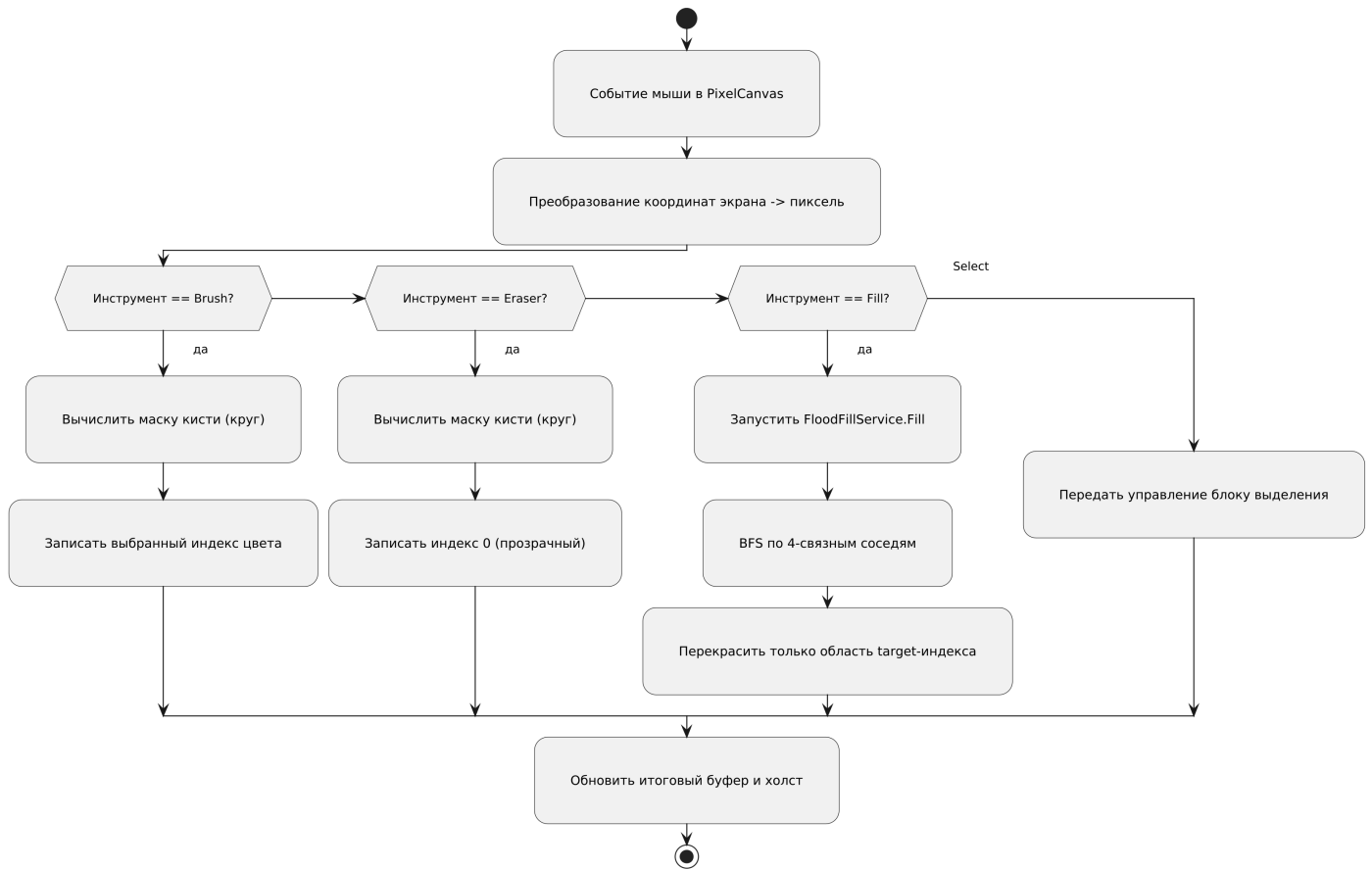


Figure 4: Инструменты и заливка

6.4 Выделение и буфер обмена фрагментов

Report/lab2/uml/lr2-selection-copy-paste.puml

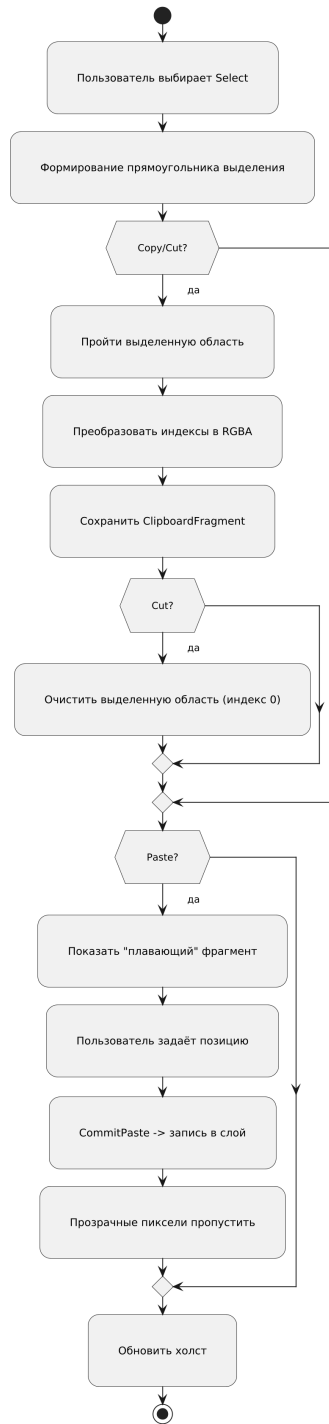


Figure 5: Выделение и copy/paste

7. Проверка работоспособности

Для проверки корректности реализации используются модульные тесты проекта `Minint.Tests`:

- `DrawingTests`;
- `FloodFillTests`;
- `FragmentServiceTests`;
- `SerializerTests`;
- `CompositorTests`;

- ExportTests.

8. Вывод

В рамках ЛР2 (вариант ИВ1) реализовано рабочее приложение-растровый редактор с собственным контейнером данных и базовым набором инструментов редактирования изображения. Практические требования ИВ1 закрыты на уровне пользовательского сценария и программной реализации.

Отдельно зафиксировано, что выбранная структура контейнера является палитровой и не повторяет буквально формулировки KB1–KB4; при этом это не противоречит задаче разработки редактора и демонстрирует полноценную обработку графических данных.

Приложение А. Исходные тексты

Сформировано автоматически скриптом Report/append_sources_to_report.py (файлов: 48).

A.1. Minint.Core/Models/MinintContainer.cs

```
namespace Minint.Core.Models;

/// <summary>
/// Top-level container: holds dimensions shared by all documents/layers,
/// and a list of documents (frames).
/// </summary>
public sealed class MinintContainer
{
    public int Width { get; set; }
    public int Height { get; set; }
    public List<MinintDocument> Documents { get; }

    public int PixelCount => Width * Height;

    public MinintContainer(int width, int height)
    {
        ArgumentOutOfRangeException.ThrowIfLessThan(width, 1);
        ArgumentOutOfRangeException.ThrowIfLessThan(height, 1);

        Width = width;
        Height = height;
        Documents = [];
    }

    /// <summary>
    /// Creates a new document with a single transparent layer and adds it to the container.
    /// </summary>
    public MinintDocument AddNewDocument(string name)
    {
        var doc = new MinintDocument(name);
        doc.Layers.Add(new MinintLayer("Layer 1", PixelCount));
        Documents.Add(doc);
        return doc;
    }
}
```

A.2. Minint.Core/Models/MinintDocument.cs

```
namespace Minint.Core.Models;

/// <summary>
/// A single document (frame) within a container.
/// Has its own palette shared by all layers, plus a list of layers.
/// </summary>
```

```

public sealed class MinintDocument
{
    public string Name { get; set; }

    /// <summary>
    /// Delay before showing the next frame during animation playback (ms).
    /// </summary>
    public uint FrameDelayMs { get; set; }

    /// <summary>
    /// Document palette. Index 0 is always <see cref="RgbaColor.Transparent"/>.
    /// All layers reference colors by index into this list.
    /// </summary>
    public List<RgbaColor> Palette { get; }

    public List<MinintLayer> Layers { get; }

    /// <summary>
    /// Reverse lookup cache: RgbaColor → palette index. Built lazily, invalidated
    /// on structural palette changes (compact, clear). Call <see cref="InvalidatePaletteCache"/>
    /// after bulk palette modifications.
    /// </summary>
    private Dictionary<RgbaColor, int>? _paletteCache;

    public MinintDocument(string name)
    {
        Name = name;
        FrameDelayMs = 100;
        Palette = [RgbaColor.Transparent];
        Layers = [];
    }

    /// <summary>
    /// Constructor for deserialization – accepts pre-built palette and layers.
    /// </summary>
    public MinintDocument(string name, uint frameDelayMs, List<RgbaColor> palette, List<MinintLayer>
        ↪ layers)
    {
        Name = name;
        FrameDelayMs = frameDelayMs;
        Palette = palette;
        Layers = layers;
    }

    /// <summary>
    /// Returns the number of bytes needed to store a single palette index on disk.
    /// </summary>
    public int IndexByteWidth => Palette.Count switch
    {
        <= 255 => 1,
        <= 65_535 => 2,
        <= 16_777_215 => 3,
        _ => 4
    };

    /// <summary>
    /// 0(1) lookup of a color in the palette. Returns the index, or -1 if not found.
    /// Lazily builds an internal dictionary on first call.
    /// </summary>
    public int FindColorCached(RgbaColor color)
    {
        var cache = EnsurePaletteCache();
        return cache.GetValueOrDefault(color, -1);
    }

    /// <summary>

```

```

/// Returns the index of <paramref name="color"/>. If absent, appends it to the palette
/// and updates the cache. O(1) amortized.
/// </summary>
public int EnsureColorCached(RgbaColor color)
{
    var cache = EnsurePaletteCache();
    if (cache.TryGetValue(color, out int idx))
        return idx;

    idx = Palette.Count;
    Palette.Add(color);
    cache[color] = idx;
    return idx;
}

/// <summary>
/// Drops the reverse lookup cache. Must be called after any operation that
/// reorders, removes, or bulk-replaces palette entries (e.g. compact, grayscale).
/// </summary>
public void InvalidatePaletteCache() => _paletteCache = null;

private Dictionary<RgbaColor, int> EnsurePaletteCache()
{
    if (_paletteCache is not null)
        return _paletteCache;

    var cache = new Dictionary<RgbaColor, int>(Palette.Count);
    for (int i = 0; i < Palette.Count; i++)
        cache.TryAdd(Palette[i], i); // first occurrence wins (for dupes)
    _paletteCache = cache;
    return cache;
}
}

```

A.3. Minint.Core/Models/MinintLayer.cs

```

using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace Minint.Core.Models;

/// <summary>
/// A single raster layer. Pixels are indices into the parent document's palette.
/// Array layout is row-major: Pixels[y * width + x].
/// </summary>
public sealed class MinintLayer : INotifyPropertyChanged
{
    private string _name;
    private bool _isVisible;
    private byte _opacity;

    public string Name
    {
        get => _name;
        set { if (_name != value) { _name = value; Notify(); } }
    }

    public bool IsVisible
    {
        get => _isVisible;
        set { if (_isVisible != value) { _isVisible = value; Notify(); } }
    }

    /// <summary>
    /// Per-layer opacity (0 = fully transparent, 255 = fully opaque).

```

```

/// Used during compositing: effective alpha = paletteColor.A * Opacity / 255.
/// </summary>
public byte Opacity
{
    get => _opacity;
    set { if (_opacity != value) { _opacity = value; Notify(); } }
}

/// <summary>
/// Palette indices, length must equal container Width * Height.
/// Index 0 = transparent by convention.
/// </summary>
public int[] Pixels { get; }

public MinintLayer(string name, int pixelCount)
{
    _name = name;
    _isVisible = true;
    _opacity = 255;
    Pixels = new int[pixelCount];
}

/// <summary>
/// Constructor for deserialization – accepts a pre-filled pixel buffer.
/// </summary>
public MinintLayer(string name, bool isVisible, byte opacity, int[] pixels)
{
    _name = name;
    _isVisible = isVisible;
    _opacity = opacity;
    Pixels = pixels;
}

public event PropertyChangedEventHandler? PropertyChanged;
private void Notify([CallerMemberName] string? prop = null)
    => PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(prop));
}

```

A.4. Minint.Core/Models/RgbaColor.cs

```

using System.Runtime.InteropServices;

namespace Minint.Core.Models;

/// <summary>
/// 4-byte RGBA color value. Equality is component-wise.
/// </summary>
[StructLayout(LayoutKind.Sequential, Pack = 1)]
public readonly record struct RgbaColor(byte R, byte G, byte B, byte A)
{
    public static readonly RgbaColor Transparent = new(0, 0, 0, 0);
    public static readonly RgbaColor Black = new(0, 0, 0, 255);
    public static readonly RgbaColor White = new(255, 255, 255, 255);

    /// <summary>
    /// Packs color into a single uint as 0xAABBGGRR (little-endian RGBA).
    /// Suitable for writing directly into BGRA bitmap buffers after byte-swap,
    /// or for use as a dictionary key.
    /// </summary>
    public uint ToPackedRgba() =>
        (uint)(R | (G << 8) | (B << 16) | (A << 24));

    public static RgbaColor FromPackedRgba(uint packed) =>
        new(
            (byte)(packed & 0xFF),

```

```

        (byte)((packed >> 8) & 0xFF),
        (byte)((packed >> 16) & 0xFF),
        (byte)((packed >> 24) & 0xFF));

    /// <summary>
    /// Packs as 0xAARRGGBB – used for Avalonia/SkiaSharp pixel buffers.
    /// </summary>
    public uint ToPackedArgb() =>
        (uint)(B | (G << 8) | (R << 16) | (A << 24));

    public override string ToString() => $"#{R:X2}{G:X2}{B:X2}{A:X2}";
}

```

A.5. Minint.Core/Services/IBmpExporter.cs

```

namespace Minint.Core.Services;

public interface IBmpExporter
{
    /// <summary>
    /// Exports a composited ARGB pixel buffer as a 32-bit BMP file.
    /// </summary>
    /// <param name="stream">Output stream.</param>
    /// <param name="pixels">Pixel data packed as 0xAARRGGBB, row-major.</param>
    /// <param name="width">Image width.</param>
    /// <param name="height">Image height.</param>
    void Export(Stream stream, uint[] pixels, int width, int height);
}

```

A.6. Minint.Core/Services/ICompositor.cs

```

using Minint.Core.Models;

namespace Minint.Core.Services;

public interface ICompositor
{
    /// <summary>
    /// Composites all visible layers of <paramref name="document"/> into a flat RGBA buffer.
    /// Result is packed as ARGB (0xAARRGGBB) per pixel, row-major, length = width * height.
    /// Layers are blended bottom-to-top with alpha compositing.
    /// </summary>
    uint[] Composite(MinintDocument document, int width, int height);
}

```

A.7. Minint.Core/Services/IDrawingService.cs

```

using Minint.Core.Models;

namespace Minint.Core.Services;

public interface IDrawingService
{
    /// <summary>
    /// Applies a circular brush stroke at (<paramref name="cx"/>, <paramref name="cy"/>)
    /// with the given <paramref name="radius"/>. Sets affected pixels to <paramref name="colorIndex"/>.
    /// </summary>
    void ApplyBrush(MinintLayer layer, int cx, int cy, int radius, int colorIndex, int width, int
        ↪ height);

    /// <summary>
    /// Applies a circular eraser at (<paramref name="cx"/>, <paramref name="cy"/>)
    /// with the given <paramref name="radius"/>. Sets affected pixels to index 0 (transparent).
    /// </summary>
}

```

```

void ApplyEraser(MinintLayer layer, int cx, int cy, int radius, int width, int height);

/// <summary>
/// Returns the set of pixel coordinates affected by a circular brush/eraser
/// centered at (<paramref name="cx"/>, <paramref name="cy"/>) with given <paramref name="radius"/>.
/// Used for tool preview overlay.
/// </summary>
List<(int X, int Y)> GetBrushMask(int cx, int cy, int radius, int width, int height);
}

```

A.8. Minint.Core/Services/IFloodFillService.cs

```

using Minint.Core.Models;

namespace Minint.Core.Services;

public interface IFloodFillService
{
    /// <summary>
    /// Flood-fills a contiguous region of identical color starting at (<paramref name="x"/>, <paramref
    ///   name="y"/>)
    /// with <paramref name="newColorIndex"/>. Uses 4-connectivity (up/down/left/right).
    /// </summary>
    void Fill(MinintLayer layer, int x, int y, int newColorIndex, int width, int height);
}

```

A.9. Minint.Core/Services/IFragmentService.cs

```

using Minint.Core.Models;

namespace Minint.Core.Services;

public interface IFragmentService
{
    /// <summary>
    /// Copies a rectangular region from one document/layer to another.
    /// Palette colors are merged: missing colors are added to the destination palette.
    /// </summary>
    /// <param name="srcDoc">Source document.</param>
    /// <param name="srcLayerIndex">Index of the source layer.</param>
    /// <param name="srcX">Source rectangle X origin.</param>
    /// <param name="srcY">Source rectangle Y origin.</param>
    /// <param name="regionWidth">Width of the region to copy.</param>
    /// <param name="regionHeight">Height of the region to copy.</param>
    /// <param name="dstDoc">Destination document.</param>
    /// <param name="dstLayerIndex">Index of the destination layer.</param>
    /// <param name="dstX">Destination X origin.</param>
    /// <param name="dstY">Destination Y origin.</param>
    /// <param name="containerWidth">Container width (shared by both docs).</param>
    /// <param name="containerHeight">Container height (shared by both docs).</param>
    void CopyFragment(
        MinintDocument srcDoc, int srcLayerIndex,
        int srcX, int srcY, int regionWidth, int regionHeight,
        MinintDocument dstDoc, int dstLayerIndex,
        int dstX, int dstY,
        int containerWidth, int containerHeight);
}

```

A.10. Minint.Core/Services/IGifExporter.cs

```

namespace Minint.Core.Services;

public interface IGifExporter
{

```

```

    /// <summary>
    /// Exports multiple frames as an animated GIF.
    /// </summary>
    /// <param name="stream">Output stream.</param>
    /// <param name="frames">Sequence of (ARGB pixels, delay in ms) per frame.</param>
    /// <param name="width">Frame width.</param>
    /// <param name="height">Frame height.</param>
    void Export(Stream stream, IReadOnlyList<(uint[] Pixels, uint DelayMs)> frames, int width, int
    ↪ height);
}

```

A.11. Minint.Core/Services/IImageEffectService.cs

```

using Minint.Core.Models;

namespace Minint.Core.Services;

public interface IImageEffectService
{
    /// <summary>
    /// Adjusts contrast of the document by modifying palette colors.
    /// <paramref name="factor"/> > 1 increases contrast, &lt; 1 decreases.
    /// Index 0 (transparent) is not modified.
    /// </summary>
    void AdjustContrast(MinintDocument document, float factor);

    /// <summary>
    /// Converts the document to grayscale by modifying palette colors.
    /// Uses ITU-R BT.601 luminance: gray = 0.299R + 0.587G + 0.114B.
    /// Index 0 (transparent) is not modified.
    /// </summary>
    void ToGrayscale(MinintDocument document);
}

```

A.12. Minint.Core/Services/IImageEffectsService.cs

```

using Minint.Core.Models;

namespace Minint.Core.Services;

public interface IImageEffectsService
{
    /// <summary>
    /// Adjusts contrast of the document by transforming its palette colors.
    /// <paramref name="factor"/> of 0 = all gray, 1 = no change, >1 = increased contrast.
    /// </summary>
    void ApplyContrast(MinintDocument doc, double factor);

    /// <summary>
    /// Converts the document to grayscale by transforming its palette colors
    /// using the luminance formula: 0.299R + 0.587G + 0.114B.
    /// </summary>
    void ApplyGrayscale(MinintDocument doc);
}

```

A.13. Minint.Core/Services/IMinintSerializer.cs

```

using Minint.Core.Models;

namespace Minint.Core.Services;

public interface IMinintSerializer
{
    /// <summary>

```



```

    /// Serializes the container to a binary .minint stream.
    /// </summary>
    void Write(Stream stream, MinintContainer container);

    /// <summary>
    /// Deserializes a .minint stream into a container.
    /// Throws <see cref="InvalidDataException"/> on format/validation errors.
    /// </summary>
    MinintContainer Read(Stream stream);
}

```

A.14. Minint.Core/Services/IPaletteService.cs

```

using Minint.Core.Models;

namespace Minint.Core.Services;

public interface IPaletteService
{
    /// <summary>
    /// Returns the index of <paramref name="color"/> in the document palette.
    /// If the color is not present, appends it and returns the new index.
    /// </summary>
    int EnsureColor(MinintDocument document, RgbaColor color);

    /// <summary>
    /// Finds index of an exact color match, or returns -1 if not found.
    /// </summary>
    int FindColor(MinintDocument document, RgbaColor color);

    /// <summary>
    /// Removes unused colors from the palette and remaps all layer pixel indices.
    /// Index 0 (transparent) is always preserved.
    /// </summary>
    void CompactPalette(MinintDocument document);
}

```

A.15. Minint.Core/Services/IPatternGenerator.cs

```

using Minint.Core.Models;

namespace Minint.Core.Services;

public enum PatternType
{
    Checkerboard,
    HorizontalGradient,
    VerticalGradient,
    HorizontalStripes,
    VerticalStripes,
    ConcentricCircles,
    Tile
}

public interface IPatternGenerator
{
    /// <summary>
    /// Generates a new document with a single layer filled with the specified pattern.
    /// </summary>
    /// <param name="type">Pattern type.</param>
    /// <param name="width">Image width in pixels.</param>
    /// <param name="height">Image height in pixels.</param>
    /// <param name="colors">Colors to use (interpretation depends on pattern type).</param>
    /// <param name="param1">Primary parameter: cell/stripe size, ring width, etc.</param>
    /// <param name="param2">Secondary parameter (optional, pattern-dependent).</param>

```

```

    MinintDocument Generate(PatternType type, int width, int height, RgbaColor[] colors, int param1,
↪    int param2 = 0);
}

```

A.16. Minint.Core/Services/Impl/Compositor.cs

```

using Minint.Core.Models;

namespace Minint.Core.Services.Impl;

public sealed class Compositor : ICompositor
{
    /// <inheritdoc />
    public uint[] Composite(MinintDocument document, int width, int height)
    {
        int pixelCount = width * height;
        var result = new uint[pixelCount]; // starts as 0x00000000 (transparent black)

        var palette = document.Palette;

        foreach (var layer in document.Layers)
        {
            if (!layer.IsVisible)
                continue;

            byte layerOpacity = layer.Opacity;
            if (layerOpacity == 0)
                continue;

            var pixels = layer.Pixels;
            for (int i = 0; i < pixelCount; i++)
            {
                int idx = pixels[i];
                if (idx == 0)
                    continue; // transparent - skip

                var src = palette[idx];

                // Effective source alpha = palette alpha * layer opacity / 255
                int srcA = src.A * layerOpacity / 255;
                if (srcA == 0)
                    continue;

                if (srcA == 255)
                {
                    // Fully opaque - fast path, no blending needed
                    result[i] = PackArgb(src.R, src.G, src.B, 255);
                    continue;
                }

                // Standard "over" alpha compositing
                uint dst = result[i];
                int dstA = (int)(dst >> 24);
                int dstR = (int)((dst >> 16) & 0xFF);
                int dstG = (int)((dst >> 8) & 0xFF);
                int dstB = (int)(dst & 0xFF);

                int outA = srcA + dstA * (255 - srcA) / 255;
                if (outA == 0)
                    continue;

                int outR = (src.R * srcA + dstR * dstA * (255 - srcA) / 255) / outA;
                int outG = (src.G * srcA + dstG * dstA * (255 - srcA) / 255) / outA;
                int outB = (src.B * srcA + dstB * dstA * (255 - srcA) / 255) / outA;
            }
        }
    }
}

```

```

        result[i] = PackArgb(
            (byte)Math.Min(outR, 255),
            (byte)Math.Min(outG, 255),
            (byte)Math.Min(outB, 255),
            (byte)Math.Min(outA, 255));
    }
}

return result;
}

private static uint PackArgb(byte r, byte g, byte b, byte a) =>
    (uint)(b | (g << 8) | (r << 16) | (a << 24));
}

```

A.17. Minint.Core/Services/Impl/DrawingService.cs

```

using Minint.Core.Models;

namespace Minint.Core.Services.Impl;

public sealed class DrawingService : IDrawingService
{
    public void ApplyBrush(MinintLayer layer, int cx, int cy, int radius, int colorIndex, int width,
        ↪ int height)
    {
        foreach (var (x, y) in GetBrushMask(cx, cy, radius, width, height))
            layer.Pixels[y * width + x] = colorIndex;
    }

    public void ApplyEraser(MinintLayer layer, int cx, int cy, int radius, int width, int height)
    {
        foreach (var (x, y) in GetBrushMask(cx, cy, radius, width, height))
            layer.Pixels[y * width + x] = 0;
    }

    public List<(int X, int Y)> GetBrushMask(int cx, int cy, int radius, int width, int height)
    {
        var mask = new List<(int, int)>();
        int r = Math.Max(radius, 0);
        int r2 = r * r;

        int xMin = Math.Max(0, cx - r);
        int xMax = Math.Min(width - 1, cx + r);
        int yMin = Math.Max(0, cy - r);
        int yMax = Math.Min(height - 1, cy + r);

        for (int py = yMin; py <= yMax; py++)
        {
            int dy = py - cy;
            for (int px = xMin; px <= xMax; px++)
            {
                int dx = px - cx;
                if (dx * dx + dy * dy <= r2)
                    mask.Add((px, py));
            }
        }

        return mask;
    }
}

```

A.18. Minint.Core/Services/Impl/FloodFillService.cs

```

using Minint.Core.Models;

```

```

namespace Minint.Core.Services.Impl;

public sealed class FloodFillService : IFloodFillService
{
    public void Fill(MinintLayer layer, int x, int y, int newColorIndex, int width, int height)
    {
        if (x < 0 || x >= width || y < 0 || y >= height)
            return;

        var pixels = layer.Pixels;
        int targetIndex = pixels[y * width + x];

        if (targetIndex == newColorIndex)
            return;

        var queue = new Queue<(int X, int Y)>();
        var visited = new bool[width * height];

        queue.Enqueue((x, y));
        visited[y * width + x] = true;

        while (queue.Count > 0)
        {
            var (cx, cy) = queue.Dequeue();
            pixels[cy * width + cx] = newColorIndex;

            Span<(int, int)> neighbors =
            [
                (cx - 1, cy), (cx + 1, cy),
                (cx, cy - 1), (cx, cy + 1)
            ];

            foreach (var (nx, ny) in neighbors)
            {
                if (nx < 0 || nx >= width || ny < 0 || ny >= height)
                    continue;
                int ni = ny * width + nx;
                if (visited[ni] || pixels[ni] != targetIndex)
                    continue;
                visited[ni] = true;
                queue.Enqueue((nx, ny));
            }
        }
    }
}

```

A.19. Minint.Core/Services/Impl/FragmentService.cs

```

using System;
using Minint.Core.Models;

namespace Minint.Core.Services.Impl;

public sealed class FragmentService : IFragmentService
{
    public void CopyFragment(
        MinintDocument srcDoc, int srcLayerIndex,
        int srcX, int srcY, int regionWidth, int regionHeight,
        MinintDocument dstDoc, int dstLayerIndex,
        int dstX, int dstY,
        int containerWidth, int containerHeight)
    {
        ArgumentOutOfRangeException.ThrowIfNegative(srcLayerIndex);
        ArgumentOutOfRangeException.ThrowIfNegative(dstLayerIndex);
    }
}

```

```

if (srcLayerIndex >= srcDoc.Layers.Count)
    throw new ArgumentOutOfRangeException(nameof(srcLayerIndex));
if (dstLayerIndex >= dstDoc.Layers.Count)
    throw new ArgumentOutOfRangeException(nameof(dstLayerIndex));

var srcLayer = srcDoc.Layers[srcLayerIndex];
var dstLayer = dstDoc.Layers[dstLayerIndex];

int clippedSrcX = Math.Max(srcX, 0);
int clippedSrcY = Math.Max(srcY, 0);
int clippedEndX = Math.Min(srcX + regionWidth, containerWidth);
int clippedEndY = Math.Min(srcY + regionHeight, containerHeight);

for (int sy = clippedSrcY; sy < clippedEndY; sy++)
{
    int dy = dstY + (sy - srcY);
    if (dy < 0 || dy >= containerHeight) continue;

    for (int sx = clippedSrcX; sx < clippedEndX; sx++)
    {
        int dx = dstX + (sx - srcX);
        if (dx < 0 || dx >= containerWidth) continue;

        int srcIdx = srcLayer.Pixels[sy * containerWidth + sx];
        if (srcIdx == 0) continue; // skip transparent

        RgbaColor color = srcDoc.Palette[srcIdx];
        int dstIdx = dstDoc.EnsureColorCached(color);
        dstLayer.Pixels[dy * containerWidth + dx] = dstIdx;
    }
}
}
}

```

A.20. Minint.Core/Services/Impl/ImageEffectsService.cs

```

using Minint.Core.Models;

namespace Minint.Core.Services.Impl;

public sealed class ImageEffectsService : IImageEffectsService
{
    public void ApplyContrast(MinintDocument doc, double factor)
    {
        for (int i = 1; i < doc.Palette.Count; i++)
        {
            var c = doc.Palette[i];
            doc.Palette[i] = new RgbaColor(
                ContrastByte(c.R, factor),
                ContrastByte(c.G, factor),
                ContrastByte(c.B, factor),
                c.A);
        }
        doc.InvalidatePaletteCache();
    }

    public void ApplyGrayscale(MinintDocument doc)
    {
        for (int i = 1; i < doc.Palette.Count; i++)
        {
            var c = doc.Palette[i];
            byte gray = (byte)Math.Clamp((int)(0.299 * c.R + 0.587 * c.G + 0.114 * c.B + 0.5), 0, 255);
            doc.Palette[i] = new RgbaColor(gray, gray, gray, c.A);
        }
        doc.InvalidatePaletteCache();
    }
}

```

```

    }

    private static byte ContrastByte(byte value, double factor)
    {
        double v = ((value / 255.0) - 0.5) * factor + 0.5;
        return (byte)Math.Clamp((int)(v * 255 + 0.5), 0, 255);
    }
}

```

A.21. Minint.Core/Services/Impl/PaletteService.cs

```

using Minint.Core.Models;

namespace Minint.Core.Services.Impl;

public sealed class PaletteService : IPaletteService
{
    public int FindColor(MinintDocument document, RgbaColor color)
        => document.FindColorCached(color);

    public int EnsureColor(MinintDocument document, RgbaColor color)
        => document.EnsureColorCached(color);

    public void CompactPalette(MinintDocument document)
    {
        var palette = document.Palette;
        if (palette.Count <= 1)
            return;

        var usedIndices = new HashSet<int> { 0 };
        foreach (var layer in document.Layers)
        {
            foreach (int idx in layer.Pixels)
                usedIndices.Add(idx);
        }

        var oldToNew = new int[palette.Count];
        var newPalette = new List<RgbaColor>(usedIndices.Count);

        newPalette.Add(palette[0]);
        oldToNew[0] = 0;

        for (int i = 1; i < palette.Count; i++)
        {
            if (usedIndices.Contains(i))
            {
                oldToNew[i] = newPalette.Count;
                newPalette.Add(palette[i]);
            }
        }

        if (newPalette.Count == palette.Count)
            return;

        palette.Clear();
        palette.AddRange(newPalette);

        foreach (var layer in document.Layers)
        {
            var px = layer.Pixels;
            for (int i = 0; i < px.Length; i++)
                px[i] = oldToNew[px[i]];
        }

        document.InvalidatePaletteCache();
    }
}

```

```

    }
}

```

A.22. Minint.Core/Services/Impl/PatternGenerator.cs

```

using System;
using Minint.Core.Models;

namespace Minint.Core.Services.Impl;

public sealed class PatternGenerator : IPatternGenerator
{
    public MinintDocument Generate(PatternType type, int width, int height, RgbaColor[] colors, int
        ↪ param1, int param2 = 0)
    {
        ArgumentOutOfRangeException.ThrowIfLessThan(width, 1);
        ArgumentOutOfRangeException.ThrowIfLessThan(height, 1);
        if (colors.Length < 2)
            throw new ArgumentException("At least two colors are required.", nameof(colors));

        var doc = new MinintDocument($"Pattern ({type})");
        var layer = new MinintLayer("Pattern", width * height);
        doc.Layers.Add(layer);

        int[] colorIndices = new int[colors.Length];
        for (int i = 0; i < colors.Length; i++)
            colorIndices[i] = doc.EnsureColorCached(colors[i]);

        int cellSize = Math.Max(param1, 1);

        switch (type)
        {
            case PatternType.Checkerboard:
                FillCheckerboard(layer.Pixels, width, height, colorIndices, cellSize);
                break;
            case PatternType.HorizontalGradient:
                FillGradient(layer.Pixels, width, height, colors[0], colors[1], doc, horizontal: true);
                break;
            case PatternType.VerticalGradient:
                FillGradient(layer.Pixels, width, height, colors[0], colors[1], doc, horizontal: false);
                break;
            case PatternType.HorizontalStripes:
                FillStripes(layer.Pixels, width, height, colorIndices, cellSize, horizontal: true);
                break;
            case PatternType.VerticalStripes:
                FillStripes(layer.Pixels, width, height, colorIndices, cellSize, horizontal: false);
                break;
            case PatternType.ConcentricCircles:
                FillCircles(layer.Pixels, width, height, colorIndices, cellSize);
                break;
            case PatternType.Tile:
                FillTile(layer.Pixels, width, height, colorIndices, cellSize, Math.Max(param2, 1));
                break;
        }

        return doc;
    }

    private static void FillCheckerboard(int[] pixels, int w, int h, int[] ci, int cell)
    {
        for (int y = 0; y < h; y++)
            for (int x = 0; x < w; x++)
                pixels[y * w + x] = ci[((x / cell) + (y / cell)) % 2 == 0 ? 0 : 1];
    }
}

```

```

private static void FillGradient(int[] pixels, int w, int h, RgbColor c0, RgbColor c1,
                                MinintDocument doc, bool horizontal)
{
    int steps = horizontal ? w : h;
    for (int s = 0; s < steps; s++)
    {
        double t = steps > 1 ? (double)s / (steps - 1) : 0;
        var c = new RgbColor(
            Lerp(c0.R, c1.R, t), Lerp(c0.G, c1.G, t),
            Lerp(c0.B, c1.B, t), Lerp(c0.A, c1.A, t));
        int idx = doc.EnsureColorCached(c);
        if (horizontal)
            for (int y = 0; y < h; y++) pixels[y * w + s] = idx;
        else
            for (int x = 0; x < w; x++) pixels[s * w + x] = idx;
    }
}

private static void FillStripes(int[] pixels, int w, int h, int[] ci, int stripe, bool horizontal)
{
    for (int y = 0; y < h; y++)
        for (int x = 0; x < w; x++)
        {
            int coord = horizontal ? y : x;
            pixels[y * w + x] = ci[(coord / stripe) % ci.Length];
        }
}

private static void FillCircles(int[] pixels, int w, int h, int[] ci, int ringWidth)
{
    double cx = (w - 1) / 2.0, cy = (h - 1) / 2.0;
    for (int y = 0; y < h; y++)
        for (int x = 0; x < w; x++)
        {
            double dist = Math.Sqrt((x - cx) * (x - cx) + (y - cy) * (y - cy));
            int ring = (int)(dist / ringWidth);
            pixels[y * w + x] = ci[ring % ci.Length];
        }
}

private static void FillTile(int[] pixels, int w, int h, int[] ci, int tileW, int tileH)
{
    for (int y = 0; y < h; y++)
        for (int x = 0; x < w; x++)
        {
            int tx = (x / tileW) % ci.Length;
            int ty = (y / tileH) % ci.Length;
            pixels[y * w + x] = ci[(tx + ty) % ci.Length];
        }
}

private static byte Lerp(byte a, byte b, double t)
    => (byte)Math.Clamp((int)(a + (b - a) * t + 0.5), 0, 255);
}

```

A.23. Minint.Infrastructure/Export/BmpExporter.cs

```

using Minint.Core.Services;

namespace Minint.Infrastructure.Export;

/// <summary>
/// Writes a 32-bit BGRA BMP (BITMAPV4HEADER) from an ARGB pixel buffer.
/// BMP rows are bottom-up, so we flip vertically during write.
/// </summary>

```



```

public sealed class BmpExporter : IBmpExporter
{
    private const int BmpFileHeaderSize = 14;
    private const int BitmapV4HeaderSize = 108;
    private const int HeadersTotal = BmpFileHeaderSize + BitmapV4HeaderSize;

    public void Export(Stream stream, uint[] pixels, int width, int height)
    {
        ArgumentNullException.ThrowIfNull(stream);
        ArgumentNullException.ThrowIfNull(pixels);
        if (pixels.Length != width * height)
            throw new ArgumentException("Pixel buffer size does not match dimensions.");

        int rowBytes = width * 4;
        int imageSize = rowBytes * height;
        int fileSize = HeadersTotal + imageSize;

        using var w = new BinaryWriter(stream, System.Text.Encoding.UTF8, leaveOpen: true);

        // BITMAPFILEHEADER (14 bytes)
        w.Write((byte)'B');
        w.Write((byte)'M');
        w.Write(fileSize);
        w.Write((ushort)0); // reserved1
        w.Write((ushort)0); // reserved2
        w.Write(HeadersTotal); // pixel data offset

        // BITMAPV4HEADER (108 bytes)
        w.Write(BitmapV4HeaderSize);
        w.Write(width);
        w.Write(height); // positive = bottom-up
        w.Write((ushort)1); // planes
        w.Write((ushort)32); // bpp
        w.Write(3); // biCompression = BI_BITFIELDS
        w.Write(imageSize);
        w.Write(2835); // X pixels per meter (~72 DPI)
        w.Write(2835); // Y pixels per meter
        w.Write(0); // colors used
        w.Write(0); // important colors

        // Channel masks (BGRA order in file)
        w.Write(0x00FF0000u); // red mask
        w.Write(0x0000FF00u); // green mask
        w.Write(0x000000FFu); // blue mask
        w.Write(0xFF000000u); // alpha mask

        // Color space type: LCS_sRGB
        w.Write(0x73524742); // 'sRGB'
        // CIEXYZTRIPLE endpoints (36 bytes zeroed)
        w.Write(new byte[36]);
        // Gamma RGB (12 bytes zeroed)
        w.Write(new byte[12]);

        // Pixel data: BMP is bottom-up, our buffer is top-down
        for (int y = height - 1; y >= 0; y--)
        {
            int rowStart = y * width;
            for (int x = 0; x < width; x++)
            {
                uint argb = pixels[rowStart + x];
                byte a = (byte)(argb >> 24);
                byte r = (byte)((argb >> 16) & 0xFF);
                byte g = (byte)((argb >> 8) & 0xFF);
                byte b = (byte)(argb & 0xFF);
                w.Write(b);
                w.Write(g);
            }
        }
    }
}

```

```

        w.Write(r);
        w.Write(a);
    }
}
}

```

A.24. Minint.Infrastructure/Export/GifExporter.cs

```

using System;
using System.Collections.Generic;
using System.IO;
using Minint.Core.Services;

namespace Minint.Infrastructure.Export;

/// <summary>
/// Self-implemented GIF89a animated exporter.
/// Quantizes each ARGB frame to 256 colors (including transparent) using popularity.
/// Uses LZW compression as required by the GIF spec.
/// </summary>
public sealed class GifExporter : IGifExporter
{
    public void Export(Stream stream, IReadOnlyList<uint[]> Pixels, uint DelayMs, int width,
        int height)
    {
        ArgumentNullException.ThrowIfNull(stream);
        if (frames.Count == 0)
            throw new ArgumentException("At least one frame is required.");

        using var w = new BinaryWriter(stream, System.Text.Encoding.UTF8, leaveOpen: true);

        WriteGifHeader(w, width, height);
        WriteNetscapeExtension(w); // infinite loop

        foreach (var (pixels, delayMs) in frames)
        {
            var (palette, indices, transparentIndex) = Quantize(pixels, width * height);
            int colorBits = GetColorBits(palette.Length);
            int tableSize = 1 << colorBits;

            WriteGraphicControlExtension(w, delayMs, transparentIndex);
            WriteImageDescriptor(w, width, height, colorBits);
            WriteColorTable(w, palette, tableSize);
            WriteLzwImageData(w, indices, colorBits);
        }

        w.Write((byte)0x3B); // GIF trailer
    }

    private static void WriteGifHeader(BinaryWriter w, int width, int height)
    {
        w.Write("GIF89a".u8.ToArray());
        w.Write((ushort)width);
        w.Write((ushort)height);
        // Global color table flag=0, color resolution=7, sort=0, gct size=0
        w.Write((byte)0x70);
        w.Write((byte)0); // background color index
        w.Write((byte)0); // pixel aspect ratio
    }

    private static void WriteNetscapeExtension(BinaryWriter w)
    {
        w.Write((byte)0x21); // extension introducer
        w.Write((byte)0xFF); // application extension
    }
}

```

```

        w.Write((byte)11); // block size
        w.Write("NETSCAPE2.0"u8.ToArray());
        w.Write((byte)3); // sub-block size
        w.Write((byte)1); // sub-block ID
        w.Write((ushort)0); // loop count: 0 = infinite
        w.Write((byte)0); // block terminator
    }

    private static void WriteGraphicControlExtension(BinaryWriter w, uint delayMs, int transparentIndex)
    {
        w.Write((byte)0x21); // extension introducer
        w.Write((byte)0xF9); // graphic control label
        w.Write((byte)4); // block size
        byte packed = (byte)(transparentIndex >= 0 ? 0x09 : 0x08);
        // disposal method=2 (restore to background), no user input, transparent flag
        w.Write(packed);
        ushort delayCs = (ushort)(delayMs / 10); // GIF delay is in centiseconds
        if (delayCs == 0 && delayMs > 0) delayCs = 1;
        w.Write(delayCs);
        w.Write((byte)(transparentIndex >= 0 ? transparentIndex : 0));
        w.Write((byte)0); // block terminator
    }

    private static void WriteImageDescriptor(BinaryWriter w, int width, int height, int colorBits)
    {
        w.Write((byte)0x2C); // image separator
        w.Write((ushort)0); // left
        w.Write((ushort)0); // top
        w.Write((ushort)width);
        w.Write((ushort)height);
        byte packed = (byte)(0x80 | (colorBits - 1)); // local color table, not interlaced
        w.Write(packed);
    }

    private static void WriteColorTable(BinaryWriter w, byte[][] palette, int tableSize)
    {
        for (int i = 0; i < tableSize; i++)
        {
            if (i < palette.Length)
            {
                w.Write(palette[i][0]); // R
                w.Write(palette[i][1]); // G
                w.Write(palette[i][2]); // B
            }
            else
            {
                w.Write((byte)0);
                w.Write((byte)0);
                w.Write((byte)0);
            }
        }
    }

    #region LZW compression

    private static void WriteLzwImageData(BinaryWriter w, byte[] indices, int colorBits)
    {
        int minCodeSize = Math.Max(colorBits, 2);
        w.Write((byte)minCodeSize);

        var output = new List<byte>();
        LzwCompress(indices, minCodeSize, output);

        int offset = 0;
        while (offset < output.Count)
        {

```

```

        int blockLen = Math.Min(255, output.Count - offset);
        w.Write((byte)blockLen);
        for (int i = 0; i < blockLen; i++)
            w.Write(output[offset + i]);
        offset += blockLen;
    }
    w.Write((byte)0); // block terminator
}

private static void LzwCompress(byte[] indices, int minCodeSize, List<byte> output)
{
    int clearCode = 1 << minCodeSize;
    int eoiCode = clearCode + 1;

    int codeSize = minCodeSize + 1;
    int nextCode = eoiCode + 1;
    int maxCode = (1 << codeSize) - 1;

    var table = new Dictionary<(int Prefix, byte Suffix), int>();

    int bitBuffer = 0;
    int bitCount = 0;

    void EmitCode(int code)
    {
        bitBuffer |= code << bitCount;
        bitCount += codeSize;
        while (bitCount >= 8)
        {
            output.Add((byte)(bitBuffer & 0xFF));
            bitBuffer >>= 8;
            bitCount -= 8;
        }
    }

    void ResetTable()
    {
        table.Clear();
        codeSize = minCodeSize + 1;
        nextCode = eoiCode + 1;
        maxCode = (1 << codeSize) - 1;
    }

    EmitCode(clearCode);
    ResetTable();

    if (indices.Length == 0)
    {
        EmitCode(eoiCode);
        if (bitCount > 0) output.Add((byte)(bitBuffer & 0xFF));
        return;
    }

    int prefix = indices[0];

    for (int i = 1; i < indices.Length; i++)
    {
        byte suffix = indices[i];
        var key = (prefix, suffix);

        if (table.TryGetValue(key, out int existing))
        {
            prefix = existing;
        }
        else
        {

```

```

        EmitCode(prefix);

        if (nextCode <= 4095)
        {
            table[key] = nextCode++;
            if (nextCode > maxCode + 1 && codeSize < 12)
            {
                codeSize++;
                maxCode = (1 << codeSize) - 1;
            }
        }
        else
        {
            EmitCode(clearCode);
            ResetTable();
        }

        prefix = suffix;
    }
}

EmitCode(prefix);
EmitCode(eoiCode);
if (bitCount > 0) output.Add((byte)(bitBuffer & 0xFF));
}

#endregion

#region Quantization

/// <summary>
/// Quantizes ARGB pixels to max 256 palette entries.
/// Reserves index 0 for transparent if any pixel has alpha < 128.
/// Uses popularity-based selection for opaque colors.
/// </summary>
private static (byte[][] Palette, byte[] Indices, int TransparentIndex) Quantize(uint[] argb, int
    count)
{
    bool hasTransparency = false;
    var colorCounts = new Dictionary<uint, int>();

    for (int i = 0; i < count; i++)
    {
        uint px = argb[i];
        byte a = (byte)(px >> 24);
        if (a < 128)
        {
            hasTransparency = true;
            continue;
        }
        uint opaque = px | 0xFF000000u;
        colorCounts.TryGetValue(opaque, out int c);
        colorCounts[opaque] = c + 1;
    }

    int transparentIndex = hasTransparency ? 0 : -1;
    int maxColors = hasTransparency ? 255 : 256;

    var sorted = new List<KeyValuePair<uint, int>>(colorCounts);
    sorted.Sort((a, b) => b.Value.CompareTo(a.Value));
    if (sorted.Count > maxColors)
        sorted.RemoveRange(maxColors, sorted.Count - maxColors);

    var palette = new List<byte[]>();
    var colorToIndex = new Dictionary<uint, byte>();

```

```

    if (hasTransparency)
    {
        palette.Add([0, 0, 0]); // transparent slot
    }

    foreach (var kv in sorted)
    {
        uint px = kv.Key;
        byte idx = (byte)palette.Count;
        palette.Add([
            (byte)((px >> 16) & 0xFF),
            (byte)((px >> 8) & 0xFF),
            (byte)(px & 0xFF)
        ]);
        colorToIndex[px] = idx;
    }

    if (palette.Count == 0)
        palette.Add([0, 0, 0]);

    var indices = new byte[count];
    for (int i = 0; i < count; i++)
    {
        uint px = argb[i];
        byte a = (byte)(px >> 24);
        if (a < 128)
        {
            indices[i] = (byte)(transparentIndex >= 0 ? transparentIndex : 0);
            continue;
        }

        uint opaque = px | 0xFF000000u;
        if (colorToIndex.TryGetValue(opaque, out byte idx))
        {
            indices[i] = idx;
        }
        else
        {
            indices[i] = FindClosest(opaque, palette, hasTransparency ? 1 : 0);
        }
    }

    return (palette.ToArray(), indices, transparentIndex);
}

private static byte FindClosest(uint argb, List<byte[]> palette, int startIdx)
{
    byte r = (byte)((argb >> 16) & 0xFF);
    byte g = (byte)((argb >> 8) & 0xFF);
    byte b = (byte)(argb & 0xFF);

    int bestIdx = startIdx;
    int bestDist = int.MaxValue;
    for (int i = startIdx; i < palette.Count; i++)
    {
        int dr = r - palette[i][0];
        int dg = g - palette[i][1];
        int db = b - palette[i][2];
        int dist = dr * dr + dg * dg + db * db;
        if (dist < bestDist)
        {
            bestDist = dist;
            bestIdx = i;
        }
    }

    return (byte)bestIdx;
}

```

```

    }

    private static int GetColorBits(int paletteCount)
    {
        int bits = 2;
        while ((1 << bits) < paletteCount) bits++;
        return Math.Min(bits, 8);
    }

    #endregion
}

```

A.25. Minint.Infrastructure/Serialization/MinintSerializer.cs

```

using System.Text;
using Minint.Core.Models;
using Minint.Core.Services;

namespace Minint.Infrastructure.Serialization;

/// <summary>
/// Self-implemented binary reader/writer for the .minint container format.
/// All multi-byte integers are little-endian. Strings are UTF-8 with a 1-byte length prefix.
/// </summary>
public sealed class MinintSerializer : IMinintSerializer
{
    private static readonly byte[] Signature = "MININT"u8.ToArray();
    private const ushort CurrentVersion = 1;
    private const int ReservedBytes = 8;
    private const int MaxNameLength = 255;

    #region Write

    public void Write(Stream stream, MinintContainer container)
    {
        ArgumentNullException.ThrowIfNull(stream);
        ArgumentNullException.ThrowIfNull(container);

        using var w = new BinaryWriter(stream, Encoding.UTF8, leaveOpen: true);

        WriteHeader(w, container);

        foreach (var doc in container.Documents)
            WriteDocument(w, doc, container.Width, container.Height);
    }

    private static void WriteHeader(BinaryWriter w, MinintContainer c)
    {
        w.Write(Signature);
        w.Write(CurrentVersion);
        w.Write((uint)c.Width);
        w.Write((uint)c.Height);
        w.Write((uint)c.Documents.Count);
        w.Write(new byte[ReservedBytes]);
    }

    private static void WriteDocument(BinaryWriter w, MinintDocument doc, int width, int height)
    {
        WritePrefixedString(w, doc.Name);
        w.Write(doc.FrameDelayMs);
        w.Write((uint)doc.Palette.Count);

        foreach (var color in doc.Palette)
        {
            w.Write(color.R);

```

```

        w.Write(color.G);
        w.Write(color.B);
        w.Write(color.A);
    }

    w.Write((uint)doc.Layers.Count);

    int byteWidth = doc.IndexByteWidth;
    foreach (var layer in doc.Layers)
        WriteLayer(w, layer, byteWidth, width * height);
}

private static void WriteLayer(BinaryWriter w, MinintLayer layer, int byteWidth, int pixelCount)
{
    WritePrefixedString(w, layer.Name);
    w.Write(layer.IsVisible ? (byte)1 : (byte)0);
    w.Write(layer.OpaCity);

    if (layer.Pixels.Length != pixelCount)
        throw new InvalidOperationException(
            $"Layer '{layer.Name}' has {layer.Pixels.Length} pixels, expected {pixelCount}.");

    for (int i = 0; i < pixelCount; i++)
        WriteIndex(w, layer.Pixels[i], byteWidth);
}

private static void WriteIndex(BinaryWriter w, int index, int byteWidth)
{
    switch (byteWidth)
    {
        case 1:
            w.Write((byte)index);
            break;
        case 2:
            w.Write((ushort)index);
            break;
        case 3:
            w.Write((byte)(index & 0xFF));
            w.Write((byte)((index >> 8) & 0xFF));
            w.Write((byte)((index >> 16) & 0xFF));
            break;
        case 4:
            w.Write(index);
            break;
    }
}

private static void WritePrefixedString(BinaryWriter w, string value)
{
    var bytes = Encoding.UTF8.GetBytes(value);
    if (bytes.Length > MaxNameLength)
        throw new InvalidOperationException(
            $"String '{value}' exceeds max length of {MaxNameLength} UTF-8 bytes.");
    w.Write((byte)bytes.Length);
    w.Write(bytes);
}

#endregion

#region Read

public MinintContainer Read(Stream stream)
{
    ArgumentNullException.ThrowIfNull(stream);

    using var r = new BinaryReader(stream, Encoding.UTF8, leaveOpen: true);

```



```

var (width, height, docCount) = ReadHeader(r);
var container = new MinintContainer(width, height);

for (int i = 0; i < docCount; i++)
    container.Documents.Add(ReadDocument(r, width, height));

return container;
}

private static (int Width, int Height, int DocCount) ReadHeader(BinaryReader r)
{
    byte[] sig = ReadExact(r, Signature.Length, "file signature");
    if (!sig.AsSpan().SequenceEqual(Signature))
        throw new InvalidDataException(
            "Invalid file signature. Expected 'MININT'.");

    ushort version = r.ReadUInt16();
    if (version != CurrentVersion)
        throw new InvalidDataException(
            $"Unsupported format version {version}. Only version {CurrentVersion} is supported.");

    uint width = r.ReadUInt32();
    uint height = r.ReadUInt32();
    uint docCount = r.ReadUInt32();

    if (width == 0 || height == 0)
        throw new InvalidDataException("Width and height must be at least 1.");
    if (width > 65_536 || height > 65_536)
        throw new InvalidDataException(
            $"Dimensions {width}x{height} exceed maximum supported size (65536).");
    if (docCount == 0)
        throw new InvalidDataException("Container must have at least 1 document.");

    byte[] reserved = ReadExact(r, ReservedBytes, "reserved bytes");
    for (int i = 0; i < reserved.Length; i++)
    {
        if (reserved[i] != 0)
            break; // non-zero reserved bytes: tolerated for forward compat
    }

    return ((int)width, (int)height, (int)docCount);
}

private static MinintDocument ReadDocument(BinaryReader r, int width, int height)
{
    string name = ReadPrefixedString(r);
    uint frameDelay = r.ReadUInt32();
    uint paletteCount = r.ReadUInt32();

    if (paletteCount == 0)
        throw new InvalidDataException("Palette must have at least 1 color.");

    var palette = new List<RgbColor>((int)paletteCount);
    for (uint i = 0; i < paletteCount; i++)
    {
        byte cr = r.ReadByte();
        byte cg = r.ReadByte();
        byte cb = r.ReadByte();
        byte ca = r.ReadByte();
        palette.Add(new RgbColor(cr, cg, cb, ca));
    }

    int byteWidth = GetIndexByteWidth((int)paletteCount);

    uint layerCount = r.ReadUInt32();

```

```

    if (layerCount == 0)
        throw new InvalidDataException($"Document '{name}' must have at least 1 layer.");

    var layers = new List<MinintLayer>((int)layerCount);
    int pixelCount = width * height;
    for (uint i = 0; i < layerCount; i++)
        layers.Add(ReadLayer(r, byteWidth, pixelCount, (int)paletteCount));

    return new MinintDocument(name, frameDelay, palette, layers);
}

private static MinintLayer ReadLayer(BinaryReader r, int byteWidth, int pixelCount, int
↪ paletteCount)
{
    string name = ReadPrefixedString(r);
    byte visByte = r.ReadByte();
    if (visByte > 1)
        throw new InvalidDataException(
            $"Layer '{name}': invalid visibility flag {visByte} (expected 0 or 1).");
    bool isVisible = visByte == 1;
    byte opacity = r.ReadByte();

    var pixels = new int[pixelCount];
    for (int i = 0; i < pixelCount; i++)
    {
        int idx = ReadIndex(r, byteWidth);
        if (idx < 0 || idx >= paletteCount)
            throw new InvalidDataException(
↪ $"Layer '{name}': pixel index {idx} at position {i} is out of palette range [0,
{paletteCount}).");
        pixels[i] = idx;
    }

    return new MinintLayer(name, isVisible, opacity, pixels);
}

private static int ReadIndex(BinaryReader r, int byteWidth)
{
    return byteWidth switch
    {
        1 => r.ReadByte(),
        2 => r.ReadUInt16(),
        3 => r.ReadByte() | (r.ReadByte() << 8) | (r.ReadByte() << 16),
        4 => r.ReadInt32(),
        _ => throw new InvalidDataException($"Invalid index byte width: {byteWidth}")
    };
}

private static string ReadPrefixedString(BinaryReader r)
{
    byte len = r.ReadByte();
    if (len == 0) return string.Empty;
    byte[] bytes = ReadExact(r, len, "string data");
    return Encoding.UTF8.GetString(bytes);
}

/// <summary>
/// Reads exactly <paramref name="count"/> bytes or throws on premature EOF.
/// </summary>
private static byte[] ReadExact(BinaryReader r, int count, string context)
{
    byte[] buf = r.ReadBytes(count);
    if (buf.Length < count)
        throw new InvalidDataException(
↪ $"Unexpected end of stream while reading {context} (expected {count} bytes, got
{buf.Length}).");
}

```

```

        return buf;
    }

#endregion

#region Helpers

/// <summary>
/// Same logic as <see cref="MinintDocument.IndexByteWidth"/>,
/// usable when only the palette count is known (during deserialization).
/// </summary>
public static int GetIndexByteWidth(int paletteCount) => paletteCount switch
{
    <= 255 => 1,
    <= 65_535 => 2,
    <= 16_777_215 => 3,
    - => 4
};

#endregion
}

```

A.26. Minint.Tests/CompositorTests.cs

```

using Minint.Core.Models;
using Minint.Core.Services.Impl;

namespace Minint.Tests;

public class CompositorTests
{
    private readonly Compositor _compositor = new();

    [Fact]
    public void Composite_EmptyLayer_AllTransparent()
    {
        var doc = new MinintDocument("test");
        doc.Layers.Add(new MinintLayer("L1", 4));

        uint[] result = _compositor.Composite(doc, 2, 2);

        Assert.All(result, px => Assert.Equal(0u, px));
    }

    [Fact]
    public void Composite_SingleOpaquePixel()
    {
        var doc = new MinintDocument("test");
        var red = new RgbaColor(255, 0, 0, 255);
        doc.EnsureColorCached(red);
        var layer = new MinintLayer("L1", 4);
        layer.Pixels[0] = 1;
        doc.Layers.Add(layer);

        uint[] result = _compositor.Composite(doc, 2, 2);

        // ARGB packed as 0xAARRGGBB
        uint expected = 0xFF_FF_00_00u;
        Assert.Equal(expected, result[0]);
        Assert.Equal(0u, result[1]); // rest is transparent
    }

    [Fact]
    public void Composite_HiddenLayer_Ignored()
    {

```

```

var doc = new MinintDocument("test");
doc.EnsureColorCached(new RgbaColor(0, 255, 0, 255));
var layer = new MinintLayer("L1", 4);
layer.Pixels[0] = 1;
layer.IsVisible = false;
doc.Layers.Add(layer);

uint[] result = _compositor.Composite(doc, 2, 2);

Assert.Equal(0u, result[0]);
}

[Fact]
public void Composite_TwoLayers_TopOverBottom()
{
    var doc = new MinintDocument("test");
    var red = new RgbaColor(255, 0, 0, 255);
    var blue = new RgbaColor(0, 0, 255, 255);
    int redIdx = doc.EnsureColorCached(red);
    int blueIdx = doc.EnsureColorCached(blue);

    var bottom = new MinintLayer("bottom", 1);
    bottom.Pixels[0] = redIdx;
    var top = new MinintLayer("top", 1);
    top.Pixels[0] = blueIdx;

    doc.Layers.Add(bottom);
    doc.Layers.Add(top);

    uint[] result = _compositor.Composite(doc, 1, 1);

    // Blue on top, fully opaque, should overwrite red
    Assert.Equal(0xFF_00_00_FFu, result[0]);
}
}

```

A.27. Minint.Tests/DrawingTests.cs

```

using Minint.Core.Models;
using Minint.Core.Services.Impl;

namespace Minint.Tests;

public class DrawingTests
{
    private readonly DrawingService _drawing = new();

    [Fact]
    public void ApplyBrush_Radius0_SetsSinglePixel()
    {
        var layer = new MinintLayer("L1", 9);
        _drawing.ApplyBrush(layer, 1, 1, 0, 1, 3, 3);

        Assert.Equal(1, layer.Pixels[1 * 3 + 1]);
        Assert.Equal(0, layer.Pixels[0]); // (0,0) untouched
    }

    [Fact]
    public void ApplyBrush_Radius1_SetsCircle()
    {
        var layer = new MinintLayer("L1", 25);
        _drawing.ApplyBrush(layer, 2, 2, 1, 1, 5, 5);

        // Center + 4 neighbors should be set
        Assert.Equal(1, layer.Pixels[2 * 5 + 2]); // center
    }
}

```

```

        Assert.Equal(1, layer.Pixels[1 * 5 + 2]); // top
        Assert.Equal(1, layer.Pixels[3 * 5 + 2]); // bottom
        Assert.Equal(1, layer.Pixels[2 * 5 + 1]); // left
        Assert.Equal(1, layer.Pixels[2 * 5 + 3]); // right
    }

    [Fact]
    public void ApplyEraser_SetsToZero()
    {
        var layer = new MinintLayer("L1", 9);
        Array.Fill(layer.Pixels, 5);
        _drawing.ApplyEraser(layer, 1, 1, 0, 3, 3);

        Assert.Equal(0, layer.Pixels[1 * 3 + 1]);
        Assert.Equal(5, layer.Pixels[0]); // untouched
    }

    [Fact]
    public void GetBrushMask_Radius0_SinglePixel()
    {
        var mask = _drawing.GetBrushMask(2, 2, 0, 5, 5);
        Assert.Single(mask);
        Assert.Equal((2, 2), mask[0]);
    }

    [Fact]
    public void GetBrushMask_OutOfBounds_Clamped()
    {
        var mask = _drawing.GetBrushMask(0, 0, 2, 3, 3);
        Assert.All(mask, p =>
        {
            Assert.InRange(p.X, 0, 2);
            Assert.InRange(p.Y, 0, 2);
        });
    }
}

```

A.28. Minint.Tests/ExportTests.cs

```

using System.Text;
using Minint.Core.Models;
using Minint.Core.Services.Impl;
using Minint.Infrastructure.Export;

namespace Minint.Tests;

public class ExportTests
{
    [Fact]
    public void BmpExport_WritesValidBmp()
    {
        var exporter = new BmpExporter();
        var pixels = new uint[] { 0xFF_FF_00_00, 0xFF_00_FF_00, 0xFF_00_00_FF, 0xFF_FF_FF_FF };

        var ms = new MemoryStream();
        exporter.Export(ms, pixels, 2, 2);

        ms.Position = 0;
        byte[] data = ms.ToArray();

        Assert.True(data.Length > 0);
        Assert.Equal((byte)'B', data[0]);
        Assert.Equal((byte)'M', data[1]);

        int fileSize = BitConverter.ToInt32(data, 2);
    }
}

```

```

        Assert.Equal(data.Length, fileSize);
    }

    [Fact]
    public void GifExport_WritesValidGif()
    {
        var exporter = new GifExporter();
        var frame1 = new uint[16]; // 4x4 transparent
        var frame2 = new uint[16];
        Array.Fill(frame2, 0xFF_FF_00_00u);

        var frames = new List<(uint[] Pixels, uint DelayMs)>
        {
            (frame1, 100),
            (frame2, 200),
        };

        var ms = new MemoryStream();
        exporter.Export(ms, frames, 4, 4);

        ms.Position = 0;
        byte[] data = ms.ToArray();

        Assert.True(data.Length > 0);
        string sig = Encoding.ASCII.GetString(data, 0, 6);
        Assert.Equal("GIF89a", sig);
        Assert.Equal(0x3B, data[^1]); // GIF trailer
    }

    [Fact]
    public void BmpExport_DimensionMismatch_Throws()
    {
        var exporter = new BmpExporter();
        var pixels = new uint[3]; // does not match 2x2

        Assert.Throws<ArgumentException>(() =>
        {
            var ms = new MemoryStream();
            exporter.Export(ms, pixels, 2, 2);
        });
    }
}

```

A.29. Minint.Tests/FloodFillTests.cs

```

using Minint.Core.Models;
using Minint.Core.Services.Impl;

namespace Minint.Tests;

public class FloodFillTests
{
    private readonly FloodFillService _fill = new();

    [Fact]
    public void Fill_EmptyLayer_FillsAll()
    {
        var layer = new MinintLayer("L1", 9); // 3x3, all zeros
        _fill.Fill(layer, 0, 0, 1, 3, 3);

        Assert.All(layer.Pixels, px => Assert.Equal(1, px));
    }

    [Fact]
    public void Fill_SameColor_NoOp()
    {
    }
}

```

```

{
    var layer = new MinintLayer("L1", 4);
    Array.Fill(layer.Pixels, 2);
    _fill.Fill(layer, 0, 0, 2, 2, 2);

    Assert.All(layer.Pixels, px => Assert.Equal(2, px));
}

[Fact]
public void Fill_Bounded_DoesNotCrossBorder()
{
    // 3x3 grid with a wall:
    // 0 0 0
    // 1 1 1
    // 0 0 0
    var layer = new MinintLayer("L1", 9);
    layer.Pixels[3] = 1; // (0,1)
    layer.Pixels[4] = 1; // (1,1)
    layer.Pixels[5] = 1; // (2,1)

    _fill.Fill(layer, 0, 0, 2, 3, 3);

    // Top row should be filled
    Assert.Equal(2, layer.Pixels[0]);
    Assert.Equal(2, layer.Pixels[1]);
    Assert.Equal(2, layer.Pixels[2]);
    // Wall untouched
    Assert.Equal(1, layer.Pixels[3]);
    // Bottom row untouched (blocked by wall)
    Assert.Equal(0, layer.Pixels[6]);
}
}

```

A.30. Minint.Tests/FragmentServiceTests.cs

```

using Minint.Core.Models;
using Minint.Core.Services.Impl;

namespace Minint.Tests;

public class FragmentServiceTests
{
    private readonly FragmentService _fragment = new();

    [Fact]
    public void CopyFragment_SameDocument_CopiesPixels()
    {
        var doc = new MinintDocument("test");
        var red = new RgbaColor(255, 0, 0, 255);
        doc.EnsureColorCached(red);

        var src = new MinintLayer("src", 16);
        src.Pixels[0] = 1; // (0,0) = red
        src.Pixels[1] = 1; // (1,0) = red
        doc.Layers.Add(src);

        var dst = new MinintLayer("dst", 16);
        doc.Layers.Add(dst);

        _fragment.CopyFragment(doc, 0, 0, 0, 2, 1, doc, 1, 2, 2, 4, 4);

        Assert.Equal(1, dst.Pixels[2 * 4 + 2]); // (2,2)
        Assert.Equal(1, dst.Pixels[2 * 4 + 3]); // (3,2)
    }
}

```

```

[Fact]
public void CopyFragment_DifferentDocuments_MergesPalette()
{
    var srcDoc = new MinintDocument("src");
    var blue = new RgbaColor(0, 0, 255, 255);
    int blueIdx = srcDoc.EnsureColorCached(blue);
    var srcLayer = new MinintLayer("L1", 4);
    srcLayer.Pixels[0] = blueIdx;
    srcDoc.Layers.Add(srcLayer);

    var dstDoc = new MinintDocument("dst");
    var dstLayer = new MinintLayer("L1", 4);
    dstDoc.Layers.Add(dstLayer);

    _fragment.CopyFragment(srcDoc, 0, 0, 0, 1, 1, dstDoc, 0, 0, 0, 2, 2);

    int dstBlueIdx = dstDoc.FindColorCached(blue);
    Assert.True(dstBlueIdx > 0);
    Assert.Equal(dstBlueIdx, dstLayer.Pixels[0]);
}

[Fact]
public void CopyFragment_TransparentPixels_Skipped()
{
    var doc = new MinintDocument("test");
    var src = new MinintLayer("src", 4); // all zeros (transparent)
    doc.Layers.Add(src);

    var dst = new MinintLayer("dst", 4);
    Array.Fill(dst.Pixels, 0);
    dst.Pixels[0] = 0; // explicitly 0
    doc.Layers.Add(dst);

    _fragment.CopyFragment(doc, 0, 0, 0, 2, 2, doc, 1, 0, 0, 2, 2);

    Assert.Equal(0, dst.Pixels[0]); // stays transparent
}
}

```

A.31. Minint.Tests/ImageEffectsTests.cs

```

using Minint.Core.Models;
using Minint.Core.Services.Impl;

namespace Minint.Tests;

public class ImageEffectsTests
{
    private readonly ImageEffectsService _effects = new();

    [Fact]
    public void ApplyGrayscale_ConvertsColors()
    {
        var doc = new MinintDocument("test");
        var red = new RgbaColor(255, 0, 0, 255);
        doc.EnsureColorCached(red);

        _effects.ApplyGrayscale(doc);

        var gray = doc.Palette[1];
        Assert.Equal(gray.R, gray.G);
        Assert.Equal(gray.G, gray.B);
        Assert.Equal(255, gray.A);
        // BT.601: 0.299*255 ≈ 76
        Assert.InRange(gray.R, 74, 78);
    }
}

```



```

}

[Fact]
public void ApplyGrayscale_PreservesTransparentIndex()
{
    var doc = new MinintDocument("test");
    doc.EnsureColorCached(new RgbaColor(100, 200, 50, 255));

    _effects.ApplyGrayscale(doc);

    Assert.Equal(RgbaColor.Transparent, doc.Palette[0]);
}

[Fact]
public void ApplyContrast_IncreasesContrast()
{
    var doc = new MinintDocument("test");
    var midGray = new RgbaColor(128, 128, 128, 255);
    var lightGray = new RgbaColor(192, 192, 192, 255);
    doc.EnsureColorCached(midGray);
    doc.EnsureColorCached(lightGray);

    _effects.ApplyContrast(doc, 2.0);

    // midGray (128) stays ~128: factor*(128-128)+128 = 128
    Assert.InRange(doc.Palette[1].R, 126, 130);
    // lightGray (192): factor*(192-128)+128 = 2*64+128 = 256 → clamped to 255
    Assert.Equal(255, doc.Palette[2].R);
}

[Fact]
public void ApplyContrast_PreservesAlpha()
{
    var doc = new MinintDocument("test");
    doc.EnsureColorCached(new RgbaColor(100, 100, 100, 200));

    _effects.ApplyContrast(doc, 1.5);

    Assert.Equal(200, doc.Palette[1].A);
}
}

```

A.32. Minint.Tests/PatternGeneratorTests.cs

```

using Minint.Core.Models;
using Minint.Core.Services;
using Minint.Core.Services.Impl;

namespace Minint.Tests;

public class PatternGeneratorTests
{
    private readonly PatternGenerator _gen = new();

    [Theory]
    [InlineData(PatternType.Checkerboard)]
    [InlineData(PatternType.HorizontalGradient)]
    [InlineData(PatternType.VerticalGradient)]
    [InlineData(PatternType.HorizontalStripes)]
    [InlineData(PatternType.VerticalStripes)]
    [InlineData(PatternType.ConcentricCircles)]
    [InlineData(PatternType.Tile)]
    public void Generate_AllTypes_ProducesValidDocument(PatternType type)
    {
        var colors = new[] { new RgbaColor(255, 0, 0, 255), new RgbaColor(0, 0, 255, 255) };
    }
}

```

```

        var doc = _gen.Generate(type, 16, 16, colors, 4, 4);

        Assert.Equal($"Pattern ({type})", doc.Name);
        Assert.Single(doc.Layers);
        Assert.Equal(256, doc.Layers[0].Pixels.Length);
        Assert.True(doc.Palette.Count >= 2);
    }

    [Fact]
    public void Generate_Checkerboard_AlternatesColors()
    {
        var colors = new[] { new RgbaColor(255, 0, 0, 255), new RgbaColor(0, 255, 0, 255) };
        var doc = _gen.Generate(PatternType.Checkerboard, 4, 4, colors, 2);

        var layer = doc.Layers[0];
        int topLeft = layer.Pixels[0];
        int topRight = layer.Pixels[2]; // cellSize=2, so (2,0) is next cell
        Assert.NotEqual(topLeft, topRight);
    }
}

```

A.33. Minint.Tests/SerializerTests.cs

```

using Minint.Core.Models;
using Minint.Infrastructure.Serialization;

namespace Minint.Tests;

public class SerializerTests
{
    private readonly MinintSerializer _serializer = new();

    [Fact]
    public void RoundTrip_EmptyDocument_PreservesStructure()
    {
        var container = new MinintContainer(32, 16);
        container.AddNewDocument("Doc1");

        var result = RoundTrip(container);

        Assert.Equal(32, result.Width);
        Assert.Equal(16, result.Height);
        Assert.Single(result.Documents);
        Assert.Equal("Doc1", result.Documents[0].Name);
        Assert.Single(result.Documents[0].Layers);
        Assert.Equal(32 * 16, result.Documents[0].Layers[0].Pixels.Length);
    }

    [Fact]
    public void RoundTrip_MultipleDocuments_PreservesAll()
    {
        var container = new MinintContainer(8, 8);
        var doc1 = container.AddNewDocument("Frame1");
        doc1.FrameDelayMs = 200;
        var doc2 = container.AddNewDocument("Frame2");
        doc2.FrameDelayMs = 500;

        var result = RoundTrip(container);

        Assert.Equal(2, result.Documents.Count);
        Assert.Equal("Frame1", result.Documents[0].Name);
        Assert.Equal(200u, result.Documents[0].FrameDelayMs);
        Assert.Equal("Frame2", result.Documents[1].Name);
        Assert.Equal(500u, result.Documents[1].FrameDelayMs);
    }
}

```

```

[Fact]
public void RoundTrip_PaletteAndPixels_Preserved()
{
    var container = new MinintContainer(4, 4);
    var doc = container.AddNewDocument("Test");
    var red = new RgbaColor(255, 0, 0, 255);
    doc.EnsureColorCached(red);

    var layer = doc.Layers[0];
    layer.Pixels[0] = 1; // red

    var result = RoundTrip(container);
    var rdoc = result.Documents[0];

    Assert.Equal(2, rdoc.Palette.Count); // transparent + red
    Assert.Equal(RgbaColor.Transparent, rdoc.Palette[0]);
    Assert.Equal(red, rdoc.Palette[1]);
    Assert.Equal(1, rdoc.Layers[0].Pixels[0]);
    Assert.Equal(0, rdoc.Layers[0].Pixels[1]);
}

[Fact]
public void RoundTrip_LayerProperties_Preserved()
{
    var container = new MinintContainer(2, 2);
    var doc = container.AddNewDocument("Test");
    doc.Layers[0].Name = "Background";
    doc.Layers[0].IsVisible = false;
    doc.Layers[0].Opacity = 128;

    var result = RoundTrip(container);
    var layer = result.Documents[0].Layers[0];

    Assert.Equal("Background", layer.Name);
    Assert.False(layer.IsVisible);
    Assert.Equal(128, layer.Opacity);
}

[Fact]
public void RoundTrip_LargePalette_Uses2ByteIndices()
{
    var container = new MinintContainer(2, 2);
    var doc = container.AddNewDocument("BigPalette");

    for (int i = 0; i < 300; i++)
        doc.EnsureColorCached(new RgbaColor((byte)(i % 256), (byte)(i / 256), 0, 255));

    Assert.Equal(2, doc.IndexByteWidth);

    int lastIdx = doc.Palette.Count - 1;
    doc.Layers[0].Pixels[0] = lastIdx;

    var result = RoundTrip(container);
    Assert.Equal(lastIdx, result.Documents[0].Layers[0].Pixels[0]);
}

[Fact]
public void Read_InvalidSignature_Throws()
{
    var ms = new MemoryStream("BADDATA"u8.ToArray());
    Assert.Throws<InvalidDataException>(() => _serializer.Read(ms));
}

[Fact]
public void Read_TruncatedStream_Throws()

```

```

{
    var ms = new MemoryStream("MINI"u8.ToArray());
    Assert.Throws<InvalidDataException>(() => _serializer.Read(ms));
}

private MinintContainer RoundTrip(MinintContainer container)
{
    var ms = new MemoryStream();
    _serializer.Write(ms, container);
    ms.Position = 0;
    return _serializer.Read(ms);
}
}

```

A.34. Minint/App.axaml.cs

```

using Avalonia;
using Avalonia.Controls.ApplicationLifetimes;
using Avalonia.Data.Core;
using Avalonia.Data.Core.Plugins;
using System.Linq;
using Avalonia.Markup.Xaml;
using Minint.ViewModels;
using Minint.Views;

namespace Minint;

public partial class App : Application
{
    public override void Initialize()
    {
        AvaloniaXamlLoader.Load(this);
    }

    public override void OnFrameworkInitializationCompleted()
    {
        if (ApplicationLifetime is IClassicDesktopStyleApplicationLifetime desktop)
        {
            // Avoid duplicate validations from both Avalonia and the CommunityToolkit.
            // More info: https://docs.avalloniaui.net/docs/guides/development-guides/data-
            validation#manage-validationplugins
            DisableAvaloniaDataAnnotationValidation();
            desktop.MainWindow = new MainWindow
            {
                DataContext = new MainWindowViewModel(),
            };
        }

        base.OnFrameworkInitializationCompleted();
    }

    private void DisableAvaloniaDataAnnotationValidation()
    {
        // Get an array of plugins to remove
        var dataValidationPluginsToRemove =
            BindingPlugins.DataValidators.OfType<DataAnnotationsValidationPlugin>().ToArray();

        // remove each entry found
        foreach (var plugin in dataValidationPluginsToRemove)
        {
            BindingPlugins.DataValidators.Remove(plugin);
        }
    }
}

```

A.35. Minint/Controls/EditableTextBlock.cs

```
using System;
using Avalonia;
using Avalonia.Controls;
using Avalonia.Input;
using Avalonia.Layout;
using Avalonia.Media;

namespace Minint.Controls;

/// <summary>
/// Shows a TextBlock by default; switches to an inline TextBox on double-click.
/// Commits on Enter or focus loss, cancels on Escape.
/// </summary>
public class EditableTextBlock : Control
{
    public static readonly StyledProperty<string> TextProperty =
        AvaloniaProperty.Register<EditableTextBlock, string>(nameof(Text), defaultBindingMode:
        ↪ Avalonia.Data.BindingMode.TwoWay);

    public string Text
    {
        get => GetValue(TextProperty);
        set => SetValue(TextProperty, value);
    }

    private readonly TextBlock _display;
    private readonly TextBox _editor;
    private bool _isEditing;

    public EditableTextBlock()
    {
        _display = new TextBlock
        {
            VerticalAlignment = VerticalAlignment.Center,
            TextTrimming = TextTrimming.CharacterEllipsis,
        };

        _editor = new TextBox
        {
            VerticalAlignment = VerticalAlignment.Center,
            Padding = new Thickness(2, 0),
            BorderThickness = new Thickness(1),
            MinWidth = 40,
            IsVisible = false,
        };

        LogicalChildren.Add(_display);
        LogicalChildren.Add(_editor);
        VisualChildren.Add(_display);
        VisualChildren.Add(_editor);

        _display.Bind(TextBlock.TextProperty, this.GetObservable(TextProperty).ToBinding());
        _editor.Bind(TextBox.TextProperty, this.GetObservable(TextProperty).ToBinding());

        _editor.KeyDown += OnEditorKeyDown;
        _editor.LostFocus += OnEditorLostFocus;
    }

    protected override void OnPointerPressed(PointerPressedEventArgs e)
    {
        base.OnPointerPressed(e);
        if (e.ClickCount == 2 && !_isEditing)
        {
            BeginEdit();
        }
    }
}
```

```

        e.Handled = true;
    }
}

private void BeginEdit()
{
    _isEditing = true;
    _editor.Text = Text;
    _display.IsVisible = false;
    _editor.IsVisible = true;
    _editor.Focus();
    _editor.SelectAll();
}

private void CommitEdit()
{
    if (!_isEditing) return;
    _isEditing = false;
    Text = _editor.Text ?? string.Empty;
    _editor.IsVisible = false;
    _display.IsVisible = true;
}

private void CancelEdit()
{
    if (!_isEditing) return;
    _isEditing = false;
    _editor.Text = Text;
    _editor.IsVisible = false;
    _display.IsVisible = true;
}

private void OnEditorKeyDown(object? sender, KeyEventArgs e)
{
    if (e.Key == Key.Enter)
    {
        CommitEdit();
        e.Handled = true;
    }
    else if (e.Key == Key.Escape)
    {
        CancelEdit();
        e.Handled = true;
    }
}

private void OnEditorLostFocus(object? sender, Avalonia.Interactivity.RoutedEventArgs e)
{
    CommitEdit();
}

protected override Size MeasureOverride(Size availableSize)
{
    _display.Measure(availableSize);
    _editor.Measure(availableSize);
    return _isEditing ? _editor.DesiredSize : _display.DesiredSize;
}

protected override Size ArrangeOverride(Size finalSize)
{
    var rect = new Rect(finalSize);
    _display.Arrange(rect);
    _editor.Arrange(rect);
    return finalSize;
}
}

```

A.36. Minint/Controls/PixelCanvas.cs

```
using System;
using System.Collections.Generic;
using Avalonia;
using Avalonia.Controls;
using Avalonia.Controls.Primitives;
using Avalonia.Input;
using Avalonia.Media;
using Avalonia.Media.Imaging;
using Avalonia.Threading;
using Minint.Core.Models;
using Minint.ViewModels;

namespace Minint.Controls;

public class PixelCanvas : Control
{
    #region Styled Properties

    public static readonly StyledProperty<WriteableBitmap?> SourceBitmapProperty =
        AvaloniaProperty.Register<PixelCanvas, WriteableBitmap?>(nameof(SourceBitmap));

    public static readonly StyledProperty<bool> ShowGridProperty =
        AvaloniaProperty.Register<PixelCanvas, bool>(nameof>ShowGrid'), defaultValue: false);

    public WriteableBitmap? SourceBitmap
    {
        get => GetValue(SourceBitmapProperty);
        set => SetValue(SourceBitmapProperty, value);
    }

    public bool ShowGrid
    {
        get => GetValue>ShowGridProperty');
        set => SetValue>ShowGridProperty', value);
    }

    #endregion

    #region Events for tool interaction

    public event Action<int, int>? ToolDown;
    public event Action<int, int>? ToolDrag;
    public event Action<(int X, int Y)?>? CursorPixelChanged;
    public Func<List<(int X, int Y)?>?> GetPreviewMask { get; set; }

    // Selection events
    public event Action<int, int>? SelectionStart;
    public event Action<int, int>? SelectionUpdate;
    public event Action<int, int>? SelectionEnd;

    // Paste events
    public event Action<int, int>? PasteMoved;
    public event Action? PasteCommitted;
    public event Action? PasteCancelled;

    /// <summary>Provides the current EditorViewModel for reading selection/paste state during
    ↪ render.</summary>
    public EditorViewModel? Editor { get; set; }

    #endregion

    private readonly Viewport _viewport = new();
    private bool _isPanning;
    private bool _isDrawing;
```

```

private bool _isSelecting;
private Point _panStart;
private double _panStartOffsetX, _panStartOffsetY;
private bool _viewportInitialized;
private int _lastBitmapWidth;
private int _lastBitmapHeight;
private (int X, int Y)? _lastCursorPixel;
private Point? _lastScreenPos;

private ScrollBar? _hScrollBar;
private ScrollBar? _vScrollBar;
private bool _suppressScrollSync;

private const double ScrollPixelsPerTick = 20.0;

public Viewport Viewport => _viewport;

static PixelCanvas()
{
    AffectsRender<PixelCanvas>(SourceBitmapProperty, ShowGridProperty);
    FocusableProperty.OverrideDefaultValue<PixelCanvas>(true);
}

public PixelCanvas()
{
    ClipToBounds = true;
}

public void AttachScrollBars(ScrollBar horizontal, ScrollBar vertical)
{
    if (_hScrollBar is not null) _hScrollBar.ValueChanged -= OnHScrollChanged;
    if (_vScrollBar is not null) _vScrollBar.ValueChanged -= OnVScrollChanged;
    _hScrollBar = horizontal;
    _vScrollBar = vertical;
    _hScrollBar.ValueChanged += OnHScrollChanged;
    _vScrollBar.ValueChanged += OnVScrollChanged;
}

#region Rendering

public override void Render(DrawingContext context)
{
    base.Render(context);
    context.FillRectangle(Brushes.Transparent, new Rect(Bounds.Size));

    var bmp = SourceBitmap;
    if (bmp is null) return;

    int imgW = bmp.PixelSize.Width;
    int imgH = bmp.PixelSize.Height;

    if (!_viewportInitialized)
    {
        _viewport.FitToView(imgW, imgH, Bounds.Width, Bounds.Height);
        _viewportInitialized = true;
    }

    DrawCheckerboard(context, imgW, imgH);

    var destRect = _viewport.ImageScreenRect(imgW, imgH);
    var srcRect = new Rect(0, 0, imgW, imgH);
    RenderOptions.SetBitmapInterpolationMode(this, BitmapInterpolationMode.None);
    context.DrawImage(bmp, srcRect, destRect);

    if (ShowGrid)
        DrawPixelGrid(context, imgW, imgH);
}

```



```

DrawToolPreview(context, imgW, imgH);
DrawSelectionOverlay(context);
DrawPastePreview(context);

int w = imgW, h = imgH;
Dispatcher.UIThread.Post(() => SyncScrollBars(w, h), DispatcherPriority.Render);
}

private void DrawCheckerboard(DrawingContext context, int imgW, int imgH)
{
    var rect = _viewport.ImageScreenRect(imgW, imgH);
    var clip = new Rect(0, 0, Bounds.Width, Bounds.Height);
    var visible = rect.Intersect(clip);
    if (visible.Width <= 0 || visible.Height <= 0) return;

    const int checkerSize = 8;
    var light = new SolidColorBrush(Color.FromRgb(204, 204, 204));
    var dark = new SolidColorBrush(Color.FromRgb(170, 170, 170));

    using (context.PushClip(visible))
    {
        context.FillRectangle(light, visible);
        double startX = visible.X - ((visible.X - rect.X) % (checkerSize * 2));
        double startY = visible.Y - ((visible.Y - rect.Y) % (checkerSize * 2));
        for (double y = startY; y < visible.Bottom; y += checkerSize)
        {
            for (double x = startX; x < visible.Right; x += checkerSize)
            {
                int col = (int)((x - rect.X) / checkerSize);
                int row = (int)((y - rect.Y) / checkerSize);
                if ((col + row) % 2 == 1)
                    context.FillRectangle(dark, new Rect(x, y, checkerSize, checkerSize));
            }
        }
    }
}

private void DrawPixelGrid(DrawingContext context, int imgW, int imgH)
{
    double zoom = _viewport.Zoom;
    if (zoom < 4) return;

    var pen = new Pen(Brushes.Black, 1);

    var clip = new Rect(0, 0, Bounds.Width, Bounds.Height);
    var imgRect = _viewport.ImageScreenRect(imgW, imgH);
    var visible = imgRect.Intersect(clip);
    if (visible.Width <= 0 || visible.Height <= 0) return;

    var (startPx, startPy) = _viewport.ScreenToPixel(visible.X, visible.Y);
    var (endPx, endPy) = _viewport.ScreenToPixel(visible.Right, visible.Bottom);
    startPx = Math.Max(0, startPx);
    startPy = Math.Max(0, startPy);
    endPx = Math.Min(imgW, endPx + 1);
    endPy = Math.Min(imgH, endPy + 1);

    using (context.PushClip(visible))
    {
        for (int px = startPx; px <= endPx; px++)
        {
            var (sx, _) = _viewport.PixelToScreen(px, 0);
            double x = Math.Floor(sx) + 0.5;
            context.DrawLine(pen, new Point(x, visible.Top), new Point(x, visible.Bottom));
        }
        for (int py = startPy; py <= endPy; py++)
    }
}

```

```

        {
            var (_, sy) = _viewport.PixelToScreen(0, py);
            double y = Math.Floor(sy) + 0.5;
            context.DrawLine(pen, new Point(visible.Left, y), new Point(visible.Right, y));
        }
    }
}

private void DrawToolPreview(DrawingContext context, int imgW, int imgH)
{
    var mask = GetPreviewMask?.Invoke();
    if (mask is null || mask.Count == 0) return;

    double zoom = _viewport.Zoom;
    var previewBrush = new SolidColorBrush(Color.FromArgb(80, 255, 255, 255));
    var outlinePen = new Pen(new SolidColorBrush(Color.FromArgb(160, 255, 255, 255)), 1);

    var clip = new Rect(0, 0, Bounds.Width, Bounds.Height);
    using (context.PushClip(clip))
    {
        foreach (var (px, py) in mask)
        {
            {
                var (sx, sy) = _viewport.PixelToScreen(px, py);
                context.FillRectangle(previewBrush, new Rect(sx, sy, zoom, zoom));
            }

            if (mask.Count > 0)
            {
                int minX = mask[0].X, maxX = mask[0].X;
                int minY = mask[0].Y, maxY = mask[0].Y;
                foreach (var (px, py) in mask)
                {
                    if (px < minX) minX = px;
                    if (px > maxX) maxX = px;
                    if (py < minY) minY = py;
                    if (py > maxY) maxY = py;
                }
                var (ox, oy) = _viewport.PixelToScreen(minX, minY);
                context.DrawRectangle(outlinePen, new Rect(ox, oy, (maxX - minX + 1) * zoom, (maxY -
↪ minY + 1) * zoom));
            }
        }
    }
}

private void DrawSelectionOverlay(DrawingContext context)
{
    var sel = Editor?.SelectionRectNormalized;
    if (sel is null) return;

    var (sx, sy, sw, sh) = sel.Value;
    double zoom = _viewport.Zoom;
    var (screenX, screenY) = _viewport.PixelToScreen(sx, sy);
    var rect = new Rect(screenX, screenY, sw * zoom, sh * zoom);

    var fillBrush = new SolidColorBrush(Color.FromArgb(40, 100, 150, 255));
    context.FillRectangle(fillBrush, rect);

    var borderPen = new Pen(new SolidColorBrush(Color.FromArgb(200, 100, 150, 255)), 1,
        new DashStyle([4, 4], 0));
    context.DrawRectangle(borderPen, rect);
}

private void DrawPastePreview(DrawingContext context)
{
    if (Editor is null || !Editor.IsPasting || Editor.Clipboard is null)
        return;
}

```

```

var pos = Editor.PastePosition!.Value;
var frag = Editor.Clipboard;
double zoom = _viewport.Zoom;

var clip = new Rect(0, 0, Bounds.Width, Bounds.Height);
using (context.PushClip(clip))
{
    for (int fy = 0; fy < frag.Height; fy++)
    {
        for (int fx = 0; fx < frag.Width; fx++)
        {
            var color = frag.Pixels[fy * frag.Width + fx];
            if (color.A == 0) continue;

            var (sx, sy) = _viewport.PixelToScreen(pos.X + fx, pos.Y + fy);
            byte dispA = (byte)(color.A * 180 / 255); // semi-transparent preview
            var brush = new SolidColorBrush(Color.FromArgb(dispA, color.R, color.G, color.B));
            context.FillRectangle(brush, new Rect(sx, sy, zoom, zoom));
        }

        // Border around the floating fragment
        var (ox, oy) = _viewport.PixelToScreen(pos.X, pos.Y);
        var borderPen = new Pen(new SolidColorBrush(Color.FromArgb(200, 255, 200, 50)), 1,
            new DashStyle([3, 3], 0));
        context.DrawRectangle(borderPen, new Rect(ox, oy, frag.Width * zoom, frag.Height * zoom));
    }
}

#endregion

#region Scrollbar Sync

private void SyncScrollBars(int imgW, int imgH)
{
    if (_hScrollBar is null || _vScrollBar is null) return;
    _suppressScrollSync = true;

    var (hMin, hMax, hVal, hView) = _viewport.GetScrollInfo(imgW, Bounds.Width, _viewport.OffsetX);
    _hScrollBar.Minimum = -hMax;
    _hScrollBar.Maximum = -hMin;
    _hScrollBar.Value = -hVal;
    _hScrollBar.ViewportSize = hView;

    var (vMin, vMax, vVal, vView) = _viewport.GetScrollInfo(imgH, Bounds.Height, _viewport.OffsetY);
    _vScrollBar.Minimum = -vMax;
    _vScrollBar.Maximum = -vMin;
    _vScrollBar.Value = -vVal;
    _vScrollBar.ViewportSize = vView;

    _suppressScrollSync = false;
}

private void OnHScrollChanged(object? sender, RangeBaseValueChangedEventArgs e)
{
    if (_suppressScrollSync) return;
    var (imgW, imgH) = GetImageSize();
    _viewport.SetOffset(-e.NewValue, _viewport.OffsetY, imgW, imgH, Bounds.Width, Bounds.Height);
    RecalcCursorPixel();
    InvalidateVisual();
}

private void OnVScrollChanged(object? sender, RangeBaseValueChangedEventArgs e)
{
    if (_suppressScrollSync) return;

```

```

        var (imgW, imgH) = GetImageSize();
        _viewport.SetOffset(_viewport.OffsetX, -e.NewValue, imgW, imgH, Bounds.Width, Bounds.Height);
        RecalcCursorPixel();
        InvalidateVisual();
    }

#endregion

#region Mouse Input

private (int W, int H) GetImageSize()
{
    var bmp = SourceBitmap;
    return bmp is not null ? (bmp.PixelSize.Width, bmp.PixelSize.Height) : (0, 0);
}

private (int X, int Y)? ScreenToPixelClamped(Point pos)
{
    var (imgW, imgH) = GetImageSize();
    if (imgW == 0) return null;
    var (px, py) = _viewport.ScreenToPixel(pos.X, pos.Y);
    if (px < 0 || px >= imgW || py < 0 || py >= imgH)
        return null;
    return (px, py);
}

private void RecalcCursorPixel()
{
    if (_lastScreenPos is null) return;
    var pixel = ScreenToPixelClamped(_lastScreenPos.Value);
    if (pixel != _lastCursorPixel)
    {
        _lastCursorPixel = pixel;
        CursorPixelChanged?.Invoke(pixel);
    }
}

protected override void OnPointerWheelChanged(PointerWheelEventArgs e)
{
    base.OnPointerWheelChanged(e);
    var (imgW, imgH) = GetImageSize();
    if (imgW == 0) return;

    bool ctrl = (e.KeyModifiers & KeyModifiers.Control) != 0;
    bool shift = (e.KeyModifiers & KeyModifiers.Shift) != 0;

    if (ctrl)
    {
        var pos = e.GetPosition(this);
        _viewport.ZoomAtPoint(pos.X, pos.Y, e.Delta.Y, imgW, imgH, Bounds.Width, Bounds.Height);
    }
    else
    {
        double dx = e.Delta.X * ScrollPixelsPerTick;
        double dy = e.Delta.Y * ScrollPixelsPerTick;
        if (shift && Math.Abs(e.Delta.X) < 0.001)
        {
            dx = dy;
            dy = 0;
        }
        _viewport.Pan(dx, dy, imgW, imgH, Bounds.Width, Bounds.Height);
    }

    RecalcCursorPixel();
    InvalidateVisual();
    e.Handled = true;
}

```

```

}

protected override void OnPointerPressed(PointerPressedEventArgs e)
{
    base.OnPointerPressed(e);
    var props = e.GetCurrentPoint(this).Properties;

    if (props.IsMiddleButtonPressed)
    {
        _isPanning = true;
        _panStart = e.GetPosition(this);
        _panStartOffsetX = _viewport.OffsetX;
        _panStartOffsetY = _viewport.OffsetY;
        e.Handled = true;
        return;
    }

    if (!props.IsLeftButtonPressed || _isPanning) return;

    var pixel = ScreenToPixelClamped(e.GetPosition(this));
    if (pixel is null) return;

    // Paste mode: left-click commits
    if (Editor is not null && Editor.IsPasting)
    {
        PasteCommitted?.Invoke();
        e.Handled = true;
        return;
    }

    // Select tool: begin rubber-band
    if (Editor is not null && Editor.ActiveTool == ToolType.Select)
    {
        _isSelecting = true;
        SelectionStart?.Invoke(pixel.Value.X, pixel.Value.Y);
        e.Handled = true;
        return;
    }

    // Regular drawing tools
    _isDrawing = true;
    ToolDown?.Invoke(pixel.Value.X, pixel.Value.Y);
    e.Handled = true;
}

protected override void OnPointerMoved(PointerEventArgs e)
{
    base.OnPointerMoved(e);
    var pos = e.GetPosition(this);
    _lastScreenPos = pos;

    if (_isPanning)
    {
        var (imgW, imgH) = GetImageSize();
        _viewport.SetOffset(
            _panStartOffsetX + (pos.X - _panStart.X),
            _panStartOffsetY + (pos.Y - _panStart.Y),
            imgW, imgH, Bounds.Width, Bounds.Height);
        RecalcCursorPixel();
        InvalidateVisual();
        e.Handled = true;
        return;
    }

    var pixel = ScreenToPixelClamped(pos);
    if (pixel != _lastCursorPixel)

```

```

{
    _lastCursorPixel = pixel;
    CursorPixelChanged?.Invoke(pixel);
}

// Paste mode: floating fragment follows cursor
if (Editor is not null && Editor.IsPasting && pixel is not null)
{
    PasteMoved?.Invoke(pixel.Value.X, pixel.Value.Y);
    InvalidateVisual();
    e.Handled = true;
    return;
}

// Selection rubber-band drag
if (_isSelecting && pixel is not null)
{
    SelectionUpdate?.Invoke(pixel.Value.X, pixel.Value.Y);
    InvalidateVisual();
    e.Handled = true;
    return;
}

if (_isDrawing && pixel is not null)
{
    ToolDrag?.Invoke(pixel.Value.X, pixel.Value.Y);
    e.Handled = true;
}
else
{
    InvalidateVisual();
}
}

protected override void OnPointerReleased(PointerReleasedEventArgs e)
{
    base.OnPointerReleased(e);

    if (_isPanning && e.InitialPressMouseButton == MouseButton.Middle)
    {
        _isPanning = false;
        e.Handled = true;
    }
    else if (_isSelecting && e.InitialPressMouseButton == MouseButton.Left)
    {
        _isSelecting = false;
        var pixel = ScreenToPixelClamped(e.GetPosition(this));
        if (pixel is not null)
            SelectionEnd?.Invoke(pixel.Value.X, pixel.Value.Y);
        InvalidateVisual();
        e.Handled = true;
    }
    else if (_isDrawing && e.InitialPressMouseButton == MouseButton.Left)
    {
        _isDrawing = false;
        e.Handled = true;
    }
}

protected override void OnPointerExited(PointerEventArgs e)
{
    base.OnPointerExited(e);
    _lastScreenPos = null;
    if (_lastCursorPixel is not null)
    {
        _lastCursorPixel = null;
    }
}

```

```

        CursorPixelChanged?.Invoke(null);
        InvalidateVisual();
    }
}

protected override void OnKeyDown(KeyEventArgs e)
{
    base.OnKeyDown(e);

    if (e.Key == Key.Escape)
    {
        if (Editor is not null && Editor.IsPasting)
        {
            PasteCancelled?.Invoke();
            InvalidateVisual();
            e.Handled = true;
        }
        else if (Editor is not null && Editor.HasSelection)
        {
            Editor.ClearSelection();
            InvalidateVisual();
            e.Handled = true;
        }
    }
    else if (e.Key == Key.Enter && Editor is not null && Editor.IsPasting)
    {
        PasteCommitted?.Invoke();
        InvalidateVisual();
        e.Handled = true;
    }
}

#endregion

protected override void OnPropertyChanged(AvaloniaPropertyChangedEventArgs change)
{
    base.OnPropertyChanged(change);
    if (change.Property == SourceBitmapProperty)
    {
        var bmp = change.GetNewValue<WriteableBitmap?>();
        int w = bmp?.PixelSize.Width ?? 0;
        int h = bmp?.PixelSize.Height ?? 0;
        if (w != _lastBitmapWidth || h != _lastBitmapHeight)
        {
            _lastBitmapWidth = w;
            _lastBitmapHeight = h;
            _viewportInitialized = false;
        }
    }
}
}

```

A.37. Minint/Controls/Viewport.cs

```

using System;
using Avalonia;

namespace Minint.Controls;

/// <summary>
/// Manages zoom level and pan offset for the pixel canvas.
/// Provides screen→pixel coordinate transforms.
/// </summary>
public sealed class Viewport
{

```

```

public double Zoom { get; set; } = 1.0;
public double OffsetX { get; set; }
public double OffsetY { get; set; }

public const double MinZoom = 0.25;
public const double MaxZoom = 128.0;

/// <summary>
/// Zoom base per 1.0 unit of wheel delta. Actual factor = Pow(base, |delta|).
/// Touchpad nudge (delta ~0.1) → ~1.01×, mouse tick (delta 1.0) → 1.10×, fast (3.0) → 1.33×.
/// </summary>
private const double ZoomBase = 1.10;

public (int X, int Y) ScreenToPixel(double screenX, double screenY) =>
    ((int)Math.Floor((screenX - OffsetX) / Zoom),
     (int)Math.Floor((screenY - OffsetY) / Zoom));

public (double X, double Y) PixelToScreen(int pixelX, int pixelY) =>
    (pixelX * Zoom + OffsetX,
     pixelY * Zoom + OffsetY);

public Rect ImageScreenRect(int imageWidth, int imageHeight) =>
    new(OffsetX, OffsetY, imageWidth * Zoom, imageHeight * Zoom);

/// <summary>
/// Zooms keeping the point under cursor fixed.
/// Uses the actual magnitude of <paramref name="delta"/> for proportional zoom.
/// </summary>
public void ZoomAtPoint(double screenX, double screenY, double delta,
                        int imageWidth, int imageHeight, double controlWidth, double controlHeight)
{
    double absDelta = Math.Abs(delta);
    double factor = delta > 0 ? Math.Pow(ZoomBase, absDelta) : 1.0 / Math.Pow(ZoomBase, absDelta);
    double newZoom = Math.Clamp(Zoom * factor, MinZoom, MaxZoom);
    if (Math.Abs(newZoom - Zoom) < 1e-12) return;

    double pixelX = (screenX - OffsetX) / Zoom;
    double pixelY = (screenY - OffsetY) / Zoom;
    Zoom = newZoom;
    OffsetX = screenX - pixelX * Zoom;
    OffsetY = screenY - pixelY * Zoom;

    ClampOffset(imageWidth, imageHeight, controlWidth, controlHeight);
}

/// <summary>
/// Pans by screen-space delta, then clamps so the image can't be scrolled out of view.
/// </summary>
public void Pan(double deltaX, double deltaY,
                int imageWidth, int imageHeight, double controlWidth, double controlHeight)
{
    OffsetX += deltaX;
    OffsetY += deltaY;
    ClampOffset(imageWidth, imageHeight, controlWidth, controlHeight);
}

/// <summary>
/// Sets offset directly (e.g. from middle-mouse drag), then clamps.
/// </summary>
public void SetOffset(double offsetX, double offsetY,
                     int imageWidth, int imageHeight, double controlWidth, double controlHeight)
{
    OffsetX = offsetX;
    OffsetY = offsetY;
    ClampOffset(imageWidth, imageHeight, controlWidth, controlHeight);
}

```



```

/// <summary>
/// Ensures at least <c>minVisible</c> pixels of the image remain on screen on each edge.
/// </summary>
public void ClampOffset(int imageWidth, int imageHeight, double controlWidth, double controlHeight)
{
    double extentW = imageWidth * Zoom;
    double extentH = imageHeight * Zoom;

    double minVisH = Math.Max(32, Math.Min(controlWidth, extentW) * 0.10);
    double minVisV = Math.Max(32, Math.Min(controlHeight, extentH) * 0.10);

    // Image right edge must be >= minVisH from left of control
    // Image left edge must be <= controlWidth - minVisH from left
    OffsetX = Math.Clamp(OffsetX, minVisH - extentW, controlWidth - minVisH);
    OffsetY = Math.Clamp(OffsetY, minVisV - extentH, controlHeight - minVisV);
}

public void FitToView(int imageWidth, int imageHeight, double controlWidth, double controlHeight)
{
    if (imageWidth <= 0 || imageHeight <= 0 || controlWidth <= 0 || controlHeight <= 0)
        return;

    double scaleX = controlWidth / imageWidth;
    double scaleY = controlHeight / imageHeight;
    Zoom = Math.Max(1.0, Math.Floor(Math.Min(scaleX, scaleY)));

    OffsetX = (controlWidth - imageWidth * Zoom) / 2.0;
    OffsetY = (controlHeight - imageHeight * Zoom) / 2.0;
}

public (double Min, double Max, double Value, double ViewportSize)
GetScrollInfo(int imageSize, double controlSize, double offset)
{
    double extent = imageSize * Zoom;
    double minVis = Math.Max(32, Math.Min(controlSize, extent) * 0.10);
    double min = minVis - extent;
    double max = controlSize - minVis;
    double viewportSize = Math.Min(controlSize, extent);
    return (min, max, offset, viewportSize);
}
}

```

A.38. Minint/Program.cs

```

using Avalonia;
using System;
using System.IO;
using Minint.Core.Models;
using Minint.Core.Services.Impl;
using Minint.Infrastructure.Serialization;

namespace Minint;

sealed class Program
{
    [STAThread]
    public static void Main(string[] args)
    {
        // TODO: remove --test branch after verification
        if (args.Length > 0 && args[0] == "--test")
        {
            RunRoundTripTest();
            RunCompositorTest();
            RunPaletteServiceTest();
        }
    }
}

```

```

        return;
    }

    BuildAvaloniaApp().StartWithClassicDesktopLifetime(args);
}

public static AppBuilder BuildAvaloniaApp()
=> AppBuilder.Configure<App>()
    .UsePlatformDetect()
    .WithInterFont()
    .LogToTrace()
    .With(new X11PlatformOptions { OverlayPopups = true });

// TODO: temporary tests – remove after verification stages.

private static void RunRoundTripTest()
{
    Console.WriteLine("=== Minint Round-Trip Test ===\n");

    var container = new MinintContainer(8, 4);

    var doc1 = container.AddNewDocument("Frame 1");
    doc1.FrameDelayMs = 200;
    doc1.Palette.Add(new RgbaColor(255, 0, 0, 255)); // idx 1 = red
    doc1.Palette.Add(new RgbaColor(0, 255, 0, 255)); // idx 2 = green
    doc1.Palette.Add(new RgbaColor(0, 0, 255, 128)); // idx 3 = semi-transparent blue
    var layer1 = doc1.Layers[0];
    for (int i = 0; i < layer1.Pixels.Length; i++)
        layer1.Pixels[i] = i % 4; // cycle 0,1,2,3

    doc1.Layers.Add(new MinintLayer("Overlay", container.PixelCount));
    var layer2 = doc1.Layers[1];
    layer2.Opacity = 128;
    layer2.Pixels[0] = 3;
    layer2.Pixels[5] = 2;

    var doc2 = container.AddNewDocument("Frame 2");
    doc2.FrameDelayMs = 150;
    doc2.Palette.Add(new RgbaColor(255, 255, 0, 255)); // idx 1 = yellow
    var layer3 = doc2.Layers[0];
    for (int i = 0; i < layer3.Pixels.Length; i++)
        layer3.Pixels[i] = i % 2;

    Console.WriteLine($"Original: {container.Width}x{container.Height}, {container.Documents.Count}
↪ docs");
    Console.WriteLine($" Doc1: palette={doc1.Palette.Count} colors, layers={doc1.Layers.Count},
↪ indexWidth={doc1.IndexByteWidth}");
    Console.WriteLine($" Doc2: palette={doc2.Palette.Count} colors, layers={doc2.Layers.Count},
↪ indexWidth={doc2.IndexByteWidth}");

    var serializer = new MinintSerializer();

    using var ms = new MemoryStream();
    serializer.Write(ms, container);
    byte[] data = ms.ToArray();
    Console.WriteLine($"Serialized: {data.Length} bytes");
    Console.WriteLine($" Signature: {System.Text.Encoding.ASCII.GetString(data, 0, 6)}");

    ms.Position = 0;
    var loaded = serializer.Read(ms);

    Assert(loaded.Width == container.Width, "Width mismatch");
    Assert(loaded.Height == container.Height, "Height mismatch");
    Assert(loaded.Documents.Count == container.Documents.Count, "Document count mismatch");

    for (int d = 0; d < container.Documents.Count; d++)

```

```

{
    var orig = container.Documents[d];
    var copy = loaded.Documents[d];
    Assert(copy.Name == orig.Name, $"Doc[{d}] name mismatch");
    Assert(copy.FrameDelayMs == orig.FrameDelayMs, $"Doc[{d}] frameDelay mismatch");
    Assert(copy.Palette.Count == orig.Palette.Count, $"Doc[{d}] palette count mismatch");

    for (int c = 0; c < orig.Palette.Count; c++)
        Assert(copy.Palette[c] == orig.Palette[c], $"Doc[{d}] palette[{c}] mismatch");

    Assert(copy.Layers.Count == orig.Layers.Count, $"Doc[{d}] layer count mismatch");

    for (int l = 0; l < orig.Layers.Count; l++)
    {
        var oLayer = orig.Layers[l];
        var cLayer = copy.Layers[l];
        Assert(cLayer.Name == oLayer.Name, $"Doc[{d}].Layer[{l}] name mismatch");
        Assert(cLayer.IsVisible == oLayer.IsVisible, $"Doc[{d}].Layer[{l}] visibility mismatch");
        Assert(cLayer.Opacity == oLayer.Opacity, $"Doc[{d}].Layer[{l}] opacity mismatch");
        Assert(cLayer.Pixels.Length == oLayer.Pixels.Length, $"Doc[{d}].Layer[{l}] pixel count
↪ mismatch");

        for (int p = 0; p < oLayer.Pixels.Length; p++)
            Assert(cLayer.Pixels[p] == oLayer.Pixels[p],
                $"Doc[{d}].Layer[{l}].Pixels[{p}] mismatch: expected {oLayer.Pixels[p]}, got
↪ {cLayer.Pixels[p]}");
    }
}

Console.WriteLine("\n✓ All assertions passed – round-trip is correct!");

// Test invalid signature
Console.WriteLine("Test: invalid signature... ");
data[0] = (byte)'X';
try
{
    serializer.Read(new MemoryStream(data));
    Console.WriteLine("FAIL (no exception)");
}
catch (InvalidDataException)
{
    Console.WriteLine("OK (InvalidDataException)");
}
data[0] = (byte)'M'; // restore

// Test truncated stream
Console.WriteLine("Test: truncated stream... ");
try
{
    serializer.Read(new MemoryStream(data, 0, 10));
    Console.WriteLine("FAIL (no exception)");
}
catch (Exception ex) when (ex is InvalidDataException or EndOfStreamException)
{
    Console.WriteLine($"OK ({ex.GetType().Name})");
}

Console.WriteLine("\n=== All tests passed ===");
}

private static void RunCompositorTest()
{
    Console.WriteLine("\n=== Compositor Test ===\n");

    var compositor = new Compositor();
    const int W = 2, H = 2;

```

```

// Document: 2 layers on a 2x2 canvas
var doc = new MinintDocument("test");
doc.Palette.Add(new RgbaColor(255, 0, 0, 255)); // idx 1 = opaque red
doc.Palette.Add(new RgbaColor(0, 0, 255, 128)); // idx 2 = semi-transparent blue

// Bottom layer: all red
var bottom = new MinintLayer("bottom", W * H);
for (int i = 0; i < bottom.Pixels.Length; i++)
    bottom.Pixels[i] = 1;
doc.Layers.Add(bottom);

// Top layer: pixel[0] = semi-blue, rest transparent
var top = new MinintLayer("top", W * H);
top.Pixels[0] = 2;
doc.Layers.Add(top);

uint[] result = compositor.Composite(doc, W, H);

// Pixel [1],[2],[3]: only bottom visible → opaque red → 0xFFFF0000 (ARGB)
uint opaqueRed = 0xFF_FF_00_00;
Assert(result[1] == opaqueRed, $"Pixel[1]: expected {opaqueRed:X8}, got {result[1]:X8}");
Assert(result[2] == opaqueRed, $"Pixel[2]: expected {opaqueRed:X8}, got {result[2]:X8}");
Assert(result[3] == opaqueRed, $"Pixel[3]: expected {opaqueRed:X8}, got {result[3]:X8}");

// Pixel [0]: red(255,0,0,255) under blue(0,0,255,128)
// srcA=128, dstA=255 → outA = 128 + 255*(255-128)/255 = 128+127 = 255
// outR = (0*128 + 255*255*(127)/255) / 255 = (0 + 255*127)/255 = 127
// outG = 0
// outB = (255*128 + 0) / 255 = 128
uint blended = result[0];
byte bA = (byte)(blended >> 24);
byte bR = (byte)((blended >> 16) & 0xFF);
byte bG = (byte)((blended >> 8) & 0xFF);
byte bB = (byte)(blended & 0xFF);

Console.WriteLine($" Blended pixel[0]: A={bA} R={bR} G={bG} B={bB}");
Assert(bA == 255, $"Pixel[0] A: expected 255, got {bA}");
Assert(bR >= 125 && bR <= 129, $"Pixel[0] R: expected ~127, got {bR}");
Assert(bG == 0, $"Pixel[0] G: expected 0, got {bG}");
Assert(bB >= 126 && bB <= 130, $"Pixel[0] B: expected ~128, got {bB}");

// Test hidden layer: hide top, result should be all red
top.IsVisible = false;
uint[] result2 = compositor.Composite(doc, W, H);
Assert(result2[0] == opaqueRed, $"Hidden top: Pixel[0] should be red, got {result2[0]:X8}");

// Test layer opacity=0: make top visible but opacity=0
top.IsVisible = true;
top.Opacity = 0;
uint[] result3 = compositor.Composite(doc, W, H);
Assert(result3[0] == opaqueRed, $"Opacity 0: Pixel[0] should be red, got {result3[0]:X8}");

// Test single transparent layer
var emptyDoc = new MinintDocument("empty");
emptyDoc.Layers.Add(new MinintLayer("bg", W * H));
uint[] result4 = compositor.Composite(emptyDoc, W, H);
Assert(result4[0] == 0, $"Empty layer: Pixel[0] should be 0x00000000, got {result4[0]:X8}");

Console.WriteLine("✓ Compositor tests passed!");
}

private static void RunPaletteServiceTest()
{
    Console.WriteLine("\n=== PaletteService Test ===\n");
}

```

```

var svc = new PaletteService();
var doc = new MinintDocument("test");

// Palette starts with [Transparent]
Assert(doc.Palette.Count == 1, "Initial palette should have 1 entry");

// EnsureColor: new color
var red = new RgbaColor(255, 0, 0, 255);
int redIdx = svc.EnsureColor(doc, red);
Assert(redIdx == 1, $"Red index: expected 1, got {redIdx}");
Assert(doc.Palette.Count == 2, $"Palette count after red: expected 2, got {doc.Palette.Count}");

// EnsureColor: same color → same index
int redIdx2 = svc.EnsureColor(doc, red);
Assert(redIdx2 == 1, $"Red re-ensure: expected 1, got {redIdx2}");
Assert(doc.Palette.Count == 2, "Palette should not grow on duplicate");

// FindColor
Assert(svc.FindColor(doc, red) == 1, "FindColor red");
Assert(svc.FindColor(doc, new RgbaColor(0, 0, 0, 255)) == -1, "FindColor missing");

// Compact: add unused color, then compact
var green = new RgbaColor(0, 255, 0, 255);
int greenIdx = svc.EnsureColor(doc, green); // idx 2
doc.Layers.Add(new MinintLayer("L", 4));
doc.Layers[0].Pixels[0] = 1; // red used
doc.Layers[0].Pixels[1] = 0; // transparent used
// green (idx 2) is NOT used by any pixel

Console.WriteLine($" Before compact: palette has {doc.Palette.Count} colors (green at idx
↪ {greenIdx})");
svc.CompactPalette(doc);
Console.WriteLine($" After compact: palette has {doc.Palette.Count} colors");

Assert(doc.Palette.Count == 2, $"After compact: expected 2 colors, got {doc.Palette.Count}");
Assert(doc.Palette[0] == RgbaColor.Transparent, "Palette[0] should be transparent");
Assert(doc.Palette[1] == red, $"Palette[1] should be red, got {doc.Palette[1]}");
Assert(doc.Layers[0].Pixels[0] == 1, "Pixel[0] should still map to red (idx 1)");

Console.WriteLine("✓ PaletteService tests passed!");
}

private static void Assert(bool condition, string message)
{
    if (!condition)
        throw new Exception($"Assertion failed: {message}");
}
}

```

A.39. Minint/ViewLocator.cs

```

using System;
using System.Diagnostics.CodeAnalysis;
using Avalonia.Controls;
using Avalonia.Controls.Templates;
using Minint.ViewModels;

namespace Minint;

/// <summary>
/// Given a view model, returns the corresponding view if possible.
/// </summary>
[RequiresUnreferencedCode(
    "Default implementation of ViewLocator involves reflection which may be trimmed away.",
    Url = "https://docs.avaloniaui.net/docs/concepts/view-locator")]

```

```

public class ViewLocator : IDataTemplate
{
    public Control? Build(object? param)
    {
        if (param is null)
            return null;

        var name = param.GetType().FullName!.Replace("ViewModel", "View", StringComparison.Ordinal);
        var type = Type.GetType(name);

        if (type != null)
        {
            return (Control)Activator.CreateInstance(type)!;
        }

        return new TextBlock { Text = "Not Found: " + name };
    }

    public bool Match(object? data)
    {
        return data is ViewModelBase;
    }
}

```

A.40. Minint/ViewModels/EditorViewModel.cs

```

using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel;
using Avalonia;
using Avalonia.Media.Imaging;
using Avalonia.Threading;
using CommunityToolkit.Mvvm.ComponentModel;
using CommunityToolkit.Mvvm.Input;
using Minint.Core.Models;
using Minint.Core.Services;
using Minint.Core.Services.Impl;

namespace Minint.ViewModels;

/// <summary>
/// Palette-independent clipboard fragment: stores resolved RGBA pixels.
/// </summary>
public sealed record ClipboardFragment(int Width, int Height, RgbaColor[] Pixels);

public partial class EditorViewModel : ViewModelBase
{
    private readonly ICompositor _compositor = new Compositor();
    private readonly IPaletteService _paletteService = new PaletteService();
    private readonly IDrawingService _drawingService = new DrawingService();
    private readonly IFloodFillService _floodFillService = new FloodFillService();
    private readonly IFragmentService _fragmentService = new FragmentService();

    [ObservableProperty]
    [NotifyPropertyChangedFor(nameof(HasContainer))]
    [NotifyPropertyChangedFor(nameof(Title))]
    private MinintContainer? _container;

    [ObservableProperty]
    private MinintDocument? _activeDocument;

    [ObservableProperty]
    private MinintLayer? _activeLayer;
}

```

```

private bool _suppressDocumentSync;

[ObservableProperty]
private WriteableBitmap? _canvasBitmap;

[ObservableProperty]
private bool _showGrid;

// Tool state
[ObservableProperty]
private ToolType _activeTool = ToolType.Brush;

[ObservableProperty]
private int _brushRadius = 1;

[ObservableProperty]
private (int X, int Y)? _previewCenter;

private Avalonia.Media.Color _previewColor = Avalonia.Media.Color.FromArgb(255, 0, 0, 0);

public Avalonia.Media.Color PreviewColor
{
    get => _previewColor;
    set
    {
        if (_previewColor == value) return;
        _previewColor = value;
        OnPropertyChanged();
        OnPropertyChanged(nameof(SelectedColor));
    }
}

public RgbaColor SelectedColor => new(_previewColor.R, _previewColor.G, _previewColor.B,
    ↪ _previewColor.A);

// Selection state (Select tool rubber-band)
[ObservableProperty]
[NotifyPropertyChangedFor(nameof(HasSelection))]
private (int X, int Y, int W, int H)? _selectionRect;

public bool HasSelection => SelectionRect is not null;

// Clipboard
[ObservableProperty]
[NotifyPropertyChangedFor(nameof(HasClipboard))]
private ClipboardFragment? _clipboard;

public bool HasClipboard => Clipboard is not null;

// Paste mode
[ObservableProperty]
[NotifyPropertyChangedFor(nameof(IsPasting))]
private (int X, int Y)? _pastePosition;

public bool IsPasting => PastePosition is not null;

[ObservableProperty]
[NotifyPropertyChangedFor(nameof(Title))]
private string? _filePath;

public bool HasContainer => Container is not null;

public string Title => FilePath is not null
    ? $"Minint - {System.IO.Path.GetFileName(FilePath)}"
    : Container is not null
        ? "Minint - Untitled"

```

```

        : "Minint";

public ObservableCollection<MinintDocument> Documents { get; } = [];
public ObservableCollection<MinintLayer> Layers { get; } = [];

#region Container / Document management

public void NewContainer(int width, int height)
{
    var c = new MinintContainer(width, height);
    c.AddNewDocument("Document 1");
    LoadContainer(c, null);
}

public void LoadContainer(MinintContainer container, string? path)
{
    Container = container;
    FilePath = path;

    SyncDocumentsList();
    SelectDocument(container.Documents.Count > 0 ? container.Documents[0] : null);
}

partial void OnActiveDocumentChanged(MinintDocument? value)
{
    if (_suppressDocumentSync) return;
    SyncLayersAndCanvas(value);
}

public void SelectDocument(MinintDocument? doc)
{
    _suppressDocumentSync = true;
    ActiveDocument = doc;
    _suppressDocumentSync = false;
    SyncLayersAndCanvas(doc);
}

public void SyncAfterExternalChange() => SyncDocumentsList();

private void SyncDocumentsList()
{
    Documents.Clear();
    if (Container is null) return;
    foreach (var doc in Container.Documents)
        Documents.Add(doc);
}

private void SyncLayersAndCanvas(MinintDocument? doc)
{
    UnsubscribeLayerVisibility();
    Layers.Clear();
    if (doc is not null)
    {
        foreach (var layer in doc.Layers)
            Layers.Add(layer);
        ActiveLayer = doc.Layers.Count > 0 ? doc.Layers[0] : null;
    }
    else
    {
        ActiveLayer = null;
    }
    SubscribeLayerVisibility();
    RefreshCanvas();
}

#endregion

```



```

#region Layer visibility change tracking

private void SubscribeLayerVisibility()
{
    foreach (var layer in Layers)
    {
        if (layer is INotifyPropertyChanged npc)
            npc.PropertyChanged += OnLayerPropertyChanged;
    }
}

private void UnsubscribeLayerVisibility()
{
    foreach (var layer in Layers)
    {
        if (layer is INotifyPropertyChanged npc)
            npc.PropertyChanged -= OnLayerPropertyChanged;
    }
}

private void OnLayerPropertyChanged(object? sender, PropertyChangedEventArgs e)
{
    if (e.PropertyName is nameof(MinintLayer.IsVisible) or nameof(MinintLayer.Opacity))
        RefreshCanvas();
}

#endregion

#region Document commands

[RelayCommand]
private void AddDocument()
{
    if (Container is null) return;
    int num = Container.Documents.Count + 1;
    var doc = Container.AddNewDocument($"Document {num}");
    Documents.Add(doc);
    SelectDocument(doc);
}

[RelayCommand]
private void RemoveDocument()
{
    if (Container is null || ActiveDocument is null) return;
    if (Container.Documents.Count <= 1) return;

    var doc = ActiveDocument;
    int idx = Container.Documents.IndexOf(doc);
    Container.Documents.Remove(doc);
    Documents.Remove(doc);

    int newIdx = Math.Min(idx, Container.Documents.Count - 1);
    SelectDocument(newIdx >= 0 ? Container.Documents[newIdx] : null);
}

[RelayCommand]
private void RenameDocument() { }

[RelayCommand]
private void MoveDocumentUp()
{
    if (Container is null || ActiveDocument is null) return;
    var doc = ActiveDocument;
    int idx = Container.Documents.IndexOf(doc);
    if (idx <= 0) return;

```

```

        (Container.Documents[idx], Container.Documents[idx - 1]) = (Container.Documents[idx - 1],
        ↪ Container.Documents[idx]);
        SyncDocumentsList();
        ReselectDocument(doc);
    }

    [RelayCommand]
    private void MoveDocumentDown()
    {
        if (Container is null || ActiveDocument is null) return;
        var doc = ActiveDocument;
        int idx = Container.Documents.IndexOf(doc);
        if (idx < 0 || idx >= Container.Documents.Count - 1) return;
        (Container.Documents[idx], Container.Documents[idx + 1]) = (Container.Documents[idx + 1],
        ↪ Container.Documents[idx]);
        SyncDocumentsList();
        ReselectDocument(doc);
    }

    private void ReselectDocument(MinintDocument doc)
    {
        // Force SelectedItem rebinding after list rebuild even when the selected object is the same
        ↪ reference.
        _suppressDocumentSync = true;
        ActiveDocument = null;
        ActiveDocument = doc;
        _suppressDocumentSync = false;
        SyncLayersAndCanvas(doc);
    }

    #endregion

    #region Layer commands

    [RelayCommand]
    private void AddLayer()
    {
        if (Container is null || ActiveDocument is null) return;
        int num = ActiveDocument.Layers.Count + 1;
        var layer = new MinintLayer($"Layer {num}", Container.PixelCount);
        ActiveDocument.Layers.Add(layer);
        Layers.Add(layer);
        ActiveLayer = layer;
        SubscribeLayerVisibility();
    }

    [RelayCommand]
    private void RemoveLayer()
    {
        if (ActiveDocument is null || ActiveLayer is null) return;
        if (ActiveDocument.Layers.Count <= 1) return;

        UnsubscribeLayerVisibility();
        var layer = ActiveLayer;
        int idx = ActiveDocument.Layers.IndexOf(layer);
        ActiveDocument.Layers.Remove(layer);
        Layers.Remove(layer);

        int newIdx = Math.Min(idx, ActiveDocument.Layers.Count - 1);
        ActiveLayer = newIdx >= 0 ? ActiveDocument.Layers[newIdx] : null;
        SubscribeLayerVisibility();
        RefreshCanvas();
    }

    [RelayCommand]
    private void MoveLayerUp()

```

```

{
    if (ActiveDocument is null || ActiveLayer is null) return;
    int idx = ActiveDocument.Layers.IndexOf(ActiveLayer);
    if (idx <= 0) return;

    UnsubscribeLayerVisibility();
    var layer = ActiveLayer;
    (ActiveDocument.Layers[idx], ActiveDocument.Layers[idx - 1]) = (ActiveDocument.Layers[idx - 1],
        ⇨ ActiveDocument.Layers[idx]);
    Layers.Move(idx, idx - 1);
    ActiveLayer = layer;
    SubscribeLayerVisibility();
    RefreshCanvas();
}

[RelayCommand]
private void MoveLayerDown()
{
    if (ActiveDocument is null || ActiveLayer is null) return;
    int idx = ActiveDocument.Layers.IndexOf(ActiveLayer);
    if (idx < 0 || idx >= ActiveDocument.Layers.Count - 1) return;

    UnsubscribeLayerVisibility();
    var layer = ActiveLayer;
    (ActiveDocument.Layers[idx], ActiveDocument.Layers[idx + 1]) = (ActiveDocument.Layers[idx + 1],
        ⇨ ActiveDocument.Layers[idx]);
    Layers.Move(idx, idx + 1);
    ActiveLayer = layer;
    SubscribeLayerVisibility();
    RefreshCanvas();
}

[RelayCommand]
private void DuplicateLayer()
{
    if (Container is null || ActiveDocument is null || ActiveLayer is null) return;
    var src = ActiveLayer;
    var dup = new MinintLayer(src.Name + " copy", src.IsVisible, src.Opacity,
        ⇨ (int[])src.Pixels.Clone());
    int idx = ActiveDocument.Layers.IndexOf(src) + 1;
    ActiveDocument.Layers.Insert(idx, dup);

    UnsubscribeLayerVisibility();
    Layers.Insert(idx, dup);
    ActiveLayer = dup;
    SubscribeLayerVisibility();
    RefreshCanvas();
}

#endregion

#region Drawing

public void OnToolDown(int px, int py)
{
    if (IsPlaying) return;
    if (Container is null || ActiveDocument is null || ActiveLayer is null)
        return;

    int w = Container.Width, h = Container.Height;
    if (px < 0 || px >= w || py < 0 || py >= h)
        return;

    if (ActiveTool == ToolType.Select) return; // handled separately

    switch (ActiveTool)

```

```

{
    case ToolType.Brush:
    {
        int colorIdx = ActiveDocument.EnsureColorCached(SelectedColor);
        _drawingService.ApplyBrush(ActiveLayer, px, py, BrushRadius, colorIdx, w, h);
        break;
    }
    case ToolType.Eraser:
        _drawingService.ApplyEraser(ActiveLayer, px, py, BrushRadius, w, h);
        break;
    case ToolType.Fill:
    {
        int colorIdx = ActiveDocument.EnsureColorCached(SelectedColor);
        _floodFillService.Fill(ActiveLayer, px, py, colorIdx, w, h);
        break;
    }
}

RefreshCanvas();
}

public void OnToolDrag(int px, int py)
{
    if (IsPlaying) return;
    if (ActiveTool is ToolType.Fill or ToolType.Select) return;
    OnToolDown(px, py);
}

public List<(int X, int Y)>? GetPreviewMask()
{
    if (PreviewCenter is null || Container is null)
        return null;
    if (ActiveTool is ToolType.Fill or ToolType.Select)
        return null;

    var (cx, cy) = PreviewCenter.Value;
    return _drawingService.GetBrushMask(cx, cy, BrushRadius, Container.Width, Container.Height);
}

[RelayCommand]
private void SelectBrush() { CancelPasteMode(); ActiveTool = ToolType.Brush; }

[RelayCommand]
private void SelectEraser() { CancelPasteMode(); ActiveTool = ToolType.Eraser; }

[RelayCommand]
private void SelectFill() { CancelPasteMode(); ActiveTool = ToolType.Fill; }

[RelayCommand]
private void SelectSelectTool() { CancelPasteMode(); ActiveTool = ToolType.Select; }

[RelayCommand]
private void ToggleGrid() => ShowGrid = !ShowGrid;

#endregion

#region Selection + Copy/Paste (A4)

/// <summary>Called by PixelCanvas when selection drag starts.</summary>
public void BeginSelection(int px, int py)
{
    if (IsPlaying) return;
    SelectionRect = (px, py, 0, 0);
}

/// <summary>Called by PixelCanvas as the user drags.</summary>

```

```

public void UpdateSelection(int px, int py)
{
    if (SelectionRect is null) return;
    var s = SelectionRect.Value;
    int x = Math.Min(s.X, px);
    int y = Math.Min(s.Y, py);
    int w = Math.Abs(px - s.X) + 1;
    int h = Math.Abs(py - s.Y) + 1;
    // Store normalized rect but keep original anchor in _selAnchor
    _selectionRectNormalized = (x, y, w, h);
}

/// <summary>Called by PixelCanvas when mouse is released.</summary>
public void FinishSelection(int px, int py)
{
    if (SelectionRect is null) return;
    var s = SelectionRect.Value;
    int x0 = Math.Min(s.X, px);
    int y0 = Math.Min(s.Y, py);
    int rw = Math.Abs(px - s.X) + 1;
    int rh = Math.Abs(py - s.Y) + 1;
    if (Container is not null)
    {
        x0 = Math.Max(0, x0);
        y0 = Math.Max(0, y0);
        rw = Math.Min(rw, Container.Width - x0);
        rh = Math.Min(rh, Container.Height - y0);
    }
    if (rw <= 0 || rh <= 0)
    {
        SelectionRect = null;
        _selectionRectNormalized = null;
        return;
    }
    SelectionRect = (x0, y0, rw, rh);
    _selectionRectNormalized = SelectionRect;
}

private (int X, int Y, int W, int H)? _selectionRectNormalized;

/// <summary>The normalized (positive W/H, clamped) selection rectangle for rendering.</summary>
public (int X, int Y, int W, int H)? SelectionRectNormalized => _selectionRectNormalized;

[RelayCommand]
private void CopySelection()
{
    if (SelectionRect is null || ActiveDocument is null || ActiveLayer is null || Container is null)
        return;

    var (sx, sy, sw, sh) = SelectionRect.Value;
    int cw = Container.Width;
    var palette = ActiveDocument.Palette;
    var srcPixels = ActiveLayer.Pixels;
    var buf = new RgbaColor[sw * sh];

    for (int dy = 0; dy < sh; dy++)
    {
        int srcRow = sy + dy;
        for (int dx = 0; dx < sw; dx++)
        {
            int srcCol = sx + dx;
            int idx = srcPixels[srcRow * cw + srcCol];
            buf[dy * sw + dx] = idx < palette.Count ? palette[idx] : RgbaColor.Transparent;
        }
    }
}

```

```

        Clipboard = new ClipboardFragment(sw, sh, buf);
    }

    [RelayCommand]
    private void PasteClipboard()
    {
        if (Clipboard is null) return;
        PastePosition = (0, 0);
    }

    public void MovePaste(int px, int py)
    {
        if (IsPlaying || !IsPasting) return;
        PastePosition = (px, py);
    }

    [RelayCommand]
    public void CommitPaste()
    {
        if (IsPlaying) return;
        if (!IsPasting || Clipboard is null || ActiveDocument is null || ActiveLayer is null ||
            ↪ Container is null)
            return;

        var (px, py) = PastePosition!.Value;
        int cw = Container.Width, ch = Container.Height;
        var frag = Clipboard;
        var dstPixels = ActiveLayer.Pixels;

        for (int fy = 0; fy < frag.Height; fy++)
        {
            int dy = py + fy;
            if (dy < 0 || dy >= ch) continue;
            for (int fx = 0; fx < frag.Width; fx++)
            {
                int dx = px + fx;
                if (dx < 0 || dx >= cw) continue;
                var color = frag.Pixels[fy * frag.Width + fx];
                if (color.A == 0) continue; // skip transparent
                int colorIdx = ActiveDocument.EnsureColorCached(color);
                dstPixels[dy * cw + dx] = colorIdx;
            }
        }

        PastePosition = null;
        SelectionRect = null;
        _selectionRectNormalized = null;
        RefreshCanvas();
    }

    [RelayCommand]
    public void CancelPaste()
    {
        PastePosition = null;
    }

    public void ClearSelection()
    {
        SelectionRect = null;
        _selectionRectNormalized = null;
    }

    private void CancelPasteMode()
    {
        PastePosition = null;
        SelectionRect = null;
    }

```

```

        _selectionRectNormalized = null;
    }

#endregion

#region Canvas rendering

public void RefreshCanvas()
{
    if (Container is null || ActiveDocument is null)
    {
        CanvasBitmap = null;
        return;
    }

    int w = Container.Width;
    int h = Container.Height;
    uint[] argb = _compositor.Composite(ActiveDocument, w, h);

    var bmp = new WriteableBitmap(
        new PixelSize(w, h),
        new Vector(96, 96),
        Avalonia.Platform.PixelFormat.Bgra8888);

    using (var fb = bmp.Lock())
    {
        unsafe
        {
            var dst = new Span<uint>((void*)fb.Address, w * h);
            for (int i = 0; i < argb.Length; i++)
            {
                uint px = argb[i];
                byte a = (byte)(px >> 24);
                byte r = (byte)((px >> 16) & 0xFF);
                byte g = (byte)((px >> 8) & 0xFF);
                byte b = (byte)(px & 0xFF);

                if (a == 255)
                {
                    dst[i] = px;
                }
                else if (a == 0)
                {
                    dst[i] = 0;
                }
                else
                {
                    r = (byte)(r * a / 255);
                    g = (byte)(g * a / 255);
                    b = (byte)(b * a / 255);
                    dst[i] = (uint)(b | (g << 8) | (r << 16) | (a << 24));
                }
            }
        }
    }

    CanvasBitmap = bmp;
}

#endregion

#region Animation playback

private DispatcherTimer? _animationTimer;
private int _animationFrameIndex;

```

```

[ObservableProperty]
[NotifyPropertyChangedFor(nameof(IsNotPlaying))]
private bool _isPlaying;

public bool IsNotPlaying => !IsPlaying;

[RelayCommand]
private void PlayAnimation()
{
    if (Container is null || Container.Documents.Count < 2) return;
    if (IsPlaying) return;

    IsPlaying = true;
    _animationFrameIndex = ActiveDocument is not null
        ? Container.Documents.IndexOf(ActiveDocument)
        : 0;
    if (_animationFrameIndex < 0) _animationFrameIndex = 0;

    AdvanceAnimationFrame();
}

[RelayCommand]
private void StopAnimation()
{
    _animationTimer?.Stop();
    _animationTimer = null;
    IsPlaying = false;
    if (ActiveDocument is not null)
        SyncLayersAndCanvas(ActiveDocument);
}

private void AdvanceAnimationFrame()
{
    if (Container is null || !IsPlaying)
    {
        StopAnimation();
        return;
    }

    var docs = Container.Documents;
    if (docs.Count == 0)
    {
        StopAnimation();
        return;
    }

    _animationFrameIndex %= docs.Count;
    var doc = docs[_animationFrameIndex];

    _suppressDocumentSync = true;
    ActiveDocument = doc;
    _suppressDocumentSync = false;
    RefreshCanvasFor(doc);

    uint delay = doc.FrameDelayMs;
    if (delay < 10) delay = 10;

    _animationTimer?.Stop();
    _animationTimer = new DispatcherTimer { Interval = TimeSpan.FromMilliseconds(delay) };
    _animationTimer.Tick += (_, _) =>
    {
        _animationTimer?.Stop();
        _animationFrameIndex++;
        AdvanceAnimationFrame();
    };
    _animationTimer.Start();
}

```



```

    }

    private void RefreshCanvasFor(MinintDocument doc)
    {
        if (Container is null)
        {
            CanvasBitmap = null;
            return;
        }

        int w = Container.Width;
        int h = Container.Height;
        uint[] argb = _compositor.Composite(doc, w, h);

        var bmp = new WriteableBitmap(
            new PixelSize(w, h),
            new Vector(96, 96),
            Avalonia.Platform.PixelFormat.Bgra8888);

        using (var fb = bmp.Lock())
        {
            unsafe
            {
                var dst = new Span<uint>((void*)fb.Address, w * h);
                for (int i = 0; i < argb.Length; i++)
                {
                    uint px = argb[i];
                    byte a2 = (byte)(px >> 24);
                    byte r2 = (byte)((px >> 16) & 0xFF);
                    byte g2 = (byte)((px >> 8) & 0xFF);
                    byte b2 = (byte)(px & 0xFF);

                    if (a2 == 255) { dst[i] = px; }
                    else if (a2 == 0) { dst[i] = 0; }
                    else
                    {
                        r2 = (byte)(r2 * a2 / 255);
                        g2 = (byte)(g2 * a2 / 255);
                        b2 = (byte)(b2 * a2 / 255);
                        dst[i] = (uint)(b2 | (g2 << 8) | (r2 << 16) | (a2 << 24));
                    }
                }
            }
        }

        CanvasBitmap = bmp;
    }

    #endregion

    public ICompositor Compositor => _compositor;
    public IPaletteService PaletteService => _paletteService;
    public IDrawingService DrawingService => _drawingService;
    public IFragmentService FragmentService => _fragmentService;
}

```

A.41. Minint/ViewModels/MainWindowViewModel.cs

```

using System;
using System.Collections.Generic;
using System.IO;
using System.Threading.Tasks;
using Avalonia.Controls;
using Avalonia.Platform.Storage;
using CommunityToolkit.Mvvm.ComponentModel;

```

```

using CommunityToolkit.Mvvm.Input;
using Minint.Core.Services;
using Minint.Core.Services.Impl;
using Minint.Infrastructure.Export;
using Minint.Infrastructure.Serialization;
using Minint.Views;

namespace Minint.ViewModels;

public partial class MainWindowViewModel : ViewModelBase
{
    private readonly MinintSerializer _serializer = new();
    private readonly IImageEffectsService _effects = new ImageEffectsService();
    private readonly IPatternGenerator _patternGen = new PatternGenerator();
    private readonly IBmpExporter _bmpExporter = new BmpExporter();
    private readonly IGifExporter _gifExporter = new GifExporter();
    private readonly ICompositor _compositor = new Compositor();

    private static readonly FilePickerFileType MinintFileType = new("Minint Files")
    {
        Patterns = ["*.minint"],
    };

    private static readonly FilePickerFileType BmpFileType = new("BMP Image")
    {
        Patterns = ["*.bmp"],
    };

    private static readonly FilePickerFileType GifFileType = new("GIF Animation")
    {
        Patterns = ["*.gif"],
    };

    [ObservableProperty]
    private EditorViewModel _editor = new();

    [ObservableProperty]
    private string _statusText = "Ready";

    public TopLevel? Owner { get; set; }

    #region File commands

    [RelayCommand]
    private async Task NewFileAsync()
    {
        if (Owner is not Window window) return;

        var dialog = new NewContainerDialog();
        var result = await dialog.ShowDialog<bool?>(window);
        if (result != true) return;

        int w = dialog.CanvasWidth;
        int h = dialog.CanvasHeight;
        Editor.NewContainer(w, h);
        StatusText = $"New {w}x{h} container created.";
    }

    [RelayCommand]
    private async Task OpenFileAsync()
    {
        if (Owner?.StorageProvider is not { } sp) return;

        var files = await sp.OpenFilePickerAsync(new FilePickerOpenOptions
        {
            Title = "Open .minint file",

```

```

        FileTypeFilter = [MinintFileType],
        AllowMultiple = false,
    });

    if (files.Count == 0) return;

    var file = files[0];
    try
    {
        await using var stream = await file.OpenReadAsync();
        var container = _serializer.Read(stream);
        var path = file.TryGetLocalPath();
        Editor.LoadContainer(container, path);
        StatusText = $"Opened {file.Name}";
    }
    catch (Exception ex)
    {
        StatusText = $"Error opening file: {ex.Message}";
    }
}

[RelayCommand]
private async Task SaveFileAsync()
{
    if (Editor.Container is null) return;

    if (Editor.FilePath is not null)
        await SaveToPathAsync(Editor.FilePath);
    else
        await SaveFileAsAsync();
}

[RelayCommand]
private async Task SaveFileAsAsync()
{
    if (Owner?.StorageProvider is not { } sp || Editor.Container is null) return;

    var file = await sp.SaveFilePickerAsync(new FilePickerSaveOptions
    {
        Title = "Save .minint file",
        DefaultExtension = "minint",
        FileTypeChoices = [MinintFileType],
        SuggestedFileName = Editor.FilePath is not null
            ? Path.GetFileName(Editor.FilePath) : "untitled.minint",
    });

    if (file is null) return;

    var path = file.TryGetLocalPath();
    if (path is null)
    {
        StatusText = "Error: could not resolve file path.";
        return;
    }

    await SaveToPathAsync(path);
}

private async Task SaveToPathAsync(string path)
{
    try
    {
        await using var fs = File.Create(path);
        _serializer.Write(fs, Editor.Container!);
        Editor.FilePath = path;
        StatusText = $"Saved {Path.GetFileName(path)}";
    }
}

```

```

    }
    catch (Exception ex)
    {
        StatusText = $"Error saving file: {ex.Message}";
    }
}

#endregion

#region Effects (A1, A2)

[RelayCommand]
private async Task ApplyContrastAsync()
{
    if (Editor.ActiveDocument is null || Owner is not Window window) return;

    var dialog = new ContrastDialog();
    var result = await dialog.ShowDialog<bool?>(window);
    if (result != true) return;

    _effects.ApplyContrast(Editor.ActiveDocument, dialog.Factor);
    Editor.RefreshCanvas();
    StatusText = $"Contrast ×{dialog.Factor:F1} applied.";
}

[RelayCommand]
private void ApplyGrayscale()
{
    if (Editor.ActiveDocument is null) return;

    _effects.ApplyGrayscale(Editor.ActiveDocument);
    Editor.RefreshCanvas();
    StatusText = "Grayscale applied.";
}

#endregion

#region Pattern generation (B4)

[RelayCommand]
private async Task GeneratePatternAsync()
{
    if (Editor.Container is null || Owner is not Window window) return;

    var dialog = new PatternDialog();
    var result = await dialog.ShowDialog<bool?>(window);
    if (result != true) return;

    try
    {
        var doc = _patternGen.Generate(
            dialog.SelectedPattern,
            Editor.Container.Width,
            Editor.Container.Height,
            [dialog.PatternColor1, dialog.PatternColor2],
            dialog.PatternParam1,
            dialog.PatternParam2);

        Editor.Container.Documents.Add(doc);
        Editor.SyncAfterExternalChange();
        Editor.SelectDocument(doc);
        StatusText = $"Pattern '{dialog.SelectedPattern}' generated.";
    }
    catch (Exception ex)
    {
        StatusText = $"Pattern generation failed: {ex.Message}";
    }
}

```

```

    }
}

#endregion

#region Export (BMP / GIF)

[RelayCommand]
private async Task ExportBmpAsync()
{
    if (Editor.Container is null || Editor.ActiveDocument is null) return;
    if (Owner?.StorageProvider is not { } sp) return;

    var file = await sp.SaveFilePickerAsync(new FilePickerSaveOptions
    {
        Title = "Export document as BMP",
        DefaultExtension = "bmp",
        FileTypeChoices = [BmpFileType],
        SuggestedFileName = $"{Editor.ActiveDocument.Name}.bmp",
    });
    if (file is null) return;

    var path = file.TryGetLocalPath();
    if (path is null) { StatusText = "Error: could not resolve file path."; return; }

    try
    {
        int w = Editor.Container.Width, h = Editor.Container.Height;
        uint[] argb = _compositor.Composite(Editor.ActiveDocument, w, h);
        await using var fs = File.Create(path);
        _bmpExporter.Export(fs, argb, w, h);
        StatusText = $"Exported BMP: {Path.GetFileName(path)}";
    }
    catch (Exception ex)
    {
        StatusText = $"BMP export failed: {ex.Message}";
    }
}

[RelayCommand]
private async Task ExportGifAsync()
{
    if (Editor.Container is null || Editor.Container.Documents.Count == 0) return;
    if (Owner?.StorageProvider is not { } sp) return;

    var file = await sp.SaveFilePickerAsync(new FilePickerSaveOptions
    {
        Title = "Export animation as GIF",
        DefaultExtension = "gif",
        FileTypeChoices = [GifFileType],
        SuggestedFileName = Editor.FilePath is not null
            ? Path.GetFileNameWithoutExtension(Editor.FilePath) + ".gif"
            : "animation.gif",
    });
    if (file is null) return;

    var path = file.TryGetLocalPath();
    if (path is null) { StatusText = "Error: could not resolve file path."; return; }

    try
    {
        int w = Editor.Container.Width, h = Editor.Container.Height;
        var frames = new List<(uint[] Pixels, uint DelayMs)>();
        foreach (var doc in Editor.Container.Documents)
        {
            uint[] argb = _compositor.Composite(doc, w, h);

```

```

        frames.Add((argb, doc.FrameDelayMs));
    }

    await using var fs = File.Create(path);
    _gifExporter.Export(fs, frames, w, h);
    StatusText = $"Exported GIF ({frames.Count} frames): {Path.GetFileName(path)}";
}
catch (Exception ex)
{
    StatusText = $"GIF export failed: {ex.Message}";
}
}

#endregion
}

```

A.42. Minint/ViewModels/ToolType.cs

```

namespace Minint.ViewModels;

public enum ToolType
{
    Brush,
    Eraser,
    Fill,
    Select
}

```

A.43. Minint/ViewModels/ToolTypeConverters.cs

```

using System;
using System.Globalization;
using Avalonia.Data.Converters;

namespace Minint.ViewModels;

/// <summary>
/// Static IValueConverter instances for binding RadioButton.IsChecked to ToolType.
/// These are one-way (read-only) – the RadioButton Command sets the actual value.
/// </summary>
public static class ToolTypeConverters
{
    public static readonly IValueConverter IsBrush = new ToolTypeConverter(ToolType.Brush);
    public static readonly IValueConverter IsEraser = new ToolTypeConverter(ToolType.Eraser);
    public static readonly IValueConverter IsFill = new ToolTypeConverter(ToolType.Fill);
    public static readonly IValueConverter IsSelect = new ToolTypeConverter(ToolType.Select);

    private sealed class ToolTypeConverter(ToolType target) : IValueConverter
    {
        public object Convert(object? value, Type targetType, object? parameter, CultureInfo culture)
            => value is ToolType t && t == target;

        public object ConvertBack(object? value, Type targetType, object? parameter, CultureInfo
            culture)
            => target;
    }
}

```

A.44. Minint/ViewModels/ViewModelBase.cs

```

using CommunityToolkit.Mvvm.ComponentModel;

namespace Minint.ViewModels;

```

```
public abstract class ViewModelBase : ObservableObject
{
}

```

A.45. Minint/Views/ContrastDialog.axaml.cs

```
using System;
using Avalonia.Controls;
using Avalonia.Interactivity;

namespace Minint.Views;

public partial class ContrastDialog : Window
{
    public double Factor => FactorSlider.Value;

    public ContrastDialog()
    {
        InitializeComponent();

        FactorSlider.PropertyChanged += (_, e) =>
        {
            if (e.Property == Slider.ValueProperty)
                FactorLabel.Text = FactorSlider.Value.ToString("F1");
        };

        OkButton.Click += (_, _) => Close(true);
        CancelButton.Click += (_, _) => Close(false);
    }
}

```

A.46. Minint/Views/MainWindow.axaml.cs

```
using System;
using Avalonia.Controls;
using Avalonia.Controls.Primitives;
using Minint.Controls;
using Minint.ViewModels;

namespace Minint.Views;

public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

    protected override void OnOpened(EventArgs e)
    {
        base.OnOpened(e);

        var canvas = this.FindControl<PixelCanvas>("Canvas");
        var hScroll = this.FindControl<ScrollBar>("HScroll");
        var vScroll = this.FindControl<ScrollBar>("VScroll");

        if (canvas is not null && hScroll is not null && vScroll is not null)
            canvas.AttachScrollBars(hScroll, vScroll);

        if (canvas is not null && DataContext is MainWindowViewModel vm)
            WireCanvasEvents(canvas, vm.Editor);
    }

    protected override void OnDataContextChanged(EventArgs e)
    {
    }
}

```

```

        base.OnDataContextChanged(e);
        if (DataContext is MainWindowViewModel vm)
            vm.Owner = this;
    }

    private static void WireCanvasEvents(PixelCanvas canvas, EditorViewModel editor)
    {
        canvas.Editor = editor;

        canvas.ToolDown += (px, py) => editor.OnToolDown(px, py);
        canvas.ToolDrag += (px, py) => editor.OnToolDrag(px, py);
        canvas.CursorPixelChanged += pixel => editor.PreviewCenter = pixel;
        canvas.GetPreviewMask = () => editor.GetPreviewMask();

        canvas.SelectionStart += (px, py) => editor.BeginSelection(px, py);
        canvas.SelectionUpdate += (px, py) => editor.UpdateSelection(px, py);
        canvas.SelectionEnd += (px, py) => { editor.FinishSelection(px, py); canvas.InvalidateVisual();
↵    };

        canvas.PasteMoved += (px, py) => editor.MovePaste(px, py);
        canvas.PasteCommitted += () => { editor.CommitPaste(); canvas.InvalidateVisual(); };
        canvas.PasteCancelled += () => { editor.CancelPaste(); canvas.InvalidateVisual(); };
    }
}

```

A.47. Minint/Views/NewContainerDialog.axaml.cs

```

using Avalonia.Controls;

namespace Minint.Views;

public partial class NewContainerDialog : Window
{
    public int CanvasWidth => (int)(WidthInput.Value ?? 64);
    public int CanvasHeight => (int)(HeightInput.Value ?? 64);

    public NewContainerDialog()
    {
        InitializeComponent();

        OkButton.Click += (_, _) => Close(true);
        CancelButton.Click += (_, _) => Close(false);
    }
}

```

A.48. Minint/Views/PatternDialog.axaml.cs

```

using System;
using System.Linq;
using Avalonia.Controls;
using Avalonia.Interactivity;
using Avalonia.Media;
using Minint.Core.Models;
using Minint.Core.Services;

namespace Minint.Views;

public partial class PatternDialog : Window
{
    public PatternType SelectedPattern =>
        PatternCombo.SelectedItem is PatternType pt ? pt : PatternType.Checkerboard;

    public RgbaColor PatternColor1
    {
        get

```



```

    {
        var c = Color1Picker.Color;
        return new RgbaColor(c.R, c.G, c.B, c.A);
    }
}

public RgbaColor PatternColor2
{
    get
    {
        var c = Color2Picker.Color;
        return new RgbaColor(c.R, c.G, c.B, c.A);
    }
}

public int PatternParam1 => (int)(Param1.Value ?? 8);
public int PatternParam2 => (int)(Param2.Value ?? 8);

public PatternDialog()
{
    InitializeComponent();

    PatternCombo.ItemsSource = Enum.GetValues<PatternType>().ToList();
    PatternCombo.SelectedIndex = 0;

    Color1Picker.Color = Color.FromRgb(0, 0, 0);
    Color2Picker.Color = Color.FromRgb(255, 255, 255);

    OkButton.Click += (_, _) => Close(true);
    CancelButton.Click += (_, _) => Close(false);
}
}

```