

LiveCode Minimum Drawing Library User Manual

Table of Contents

- 1 Introduction
- 2 Getting Started
- 3 Core Concepts
- 4 Main Functions
- 5 Drawing Tools
- 6 Color Management
- 7 Text and Number Rendering
- 8 Visual Effects
- 9 Utility Functions
- 10 Debugging

LiveCode Minimum Drawing Library User Manual

Chapter 1: Introduction

The LiveCode Minimum Drawing Library (MDL) is a powerful and versatile toolkit designed to enhance graphical capabilities within LiveCode projects. Originally developed by UDI and adapted for broader use, this library provides developers with a comprehensive set of tools for creating, manipulating, and rendering graphics efficiently.

Key features of the MDL include:

- Low-level pixel manipulation
- Shape drawing (lines, rectangles, ovals, diamonds, stars)
- Text rendering with customizable vector fonts
- Color management and blending
- Visual effects for transitions
- Utility functions for coordinate and rectangle manipulations

Whether you're creating simple diagrams or complex animations, the MDL offers the flexibility and performance to bring your visual ideas to life within the LiveCode environment.

Chapter 2: Getting Started

2.1 Installation

To incorporate the MDL into your LiveCode project:

- 1 Open your LiveCode stack script.
- 2 Copy the entire MDL script.
- 3 Paste the script into your stack script area.
- 4 Save your stack.

2.2 Basic Setup

Before you can start drawing, you need to set up your environment:

- 1 Create an image object:
 - In your LiveCode stack, create a new image object.
 - Name this image object "image 1" (or choose a name of your preference).
 - This image will serve as your drawing canvas.

- 2 Initialize the drawing buffer: `livecode`

Copy

```
MdMakeBuffer "image 1"
```

This command creates a drawing buffer based on the dimensions of your image object.

- 3 Set initial colors (optional): `livecode`

Copy

```
MdSetPenColor "255,0,0" -- Set pen color to red
```

- 4 `MdSetBgColor "255,255,255" -- Set background to white`

2.3 Basic Drawing Example

Here's a simple example to draw a red rectangle:

`livecode`

Copy

```
-- Initialize buffer
MdMakeBuffer "image 1"

-- Set color and draw
MdSetPenColor "255,0,0"
MdDrawRect "50,50,150,150", true
```

```
-- Update the image  
MdUpdate "image 1"
```

2.4 Tips for Efficient Use

- Always call `MdUpdate` after your drawing operations to see the results.
- Use `MdPushBuffer` and `MdPopBuffer` to save and restore buffer states for complex drawings.
- Experiment with different visual effects in `MdUpdate` for smooth transitions.

Chapter 3: Core Concepts

Understanding the core concepts of the MDL is crucial for effective use of the library.

3.1 The Drawing Buffer

The drawing buffer is the heart of the MDL. It's an in-memory representation of your image, where all drawing operations are performed.

Key points about the buffer:

- It's created with `MdMakeBuffer` or `MdNewBuffer`.
- Its size matches the dimensions of your image object.
- All drawing operations modify this buffer, not the image directly.
- The buffer content is applied to the image when you call `MdUpdate`.

3.2 Coordinate System

The MDL uses a coordinate system where:

- The origin (0,0) is at the top-left corner of the image.
- X-coordinates increase from left to right.
- Y-coordinates increase from top to bottom.

3.3 Color Representation

Colors in the MDL can be specified in several formats:

- RGB: "255,255,255" (white)
- RGBA: "255,255,255,128" (semi-transparent white)
- Hexadecimal: "#FFFFFF" (white)

The library provides functions to work with colors, including blending and conversion between formats.

3.4 Drawing Primitives

The MDL offers various primitive shapes for drawing:

- Pixels: The smallest unit, set with `MdSetPixel`.

- Lines: Drawn with `MdDrawLine`.
- Rectangles: Created using `MdDrawRect`.
- Ovals and Circles: Rendered with `MdDrawOval`.
- More complex shapes: Diamonds, stars, etc.

These primitives form the building blocks for more complex graphics.

3.5 Text and Number Rendering

The MDL includes a custom vector font system for rendering text and numbers. This system allows for flexible text drawing without relying on system fonts.

Understanding these core concepts will provide a solid foundation for using the Minimum Drawing Library effectively in your LiveCode projects.

Chapter 4: Main Functions

The MDL provides several core functions that form the foundation of its functionality. Understanding these functions is crucial for effective use of the library.

4.1 Buffer Management

MdMakeBuffer pImageName

Initializes the drawing buffer based on an existing image object.

livecode

Copy

```
MdMakeBuffer "image 1"
```

MdNewBuffer pWidth, pHeight

Creates a new buffer with specified dimensions.

livecode

Copy

```
MdNewBuffer 800, 600
```

MdUpdate pImageName, pEffect, pSpeed

Updates the image object with the current buffer content. Optionally applies visual effects.

livecode

Copy

```
MdUpdate "image 1"
MdUpdate "image 1", "pushLeft", "fast"
```

4.2 Buffer Manipulation

MdPushBuffer

Saves the current buffer state to temporary memory.

livecode

Copy

MdPushBuffer

MdPopBuffer

Restores the buffer state from temporary memory.

livecode

Copy

MdPopBuffer

4.3 Buffer Information

MdGetBufferWidth

Returns the width of the current buffer.

MdGetBufferHeight

Returns the height of the current buffer.

MdGetBufferRect

Returns the dimensions of the buffer as a rectangle.

Chapter 5: Drawing Tools

The MDL offers a variety of drawing tools for creating both simple and complex graphics.

5.1 Line Drawing

MdDrawLine pPoints

Draws a line between specified points.

livecode

Copy

```
MdDrawLine "0,0,100,100"
```

MdEraseLine pPoints

Erases a line using the background color.

5.2 Shape Drawing

MdDrawRect pRect, pFillBool

Draws a rectangle. If pFillBool is true, the rectangle is filled.

livecode

Copy

```
MdDrawRect "50,50,150,150", true
```

MdDrawOval pRect, pFillBool

Draws an oval or circle within the specified rectangle.

livecode

Copy

```
MdDrawOval "50,50,150,150", false
```

MdDrawDia pRect, pFillBool

Draws a diamond shape.

MdDrawStar pRect, pFillBool

Draws a star shape.

5.3 Pixel Operations

MdSetPixel pLoc

Sets a single pixel at the specified location.

livecode

Copy

```
MdSetPixel "100,100"
```

MdErasePixel pLoc

Erases a single pixel (sets it to the background color).

MdGetPixel pLoc

Returns the color of the pixel at the specified location.

5.4 Complex Drawing

MdDrawPix pSrcPix, pDstLoc

Draws a pixel array (pix) onto the buffer at the specified location.

MdDrawPixPat pSrcPix, pDstRect

Draws a pixel array as a repeating pattern within the specified rectangle.

Chapter 6: Color Management

Effective color management is crucial for creating visually appealing graphics. The MDL provides several functions for working with colors.

6.1 Setting Colors

MdSetPenColor pRGBN

Sets the current pen color for drawing operations.

livecode

Copy

```
MdSetPenColor "255,0,0" -- Set pen color to red
MdSetPenColor "#00FF00" -- Set pen color to green using hex
```

MdSetBgColor pRGBN

Sets the current background color.

livecode

Copy

```
MdSetBgColor "0,0,255" -- Set background color to blue
```

6.2 Getting Colors

MdGetPenColor

Returns the current pen color.

MdGetBgColor

Returns the current background color.

6.3 Color Blending

MdGetBlendColor pColor1, pColor2, pBlend

Returns a color that is a blend between two specified colors.

livecode

Copy

```
put MdGetBlendColor("255,0,0", "0,0,255", 5) into tPurple
-- Returns a 50% blend between red and blue
```

MdGetBlendColorBin pColor1, pColor2, pBlend

Similar to MdGetBlendColor, but works with binary color data for improved performance.

6.4 Alpha Channel Manipulation

MdSetPixAlpha @rPix, pAlphaValue, pRect

Sets the alpha value for a specified region of a pixel array.

MdSetBufferAlpha pAlphaValue, pRect

Sets the alpha value for a specified region of the current buffer.

Understanding these color management functions allows for sophisticated color effects and blending in your graphics projects.

Chapter 7: Text and Number Rendering

The MDL provides custom functions for rendering text and numbers, offering flexibility and control over text display.

7.1 Text Rendering

MdDrawText pText, pLoc, pTextLen, pTextStyle

Draws text using a built-in vector font.

Parameters:

- **pText**: The text to draw
- **pLoc**: Starting location "x,y"
- **pTextLen**: Maximum text length (for alignment)
- **pTextStyle**: Styling options (e.g., "bold", "center", "X:1.5", "Y:2")

livecode

Copy

```
MdDrawText "Hello, World!", "10,20", 15, "bold center"
```

7.2 Number Rendering

MdDrawNumber pN, pLoc, pTextLen, pWeight, pAlign

Draws numbers using a 7-segment display style.

Parameters:

- **pN**: The number to draw
- **pLoc**: Starting location "x,y"
- **pTextLen**: Maximum text length (for alignment)
- **pWeight**: Font weight (0-3)
- **pAlign**: Alignment ("right" or left by default)

livecode

Copy

```
MdDrawNumber 12345, "50,50", 5, 2, "right"
```

7.3 Custom Segment Display

MdDrawSegment pPattern, pLoc

Draws a custom 7-segment display pattern.

livecode

Copy

```
MdDrawSegment "1347", "100,100" -- Draws a "7"
```

Chapter 8: Visual Effects

The MDL includes a variety of visual effects that can be applied when updating the image, allowing for smooth transitions and animations.

8.1 Available Effects

MdGetEffectList

Returns a list of all available visual effects.

8.2 Applying Effects

Effects are applied using the `MdUpdate` function:

livecode

Copy

```
MdUpdate "image 1", "pushLeft", "fast"
```

8.3 Categories of Effects

- 1 Push Effects: "pushUp", "pushDown", "pushLeft", "pushRight"
- 2 Scroll Effects: "scrollUp", "scrollDown", "scrollLeft", "scrollRight"
- 3 Wipe Effects: "wipeUp", "wipeDown", "wipeLeft", "wipeRight", "wipeOpen", "wipeClose"
- 4 Reveal Effects: "revealUp", "revealDown", "revealLeft", "revealRight"
- 5 Square Effects: "squareOpen", "squareClose"
- 6 Clam Effects: "clamOpen", "clamClose"
- 7 Dissolve Effects: "dissolve", "dissolveNight", "dissolveDay"

8.4 Customizing Effect Speed

The speed parameter in `MdUpdate` can be set to "slow", "fast", or omitted for normal speed.

Chapter 9: Utility Functions

The MDL provides several utility functions to assist with common operations in graphics programming.

9.1 Rectangle Manipulation

MdPackRect pX1, pY1, pX2, pY2

Creates a rect string from individual coordinates.

MdUnpackRect pRect, @rX1, @rY1, @rX2, @rY2

Extracts individual coordinates from a rect string.

MdOffsetRect @rRect, pX, pY

Moves a rectangle by the specified offset.

MdInsetRect @rRect, pX, pY

Shrinks or enlarges a rectangle from its center.

MdZoomRect @rRect, pZoomX, pZoomY

Scales a rectangle while keeping its top-left corner fixed.

9.2 Point Operations

MdPtInRect pLoc, pRect

Checks if a point is inside a given rectangle.

MdOffsetLoc @rLoc, pX, pY

Moves a point by the specified offset.

9.3 Data Manipulation

MdPackData pLen, pData

Repeats data to fill a specified length.

MdSwap @p1, @p2

Swaps the values of two variables.

MdShuffleItems @rItems

Randomly shuffles a list of items.

MdGetShuffleItems pN

Returns a shuffled list of numbers from 1 to N.

9.4 Help Function

MdGetHelp

Returns a simple help text for the MDL functions.

These utility functions can significantly simplify many common tasks in graphics programming, making your code more efficient and readable.

Chapter 10: Debugging

Effective debugging is crucial for developing robust graphics applications. The MDL includes built-in debugging capabilities to help you identify and resolve issues in your code.

10.1 Debug Logging

debugLog pMessage

Outputs debug messages to a field named "DebugOutput" in your LiveCode stack.

livecode

Copy

```
debugLog "Drawing rectangle at coordinates: 50,50,150,150"
```

To use this function effectively:

- 1 Create a field named "DebugOutput" in your LiveCode stack.
- 2 Call `debugLog` at key points in your code to track the execution flow and state of variables.

10.2 Best Practices for Debugging

- 1 **Logging Buffer Operations:** Log the dimensions and key properties of your buffer when initializing or modifying it. livecode

Copy

```
debugLog "Buffer initialized: " & MdGetBufferWidth() & "x" & MdGetBufferHeight()
```

- 2 **Tracking Drawing Operations:** Log the parameters of drawing operations to ensure they're being called with the expected values. livecode

Copy

```
debugLog "Drawing line from " & item 1 to 2 of pPoints & " to " & item 3 to 4 of pPoints
```

- 3 **Monitoring Color Changes:** Log color changes to track unexpected color shifts in your graphics. `livecode`

Copy

```
debugLog "Pen color set to: " & MdGetPenColor()
```

- 4 **Checking Effect Applications:** When applying visual effects, log the effect name and speed to verify the correct effect is being used. `livecode`

Copy

```
debugLog "Applying effect: " & pEffect & " at speed: " & pSpeed
```

10.3 Common Debugging Scenarios

- 1 **Invisible Graphics:** If your drawings aren't visible, check:
 - Buffer dimensions match your expectations
 - Pen color differs from background color
 - Drawing coordinates are within the buffer bounds
- 2 **Unexpected Colors:** Verify:
 - Correct color values are being passed to color setting functions
 - Alpha channel values (if applicable)
- 3 **Performance Issues:** Look for:
 - Excessive buffer manipulations (push/pop operations)
 - Inefficient use of drawing primitives
- 4 **Effect Glitches:** Ensure:
 - Effect names are spelled correctly
 - Buffer and image dimensions match

10.4 Advanced Debugging Techniques

- 1 **Visual Debugging:** Use `MdDrawRect` or `MdDrawLine` to visualize bounding boxes or paths in your graphics.
- 2 **State Logging:** At key points in your code, log the entire state of your drawing environment: `livecode`

Copy

```
command logDrawingState  
3 debugLog "=== Drawing State ==="
```

```
4    debugLog "Buffer: " & MdGetBufferWidth() & "x" &
    MdGetBufferHeight()
5    debugLog "Pen Color: " & MdGetPenColor()
6    debugLog "BG Color: " & MdGetBgColor()
7    debugLog "=====
8    end logDrawingState
```

- 9 Incremental Drawing:** For complex graphics, update the image after each major component is drawn to isolate issues.

10.5 Optimizing Your Graphics Code

Use the debugging tools not just for fixing errors, but also for optimizing your graphics routines:

- 1 Identify redundant drawing operations.
- 2 Look for opportunities to use more efficient primitives (e.g., `MdDrawPixPat` instead of multiple `MdDrawRect` calls).
- 3 Minimize buffer push/pop operations where possible.

By mastering the debugging capabilities of the MDL, you'll be able to create more robust and efficient graphics code, leading to smoother, more responsive LiveCode applications.

This concludes our comprehensive manual for the LiveCode Minimum Drawing Library. With these tools and techniques at your disposal, you're well-equipped to create sophisticated graphics in your LiveCode projects. Remember that practice and experimentation are key to mastering these concepts. Happy coding!