Gateless Gate: Interactive Shape Editing

Right-Click Edit Mode for the Erae Touch II Layout Editor

Feature Implementation Report — Thesis Appendix

George Redpath

NTNU / Aalborg University

February 26, 2026

Abstract

This document describes the design and implementation of the **Edit Shape** mode in the Erae Shape Editor, a JUCE-based visual layout editor for the Erae Touch II multi-touch surface. The feature allows users to right-click any existing shape and interactively modify its geometry through simultaneous cell painting, erasing, and handle-based resizing—all without mode switching. Non-pixel shapes (rectangles, circles, hexagons, polygons) are transparently auto-converted to pixel representations on first edit. The implementation spans four source files, introduces a new undo action type, and adds a `replaceShape()` primitive to the layout model. This work forms part of the *Gateless Gate* project, a modular Eurorack synthesizer system combining FPGA-based DSP with touch-based control surfaces.

1 Context: The Gateless Gate Project

The Gateless Gate is a 104 HP Eurorack modular synthesizer built around a Zynq-7020 FPGA (Zybo Z7-20 board) containing 302 SystemVerilog DSP modules, 56 audio channels (28 ADC + 28 DAC), and a patent-pending 1-Wire automatic patch topology detection system. The project encompasses approximately 116,000 lines of code across FPGA gateware, embedded firmware, and desktop software.

The **Erae Shape Editor** serves as a complementary touch-based control interface. Where the Gateless Gate hardware uses physical rotary encoders and illuminated jacks, the Erae Touch II provides a 42×24 grid multi-touch surface capable of continuous pressure and position tracking. The editor application—built as a JUCE VST3/Standalone plugin—lets musicians design custom touch layouts with shapes, colors, MIDI behaviors, and visual feedback styles, then push them to the hardware surface over USB SysEx.

2 Motivation

Prior to this work, the editor supported five shape types—Rectangle, Circle, Hexagon, Polygon, and Pixel—each created through dedicated drawing tools. However, **modifying an existing shape's geometry** required deleting it and redrawing from scratch, losing all associated behavior configuration (MIDI mappings, velocity curves, visual style, z-ordering).

Musicians frequently need to fine-tune shape boundaries: adjusting pad sizes by a pixel or two, carving notches for adjacent shapes, or adding extensions to irregular freeform layouts. The lack of in-place editing was a significant workflow friction point.

The solution is an **Edit Shape** mode that provides three simultaneous editing actions without mode switching:

1. **Paint** cells onto the shape (left-click/drag)
2. **Erase** cells from the shape (right-click/drag)
3. **Resize** via drag handles (corners and edges)

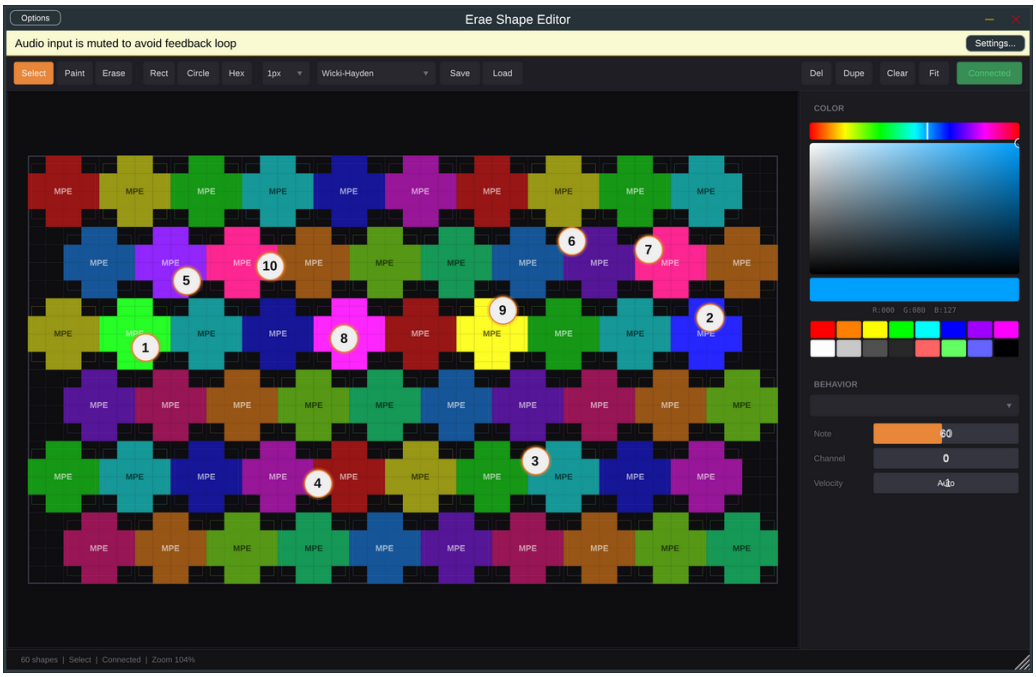

Figure 1: Erae Shape Editor showing a Wicki-Hayden isomorphic keyboard layout with 10-finger multitouch tracking, per-finger color palette, and real-time hardware rendering.

3 Interaction Design

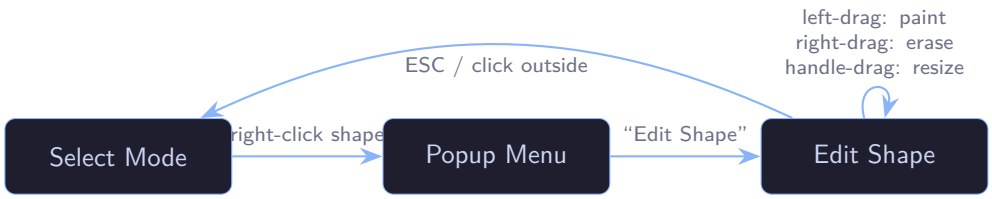

Figure 2: State transition diagram for the Edit Shape interaction.

The interaction flow is:

1. In **Select mode**, the user right-clicks a shape. A context menu appears with a single option: *Edit Shape*.

2. Selecting it enters **Edit Shape mode** for that specific shape. The canvas displays:
   - A per-cell grid overlay highlighting every cell belonging to the shape
   - Eight resize handles (four corners, four edge midpoints)
   - An "EDIT SHAPE (ESC to finish)" indicator

3. The user modifies the shape freely:
   - **Left-click/drag**: adds grid cells to the shape (live update)
   - **Right-click/drag**: removes grid cells from the shape (live update)
   - **Handle drag**: scales all cells proportionally to fit the new bounding box

4. **ESC** or clicking > 3 cells outside the bounding box commits the edit and returns to Select mode.

5. **Ctrl+Z** reverses the entire editing session as a single undo step.

4 Auto-Conversion Strategy

A key design decision is handling non-pixel shapes. Rectangles, circles, hexagons, and polygons store geometry parametrically (origin, width/height, radius, vertex list), not as discrete cells. When the user paints or erases a single cell, the shape must be converted to a `PixelShape` to support arbitrary cell-level editing.

Table 1: Auto-conversion behavior by edit action type.

Action	Converts to Pixel?	Reason
Paint cell	Yes (on first cell edit)	Cell-level granularity requires pixel storage
Erase cell	Yes (on first cell edit)	Removing a cell breaks parametric geometry
Handle resize	Yes (cells are scaled)	Scaled cells no longer fit parametric form

The conversion captures the shape's current `gridPixels()` output—the rasterized set of integer grid coordinates—and constructs a `PixelShape` with equivalent cells. All visual properties (color, active color, behavior, behavior parameters, z-order, visual style) are preserved through the conversion.

Undo reverses the conversion entirely, restoring the original parametric shape.

5 Architecture

The implementation touches four files across three layers of the application architecture:

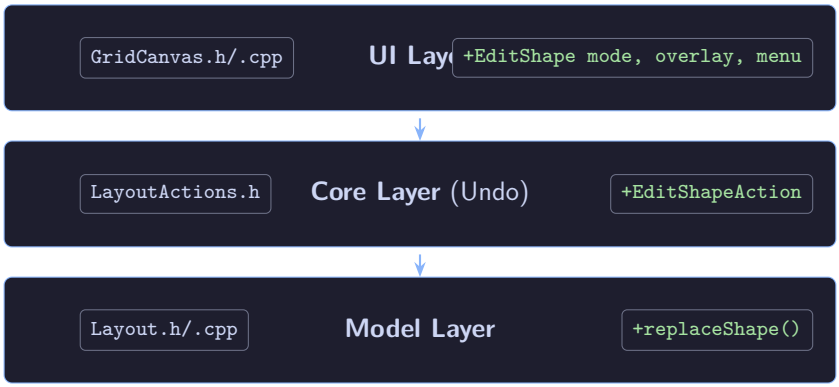

Figure 3: Architectural layers and files modified. Green text indicates additions.

5.1 Model Layer: `Layout.h/.cpp`

A single new method was added to the `Layout` class:

Listing 1: The replaceShape method preserves z-order position.

```
void Layout::replaceShape(const std::string& id,
                          std::unique_ptr<Shape> newShape)
{
    for (auto& s : shapes_) {
        if (s->id == id) {
            s = std::move(newShape);
            sortByZOrder();
            notifyListeners();
            return;
        }
```

```
    }
}
```

This swaps a shape in-place by ID, preserving its position in the shape vector (and thus its z-order after re-sorting). This is preferable to `extractShape()` followed by `addShape()`, which would lose the shape's position among peers of equal z-order.

5.2 Core Layer: `EditShapeAction`

A new undo action stores cloned copies of both the original and final shape states:

Listing 2: EditShapeAction uses clones for repeatable perform/undo.

```
class EditShapeAction : public UndoableAction {
    Layout& layout_;
    std::string id_;
    std::unique_ptr<Shape> oldShape_;  // clone of original
    std::unique_ptr<Shape> newShape_;  // clone of edited result

    void perform() override {
        layout_.replaceShape(id_, newShape_->clone());
    }
    void undo() override {
        layout_.replaceShape(id_, oldShape_->clone());
    }
};
```

Cloning on every perform/undo ensures the action remains valid across multiple undo/redo cycles. The memory cost is minimal—each `PixelShape` stores only a vector of (x, y) integer pairs.

5.3 UI Layer: `GridCanvas`

The bulk of the implementation lives in the canvas component. The `ToolMode` enum gains a new value:

```
enum class ToolMode {
    Select, Paint, Erase, DrawRect, DrawCircle,
    DrawHex, DrawPoly, DrawPixel, EditShape
};
```

Five new member variables track the editing session:

Table 2: Edit-mode state variables.

Variable	Type	Purpose
`editingShapeId_`	`string`	ID of the shape being edited (empty = not editing)
`editOrigShape_`	`unique_ptr<Shape>`	Clone of shape before any edits (for undo)
`editCells_`	`set<pair<int,int>>`	Current cell set in absolute grid coordinates
`editConverted_`	`bool`	Whether we auto-converted from a parametric shape
`editDraggingHandle_`	`HandlePos`	Which resize handle is being dragged

6 Key Implementation Details

6.1 Live Synchronization

Every paint or erase action immediately synchronizes the in-memory cell set back to the layout model via `syncEditCellsToShape()`. This method:

1. Computes the bounding box origin $(\min x, \min y)$ from `editCells_`
2. Builds relative cell coordinates: $(c_x - \min x,\ c_y - \min y)$
3. If the shape is not yet a `PixelShape`, constructs one and calls `replaceShape()`
4. If already a `PixelShape`, updates `relCells` in-place and notifies listeners

This provides immediate visual feedback: the shape redraws on every cell change, and widgets (pressure glow, fill bars) continue to render correctly because the layout model stays consistent.

6.2 Handle-Based Scaling

When the user drags a resize handle, all cells are proportionally scaled to fit the new bounding box. The algorithm:

1. Compute the old bounding box (x_0, y_0, w_0, h_0) from current cells
2. Compute the new bounding box (x_1, y_1, w_1, h_1) from the handle delta
3. For each cell (c_x, c_y), map to the new box:

$$c'_x = \lfloor x_1 + (c_x - x_0) \cdot w_1 / w_0 \rfloor, \quad c'_y = \lfloor y_1 + (c_y - y_0) \cdot h_1 / h_0 \rfloor$$

4. Clamp to grid bounds $(0 \le c' < 42,\ 0 \le c' < 24)$

This produces nearest-neighbor scaling of the cell pattern, which preserves shape topology while allowing arbitrary aspect ratio changes through all eight handle positions.

6.3 Context Menu

The right-click context menu uses JUCE's asynchronous popup system to avoid blocking the message thread:

Listing 3: Asynchronous popup menu with lambda callback.

```
juce::PopupMenu menu;
menu.addItem(1, "Edit Shape");
menu.showMenuAsync(
    juce::PopupMenu::Options().withTargetScreenArea(
        juce::Rectangle<int>(e.getScreenX(),
                             e.getScreenY(), 1, 1)),
    [this, shapeId](int result) {
        if (result == 1)
            enterEditMode(shapeId);
    });
```

The lambda captures the shape ID by value, ensuring correctness even if the layout changes between menu display and selection.

7 Changes Summary

8 Relation to Thesis

This feature contributes to the thesis in two dimensions:

Table 3: Summary of all file modifications.

File	Changes
`Layout.h`	Added `replaceShape()` declaration
`Layout.cpp`	Implemented `replaceShape()` (10 lines)
`LayoutActions.h`	Added `EditShapeAction` class (28 lines)
`GridCanvas.h`	Added `EditShape` enum value, 5 state variables, 6 method declarations
`GridCanvas.cpp`	Right-click menu, `enterEditMode()`, `exitEditMode()`, `syncEditCellsToShape()`, `editHitTestHandle()`, `editBBoxScreen()`, `drawEditModeOverlay()`, mouse/key handling (∼200 lines)
`PluginEditor.cpp`	Added `EditShape` case to status bar switch
`README.md`	Documented Edit Shape mode and keyboard shortcuts

Interface Design for Electronic Music. The Edit Shape mode demonstrates a *modeless editing paradigm* where paint, erase, and resize are available simultaneously through different mouse buttons and handle affordances. This contrasts with traditional modal tool selection (the existing toolbar) and provides a more fluid creative workflow for fine-tuning touch layouts. The approach is informed by pixel art editors (Aseprite, Piskel) adapted to the constraints of a grid-quantized musical control surface.

Software Architecture. The implementation follows the existing command pattern for undo/redo, demonstrating how a complex multi-action editing session (potentially hundreds of cell changes) collapses into a single undoable transaction. The auto-conversion from parametric to pixel representation is transparent to the user and fully reversible, illustrating the principle that *the model should serve the interaction, not constrain it.*

9 Future Work

- **Brush size in edit mode**: Currently fixed at 1 cell; could respect the global brush size setting
- **Non-destructive resize**: Keep parametric shapes parametric when only resizing (no cell paint/erase)
- **Selection within edit mode**: Rectangular selection of cells for move/copy within the shape
- **Symmetry tools**: Mirror paint strokes horizontally/vertically for symmetric pad layouts