Floor Plan Shape Implementation Guide

Overview

This document provides a comprehensive technical explanation of how complex floor plan shapes (L-Shaped, H-Shaped, M-Shaped) are implemented in the ArchitectPro application. The implementation uses a section-based approach combined with proportional room allocation algorithms to ensure zero overlapping and uniform distribution.

Architecture Design

Core Concepts

1. **Sections**: Complex shapes are decomposed into rectangular sections
2. **Room Allocation**: Rooms are assigned to sections based on spatial requirements
3. **Proportional Distribution**: Percentage-based sizing ensures consistent layouts
4. **Boundary Enforcement**: Strict dimension clamping prevents overlapping

System Components

```
lib/
├── types.ts              # TypeScript interfaces
├── shape-layout-utils.ts # Core shape generation algorithms
└── floor-plan-utils.ts   # Utility functions

app/designer/_components/
├── floor-plan-canvas.tsx # SVG rendering with shape overlay
├── controls-panel.tsx    # User interface controls
└── designer-workspace.tsx # State management
```

Algorithm Implementation

Phase 1: Section Generation

The `generateSections()` function decomposes a shape into rectangular sections:

```
export function generateSections(
  shape: FloorPlanShape,
  totalWidth: number,
  totalHeight: number
): Section[] {
  switch (shape) {
    case 'Rectangular':
      return generateRectangularSections(totalWidth, totalHeight);
    case 'L-Shaped':
      return generateLShapedSections(totalWidth, totalHeight);
    case 'H-Shaped':
      return generateHShapedSections(totalWidth, totalHeight);
    case 'M-Shaped':
      return generateMShapedSections(totalWidth, totalHeight);
    default:
      return generateRectangularSections(totalWidth, totalHeight);
  }
}
```

L-Shaped Section Logic

```
function generateLShapedSections(
  totalWidth: number,
  totalHeight: number
): Section[] {
  // Horizontal section spans full width, 60% of height
  const horizontalHeight = totalHeight * 0.6;

  // Vertical section is 60% of width, 40% of height
  const verticalWidth = totalWidth * 0.6;
  const verticalHeight = totalHeight * 0.4;

  return [
    {
      id: 'horizontal',
      x: 0,
      y: 0,
      width: totalWidth,
      height: horizontalHeight,
      name: 'Horizontal Wing',
    },
    {
      id: 'vertical',
      x: 0,
      y: horizontalHeight,
      width: verticalWidth,
      height: verticalHeight,
      name: 'Vertical Wing',
    },
  ];
}
```

Key Parameters:

- Horizontal section: 100% width × 60% height
- Vertical section: 60% width × 40% height
- Sections are positioned to form an L-shape

H-Shaped Section Logic

```
function generateHShapedSections(
  totalWidth: number,
  totalHeight: number
): Section[] {
  const wingWidth = totalWidth * 0.3;
  const bridgeWidth = totalWidth * 0.4;
  const bridgeHeight = totalHeight * 0.3;
  const bridgeOffset = totalHeight * 0.35;

  return [
    {
      id: 'left-wing',
      x: 0,
      y: 0,
      width: wingWidth,
      height: totalHeight,
      name: 'Left Wing',
    },
    {
      id: 'bridge',
      x: wingWidth,
      y: bridgeOffset,
      width: bridgeWidth,
      height: bridgeHeight,
      name: 'Central Bridge',
    },
    {
      id: 'right-wing',
      x: wingWidth + bridgeWidth,
      y: 0,
      width: wingWidth,
      height: totalHeight,
      name: 'Right Wing',
    },
  ];
}
```

Key Parameters:

- Left wing: 30% width × 100% height
- Bridge: 40% width × 30% height (centered vertically)
- Right wing: 30% width × 100% height

M-Shaped Section Logic

```
function generateMShapedSections(
  totalWidth: number,
  totalHeight: number
): Section[] {
  const wingWidth = totalWidth * 0.25;
  const centerWidth = totalWidth * 0.5;
  const wingHeight = totalHeight * 0.6;

  return [
    {
      id: 'center',
      x: wingWidth,
      y: 0,
      width: centerWidth,
      height: totalHeight,
      name: 'Central Section',
    },
    {
      id: 'left-wing',
      x: 0,
      y: 0,
      width: wingWidth,
      height: wingHeight,
      name: 'Left Wing',
    },
    {
      id: 'right-wing',
      x: wingWidth + centerWidth,
      y: 0,
      width: wingWidth,
      height: wingHeight,
      name: 'Right Wing',
    },
  ];
}
```

Key Parameters:
- Central section: 50% width × 100% height
- Left wing: 25% width × 60% height
- Right wing: 25% width × 60% height

Phase 2: Room Placement

The `placeRoomsInShape()` function assigns rooms to sections:

```
export function placeRoomsInShape(
  shape: FloorPlanShape,
  sections: Section[],
  totalWidth: number,
  totalHeight: number
): Room[] {
  switch (shape) {
    case 'Rectangular':
      return placeRoomsRectangular(sections[0], 1);
    case 'L-Shaped':
      return placeRoomsLShaped(sections, 1);
    case 'H-Shaped':
      return placeRoomsHShaped(sections, 1);
    case 'M-Shaped':
      return placeRoomsMShaped(sections, 1);
    default:
      return [];
  }
}
```

Anti-Overlap Algorithm for L-Shaped Layout

```
function placeRoomsLShaped(sections: Section[], startId: number): Room[] {
  const rooms: Room[] = [];
  const padding = 0.5; // 0.5m spacing between rooms
  const horizontal = sections[0];
  const vertical = sections[1];

  // Step 1: Calculate available space (subtract padding)
  const horizontalAvailWidth = horizontal.width - padding * 2;
  const horizontalAvailHeight = horizontal.height - padding * 2;
  const verticalAvailWidth = vertical.width - padding * 2;
  const verticalAvailHeight = vertical.height - padding * 2;

  // Step 2: Allocate percentage-based widths
  const livingWidth = horizontalAvailWidth * 0.55; // 55% for living
  const kitchenWidth = horizontalAvailWidth * 0.40; // 40% for kitchen

  // Step 3: Calculate room dimensions with architectural constraints
  const livingDims = calculateRoomDimensions('living', livingWidth, horizontalAvail-
Height);
  const actualLivingWidth = Math.min(livingDims.width, livingWidth);
  const actualLivingHeight = Math.min(livingDims.height, horizontalAvailHeight);

  // Step 4: Place living room with padding
  rooms.push({
    id: `room-${startId++}`,
    name: 'Living Room',
    type: 'living',
    x: horizontal.x + padding,
    y: horizontal.y + padding,
    width: actualLivingWidth,
    height: actualLivingHeight,
    color: '#BFDBFE',
    floor: 1,
  });

  // Step 5: Calculate kitchen dimensions
  const kitchenDims = calculateRoomDimensions('kitchen', kitchenWidth, horizonta-
lAvailHeight);
  const actualKitchenWidth = Math.min(kitchenDims.width, kitchenWidth);
  const actualKitchenHeight = Math.min(kitchenDims.height, horizontalAvailHeight);

  // Step 6: Place kitchen AFTER living room (sequential positioning)
  rooms.push({
    id: `room-${startId++}`,
    name: 'Kitchen',
    type: 'kitchen',
    x: horizontal.x + actualLivingWidth + padding * 2, // Ensures gap
    y: horizontal.y + padding,
    width: actualKitchenWidth,
    height: actualKitchenHeight,
    color: '#FDE68A',
    floor: 1,
  });

  // Step 7: Place garage in vertical section
  const garageWidth = verticalAvailWidth * 0.65;
  const garageDims = calculateRoomDimensions('garage', garageWidth, verticalAvail-
Height);
  const actualGarageWidth = Math.min(garageDims.width, garageWidth);
  const actualGarageHeight = Math.min(garageDims.height, verticalAvailHeight);

  rooms.push({
```

```
    id: `room-${startId++}`,
    name: 'Garage',
    type: 'garage',
    x: vertical.x + padding,
    y: vertical.y + padding,
    width: actualGarageWidth,
    height: actualGarageHeight,
    color: '#9CA3AF',
    floor: 1,
  });

  // Step 8: Place maid's room AFTER garage
  const maidWidth = verticalAvailWidth * 0.30;
  const maidDims = calculateRoomDimensions('maid', maidWidth, verticalAvailHeight);
  const actualMaidWidth = Math.min(maidDims.width, maidWidth);
  const actualMaidHeight = Math.min(maidDims.height, verticalAvailHeight);

  rooms.push({
    id: `room-${startId++}`,
    name: "Maid's Room",
    type: 'maid-room',
    x: vertical.x + actualGarageWidth + padding * 2, // Ensures gap
    y: vertical.y + padding,
    width: actualMaidWidth,
    height: actualMaidHeight,
    color: '#E9D5FF',
    floor: 1,
  });

  return rooms;
}
```

Anti-Overlap Mechanisms:

1. **Padding System**: 0.5m minimum gap between all rooms
2. **Available Space Calculation**: Subtracts padding from section dimensions before allocation
3. **Percentage-Based Sizing**: Fixed percentages prevent size conflicts
4. **Dimension Clamping**: `Math.min()` ensures rooms never exceed allocated space
5. **Sequential Positioning**: `x = previousX + previousWidth + padding * 2` prevents overlaps

Phase 3: Room Dimension Calculation

The `calculateRoomDimensions()` function enforces architectural standards:

```
export function calculateRoomDimensions(
  roomKey: string,
  availableWidth: number,
  availableHeight: number
): { width: number; height: number; area: number } {
  const requirements = REQUIRED_ROOMS[roomKey];
  if (!requirements) {
    return { width: availableWidth, height: availableHeight, area: 0 };
  }

  const availableArea = availableWidth * availableHeight;
  const minArea = requirements.minWidth * requirements.minHeight;

  // Use minimum dimensions if space is tight
  if (availableArea <= minArea * 1.2) {
    return {
      width: requirements.minWidth,
      height: requirements.minHeight,
      area: minArea,
    };
  }

  // Calculate optimal dimensions based on aspect ratio
  const targetArea = Math.min(availableArea * 0.8, minArea * 1.5);
  let width = Math.sqrt(targetArea * requirements.preferredAspectRatio);
  let height = targetArea / width;

  // Ensure dimensions don't exceed available space
  if (width > availableWidth) {
    width = availableWidth * 0.9;
    height = targetArea / width;
  }
  if (height > availableHeight) {
    height = availableHeight * 0.9;
    width = targetArea / height;
  }

  return {
    width: Math.max(width, requirements.minWidth),
    height: Math.max(height, requirements.minHeight),
    area: width * height,
  };
}
```

Minimum Room Requirements:

```
export const REQUIRED_ROOMS: Record<string, RoomRequirements> = {
  living: {
    name: 'Living Room',
    type: 'living',
    minWidth: 3,
    minHeight: 4,
    preferredAspectRatio: 1.5,
  },
  kitchen: {
    name: 'Kitchen',
    type: 'kitchen',
    minWidth: 2.5,
    minHeight: 3.2,
    preferredAspectRatio: 1.2,
  },
  maid: {
    name: "Maid's Room",
    type: 'maid-room',
    minWidth: 2.8,
    minHeight: 2.8,
    preferredAspectRatio: 1.0,
  },
  garage: {
    name: 'Garage',
    type: 'garage',
    minWidth: 5,
    minHeight: 3,
    preferredAspectRatio: 1.8,
  },
};
```

Phase 4: Visual Rendering

The floor plan canvas renders shapes with visual indicators:

```
{/* Section outlines - subtle gray */}
{sections && sections?.length > 0 && (
  <g className="sections-group">
    {sections?.map?.((section) => (
      <g key={section?.id}>
        <rect
          x={section?.x}
          y={section?.y}
          width={section?.width}
          height={section?.height}
          fill="none"
          stroke="#94A3B8"
          strokeWidth="0.12"
          strokeDasharray="0.4 0.3"
          opacity="0.5"
        />
        {section?.name && (
          <text
            x={section?.x + section?.width / 2}
            y={section?.y + 0.5}
            fontSize="0.35"
            fill="#64748B"
            textAnchor="middle"
            fontFamily="Arial, sans-serif"
            fontWeight="500"
          >
            {section?.name}
          </text>
        )}
      </g>
    ))}

    {/* Prominent shape outline - blue dotted line */}
    {shape !== 'Rectangular' && (
      <g className="shape-outline-overlay">
        {sections?.map?.((section) => (
          <rect
            key={`outline-${section?.id}`}
            x={section?.x}
            y={section?.y}
            width={section?.width}
            height={section?.height}
            fill="none"
            stroke="#2563EB"
            strokeWidth="0.25"
            strokeDasharray="0.6 0.4"
            opacity="0.8"
            style={{
              filter: 'drop-shadow(0 0 0.2px rgba(37, 99, 235, 0.5))',
            }}
          />
        ))}
      </g>
    )}
  </g>
)}
```

Visual Layer Hierarchy:

1. Grid background (bottom)
2. Room rectangles

3. Subtle section outlines (gray, dashed)
4. Prominent shape outline (blue, thicker)
5. Section labels (top)

Styling Properties:
- **Subtle outlines**: `stroke-width: 0.12`, `opacity: 0.5`, gray color
- **Prominent outline**: `stroke-width: 0.25`, `opacity: 0.8`, blue color
- **Drop shadow**: Creates glow effect for better visibility
- **Dotted pattern**: `strokeDasharray: "0.6 0.4"` for clear distinction

Validation System

The `validateShapeLayout()` function ensures correctness:

```
export function validateShapeLayout(rooms: Room[], sections: Section[]): boolean {
  // Check if all required room types are present
  const requiredTypes: string[] = ['living', 'kitchen', 'maid-room', 'garage'];
  const presentTypes = new Set<string>(rooms.map((r) => r.type));

  for (const reqType of requiredTypes) {
    if (!presentTypes.has(reqType)) {
      return false;
    }
  }

  // Check if rooms don't overlap and fit within sections
  for (const room of rooms) {
    const inSection = sections.some(
      (section) =>
        room.x >= section.x &&
        room.x + room.width <= section.x + section.width &&
        room.y >= section.y &&
        room.y + room.height <= section.y + section.height
    );
    if (!inSection) {
      return false;
    }
  }

  return true;
}
```

Validation Checks:
1. All required room types exist
2. Each room fits entirely within a section
3. No rooms extend beyond section boundaries

Performance Considerations

Computational Complexity

- **Section Generation**: O(1) - Fixed number of sections per shape

- **Room Placement**: O(n) where n = number of rooms (typically 4)
- **Dimension Calculation**: O(1) per room
- **Validation**: O(n × m) where n = rooms, m = sections

Optimization Strategies

1. **Memoization**: Pre-calculate section layouts for common dimensions
2. **Lazy Rendering**: Only render visible sections in large layouts
3. **SVG Optimization**: Use CSS transitions instead of JavaScript animations

---

Testing & Debugging

Unit Tests

```
describe('Shape Layout Utils', () => {
  test('L-shaped sections have correct proportions', () => {
    const sections = generateSections('L-Shaped', 15, 12);
    expect(sections).toHaveLength(2);
    expect(sections[0].width).toBe(15);
    expect(sections[0].height).toBe(7.2); // 60% of 12
    expect(sections[1].width).toBe(9); // 60% of 15
    expect(sections[1].height).toBe(4.8); // 40% of 12
  });

  test('Rooms do not overlap in L-shaped layout', () => {
    const sections = generateSections('L-Shaped', 15, 12);
    const rooms = placeRoomsLShaped(sections, 1);

    // Check horizontal section rooms
    const living = rooms[0];
    const kitchen = rooms[1];
    expect(living.x + living.width + 0.5).toBeLessThanOrEqual(kitchen.x);

    // Validate all rooms fit in sections
    expect(validateShapeLayout(rooms, sections)).toBe(true);
  });
});
```

Visual Debugging

1. **Console Logging**: Add debug output for room coordinates
2. **Color Coding**: Use distinct colors for debugging overlaps
3. **Boundary Visualization**: Render section boundaries in debug mode

---

Extension Points

Adding New Shapes

1. Add shape type to `FloorPlanShape` enum in `types.ts`
2. Implement `generate[Shape]Sections()` function
3. Implement `placeRooms[Shape]()` function
4. Update switch statements in `generateSections()` and `placeRoomsInShape()`

5. Add visual rendering logic in `floor-plan-canvas.tsx`

Customizing Room Allocation

```
// Example: Prioritize living room size
const livingWidth = horizontalAvailWidth * 0.65; // Increase from 55%
const kitchenWidth = horizontalAvailWidth * 0.30; // Decrease from 40%
```

Dynamic Padding Adjustment

```
// Scale padding based on overall dimensions
const padding = Math.max(0.5, totalWidth * 0.02);
```

Common Issues & Solutions

Issue: Rooms Overlapping

Cause: Insufficient padding or incorrect sequential positioning
Solution: Verify `padding * 2` is used when positioning adjacent rooms

Issue: Rooms Too Small

Cause: Percentage allocation doesn't account for minimum dimensions
Solution: Use `Math.max()` to enforce minimum room sizes

Issue: Shape Outline Not Visible

Cause: Z-index issues or opacity too low
Solution: Ensure shape outline is rendered after rooms, increase opacity

---

References

- **Architectural Standards**: Neufert Architects' Data (41st Edition)
- **SVG Specification**: W3C SVG 1.1 Specification
- **React Best Practices**: React TypeScript Cheatsheet

Document Version: 1.0
Last Updated: December 25, 2024
Implementation: ArchitectPro v2.0