CLI Visualization System Documentation

Overview

The UtilityFog CLI Visualization system provides comprehensive command-line tools for visualizing fractal tree structures, message flows, and state transitions. It supports multiple rendering modes, interactive visualization, and export to various formats including HTML, SVG, and JSON.

Architecture

Core Components

1. **Data Models**: TreeNode, MessageFlow, StateTransition, VisualizationData
2. **Renderers**: TreeRenderer, FlowRenderer, StateRenderer, InteractiveRenderer
3. **Exporters**: HTMLExporter, SVGExporter, TextExporter, JSONExporter
4. **CLI Interface**: VisualizationCLI with command-line argument parsing

Key Features

- **Multiple Visualization Types**: Tree structure, message flows, state transitions
- **Interactive Mode**: Real-time visualization with keyboard navigation
- **Export Formats**: HTML reports, SVG diagrams, plain text, JSON data
- **Customizable Display**: Color schemes, dimensions, time windows
- **Demo Data Generation**: Built-in sample data generator for testing

Usage

Installation

The CLI visualization system is part of the UtilityFog frontend package:

```
# Install dependencies
pip install -r requirements.txt

# Run from the project root
python -m utilityfog_frontend.cli_viz --help
```

Basic Commands

Tree Structure Visualization

Display the hierarchical structure of nodes:

```
# Basic tree view
python -m utilityfog_frontend.cli_viz tree --input data.json

# Customized tree view
python -m utilityfog_frontend.cli_viz tree \
    --input data.json \
    --width 120 \
    --height 40 \
    --color-scheme dark \
    --show-ids
```

Message Flow Visualization

Display message flows between nodes:

```
# Basic flow view
python -m utilityfog_frontend.cli_viz flow --input data.json

# Flow view with time window
python -m utilityfog_frontend.cli_viz flow \
    --input data.json \
    --time-window 300 \
    --color-scheme colorblind
```

State Transition Visualization

Display state changes over time:

```
# Basic state view
python -m utilityfog_frontend.cli_viz state --input data.json

# State view with extended time window
python -m utilityfog_frontend.cli_viz state \
    --input data.json \
    --time-window 600
```

Interactive Mode

Real-time visualization with keyboard controls:

```
# Interactive mode
python -m utilityfog_frontend.cli_viz interactive \
    --input data.json \
    --refresh-rate 2.0

# Controls:
# [t] - Switch to tree view
# [f] - Switch to flow view
# [s] - Switch to state view
# [r] - Refresh data
# [q] - Quit
```

Export Functionality

HTML Report Export

Generate interactive HTML reports:

```
# Basic HTML export
python -m utilityfog_frontend.cli_viz export \
    --input data.json \
    --output report.html \
    --format html
```

The HTML export includes:
- Interactive tabbed interface
- Tree structure with expandable nodes
- Message flow timeline
- State transition history
- Statistics dashboard
- Responsive design with CSS styling

SVG Diagram Export

Generate scalable vector graphics:

```
# SVG export with custom dimensions
python -m utilityfog_frontend.cli_viz export \
    --input data.json \
    --output diagram.svg \
    --format svg \
    --width 1200 \
    --height 800
```

Text Report Export

Generate plain text reports:

```
# Text export
python -m utilityfog_frontend.cli_viz export \
    --input data.json \
    --output report.txt \
    --format text
```

JSON Data Export

Export processed visualization data:

```
# JSON export
python -m utilityfog_frontend.cli_viz export \
    --input data.json \
    --output processed.json \
    --format json
```

Demo Data Generation

Generate sample data for testing:

```
# Generate demo data
python -m utilityfog_frontend.cli_viz demo \
    --nodes 20 \
    --messages 50 \
    --transitions 30 \
    --output demo_data.json
```

Data Format

Input JSON Structure

The visualization system expects JSON data in the following format:

```json
{
  "timestamp": 1632150000.0,
  "nodes": {
    "node_001": {
      "id": "node_001",
      "name": "Root Node",
      "state": "active",
      "parent_id": null,
      "children": ["node_002", "node_003"],
      "position": [400, 100],
      "metadata": {
        "cpu_usage": 45.2,
        "memory_mb": 512
      },
      "last_updated": 1632149900.0
    }
  },
  "messages": [
    {
      "id": "msg_001",
      "source_id": "node_001",
      "target_id": "node_002",
      "message_type": "coordination",
      "content": "heartbeat",
      "timestamp": 1632149950.0,
      "status": "delivered",
      "metadata": {
        "size_bytes": 256
      }
    }
  ],
  "transitions": [
    {
      "node_id": "node_002",
      "from_state": "inactive",
      "to_state": "active",
      "timestamp": 1632149900.0,
      "trigger": "heartbeat",
      "metadata": {
        "duration_ms": 150
      }
    }
  ],
  "metadata": {
    "version": "1.0",
    "source": "simulation"
  }
}
```

Node States

Supported node states:
- `active` : Node is operational and processing
- `inactive` : Node is idle or offline
- `processing` : Node is actively processing tasks
- `error` : Node has encountered an error
- `unknown` : Node state is undetermined

Message Types

Supported message types:
- `coordination` : Coordination protocol messages
- `data` : Data transfer messages
- `heartbeat` : Health check messages
- `error` : Error notification messages
- `control` : Control and command messages

Rendering Options

Color Schemes

Default Scheme

- Active: Green
- Inactive: Gray
- Processing: Yellow
- Error: Red
- Unknown: Blue

Dark Scheme

- Optimized for dark terminals
- High contrast colors
- Bright accent colors

Colorblind Scheme

- Colorblind-friendly palette
- Distinct patterns and symbols
- Accessible color combinations

Display Customization

Tree Renderer Options

- `width` : Display width in characters (default: 80)
- `height` : Display height in lines (default: 24)
- `show_ids` : Show node IDs alongside names
- `color_scheme` : Color scheme selection

Flow Renderer Options

- `time_window` : Time window for message display (seconds)
- Message status indicators
- Flow direction arrows
- Timestamp information

State Renderer Options

- `time_window` : Time window for transition display (seconds)
- State change indicators
- Transition triggers
- Duration information

Integration

System Integration

The CLI visualization system integrates with other UtilityFog components:

```python
from utilityfog_frontend.cli_viz import VisualizationData, TreeNode, MessageFlow

# Create visualization data
viz_data = VisualizationData()

# Add nodes from coordination system
for node_id, node_info in coordination_system.get_nodes():
    tree_node = TreeNode(
        id=node_id,
        name=node_info.name,
        state=NodeState(node_info.status),
        metadata=node_info.metrics
    )
    viz_data.add_node(tree_node)

# Add messages from messaging system
for message in messaging_system.get_recent_messages():
    flow = MessageFlow(
        id=message.id,
        source_id=message.sender,
        target_id=message.receiver,
        message_type=MessageType(message.type),
        content=message.payload
    )
    viz_data.add_message(flow)
```

Telemetry Integration

Integration with the telemetry system (FT-008):

```python
from utilityfog_frontend.telemetry import TelemetryCollector
from utilityfog_frontend.cli_viz import VisualizationData

def create_viz_from_telemetry(collector: TelemetryCollector) -> VisualizationData:
    """Create visualization data from telemetry collector."""
    viz_data = VisualizationData()

    # Extract node information from metrics
    snapshot = collector.get_snapshot()

    # Convert telemetry events to visualization events
    for event in collector.get_events():
        if event.name == "node_state_change":
            transition = StateTransition(
                node_id=event.value["node_id"],
                from_state=NodeState(event.value["from_state"]),
                to_state=NodeState(event.value["to_state"]),
                timestamp=event.timestamp,
                trigger=event.value.get("trigger", "unknown")
            )
            viz_data.add_transition(transition)

    return viz_data
```

Advanced Features

Custom Renderers

Create custom renderers for specialized visualization needs:

```python
from utilityfog_frontend.cli_viz.renderer import BaseRenderer

class CustomRenderer(BaseRenderer):
    """Custom renderer for specialized visualization."""

    def render(self, data: VisualizationData) -> str:
        """Render custom visualization."""
        lines = []
        lines.append("CUSTOM VISUALIZATION")
        lines.append("=" * self.width)

        # Custom rendering logic
        for node in data.nodes.values():
            lines.append(f"Node: {node.name} ({node.state.value})")

        return "\n".join(lines)
```

Custom Exporters

Create custom exporters for specialized output formats:

```python
from utilityfog_frontend.cli_viz.exporters import BaseExporter

class CustomExporter(BaseExporter):
    """Custom exporter for specialized format."""

    def export(self, data: VisualizationData, output_path: str) -> bool:
        """Export to custom format."""
        try:
            # Custom export logic
            with open(output_path, 'w') as f:
                f.write("Custom format data\n")
                for node in data.nodes.values():
                    f.write(f"{node.id},{node.name},{node.state.value}\n")
            return True
        except Exception as e:
            print(f"Export error: {e}")
            return False
```

Programmatic Usage

Use the visualization system programmatically:

```python
import asyncio
from utilityfog_frontend.cli_viz import VisualizationCLI

async def automated_visualization():
    """Automated visualization workflow."""
    cli = VisualizationCLI()

    # Generate demo data
    demo_args = type('Args', (), {
        'nodes': 15,
        'messages': 30,
        'transitions': 20,
        'output': 'auto_demo.json'
    })()
    cli.cmd_demo(demo_args)

    # Export HTML report
    export_args = type('Args', (), {
        'input': 'auto_demo.json',
        'output': 'auto_report.html',
        'format': 'html',
        'width': 800,
        'height': 600
    })()
    cli.cmd_export(export_args)

    print("Automated visualization complete")

# Run automated workflow
asyncio.run(automated_visualization())
```

Performance Considerations

Memory Usage

- Node data is kept in memory for fast access
- Message and transition history is limited (configurable)
- Large datasets may require data streaming or pagination

Rendering Performance

- ASCII rendering is optimized for terminal display
- SVG generation scales with node count
- HTML export includes embedded CSS/JS for self-contained reports

Interactive Mode

- Refresh rate affects CPU usage
- Terminal I/O may have platform-specific limitations
- Large datasets may cause display lag

Troubleshooting

Common Issues

1. **Import Errors**: Ensure all dependencies are installed
2. **File Not Found**: Check input file paths and permissions

3. **Display Issues**: Verify terminal size and color support
4. **Interactive Mode**: Requires Unix-like system with termios

Debug Mode

Enable debug output for troubleshooting:

```
import logging
logging.getLogger('utilityfog_frontend.cli_viz').setLevel(logging.DEBUG)
```

Performance Tuning

- Reduce time windows for large datasets
- Use appropriate display dimensions
- Limit node metadata for better performance

API Reference

Core Classes

VisualizationData

Container for all visualization data.

Methods:
- `add_node(node)` : Add a tree node
- `add_message(message)` : Add a message flow
- `add_transition(transition)` : Add a state transition
- `get_active_nodes()` : Get nodes in active state
- `get_recent_messages(seconds)` : Get recent messages
- `get_node_hierarchy()` : Get hierarchical structure

TreeNode

Represents a node in the fractal tree.

Properties:
- `id` : Unique node identifier
- `name` : Display name
- `state` : Current node state
- `parent_id` : Parent node ID
- `children` : List of child node IDs
- `metadata` : Additional node data

MessageFlow

Represents a message between nodes.

Properties:
- `source_id` : Source node ID
- `target_id` : Target node ID
- `message_type` : Type of message
- `content` : Message content
- `status` : Delivery status

StateTransition

Represents a state change event.

Properties:
- `node_id`: Node that changed state
- `from_state`: Previous state
- `to_state`: New state
- `trigger`: What caused the transition

Renderers

TreeRenderer

Renders hierarchical tree structure.

FlowRenderer

Renders message flows between nodes.

StateRenderer

Renders state transitions over time.

InteractiveRenderer

Provides interactive real-time visualization.

Exporters

HTMLExporter

Exports interactive HTML reports.

SVGExporter

Exports scalable vector graphics.

TextExporter

Exports plain text reports.

JSONExporter

Exports structured JSON data.

Future Enhancements

- **3D Visualization**: WebGL-based 3D tree rendering
- **Real-time Streaming**: Live data feed integration
- **Advanced Analytics**: Statistical analysis and pattern detection
- **Custom Themes**: User-defined color schemes and layouts
- **Plugin System**: Extensible renderer and exporter plugins
- **Performance Optimization**: Lazy loading and data virtualization
- **Mobile Support**: Responsive design for mobile devices
- **Collaboration Features**: Shared visualization sessions