# Deep Dive into the **LearnGitBranching** Codebase

**LearnGitBranching** is an open-source interactive Git tutorial featuring a *visual git branch tree* that animates in response to user commands. This report provides a **comprehensive, file-by-file breakdown** of how the codebase works – focusing especially on **the UI/UX of the animated git branch tree** – so that others can replicate a similar project with the same tech stack, step by step, **without errors**.

We will cover the **tech stack**, overall **architecture**, key modules with **pseudocode**, and even include **architectural diagrams** for clarity. Short, focused sections and lists are used for readability.

## Tech Stack Overview

LearnGitBranching is built with a mix of classic and modern web technologies:

* **HTML5/CSS3** for the base layout and styling (with custom CSS for the visualization canvas).
* **JavaScript (ES5/ES6)** as the primary language. The project uses both older JS patterns and newer syntax due to its long evolution.
* **Libraries/Frameworks**:
* **Backbone.js** – an older MVC framework used for some views and models.
* **React** – modern library for UI; used for certain interface components (e.g. helper bars, modals) alongside Flux architecture.
* **Flux/Dispatcher** – for state management; you’ll find a central dispatcher and stores (e.g. GlobalStateStore, LevelStore) to manage app state.
* **jQuery & jQuery-UI** – for DOM manipulation and possibly UI effects (e.g. dragging, modals).
* **Underscore (lodash)** – utility library for functional helpers.
* **Q (Promises)** – used for managing asynchronous sequences (like animations) in earlier code.
* **Raphaël (SVG library)** – included for drawing vector graphics【93†】. (The codebase uses Raphaël to draw commit nodes and connecting lines as scalable vectors, enabling smooth animations and curved lines).
* **Marked** – to render Markdown content (for tutorial text and level descriptions).
* **Build & Tooling**: Initially built with Gulp, now uses **Vite** for bundling (see vite.config.js). Testing is set up with Jest (there’s a \_\_tests\_\_ directory). Package management is via Yarn (see yarn.lock).

**👉 Why this mix?** The project started pre-2015, using Backbone for structure and Raphaël for SVG. Over time, it adopted React and Flux for new UI components, resulting in a hybrid architecture. This is actually instructive for replication: you can either recreate it with the same libraries, or choose a modern stack (e.g. pure React+D3) to achieve similar results.

*(Source: see project README for description and package.json for dependencies.)*

## High-Level Architecture

At a high level, LearnGitBranching can be seen as two major pieces: **(1) a Git logic engine** that simulates git operations on an internal commit graph, and **(2) a Visualization/UI layer** that displays this graph and animates changes. Surrounding these are supporting components for user interaction and game-like levels. The flow looks like this:

flowchart LR  
 subgraph User Interface  
 CLI[Command Line Input] -->|user types commands| Controller  
 Controller[Command Controller] -->|dispatch| GitEngine  
 UIViews[UI Views (React/Backbone)] -->|render| VisualTree  
 end  
 subgraph Git Simulation  
 GitEngine[Git Engine & Commands] --> GraphModel[Commit Graph Model]  
 end  
 subgraph Visualization  
 GraphModel -->|diff changes| Animations  
 GraphModel -->|update| VisualTree[Visual Branch Tree]  
 Animations -->|animate DOM| VisualTree  
 end  
 subgraph Levels  
 LevelData[Level Definitions] --> GraphModel  
 LevelData --> UIViews  
 end

**Explanation:** The **Command Controller** (which listens to the command-line UI) sends user commands to the **Git Engine**. The Git Engine updates the **Commit Graph Model** (the internal representation of commits, branches, HEAD, etc.). The **Visual Tree** UI (the commit graph canvas) is then updated to reflect the new model state. An **Animation system** computes the difference between the old and new graph states and produces animations to smoothly transition the visual elements (e.g. moving HEAD, creating a new commit node with a bounce). Meanwhile, **UI Views** (like toolbars, modals) update via React/Flux or Backbone to reflect any state changes (like level progress or messages). The **Levels system** provides predefined scenarios (initial states and goals), which feed into the Graph Model setup and also display instructions in the UI.

Next, we’ll dive into each part in detail, focusing on how the **animated git branch tree UI is created and works**.

## Commit Graph Data Model (Git Engine & Graph)

The project’s core is a simulation of Git’s data structures: commits, branches, HEAD, etc. These are managed by the **Git Engine** modules under src/js/git/ and the **Graph Model** under src/js/graph/. Key elements:

* **GitEngine (src/js/git/index.js)**: The main logic that processes Git commands. It includes implementations for commands like commit, branch, checkout, merge, etc. Each command updates an in-memory representation of the repository. This module uses helper sub-modules:
* commands.js – defines the behavior of individual git commands.
* gitShim.js – possibly a layer to simulate Git commands for Mercurial (there are Mercurial references, hinting at multi-VCS support).
* headless.js – runs git commands without a UI (for testing or background logic).
* **Graph (src/js/graph/index.js)**: Manages the *structure* of the commit graph. It can create or retrieve commit and branch objects from a JSON representation. For example, it can take a JSON like {"branches": {...}, "commits": {...}, "HEAD": {...}} and instantiate the corresponding commit graph in memory. It ensures unique commit IDs, sets parent-child relationships, etc.
* **TreeCompare (src/js/graph/treeCompare.js)**: A utility that **diffs two commit graph states** (before and after a command). This is critical for the animation system – by comparing the “before” and “after” states of the graph, the code can determine what changed (e.g. a new commit was added, a branch moved) and generate the appropriate animation sequence.
* **Data Structures**: Commits and branches might be represented as simple JS objects or Backbone models. Each commit has an ID (like C0, C1…), parent commit references, and flags (e.g. rootCommit: true for the first commit). Each branch is just a pointer to a commit (the branch’s “target”). (In Git, a branch is a name for a commit; the code follows this, representing branches as objects with an id and a target commit id). For example, a simple repo with one commit and two branches might be represented in JSON as:
* {  
   "branches": {   
   "main": {"target": "C0"},   
   "bugFix": {"target": "C0"}   
   },  
   "commits": {   
   "C0": {"id": "C0", "parents": [], "rootCommit": true}   
   },  
   "HEAD": {"target": "main"}   
  }
* This means both “main” and “bugFix” point to commit C0, and HEAD is on the “main” branch. The Graph module can build this in-memory and also update it when commands run.
* **Integration with Visuals**: The GitEngine doesn’t directly manipulate the DOM; it purely updates the data. After each command, it triggers the visual update process: it will call functions to update the **Visual Tree** (see next section), often via the Animation Factory. For instance, after creating a commit, the GitEngine might call something like AnimationFactory.startAnimation("commitCreated", newCommitObject) to queue the visual creation of that commit. The code uses a **Dispatcher/Action** pattern (Flux) here: e.g., a CommandLineAction is dispatched when a command completes, carrying the new state or diff, which the visualization store picks up.

**💡 Recipe Step 1: Build a Git Data Model.** To replicate: first implement classes or structures for **Commit** and **Branch**. Create a **GitEngine** module that can execute commands (start with basics: commit, branch, checkout) and update a **Graph** structure. Ensure you can represent multiple branches and commits, and that each commit knows its parents and children. This model should be decoupled from the UI (so it could run “headlessly”). Also implement a way to **compare states** or at least identify what changed after each command (you can store a copy of the previous state or use events to know what was added).

## Visualization System (Animating the Git Branch Tree)

This is the heart of the **UI/UX**: showing the commit tree and animating it. The visualization code is primarily under src/js/visuals/. It’s structured in an **object-oriented manner**, with classes for each visual element of the git graph: **Tree**, **Branch**, **Node (commit)**, **Edge (connection line)**, and **Tag (labels like branch names or HEAD)**. These correspond to the conceptual elements of a commit graph. Let’s break down the key components and their interactions:

### Visualization Container and Base Class

* **Visualization (Visual Tree)**: The main container for the git graph visualization. In code, this is often referred to as gitVisuals or Visuals. It represents the entire commit tree canvas. When the app initializes or a level is loaded, a Visualization instance is created and attached to a DOM element in the page (likely a dedicated <div> acting as the canvas).
* *Tech detail:* The codebase uses a custom class (likely called Visualization or TreeVisual) that extends a base class VisBase. This class acts as the root of all visual elements. It holds collections of branches, commits, etc., and provides utility methods to add new branches or find commits.
* It likely also sets up the drawing area (for example, if using Raphael, it might initialize the Raphael paper here, or if using plain DOM, it might set a relative positioning context).
* The **coordinate system** for the visual tree is grid-like: each commit is placed at an (x, y) coordinate where x corresponds to the branch index and y corresponds to the generation (commit level). For example, the code defines a constant spacing: ROW\_HEIGHT (vertical gap between commit levels) and ROW\_WIDTH (horizontal gap between branch columns). So the formula for a commit’s position is essentially:
  + **x = branchIndex \* ROW\_WIDTH**
  + **y = commitLevel \* ROW\_HEIGHT** (This is exactly what the VisNode.position() method computes internally – each new commit gets positioned based on its branch and depth.)
* *In code:* when a new commit node is initialized, it calculates its coordinates, e.g. this.meta.origin.x = this.br \* constants.ROW\_WIDTH; this.meta.origin.y = this.level \* constants.ROW\_HEIGHT; – meaning branch spacing and row height govern the layout.
* **VisBase (src/js/visuals/visBase.js)**: An abstract base class that all visual elements inherit from. It provides common functionality:
* **Element Creation**: In VisBase.init(), it creates a containerDiv (an HTML <div> element) for the visual element if none is provided. It sets a CSS class (e.g. gitVis) on this container. All visual elements (nodes, branches, etc.) ultimately use a <div> for their DOM element, styled appropriately (or act as a container for Raphael SVG elements).
* **Hierarchy**: It stores references like this.parent (the parent visual in the hierarchy), this.rootTree (reference to the top Visualization), and this.gitVisuals (possibly the same as rootTree or a central object managing visuals).
* **Position and Size**: It has default meta.origin coordinates (x:0, y:0) and might have basic height() and width() methods that can be overridden. For example, a base node might define width() as a fixed constant (like node diameter) and height() as the fixed row height.
* **Event Handling**: If the element is the root (the whole tree), VisBase sets up an EventBaton (in this.eventBaton = new EventBaton(this) for the root). The EventBaton is a utility to manage input focus between components. In practice, this likely ensures keyboard events (arrow keys, etc.) are handled by the right component (for example, when the user is in a modal vs. the main view). This is part of the UX management (not vital for drawing the tree, but important for overall UI responsiveness).
* **Animation Queue**: Each visual element (especially the root) holds an animationQueue (just an array). Animations affecting that element can be queued here. Typically, only the root Visualization might use the queue, or each element queues its own small animation (like a node bounce). This ties into the animation system discussed later.

*Pseudocode (simplified):*

class VisBase {  
 init(params={}) {  
 this.containerDiv = params.containerDiv || document.createElement('div');  
 this.el = this.containerDiv;  
 if (!params.containerDiv) this.containerDiv.className = 'gitVis';  
 // Set dimensions if provided  
 if (params.height) this.el.style.height = params.height;  
 if (params.width) this.el.style.width = params.width;  
 // Hierarchy links  
 this.parent = params.parent || null;  
 this.type = params.type; // e.g. 'node', 'branch', 'tag', 'tree'  
 this.rootTree = params.rootTree || (this.type === 'tree' ? this : null);  
 this.gitVisuals = params.gitVisuals || (this.parent ? this.parent.gitVisuals : null);  
 // Animation handling  
 this.animationQueue = [];  
 // Event baton for root  
 this.isRoot = (this.type === 'tree');  
 if (this.isRoot) {  
 this.eventBaton = new EventBaton(this);  
 }  
 // ... other initialization ...  
 }  
 // Default size  
 width() { return this.baseWidth || 0; }  
 height() { return this.baseHeight || 0; }  
 // (Note: specific classes override width/height)  
}

### Visual Elements: Branches, Nodes, Edges, Tags

Now, let’s see how each type of element is implemented and how they collaborate to form the tree:

* **VisBranch (src/js/visuals/visBranch.js)** – represents a Git **branch** in the visual. In the UI, a branch is essentially a vertical column of commits with a label (the branch name) pointing to the latest commit. The VisBranch class manages the collection of commit nodes on that branch and the branch label tag.
* When a new branch is created (e.g. via git branch command), the code will do something like:
* let newBranch = new VisBranch({ parent: visualization, label: 'bugFix', parentBranch: currentBranch });
* Here, parent is the Visualization (root), and parentBranch is the branch from which this new branch splits (if any).
* **Initialization** (VisBranch.init()): It calls the base init (setting up its own this.el div and linking to parent). Then:
  + It assigns a unique ID for the branch (via this.gitVisuals.uniqueIdForType('branch') – an internal counter).
  + It appends its DOM element to the parent (so the branch’s <div> is placed inside the main visualization container).
  + It maintains a list nodeCollection which will hold all commit nodes on this branch. Initially, this might be empty or inherit some nodes if branching off an existing commit.
  + It sets this.label to the branch name (for display).
  + It registers itself in the parent Visualization’s branch list (parent.assignBranchCollection(this) presumably adds it to a global list of branches).
  + **Branch’s tip commit**: Immediately upon creation, VisBranch creates a **VisNode** to represent the branch’s tip. This is important: when we create a new branch in Git, no new commit is made; the branch just points to an existing commit. The visualization must reflect that by showing the branch label on that commit. The code does:
  + this.node = new VisNode({  
     parent: this,   
     id: this.id,  
     label: this.label,  
     parentNode: params.parentBranch ? params.parentBranch.node : undefined,  
     branch: params.targetBranch || (this.parent.branchCollection.length - 1),  
     height: params.parent.getHeight(),  
     tags: [], // no tags initially  
     commitColor: utils.getColorFromHash(this.label, constants.BRANCH\_COLOR),  
     tagColor: utils.getColorFromHash(this.label, constants.TAG\_COLOR),  
     // possibly arrow style for edges...  
    });
  + This creates a VisNode for the branch pointer:
    - parent: this (so the node is added to the branch’s DOM container).
    - parentNode: if the new branch splits from an existing branch, parentBranch.node is the node from which we branched. This tells the new node who its ancestor is.
    - branch: <index>: the horizontal position for this branch (the code uses either targetBranch if specified or defaults to the new branch’s index = total branches count - 1, which effectively places it in the next column).
    - height: parent.getHeight(): This sets the vertical position. Likely, parent.getHeight() returns the current maximum commit level (or number of commits) in the entire graph. Using that ensures the new branch’s initial node is placed at the correct vertical position (aligned with the commit it points to). For instance, if you branch off the latest commit (which is at level 3), getHeight() might return 3, so the new branch’s node appears at y = 3 \* ROW\_HEIGHT, aligning exactly with that commit’s row.
    - commitColor and tagColor: Each branch is given consistent colors derived from its name (using a hash function). So all commits on branch "main" might be blue, on "bugFix" green, etc., and tags for those branches tinted accordingly. This is a UX choice to help identify branches visually. (The code uses utils.getColorFromHash(name, baseColor) to generate a color【**source code** shows using base constants like BRANCH\_COLOR and TAG\_COLOR seeds】.)
  + The new VisNode added becomes this.node (the branch’s tip). Immediately after creation, the code also creates a **VisTag** for the branch name (see VisTag below), attaching it to that node.
  + The branch’s nodeCollection now contains this one node (and will grow as commits are added on this branch).
* The VisBranch class also handles when the branch moves (e.g., after a commit on that branch, the tip moves to the new commit node). It likely has a method to update this.node to the latest commit, and possibly to remove old branch labels or detach HEAD if needed.

*UX note:* VisBranch itself might not have a visible representation besides containing nodes. The branch name label is actually a VisTag attached to the commit node. So visually, “branch = column + label on commit”.

* **VisNode (src/js/visuals/visNode.js)** – represents a **commit** in the visualization. This is typically drawn as a colored circle or dot with a small label (commit ID or message, though by default LGB doesn’t show commit messages in the graph, just IDs on hover).
* **Initialization** (VisNode.init()): Like others, calls base init (which creates a div .gitVis). Then:
  + Sets a unique id for the node (likely matching the commit ID like "C1").
  + Attaches itself to its parent DOM: this.parent.el.appendChild(this.el). If parent is a VisBranch, this means the commit’s element is inside the branch’s container in the DOM (thus automatically aligning by branch column if the parent branch’s container is positioned appropriately).
  + **Branch Index**: Stores this.br = params.branch – a number indicating which branch column. (This is used to compute x coordinate).
  + **Level (Depth)**: Computes this.level = (this.parent.level || 0) + 1. If the parent is a branch, the branch might have a level property (possibly the branch’s own generation relative to tree). In practice, the tree might have level 0, branch level 0 or 1, and node = branch level + 1. This is a bit subtle, but effectively it ensures that commits have an incremented level value to indicate vertical placement. If branch’s level was 0, then the first commit node gets level 1.
  + **Parent Commit Link**: this.parentNode = params.parentNode || null. If this commit was created as a child of an existing commit, parentNode points to that VisNode. This is used to draw the connecting edge between commits.
  + **Root Reference**: this.root = this.parent.root – quick access to the Visualization root.
  + **Collection**: If a collection (like an array of all nodes on this branch) is passed, it uses it; otherwise it uses this.parent.nodeCollection. It then pushes itself into that collection.
  + **Arrow drawing settings**: (Arrows meaning the direction indicator on edges) – uses defaults if not provided. Typically, arrows might be drawn on merge lines or so.
  + **Geometry**:
  + this.branchSpacing = params.branchSpacing || constants.ROW\_WIDTH – horizontal spacing (defaults to a constant, say 50px).
  + this.baseWidth = params.width || constants.NODE\_WIDTH – the base width of the node graphic (if the commit is drawn as a shape; could correspond to diameter).
  + **Positioning & Rendering**: It immediately calls this.position() and this.render().
  + position(): calculates its coordinates as described earlier. For instance:
  + const x = this.br \* this.branchSpacing;  
    const y = this.level \* constants.ROW\_HEIGHT;  
    this.meta.origin.x = x;  
    this.meta.origin.y = y;  
    // Additionally, update this.el.style.left/top to position the element.  
    this.el.style.left = x + 'px';  
    this.el.style.top = y + 'px';
  + So if this node is on branch index 1 and level 3, and ROW\_WIDTH=60, ROW\_HEIGHT=30, it will be positioned at (60, 90) within the container. This places commits in a grid layout.
  + render(): likely draws the commit as a shape. If using Raphaël, this might create a circle on the Raphael canvas at the given coordinates. If using pure DOM/CSS, it might style the this.el div to look like a node:
    - e.g., set this.el.className = 'gitNode'; and CSS for .gitNode could make it a small circle (using border-radius: 50%, background color, fixed size). Alternatively, the code could directly style it: this.el.style.background = commitColor and set width/height = diameter.
    - The commitColor set earlier ensures each branch’s commits have a consistent color (as computed in branch creation).
  + After rendering the node itself, the code attaches a **VisTag** for this node if needed. Indeed, in VisNode.init, we see:
  + this.tag = new VisTag({  
     parent: this.parent, // attach tag to the branch container (or possibly directly to node’s parent)  
     targetNode: this,   
     inline: true,  
     offset: 0,  
     color: params.tagColor  
    });  
    this.tags = [this.tag];
  + This creates a **tag label** for the commit. However, by default inline: true might indicate this tag is not a branch name but something like a commit ID or HEAD label that sits “inline” with the node. In practice, for normal commits, LGB doesn’t show a label unless it’s HEAD or a branch. So this default tag might actually be repurposed as the branch label for branch’s tip node:
    - If this node is the tip of a branch, the branch’s name will be printed next to it (that’s likely this VisTag).
    - If params.tag was passed (e.g. if this commit should have a detached HEAD label), then they call this.detachTag(params.tag) to attach an extra tag (like “HEAD”) floating above. So each commit can have one or multiple tags (branch names, HEAD, etc.) attached.
  + After initialization, a VisNode is fully drawn at the correct position. Initially, it might be drawn invisible or with a certain transform (for animation). The Animation system often works by immediately creating elements but maybe scaling them down or hiding them, then animating them in. It’s possible that render() does not yet reveal it until an animation runs (depending on how animations are queued).
* **Edges**: Note that a commit alone doesn’t show its connection. The connecting line is a separate VisEdge object (see below). However, VisNode might be aware of its parent commit. Possibly in VisNode.position() or after creation, if this.parentNode exists, it will create a VisEdge from this.parentNode to itself. The code likely calls something like:
* new VisEdge({ parent: this.parent, source: this.parentNode, target: this });
* This would append a line in the branch container or root container connecting the two commit nodes. We’ll cover VisEdge next.

*Pseudocode summary:*

class VisNode extends VisBase {  
 init(params) {  
 params.type = 'node';  
 super.init(params); // creates this.el, sets parent etc.  
 this.id = params.id || generateNodeId();  
 this.parent = params.parent; // VisBranch  
 this.parent.el.appendChild(this.el); // attach node DOM to branch DOM  
 this.br = params.branch; // branch index  
 this.level = (this.parent.level || 0) + 1;  
 this.parentNode = params.parentNode || null; // link to parent commit's VisNode  
 this.root = this.parent.root;  
 this.collection = params.collection || this.parent.nodeCollection;  
 this.collection.push(this);  
 // visual styles  
 this.branchSpacing = params.branchSpacing || constants.ROW\_WIDTH;  
 this.baseWidth = params.width || constants.NODE\_WIDTH;  
 // Compute position and place element  
 this.position(); // sets this.meta.origin and maybe CSS left/top  
 // Render the commit dot (color, shape)  
 this.render();   
 // Attach tag (label) if needed  
 this.tag = new VisTag({ parent: this.parent, targetNode: this, inline: true, color: params.tagColor || '#...' });  
 this.tags = [ this.tag ];  
 if (params.tag) {  
 this.detachTag(params.tag); // attach an additional tag like HEAD  
 }  
 }  
 position() {  
 const x = this.br \* this.branchSpacing;  
 const y = this.level \* constants.ROW\_HEIGHT;  
 this.meta.origin = { x, y };  
 this.el.style.left = x + 'px';  
 this.el.style.top = y + 'px';  
 }  
 render() {  
 // e.g., style as a circle with commitColor  
 this.el.className += ' gitNode';  
 this.el.style.backgroundColor = params.commitColor || constants.DEFAULT\_COMMIT\_COLOR;  
 this.el.style.width = this.baseWidth + 'px';  
 this.el.style.height = this.baseWidth + 'px';  
 // possibly use Raphael to draw more complex shapes  
 }  
}

* **VisEdge (src/js/visuals/visEdge.js)** – represents the **line (edge) connecting two commits**. In git graphs, each commit (except root) has a parent, so we draw a line from the child commit up (or diagonally) to its parent commit. Also, merges have two parents, so a commit might have an extra edge from a second parent branch.
* **Initialization**: The VisEdge connects a source node to a target node. For a normal commit, the source might be the parent commit and target the new commit. For a merge commit with two parents, there might be two VisEdges created.
  + The VisEdge is also a VisBase subclass (type 'edge'). It creates a this.el (likely not a full div, maybe an SVG element or a thin div). It appends to parent container (which might be the branch or the root container – possibly root to allow crossing branches).
  + It stores this.source and this.target as references to the nodes it connects.
  + It may set some default width or path (initially a straight line of zero length).
  + Then it calls this.changeLevel(0) and this.render(). The changeLevel(0) might be a method to initialize the edge’s path or z-index.
  + render() likely uses Raphaël or Canvas to actually draw the line:
  + If Raphaël is used, VisEdge.render() could compute a path string from source to target coordinates. For example, for a simple straight line: M x1,y1 L x2,y2. But LGB often draws **curved** lines (especially for merges or if branches are spaced out). They might use a quadratic Bezier curve for nicer visuals. The constant curveControlPointOffset (50) suggests edges might curve. Possibly they draw a slight curve between commits using a control point offset horizontally.
  + Using Raphaël, one would do paper.path("M ... C ...") and style it (stroke color equals branch color, etc.). The Raphaël paper could be initiated in the Visualization root and passed down or globally accessible.
  + If not using Raphaël for edges, they might create an HTML <canvas> or use an <svg> element. But given Raphaël is included, it’s likely being used for edges (and possibly for drawing commit dots as vector circles, though commit dots could be just divs).
  + The edge also needs to update if the connected nodes move (like in an animation, if a node moves, the edge should animate too). Possibly the edges listen to node movements or are recomputed after each layout change. In our case, since nodes don’t move once placed (except in rebase animations), edges are mostly static lines drawn once per commit.
* **Example**: If a new commit C2 is created on branch main (with parent C1), the process is: new VisNode for C2 is created at position (x=0, y=2\*ROW\_HEIGHT, say) and then new VisEdge({ parent: mainBranch, source: C1Node, target: C2Node }) is created. This edge’s render will draw a vertical line (since same branch, x coordinates equal) from C2 up to C1. If a new branch commit is created (branching), e.g. commit C3 on bugFix branched from C1 on main, then:
  + bugFix branch’s first node C3 will have parentNode = C1 (on main).
  + An edge connecting C1 (source) to C3 (target) will be drawn, which will be diagonal (since different branch columns: C1 at x=0, C3 at x=1\*ROW\_WIDTH).
  + The edge might be drawn as a diagonal or possibly broken curve. LGB typically shows a diagonal line between the two. If a merge commit is made (merging bugFix into main, creating C4), then C4 will have two parents: one on main (C1 perhaps) and one on bugFix (C3). The visualization will draw two edges: one straight up to the main parent, and another angled from the side branch. This system cleanly separates visuals: commits (nodes) don’t draw their own lines; edges handle lines, making it easier to animate or style them separately.
* **VisTag (src/js/visuals/visTag.js)** – represents any **label or tag** attached to commits. This includes branch name labels (e.g. a rectangular label “main” next to a commit), the HEAD pointer, remote branch labels (o/main), and possibly commit message popups.
* Each VisTag is usually a small DOM element (perhaps a <span> or <div> with a special class like .gitTag or similar), absolutely positioned relative to its parent (which could be a branch container or the node itself).
* **Initialization**: It takes:
  + parent: Typically a VisBranch or Visualization container to append the tag.
  + targetNode: the VisNode it points to.
  + inline: a boolean indicating if the tag should sit *on the same horizontal level* as the node (inline=true means right beside the commit dot; inline=false might mean offset above it).
  + offset: pixel offset to adjust positioning (like to avoid overlapping multiple tags).
  + color: the background color for the tag (tags often share the branch color or a tag-specific color).
  + It creates an element (e.g. a <div class="gitTag">BranchName</div>) with the label text and styles it (background color, maybe a border).
  + Positions it: if inline, it might set the tag’s left position a bit to the right of the node; if not inline (like HEAD tag might hover above), it sets a bottom offset.
  + For HEAD, they often draw it as a small orange tag above the commit.
  + The code handles multiple tags on one node by stacking them with slight offsets.
* **Usage**: When a branch is created, a VisTag for the branch name is attached to the branch’s tip commit node (inline). When HEAD is on a branch, HEAD is typically not shown separately (the branch name itself is highlighted). But when HEAD is detached (pointing directly to a commit), a HEAD tag is attached to that commit (non-inline, above it).
* For remote branches (like o/main), tags would be drawn perhaps in a different color or shape (the code has constants for remote tag colors).
* The tags are crucial for UX to identify which commit is the tip of which branch. The code likely updates tag text and position whenever a branch moves (e.g., on commit, the branch’s tag needs to move to the new commit’s position – in practice, since they destroy/create VisNode and VisEdge on commit, they might also destroy the old tag and create a new one on the new node).
* **Putting it together**:
* The **Visualization (root)** contains a collection of VisBranch objects (branchCollection) and might also have a collection of all VisNodes (commitCollection) and VisTags.
* Initially, when the app loads a level or the sandbox, the Visualization is created and one initial branch (usually “main”) is set up with a root commit:
  + E.g., for a fresh repo: create main branch (VisBranch) with label:'main' and no parent branch → this will in turn create a root VisNode (C0) with no parent commit (so it’s drawn at the top of main column).
  + Mark that commit as a root commit (sometimes drawn differently, e.g., no parent edge).
  + HEAD is pointing to main, so possibly no separate HEAD tag is shown (since HEAD is just on main).
* If a tutorial level defines a more complex starting tree (some levels start with multiple branches pre-made), the code will build that by reading the JSON and calling appropriate VisBranch/VisNode creations to mirror the structure.
* With the structure in place, the visual tree is static until a command comes in.

**💡 Recipe Step 2: Implement the Visual Elements.** To replicate, create classes (or React components) for: - **Branch**: manages a list of Node components and has a name. - **Node (Commit)**: knows its position and parent, draws a dot. - **Edge**: draws a line between two nodes. - **Tag**: draws labels for branch names or HEAD. You can use an SVG library (like D3 or direct <svg> elements) to draw nodes and paths, or use HTML/CSS for nodes and an <svg> overlay for edges. The key is to maintain a consistent coordinate grid and update it as needed.

### Animation System: Bringing the Graph to Life

One of the most impressive UX aspects is how the commit tree animates when changes occur. The project achieves this with a custom **Animation Queue system** (under src/js/visuals/animation/). The core idea is that when a Git command changes the graph, a sequence of animations is generated to smoothly transition the old graph to the new graph state, rather than instantly redrawing.

Key parts of the animation system:

* **AnimationFactory (animationFactory.js)**: This module defines **factory functions** for different types of animations, such as:
* *commit birth animation* – e.g., popping in a new commit node with a bounce.
* *branch creation animation* – maybe a fade-in of a new branch label or a highlighting of the branch line.
* *checkout animation* – moving the HEAD tag from one branch to another.
* *merge animation* – perhaps highlighting two parent connections converging.
* *rebase animation* – moving commits from one branch to another, etc.

The AnimationFactory is essentially an object with methods like makeCommitBirthAnimation(gitVisuals, visNode), makeBranchMoveAnimation(branch, newTargetNode), etc. These methods create **Animation objects** (see below) configured with the specific DOM changes needed.

For example, a simplified pseudo for commit birth:

AnimationFactory.makeCommitBirthAnimation = function(gitVisuals, visNode) {  
 const time = GRAPHICS.defaultAnimationTime; // e.g. 400ms  
 const bounceTime = time \* 2;  
 // Define the actual animation behavior as a closure:  
 const closure = function() {  
 // e.g., animate the node from scale 0 to 1 (bounce effect)  
 $(visNode.el).css({ transform: 'scale(0)' });  
 $(visNode.el).animate({ transform: 'scale(1)' }, time, 'swing');  
 // Or use Raphael: visNode.circle.animate({ "r": finalRadius }, time, "bounce");  
 };  
 return new Animation({ duration: bounceTime, closure: closure });  
};

This returns an Animation model with the closure to run and a set duration.

The actual code uses promises (Q library) and might be more complex (ensuring the bounce overshoots then settles, etc.). But conceptually, it prepares a function that when called will carry out the visual change.

The AnimationFactory likely has a method to **queue up animations** given the difference in tree states. Possibly it has a function like AnimationFactory.applyTreeDiff(oldTree, newTree, gitVisuals) that: - Uses TreeCompare to identify changes (e.g., “commit X added”, “branch Y moved from commit A to B”). - For each change, push the corresponding animation into an AnimationQueue. - E.g., if a commit was added: queue a commit birth animation for that node, and an edge drawing animation for its connecting edge (maybe edges appear with a draw line animation). - If HEAD moved: queue an animation to move the HEAD tag from the old node to new node (this could be a simple position change of the HEAD element). - If a branch label moved (because the branch tip moved on commit): possibly an animation to fade out the tag at old commit and fade in on new commit, or simply instant reposition (the commit birth might implicitly carry the branch tag along). - If a commit was removed (like in reset or rebase drop): queue an animation to fade out or shrink that commit node and remove it. - If multiple things changed, it sequences them nicely (e.g., perhaps animate commit births one by one if multiple were created).

* **Animation and AnimationQueue (animation/index.js)**: These are defined as Backbone models:
* **Animation**: A model representing one atomic animation. It has attributes like duration and a closure (the function to run). It has a method .run() which simply executes the closure, and probably deals with any promise it returns or any cleanup.
* **AnimationQueue**: A model that manages a list of Animations. Key features of this queue:
  + It holds an array animations[] and an index pointer.
  + You can enqueue animations (probably by queue.add(animation) which pushes into the array).
  + It has methods to play through the queue sequentially. Typically, something like:
  + AnimationQueue.playAll = function(callback) {  
     // play each animation in sequence  
     if (this.index < this.animations.length) {  
     let anim = this.animations[this.index++];  
     anim.run();  
     setTimeout(() => { this.playAll(callback); }, anim.get('duration'));  
     } else {  
     // done all animations  
     if (callback) callback();  
     this.index = 0;  
     this.animations = [];  
     }  
    };
  + The real code uses promises (Q) for chaining. It might wrap each animation’s completion in a promise that resolves after duration, then use then to trigger the next. Also, if an animation’s closure returns a promise (for complex animations using CSS transitions or such), the queue likely waits (promiseBased logic was hinted, meaning it can integrate asynchronous animations).
  + It has thenFinish() to mark the final callback, and perhaps finish() to clean up.
  + The queue may optionally have a defer mode or promiseBased flag (from the code, possibly to handle waiting for user input in interactive sequences).
* **Global State Integration**: While animations run, the UI usually should prevent new commands. The code likely dispatches an action to indicate “animation in progress”. Indeed, Animation/index.js imports GlobalStateActions. Likely before playing the queue, it calls something like GlobalStateActions.startAnimating() and after completion calls GlobalStateActions.stopAnimating(), which the GlobalStateStore uses to disable the command input field or show a loading indicator. This ensures the user can’t spam commands and break the sequence. Once animations finish, input is re-enabled.
* **Example of an Animation Sequence** (Commit scenario):
* User types git commit.
* GitEngine creates a new commit in the data model (say C2 on main).
* The visualization diff finds: “New commit C2 on branch main” (also “branch main moved from C1 to C2”, and “HEAD moved from C1 to C2” if HEAD was on main).
* Animations prepared:
  + **Commit Birth** for C2: Create the VisNode for C2 (initially with scale 0 or opacity 0), then animate it to appear (pop out).
  + **Edge Draw** for C2: Create VisEdge from C1 to C2 (maybe initially invisible or very short), animate it (e.g., draw the line downward).
  + **Branch label move**: Actually the branch “main” label will now point to C2 instead of C1. The code might simply remove the tag from C1 and attach a new tag to C2. This could be instantaneous or animated (perhaps a slide).
  + **HEAD move**: if HEAD was on main, HEAD technically still on main, so no separate HEAD tag move. If HEAD was detached and user committed (creating a detached commit), then HEAD tag would move to the new commit – they’d animate the HEAD tag’s position.
* The AnimationQueue executes these in a nice order. Possibly they do the edge draw slightly after the node appears, etc., to create a smooth effect. All within ~400ms – it feels fluid to the user.
* **Parallel vs Sequential**: Most animations in LGB appear sequential but quickly done. The code might allow some overlap (like edge drawing while node bounces). Given the use of promises, it’s possible they trigger animations concurrently and wait for all to finish. The AnimationQueue class can likely handle both sequential and parallel (the presence of defer and promise logic hints at some flexibility, maybe certain animations wait on user action in interactive rebase).
* **Other animations**:
* **Branch creation** (no commit, just git branch): This doesn’t add a commit node, just adds a new branch tag on an existing commit. The animation might be a simple tag fade-in next to that commit.
* **Checkout**: This might highlight the HEAD movement. Possibly they flash the target branch label or animate a HEAD tag if detached.
* **Merge**: When a merge commit is created, the animations include the commit birth (with two edges). They might also briefly highlight the merged branch’s commits or animate the merge line differently.
* **Rebase**: This is more complex – rebase essentially removes some commits and re-adds them elsewhere. The UI often shows the commits being **copied** to the new location:
  + LGB likely implements rebase by *removing the original commits (or marking them as dropped)* and animating new commits on the target branch. They might fade out the old ones or draw the new ones with dashed outline first, etc. There is also an interactive rebase view implemented (rebaseView.js) which suggests some UI for picking commits.
  + For a simpler user perspective: rebase = new commits appear on new base, then old commits disappear. Animations would reflect that order.
* **Reset/Checkout detached**: If HEAD is moved to an older commit (detach or a reset --hard), the branch label might move or HEAD tag appears. Possibly the commits that were ahead are grayed out or something. (Not sure if LGB animates branch moves backwards or just jumps.)
* **User Interaction During Animations**: The user typically cannot issue a new command until animations finish (as mentioned, input is locked). However, they can *observe* and possibly click on commits to see messages (LGB on click shows the commit’s message/description). The code for clicking commits might be in the visualization: an event on VisNode element triggers a modal or tooltip. This likely uses the EventBaton and Backbone Views for modals.

**💡 Recipe Step 3: Implement Animations.** In replication, one could use CSS transitions or a library like GSAP for animations. The approach: - Determine state changes after each command. - For each change, define an animation: e.g., for a new node, create it with scale:0 then animate to scale 1; for a moved label, animate its position or opacity. - Either chain these via JavaScript (Promise/async or callback) or trigger simultaneous CSS transitions. - Ensure to **disable input** during animations and re-enable after – this is crucial for UX so the user doesn’t interrupt the sequence.

## User Interface & Controls

Aside from the animated branch tree, the application includes a rich UI for interacting with the tutorial. Notably: - **Command Line Input**: At the bottom (or top), there’s a text input where users type git commands. This is managed by a view (likely CommandLineStore and CommandView). When the user presses Enter, the command is parsed and dispatched. The code uses something like: - A React/Backbone view for the command line (e.g. CommandView.jsx). - CommandLineStore to hold the current input and command history. - On submit, an action CommandLineActions.submit(command) is fired, which the GitEngine listens to (or a controller in src/js/commands/index.js picks up and routes to GitEngine). - The command is executed, and either the state updates (leading to animations) or an error occurs (which is caught and perhaps displayed to the user – the errors.js utility defines custom error messages for common mistakes). - **Command History**: There is a panel showing past commands (in the UI, above the input). That is likely a React component (CommandHistoryView.jsx) that listens to the CommandLineStore. - **Output/Explanation**: After some commands (especially in tutorial levels), the UI might show messages or prompts. For example, if a level expects you to do something, hitting the wrong command might show a hint. The intl/strings.js contains lots of localized strings, and intl/index.js helps fetch the right language strings. These messages are displayed in modals or alerts: - ModalAlert and ModalConfirm are views (possibly Backbone or React) for showing prompts (like the initial tutorial instructions, or when you finish a level). - They use templates defined in template.index.html (the HTML has script templates for modals). - The UI calls things like Main.showModal(messageMarkdown) to display them. - **Level Selection and Progress**: The top menu (levels dropdown, etc.) is handled by LevelStore, LevelActions, and React views in react\_views/LevelToolbarView.jsx and levelDropdownView.js. This manages which level is loaded, and calls the LevelActions to load a level’s JSON from src/levels/. - When a level is loaded, the current visualization is reset and rebuilt according to the level’s starting tree (provided as startTree or by applying initial commands). - The goal tree (target end state) is stored, so the app can check if the user’s current state matches the goal (to mark level complete). This check is likely done by TreeCompare or similar mechanism to compare current graph vs goal graph string. - If complete, a modal “Level Completed!” is shown, and maybe LevelActions.completeLevel updates the UI (like unlocking next level). - **Helper Bar / Cheat Sheet**: There’s a sidebar or top bar that shows available commands and their usage. Files like CommandsHelperBarView.jsx and IntlHelperBarView.jsx suggest a dynamic display of help text, possibly localized. - **Keyboard Shortcuts**: The KeyboardListener (in util/keyboard.js) is used to capture keyboard events (like maybe arrows to navigate command history, or certain shortcuts to reset level, etc.). The EventBaton coordinates whether the keyboard events go to the command line (when focused) or elsewhere. - **Responsive Layout**: Constants in constants.js (VIEWPORT settings) indicate minimum width/height and zoom limits. The app likely adjusts zoom or pan if the commit tree grows large. Possibly, it allows zooming out to see the whole graph. The zoomLevel.js and some UI control might handle that (like +/- buttons or pinch on mobile). - **Styling**: The CSS (src/style/main.css, etc.) defines classes like: - .gitVis – base class for visual containers. - .gitNode – commit dot styling (size, border). - .gitEdge – perhaps styling for edges if drawn via HTML (or none if via SVG). - .gitTag – styling for labels (background color, rounded corners, small padding). - There are likely color definitions for branch colors (maybe generated on the fly though). - The CSS and constants ensure the look: e.g., default commit fill (rectFill) is a light color, HEAD tag fill (headRectFill) is a specific color (blue maybe), stroke width for commit outline, etc., which are used if Raphaël draws shapes.

**💡 Recipe Step 4: Build the UI and Controls.** In replication: - Create a simple web page with a **canvas area** (div or svg) for the visualization, and an input for commands. - Implement an event flow: on pressing Enter in input, call a function that processes the command (using your GitEngine from step 1), then triggers update of the visuals (from step 2) with animations (step 3). - Add UI components as needed: a panel to show command history (just a list appending each command), and an area to show messages/hints (you can start with console logs or alerts, then style nicely). - For level progression, you can hard-code one scenario or use the provided JSON structure idea to define initial/goal states.

## File-by-File: Code Walkthrough with Pseudocode

This section serves as a *recipe-like* walkthrough of the **key files** related to the animated branch tree, explaining their role and structure in pseudocode form. Follow this as a guideline to implement similar files in your project:

* **src/js/visuals/visBase.js** – *Visual Base Class*. Provides common properties and initialization for all visual elements.
* **Methods/Properties**:
  + init(params): Create this.el (a <div>), set class name, size (width/height if given), store hierarchy links (parent, rootTree, etc.), init animationQueue array. If type === 'tree', set up eventBaton for keyboard event management.
  + animationQueue: Array to hold pending Animation objects (used mostly at root).
  + isRoot: Boolean if this element is the main tree.
  + width()/height(): Default to 0 or to base dimensions; overridden by subclasses (e.g., commit node width).
* *Note:* The class likely uses a small inheritance helper since ES6 class extends might not be fully used. In the code, we see usage of VisBranch.super\_.prototype.init.apply(this, ...) which hints that inheritance is set up such that VisBranch.super\_ = VisBase. There might be a util like utils.inherits(VisBranch, VisBase) at the bottom of each file to wire up the prototype chain.
* **src/js/visuals/visBranch.js** – *Visual Branch Class*. Represents a branch column and its branch label.
* **Properties**:
  + id: Unique branch identifier (could be just the name or an internal numeric ID).
  + label: Branch name (string, e.g. "main").
  + nodeCollection: Array to hold VisNode objects that belong to this branch (commits in this branch).
  + node: Reference to the latest VisNode (tip of branch).
* **Methods**:
  + init(params):
  + params.type = 'branch'; call base init.
  + Set this.id (via global counter). Append this.el to parent container.
  + Set this.label from params.
  + Initialize nodeCollection (if branching from existing branch, might copy or start new).
  + Register with parent Visualization’s branch list.
  + Create initial commit node: this.node = new VisNode({ parent: this, ... parentNode: params.parentBranch?.node ... }) as described earlier.
  + (Potentially) create a branch label tag: The VisNode creation will attach a VisTag for the branch name automatically. If not, VisBranch might do it after node creation: e.g., this.tag = new VisTag({ parent: this, targetNode: this.node, label: this.label, color: branchColor }). But likely the VisNode’s inline tag already covers it.
  + getLatestNodesCollection(): (speculative) Maybe returns an array of nodes at the latest commit index (since branches could have multiple “strands” if merges? But likely one per branch).
  + assignBranchCollection(branch): A Visualization method that adds this new branch to an internal list branchCollection.
  + Other utility methods possibly like moving the branch pointer (but moving pointer is done by adding node, so branch doesn’t “move”, the tag does).
* **src/js/visuals/visNode.js** – *Visual Node (Commit) Class*. Represents a commit node graphic.
* **Properties**:
  + id: Commit identifier (like "C0", "C1", etc., or could be a unique internal number).
  + parent: The VisBranch container this node is attached to (visual parent).
  + parentNode: Reference to this commit’s parent commit’s VisNode (if any).
  + br: Branch index (number) this node is on.
  + level: Generation index (0 for root commit, increasing downwards).
  + commitColor: Color for the commit dot (usually derived from branch color).
  + tags: Array of VisTag labels attached (e.g., branch names, HEAD).
* **Methods**:
  + init(params):
  + params.type = 'node'; call base init.
  + Append to parent branch’s DOM.
  + Set this.br, this.level, this.parentNode as described.
  + Add self to branch’s nodeCollection.
  + Configure visual style settings (branchSpacing, baseWidth, colors).
  + Call this.position() to compute coordinates and set CSS position.
  + Call this.render() to create the DOM visuals (e.g., styling the .gitNode dot, perhaps using Raphaël).
  + Create a VisTag for branch label (inline) and store in this.tags. If a special tag (HEAD) is passed in params, attach it via detachTag() (which likely creates a non-inline tag positioned above).
  + position(): Compute (x, y) from this.br and this.level and update this.el style or internal meta.
  + render(): Actually draw the node. For instance, could set this.el.innerHTML = '<div class="commit-dot"></div>' or use an SVG circle. The code references constants like nodeRadius, rectStrokeWidth, etc., which suggests possibly Raphaël was used to draw a circle with a white border. If they used Raphaël:
  + They might store a this.circle = paper.circle(x, y, nodeRadius).attr({ fill: commitColor, stroke: "#FFF", "stroke-width": 3 }). In that case, this.el might still be a container but the actual circle is drawn on the Raphael canvas. The integration between Raphael and the DOM might be via absolutely positioning an SVG canvas under the .gitVis container.
  + Alternatively, if pure DOM, they might simply style the .gitNode div with border and background to look like that.
  + Possibly detachTag(tagName): if a tagName (like "HEAD") is given, it creates a VisTag for HEAD and positions it above this node. This is used when HEAD is not on a branch.
* **src/js/visuals/visEdge.js** – *Visual Edge Class*. Represents a connection line between commits.
* **Properties**:
  + source: VisNode (parent commit).
  + target: VisNode (child commit).
  + (If edges have direction, maybe outgoing flag or similar).
  + Possibly stores the SVG path element or DOM element that represents the line.
* **Methods**:
  + init(params):
  + params.type = 'edge'; call base init.
  + Append to parent container (likely the Visualization root or branch – but consider, if connecting commits on different branches, maybe better to append to root so it’s not cut off by branch container boundaries).
  + Set this.source = params.source, this.target = params.target.
  + Compute initial geometry: perhaps set this.baseWidth = 0 (for initial length).
  + Call this.changeLevel(0) – possibly a method that adjusts z-index or layering (ensuring edges appear behind nodes or something).
  + Call this.render():
    - This likely draws the line. Possibly calculates a path from source to target coordinates. If using Raphaël:
    - let x1 = source.meta.origin.x, y1 = source.meta.origin.y;  
      let x2 = target.meta.origin.x, y2 = target.meta.origin.y;  
      // If same branch (x1 == x2): draw straight line down.  
      // If different branch: draw a diagonal or curve.  
      let pathStr;  
      if (x1 === x2) {  
       pathStr = `M${x1},${y1} L${x2},${y2}`;  
      } else {  
       // Draw a curved path: e.g., quadratic Bezier with control point offset  
       let midY = (y1 + y2) / 2;  
       pathStr = `M${x2},${y2} Q${x2},${midY} ${x1},${y1}`;  
      }  
      this.path = paper.path(pathStr).attr({ stroke: sourceBranchColor, 'stroke-width': 2 });
    - The curveControlPointOffset constant could be used to tweak the Q control point x coordinate for nicer curves.
    - If not using Raphaël, they might absolutely position a long narrow div rotated to connect the dots – but curves would be hard that way, so Raphaël or <svg> path is more plausible.
  + The edge might also handle arrowheads if arrowsEnabled is true (drawing a small arrow at the end of the line, possibly using an SVG marker or by drawing a triangular path).
  + Possibly a method update(): If either source or target moves (during certain animations like rebase), update the path.
* **src/js/visuals/visTag.js** – *Visual Tag Class*. Represents a label attached to a commit.
* **Properties**:
  + targetNode: the VisNode this tag is for.
  + label or uses targetNode.parent.label for branch name (the code passes color and maybe the label implicitly).
  + inline: boolean (positioning mode).
  + offset: number of pixels to offset multiple tags.
* **Methods**:
  + init(params):
  + params.type = 'tag'; call base init.
  + Append to parent (which could be a branch container or maybe directly to the node’s element – but likely branch container for easier positioning in the same coordinate space).
  + Create a small DOM element (e.g., this.el = document.createElement('span') with class "gitTag").
  + Set text: if params.text provided use it, else if targetNode has a branch label use that, else if there’s an implied HEAD etc.
  + Style: background color = params.color (often the branch color or a constant for HEAD), perhaps a border, small padding, border-radius for the capsule shape.
  + Position it relative to the commit:
    - If inline: true, position it to the right of the commit dot. Typically an inline tag might be centered vertically with the dot, just shifted right. E.g., this.el.style.left = targetNode.meta.origin.x + nodeWidth + 5px; this.el.style.top = targetNode.meta.origin.y - someOffset;.
    - If inline: false, maybe position above the node: left = x; top = y - 20px.
    - If offset is given (for stacking), adjust top or left accordingly.
  + Possibly, if the tag is HEAD and the branch also has a tag, offset them not to overlap (maybe offset: 15 pixels vertically).
  + The tag might also attach event handlers (perhaps clicking a branch tag could highlight or offer to delete branch in sandbox mode).
  + render(): if not done in init, possibly fill innerHTML or further style adjustments.
* **src/js/visuals/visualization.js** – *Visualization Controller*. Orchestrates creation of all above and provides high-level methods.
* **Properties**:
  + branchCollection: list of all VisBranch.
  + commitCollection: list of all VisNode (if maintained globally).
  + tagCollection: list of all VisTag.
  + Possibly a reference to the Raphael paper or main SVG canvas context.
  + uniqueCounter for generating unique IDs.
* **Methods**:
  + init(params):
  + params.type = 'tree'; call base init (which sets up containerDiv etc.).
  + Initialize empty collections (branches, commits, tags).
  + If params has an initial tree structure, build it: e.g. iterate through each branch in data, create VisBranch and corresponding commits.
  + Provide utility functions:
    - createBranch(name, parentBranch): convenience to add a new branch to the visual. Uses new VisBranch({parent:this, label:name, parentBranch }).
    - removeBranch(name): removes branch visuals (perhaps not used often, except maybe in level resets).
    - addCommit(branchName): create a new commit on given branch. Finds the VisBranch, and does new VisNode({ parent: branch, parentNode: branch.node, branch: branch.index, ... }) and also updates branch.node to this new node. Also creates VisEdge from old branch.node to new node. Essentially wraps the lower-level calls.
    - moveHEAD(toBranchOrCommit): handles visuals when HEAD changes – e.g., highlight the HEAD tag on a branch or attach HEAD tag to a commit if detached.
    - uniqueIdForType(type): generates incremental IDs for 'node', 'branch', etc., to ensure each visual has a unique identifier (used in linking data structures or debugging).
  + Possibly clear() to remove all visuals (when loading a new level or resetting sandbox).
  + Many of these might be triggered by actions from the GitEngine:
  + For example, when GitEngine performs a commit, it could call visualization.addCommit(currentBranchName) which handles all visual parts; **however**, in this codebase, instead of directly calling such methods, they often rely on the diff+animation system. So it might be that visualization.js mostly sets up initial state and the AnimationFactory takes care of calling the right methods at animation time.
  + *Important:* The Visualization class might also subscribe to the Flux store or events. For instance, when LevelStore says "new level loaded with tree X", the Visualization will construct that tree. Or when CommandLineStore says "executed command", Visualization might get the diff and produce animations. Given the code uses Backbone and Flux, the integration might be via events rather than direct calls.
* **src/js/visuals/animation/index.js** – *Animation Models*. Defines:
* Animation (Backbone Model): with defaults duration (ms), closure (function to execute). It has initialize() which ensures closure is provided, and run() which simply calls the closure (and likely sets a timer or returns a promise).
* AnimationQueue (Backbone Model): with defaults animations: [], index: 0, callback: null, defer: false, promiseBased: false.
  + initialize(): set animations to empty array.
  + add(animation): push into animations array.
  + playNext() or similar: runs animations[index] and increments index. It might respect defer (which if true, maybe means wait for a user confirmation to proceed at certain breakpoints).
  + Uses Q.promise to chain animations end to calling finish() (calls final callback and resets).
  + The code also logs uncaught errors in animation promises to console, etc., as seen by snippet (fail function printing 'uncaught error').
  + Likely start() or playAll() is called to begin the sequence. Possibly GlobalStateActions are dispatched here: e.g.,
  + GlobalStateActions.startAnimation();   
    this.playAll(() => {  
     GlobalStateActions.endAnimation();  
     if (this.get('callback')) this.get('callback')();  
    });
  + This would lock UI until done and then call the provided callback (which might signal the command is fully applied).
* **src/js/visuals/animation/animationFactory.js** – *Animation Factory*. Contains a set of static functions or a singleton object that produces animations given certain events. Some likely functions:
* AnimationFactory.commitBirth(gitVisuals, visNode): returns an Animation that scales a new node from 0 to full size (with bounce).
* AnimationFactory.drawEdge(visEdge): returns an Animation that draws an edge (could animate the stroke drawing from source to target).
* AnimationFactory.fadeInTag(visTag): Animation to fade in a new branch label.
* AnimationFactory.moveTag(visTag, targetNode): if a tag (like HEAD) needs to move, animate its position change.
* AnimationFactory.removeNode(visNode): animate removing a node (e.g., shrinking or fading).
* Possibly composite ones like AnimationFactory.rebaseSequence(oldNodes, newNodes) which might orchestrate multiple animations (but likely the logic of rebase is handled at a higher level by issuing multiple simple animations in sequence).
* Internally, it likely uses the above AnimationQueue to schedule multiple animations. For example, if a command results in multiple changes, the factory might push several animations into the queue.
* It may also coordinate delays: e.g., after creating a commit (pop and edge draw), maybe a slight delay then highlight the branch tag. They might insert small timeout animations (an Animation whose closure is empty but has a duration, just to create a pause).
* The AnimationFactory is probably called from the Git module or from a controller after a command is executed. E.g., after running a command, the code might do:
* let diff = TreeCompare.compare(oldTree, newTree);  
  AnimationFactory.animateDiff(diff, visualization);
* And animateDiff will create a new AnimationQueue, fill it, and run it.
* **Other relevant files**:
* src/js/level/index.js – likely orchestrates building a level. It may take a level JSON (like the exports.level we saw) and:
  + Parse the startTree or apply initial commands to set up the GitEngine’s state.
  + Then call Visualization to render that state.
  + Possibly show the level’s intro dialog.
  + Check for goal after each command.
* src/js/views/gitDemonstrationView.js – This seems to be the main view that embeds everything. It probably extends a Backbone View (or uses the ContainedBase view from views/index.js). It likely:
  + Creates the Visualization instance: indeed we saw it requires Visualization = require('../visuals/visualization').Visualization. It might do this.vis = Visualization({...}) to initialize the visual area.
  + It might also handle the text console area (ModalTerminal).
  + Essentially, it’s the container for the tutorial interface, bringing together the terminal and the visualization side by side.
  + Pseudocode of its initialize might be:
  + initialize(options) {  
     // Set up DOM from template  
     this.$el.html(this.template());  
     // Create sub-views:  
     this.terminalView = new ModalTerminal({ el: this.$('.terminal-container') });  
     this.visualization = new Visuals(Object.assign({}, options.visualizationOpts, { containerDiv: this.$('.visualizer')[0] }));  
     // ...subscribe to events like command executed, etc.  
    }
  + If using React for some parts, this view might mount React components into certain regions.
* src/js/actions/GlobalStateActions.js & GlobalStateStore.js – manage global UI state (like whether a modal is open, or animating).
  + Possibly has actions like startLevel, resetLevel, toggleUIElement, etc., in addition to startAnimation/stopAnimation.
  + The store might have booleans like isAnimating, isModalOpen, which the React views or backbone views observe to disable/enable parts of UI.
* src/js/util/ – various helpers (debounce, throttle, etc.). Notably:
  + util/constants.js – contains values used across UI:
  + GRAPHICS.defaultAnimationTime (duration of animations, 400ms by default).
  + GRAPHICS.defaultEasing (easing function, e.g., 'easeInOut').
  + GRAPHICS.nodeRadius (radius of commit circle) and NODE\_WIDTH possibly doubling that.
  + Colors for commit fills and outlines: rectFill, headRectFill, etc.
  + Arrow settings.
  + Other UI constants like minZoom, etc.
  + util/utils.js – could contain getColorFromHash(name, baseColor) that generates a color for a branch name. Possibly it hashes the name string to a number and then uses HSB adjustments on a base color (ensuring each branch gets a distinct hue). This is how branch colors are consistent but arbitrary.
  + util/errors.js – defines error messages for common mistakes (like trying to checkout a non-existent branch).
  + util/eventBaton.js – class that manages focus between components (e.g., who handles keyboard events – the command line or an open modal, etc.). The root visualization might hold it, passing it to modals when needed.
  + util/keyboard.js – sets up key bindings (like up/down for command history, or maybe shortcuts to skip levels, etc.).
* **React Components (src/js/react\_views/\*.jsx)** – not the main focus of this tree animation discussion, but for completeness:
* MainHelperBarView.jsx, CommandsHelperBarView.jsx, etc., are likely small info panels in the UI (maybe listing available commands or internationalization help).
* They connect to stores like LocaleStore (for language changes) and CommandLineStore (for suggestions).
* Using React for these parts suggests they can update without full page refresh (e.g., dynamically show translation or command usage hints as user types).
* **Styling Files**:
* src/style/main.css – global styles. Key classes likely:
  + .gitVis – base style for visualization containers (maybe position: relative; overflow: visible;).
  + .gitNode – style for commit dot (width/height = 34px say (2\*radius), border radius 50%, background-color from script, border white).
  + .gitTag – style for labels (display: inline-block; padding: 2px 4px; font-size: 12px; background: branchColor; color: #fff; border-radius: 4px; margin-left: 5px; user-select: none).
  + .gitTag.head or similar for HEAD tags (maybe a different color like headRectFill which might be purple or orange to stand out).
  + .gitEdge if exists, or edges might be drawn in an <svg> that is absolutely positioned under .gitVis container (with pointer-events none, so clicks go to nodes).
  + Layout CSS for positioning the console and visual next to each other (flex containers etc., given classes like box flex1 seen in templates).
  + Rainbows.css – possibly an easter egg or styling for celebratory animations (maybe confetti when level complete?).

With these files understood, one can navigate the codebase effectively or rebuild a similar architecture.

## Conclusion & Replication Guide

Recreating **LearnGitBranching** from scratch is a non-trivial project, but with this deep dive, you have a clear *recipe*:

1. **Set up Data Structures and Logic** – Implement the Git simulation (Commits, Branches, HEAD) and ensure you can apply commands to mutate the state. Keep track of state changes.
2. **Design the Visualization Layer** – Decide on a rendering approach (SVG or DOM). Implement visual classes for nodes, branches, edges, tags akin to VisNode, VisBranch, etc. Make sure you can lay out commits in a grid and draw connecting lines correctly.
3. **Integrate the Two with an Animation System** – After each command, compute what changed and animate it. Start with simple transitions (e.g., fade in a new commit) and build up (slide branch labels, etc.). Use a queue/promise system to sequence animations if multiple happen together.
4. **Build the UI** – Create the input console and link it to your command execution. Display helpful messages or errors to the user. If building a tutorial, create a way to load predefined scenarios (levels) and check goals.
5. **Polish the UX** – Add nice touches: color-code branches, bounce new commits, highlight merges. Ensure the interface is responsive (maybe allow panning/zooming the visualization if it overflows).
6. **Test with Various Git Scenarios** – Try branching, merging, rebasing to see if visuals hold up. Fine-tune animation timings (the given default is ~400ms with ease-in-out which feels snappy).

By following the structure above – which mirrors the LearnGitBranching codebase – you can replicate an **exact project** of a visual Git branching game. The key is careful management of state and visuals, and a robust animation queue to tie them together. Happy coding!

**Sources:**

* LearnGitBranching README – project purpose and high-level idea.
* Package.json – confirming libraries used (Backbone, React, jQuery, Raphaël, etc.).  
  *(The code analysis is based on the user-provided repository files.)*