# Detailed Explanation of the WebGraph Program

This program is designed to fetch and visualize a hierarchical structure of web pages starting from a given URL. It utilizes SFML for graphical rendering and libcurl for HTTP requests. The key aspects of the program include polymorphism with multiple levels of inheritance, separate module or header files for class declarations and definitions, and the usage of standard library containers and algorithms.

1. Polymorphism with Multiple Levels of Inheritance

The program uses polymorphism and inheritance through the PageNode base class and its derived classes: RootNode, FirstDepthNode, SecondDepthNode, and ThirdDepthNode.

PageNode (Base Class):

- This class provides the common interface for all types of nodes. It holds the URL, position, depth, and child nodes.
- It declares a pure virtual function draw that must be overridden by derived classes to handle their specific drawing logic.
- Header: PageNode.h
- Implementation: PageNode.cpp

RootNode (First Derived Class):

- Inherits from PageNode and represents the root URL.
- Overrides the draw method to define how the root node should be displayed.
- Header: RootNode.h
- Implementation: RootNode.cpp

FirstDepthNode (Second Derived Class):

- Inherits from PageNode and represents the first level of URLs.
- Overrides the draw method to define how these nodes should be displayed.
- Header: FirstDepthNode.h
- Implementation: FirstDepthNode.cpp

SecondDepthNode (Third Derived Class):

- Inherits from PageNode and represents the second level of URLs.
- Overrides the draw method to define how these nodes should be displayed.
- Header: SecondDepthNode.h
- Implementation: SecondDepthNode.cpp

ThirdDepthNode (Fourth Derived Class):

- Inherits from PageNode and represents the third level of URLs.
- Overrides the draw method to define how these nodes should be displayed.
- Header: ThirdDepthNode.h
- Implementation: ThirdDepthNode.cpp

2. Classes Declared and Defined in Separate Module or Header Files

Each class is declared in a header file and defined in a corresponding implementation file. This separation of interface and implementation improves code organization and maintainability.

Headers:
- PageNode.h
- RootNode.h
- FirstDepthNode.h
- SecondDepthNode.h
- ThirdDepthNode.h
- HTML_FETCHER.h

Implementations:
- PageNode.cpp
- RootNode.cpp
- FirstDepthNode.cpp
- SecondDepthNode.cpp
- ThirdDepthNode.cpp
- HTML_FETCHER.cpp

3. Usage of Standard Library Containers and Algorithms
The program makes extensive use of standard library containers and algorithms:

Containers:
- std::vector is used to store child nodes in PageNode and derived classes.
- std::vector is also used to store URLs in HTML_FETCHER.cpp.

Algorithms:
- The program uses the push_back method to add elements to std::vector.
- It employs STL iterators for iterating over std::vector elements.
- It uses std::regex to match and transform URLs.

Program Flow

1. Initialization:

- The user is prompted to enter a website address.
- This URL is added to a vector of initial URLs.

2. Fetching Links:

- fetchNestedLinks is called with the initial URLs and a specified depth (3 in this case).
- This function recursively fetches HTML content from URLs, extracts links, and creates a hierarchy of PageNode objects.

3. Rendering:

- An SFML window is created for graphical visualization.
- The font is loaded from a predefined list of paths.
- The positions for the nodes are set using setPositionForNodes.
- The main rendering loop handles events (mouse movement, zooming, and window close) and draws the nodes and their links.

4. Node Drawing:

- Each node type (RootNode, FirstDepthNode, SecondDepthNode, ThirdDepthNode) has its own implementation of the draw method, determining its appearance.
- Lines are drawn between parent and child nodes to represent the hierarchy

5. Cleanup:

- After the window is closed, dynamically allocated PageNode objects are deleted to free memory.

## Detailed Explanation of Fetching Links

[step 2 in program flow]

Fetching links in this program involves retrieving the HTML content of web pages, parsing that content to extract links, and recursively building a hierarchical structure of these links up to a specified depth. This process uses several key components and external libraries, including cURL for HTTP requests and Gumbo for HTML parsing. Here's a detailed breakdown of how this works:

### 1. Fetching HTML Content

The program fetches the HTML content of a given URL using cURL. This is encapsulated in the fetchHTML function:

```
std::string fetchHTML(const std::string& url) {
    CURL* curl;
    CURLcode res;
    std::string readBuffer;

    curl = curl_easy_init();
    if (curl) {
        curl_easy_setopt(curl, CURLOPT_URL, url.c_str());
        curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, WriteCallback);
        curl_easy_setopt(curl, CURLOPT_WRITEDATA, &readBuffer);
        res = curl_easy_perform(curl);
        if (res != CURLE_OK) {
            std::cerr << "cURL error: " << curl_easy_strerror(res) << " for URL: " << url <<
std::endl;
        }
        curl_easy_cleanup(curl);
    }
```

```
    return readBuffer;
}
```

- curl_easy_init initializes a cURL session.
- curl_easy_setopt sets various options:
    - CURLOPT_URL sets the URL to fetch.
    - CURLOPT_WRITEFUNCTION specifies a callback function (WriteCallback) to handle incoming data.
    - CURLOPT_WRITEDATA specifies a pointer to pass to the write callback function.
- curl_easy_perform performs the request and stores the result in readBuffer.
- If an error occurs, curl_easy_strerror(res) prints an error message.
- curl_easy_cleanup cleans up the cURL session.

The WriteCallback function is defined as follows:

```
static size_t WriteCallback(void* contents, size_t size, size_t nmemb, void* userp) {
    ((std::string*)userp)->append((char*)contents, size * nmemb);
    return size * nmemb;
}
```

- This callback appends the fetched data to a std::string object.

2. Extracting Links from HTML

After fetching the HTML, the program uses the Gumbo parser to extract links. This is done in the extractLinks function:

```
std::vector<std::string> extractLinks(const std::string& base_url, const std::string& html) {
    GumboOutput* output = gumbo_parse(html.c_str());
    std::vector<std::string> links;
    searchForLinks(output->root, links);
    gumbo_destroy_output(&kGumboDefaultOptions, output);

    std::vector<std::string> absolute_links;
    std::regex url_regex("(https?|ftp)://([^\\s/$.?#].[^\\s]*)");
    std::smatch url_match_result;

    for (const std::string& link : links) {
        if (std::regex_match(link, url_match_result, url_regex)) {
            absolute_links.push_back(link);  // It's already an absolute URL
        } else {
            // Assume the link is a relative URL
            std::string absolute_link = base_url + link;
            absolute_links.push_back(absolute_link);
        }
    }
    return absolute_links;
}
```

- gumbo_parse parses the HTML content.
- searchForLinks recursively searches for anchor (<a>) tags and extracts their href attributes.
- gumbo_destroy_output cleans up the Gumbo output.

The searchForLinks function is defined as follows:

```
void searchForLinks(GumboNode* node, std::vector<std::string>& links) {
    if (node->type != GUMBO_NODE_ELEMENT) {
        return;
    }

    GumboAttribute* href;
    if (node->v.element.tag == GUMBO_TAG_A &&
        (href = gumbo_get_attribute(&node->v.element.attributes, "href"))) {
        links.push_back(href->value);
    }

    const GumboVector* children = &node->v.element.children;
    for (unsigned int i = 0; i < children->length; ++i) {
        searchForLinks(static_cast<GumboNode*>(children->data[i]), links);
    }
}
```

- This function checks if a node is an anchor tag (GUMBO_TAG_A).
- If so, it extracts the href attribute and adds it to the links vector.
- It then recursively processes all child nodes.

3. Creating Page Nodes

The program defines several types of nodes to represent pages at different depths in the hierarchy. Each type of node is a subclass of PageNode. The createNode function creates the appropriate type of node based on the current depth:

```
PageNode* createNode(const std::string& url, int depth) {
    switch (depth) {
        case 0:
            return new RootNode(url);
        case 1:
            return new FirstDepthNode(url);
        case 2:
            return new SecondDepthNode(url);
        case 3:
            return new ThirdDepthNode(url);
        default:
            return nullptr;
    }
}
```

- Depending on the depth parameter, it returns an instance of RootNode, FirstDepthNode, SecondDepthNode, or ThirdDepthNode

4. Fetching Nested Links

The core function for fetching links and building the hierarchy is fetchNestedLinks:

```
std::vector<PageNode*> fetchNestedLinks(const std::vector<std::string>& urls, int depth) {
    std::vector<PageNode*> result;

    if (depth < 0) {
        return result;
    }

    for (const auto& url : urls) {
        if (url.empty()) {
            continue;  // Skip empty URLs
        }

        PageNode* node = createNode(url, depth);
        std::string html = fetchHTML(url);

        if (html.empty()) {
            continue;  // Skip if failed to fetch HTML
        }

        std::vector<std::string> links = extractLinks(url, html);

        if (depth > 0) {
            std::vector<PageNode*> nestedLinks = fetchNestedLinks(links, depth - 1);
            for (auto nestedNode : nestedLinks) {
                node->addChild(nestedNode);
            }
        }

        result.push_back(node);
    }

    return result;
}
```

- It takes a list of URLs and a depth parameter.
- If depth is negative, it returns an empty result.
- For each URL:
    - It creates a node using createNode.
    - Fetches the HTML content using fetchHTML.
    - Extracts links from the HTML using extractLinks.
    - Recursively fetches nested links if depth is greater than 0.
    - Adds each nested link as a child of the current node using node->addChild.
    - Adds the current node to the result list.
- Returns the list of nodes.

5. Integrating Fetching and Rendering
In main.cpp, the fetched nodes are used to build the graph structure that is later rendered:

```
std::vector<PageNode*> nestedLinks = fetchNestedLinks(initialUrls, 3);
if (nestedLinks.empty()) {
    std::cerr << "No links found. Exiting..." << std::endl;
```

```
    return -1;
}

// Positioning and rendering code follows...
```

- initialUrls contains the starting URL(s) entered by the user.
- The fetchNestedLinks function is called with a depth of 3 to build the hierarchical structure.
- If no links are found, the program exits with an error message.
- If links are found, they are positioned and rendered as described in the rendering explanation.

Conclusion: Explanation of Fetching Links

The process of fetching links in this program involves multiple steps:
Fetching HTML content from given URLs using cURL.
Parsing the HTML content to extract links using the Gumbo parser.
Creating hierarchical PageNode objects to represent the structure.
Recursively fetching nested links up to a specified depth.
This process is encapsulated in functions that abstract the details of HTTP requests and HTML parsing, making the code modular and maintainable. The fetched links are then integrated into the rendering process to visualize the hierarchy.

## Detailed Explanation of Rendering

[step 2 in program flow]

Rendering in this program involves visualizing a hierarchical graph structure of web pages using SFML (Simple and Fast Multimedia Library). The rendering process includes setting up the window, handling user input events, positioning the nodes, and drawing the nodes and the connecting lines. Here's a step-by-step breakdown of the rendering process:

1. Setting Up the Window

In the main.cpp file, the program initializes an SFML window for rendering the graph:

```
sf::RenderWindow window(sf::VideoMode(WINDOW_WIDTH, WINDOW_HEIGHT), "Graph Visualization");
```

- WINDOW_WIDTH and WINDOW_HEIGHT are constants defining the size of the window.
- "Graph Visualization" is the title of the window.

2. Loading the Font

The program attempts to load a font for rendering text labels on the nodes. It tries several paths to ensure compatibility across different operating systems:

```
sf::Font font;
```

```
std::vector<std::string> fontPaths = {
    "/usr/share/fonts/truetype/dejavu/DejaVuSans.ttf",
    "/Library/Fonts/Arial Unicode.ttf",
    "C:\\Windows\\Fonts\\arial.ttf"
};

bool fontLoaded = false;
for (const auto& fontPath : fontPaths) {
    if (font.loadFromFile(fontPath)) {
        fontLoaded = true;
        break;
    }
}

if (!fontLoaded) {
    std::cerr << "Failed to load any of the default fonts" << std::endl;
    return -1;
}
```

- The program checks each path in fontPaths and attempts to load the font using font.loadFromFile(fontPath).
- If none of the fonts can be loaded, the program prints an error message and exits.

3. Positioning the Nodes

Before rendering, the program calculates positions for each node to visualize the hierarchical structure:

```
sf::Vector2f startPosition(WINDOW_WIDTH / 2, WINDOW_HEIGHT / 2);
float verticalSpacing = 150.0f;
setPositionForNodes(nestedLinks[0], startPosition, verticalSpacing, 0);
```

- startPosition is the initial position of the root node, centered in the window.
- verticalSpacing defines the initial distance between levels in the hierarchy.
- setPositionForNodes is a recursive function that positions each node and its children.

The setPositionForNodes function is defined as follows:

```
void setPositionForNodes(PageNode* node, sf::Vector2f& startPos, float verticalSpacing, int
depth) {
    static const float angleIncrement = 30.0f;
    float angle = 0.0f;

    for (auto& child : node->getChildren()) {
        float xOffset = cos(angle) * verticalSpacing;
        float yOffset = sin(angle) * verticalSpacing;
        sf::Vector2f childPos = startPos + sf::Vector2f(xOffset, yOffset);
        child->setPosition(childPos);
        setPositionForNodes(child, childPos, verticalSpacing / 1.5f, depth + 1);
```

```
        angle += angleIncrement;
    }
}
```

- The function positions each child node around its parent in a circular pattern.
- The angle between child nodes is incremented by angleIncrement (30 degrees).
- verticalSpacing decreases with depth to create a tree-like structure.

4. Handling User Input Events

The main loop of the program handles user input events to interact with the visualization:

```
while (window.isOpen()) {
    sf::Event event;
    while (window.pollEvent(event)) {
        if (event.type == sf::Event::Closed) {
            window.close();
        } else if (event.type == sf::Event::MouseMoved) {
            mousePosition = static_cast<sf::Vector2f>(sf::Mouse::getPosition(window));
        } else if (event.type == sf::Event::MouseWheelScrolled) {
            if (event.mouseWheelScroll.delta > 0) {
                zoom *= ZOOM_FACTOR;
            } else {
                zoom /= ZOOM_FACTOR;
            }
        }
    }
```

- window.pollEvent(event) retrieves all pending events.
- sf::Event::Closed closes the window when the user attempts to close it.
- sf::Event::MouseMoved updates the mousePosition to the current mouse position.
- sf::Event::MouseWheelScrolled adjusts the zoom factor to zoom in and out based on the mouse wheel movement.

5. Drawing the Nodes and Links

The drawPageLinks function recursively draws each node and the lines connecting them:

```
void drawPageLinks(sf::RenderWindow& window, const PageNode* node, sf::Font& font, const
sf::Vector2f& mousePosition, float zoom) {
    node->draw(window, font, mousePosition, zoom);
    for (const auto& child : node->getChildren()) {
        sf::Vertex line[] = {
            sf::Vertex(node->getPosition(), sf::Color::Black),
            sf::Vertex(child->getPosition(), sf::Color::Black)
        };
        window.draw(line, 2, sf::Lines);
        drawPageLinks(window, child, font, mousePosition, zoom);
    }
```

```
}
```

- The node->draw method is called to render the node itself. This method is overridden in each derived class to define specific drawing behavior.
- Lines (sf::Vertex line[]) are drawn between the node and its children using window.draw(line, 2, sf::Lines).
- The function recursively calls itself for each child node to render the entire hierarchy.

Each node type (e.g., RootNode, FirstDepthNode) has its own implementation of the draw method:

```
void RootNode::draw(sf::RenderWindow& window, sf::Font& font, const sf::Vector2f&
mousePosition, float zoom) const {
    sf::CircleShape shape(10 * zoom);
    shape.setFillColor(sf::Color::Black);
    shape.setPosition(position);
    window.draw(shape);

    sf::Text text;
    text.setFont(font);
    text.setString(url);
    text.setCharacterSize(12);
    text.setFillColor(sf::Color::Black);
    text.setPosition(position.x + 15 * zoom, position.y);
    window.draw(text);
}
```

- A sf::CircleShape is created to represent the node visually.
- shape.setFillColor(sf::Color::Black) sets the color of the node.
- shape.setPosition(position) positions the node at its calculated position.
- The node is drawn using window.draw(shape).
- A sf::Text object is created to display the node's URL.
- The text is drawn using window.draw(text).

6. Main Rendering Loop

The main rendering loop clears the window, draws all nodes and links, and displays the result:

```
while (window.isOpen()) {
    sf::Event event;
    while (window.pollEvent(event)) {
        // Event handling
    }

    window.clear(sf::Color::White);

    for (const auto& node : nestedLinks) {
        drawPageLinks(window, node, font, mousePosition, zoom);
    }
```

```
    window.display();
}
```

- window.clear(sf::Color::White) clears the window with a white background.
- drawPageLinks is called for each top-level node in nestedLinks to draw the entire graph.
- window.display() updates the window to show the rendered frame.

7. Cleanup

After the window is closed, the program cleans up dynamically allocated PageNode objects:

```
for (auto node : nestedLinks) {
    delete node;
}
```

- This ensures that all memory allocated for the nodes is properly released.

Conclusion: Explanation of Rendering

The rendering process in this program is carefully structured to visualize a hierarchical web of nodes using SFML. It involves setting up the window, loading fonts, positioning nodes, handling user input, and recursively drawing nodes and their connections. The use of polymorphism allows different types of nodes to be rendered differently while maintaining a consistent interface. This design ensures flexibility and extensibility in the rendering logic.

## Detailed Explanation of Node Drawing

[step 2 in program flow]

In this program, the node drawing mechanism involves rendering different types of PageNode objects onto an SFML window. Each node type is represented by a specific class derived from the abstract base class PageNode. The drawing process is responsible for visually representing nodes and their connections on the screen, using SFML for rendering.

Class Structure for Nodes

The program defines several classes to represent nodes at different depths:

- PageNode: Abstract base class.
- RootNode: Derived class representing the root node.
- FirstDepthNode: Derived class representing nodes at the first depth level.
- SecondDepthNode: Derived class representing nodes at the second depth level.
- ThirdDepthNode: Derived class representing nodes at the third depth level.

Each of these classes has its own implementation of the draw method, which handles the specifics of rendering the node.

Base Class: PageNode

The PageNode class provides the common interface and storage for node attributes like URL, position, and children:

PageNode.h:

```
#ifndef PageNode_h
#define PageNode_h

#include <string>
#include <vector>
#include <SFML/Graphics.hpp>

class PageNode {
public:
    PageNode(const std::string& url, int depth);
    virtual ~PageNode();

    void addChild(PageNode* child);
    const std::vector<PageNode*>& getChildren() const;

    virtual void draw(sf::RenderWindow& window, sf::Font& font, const sf::Vector2f&
mousePosition, float zoom) const = 0;

    void setPosition(const sf::Vector2f& pos);
    sf::Vector2f getPosition() const;

    int getDepth() const;
    std::string getUrl() const;

protected:
    std::string url;
    std::vector<PageNode*> children;
    sf::Vector2f position;
    int depth;
};

#endif /* PageNode_h */
```

PageNode.cpp:

```
#include "PageNode.h"
```

```
PageNode::PageNode(const std::string& url, int depth)
    : url(url), position(0, 0), depth(depth) {}

PageNode::~PageNode() {
    for (auto child : children) {
        delete child;
    }
}

void PageNode::addChild(PageNode* child) {
    children.push_back(child);
}

const std::vector<PageNode*>& PageNode::getChildren() const {
    return children;
}

void PageNode::setPosition(const sf::Vector2f& pos) {
    position = pos;
}

sf::Vector2f PageNode::getPosition() const {
    return position;
}

int PageNode::getDepth() const {
    return depth;
}

std::string PageNode::getUrl() const {
    return url;
}
```

Derived Classes for Specific Node Types

Each derived class overrides the draw method to provide specific rendering logic.

RootNode

RootNode.h:

```
#ifndef RootNode_h
#define RootNode_h

#include "PageNode.h"

class RootNode : public PageNode {
public:
    RootNode(const std::string& url);
```

```
    void draw(sf::RenderWindow& window, sf::Font& font, const sf::Vector2f& mousePosition,
float zoom) const override;
};

#endif /* RootNode_h */
```

RootNode.cpp:

```
#include "RootNode.h"

RootNode::RootNode(const std::string& url)
    : PageNode(url, 0) {}

void RootNode::draw(sf::RenderWindow& window, sf::Font& font, const sf::Vector2f&
mousePosition, float zoom) const {
    sf::CircleShape shape(10 * zoom);
    shape.setFillColor(sf::Color::Black);
    shape.setPosition(position);
    window.draw(shape);

    sf::Text text;
    text.setFont(font);
    text.setString(url);
    text.setCharacterSize(12);
    text.setFillColor(sf::Color::Black);
    text.setPosition(position.x + 15 * zoom, position.y);
    window.draw(text);
}
```

FirstDepthNode

FirstDepthNode.h:

```
#ifndef FirstDepthNode_h
#define FirstDepthNode_h

#include "PageNode.h"

class FirstDepthNode : public PageNode {
public:
    FirstDepthNode(const std::string& url);

    void draw(sf::RenderWindow& window, sf::Font& font, const sf::Vector2f& mousePosition,
float zoom) const override;
};

#endif /* FirstDepthNode_h */
```

FirstDepthNode.cpp:

```
#include "FirstDepthNode.h"
```

```
FirstDepthNode::FirstDepthNode(const std::string& url)
    : PageNode(url, 1) {}

void FirstDepthNode::draw(sf::RenderWindow& window, sf::Font& font, const sf::Vector2f&
mousePosition, float zoom) const {
    sf::CircleShape shape(6 * zoom);
    shape.setFillColor(sf::Color::Red);
    shape.setPosition(position);
    window.draw(shape);

    sf::Text text;
    text.setFont(font);
    text.setString(url);
    text.setCharacterSize(12);
    text.setFillColor(sf::Color::Black);
    text.setPosition(position.x + 10 * zoom, position.y);
    window.draw(text);
}
```

SecondDepthNode

SecondDepthNode.h:

```
#ifndef SecondDepthNode_h
#define SecondDepthNode_h

#include "PageNode.h"

class SecondDepthNode : public PageNode {
public:
    SecondDepthNode(const std::string& url);

    void draw(sf::RenderWindow& window, sf::Font& font, const sf::Vector2f& mousePosition,
float zoom) const override;
};

#endif /* SecondDepthNode_h */
```

SecondDepthNode.cpp:

```
#include "SecondDepthNode.h"

SecondDepthNode::SecondDepthNode(const std::string& url)
    : PageNode(url, 2) {}

void SecondDepthNode::draw(sf::RenderWindow& window, sf::Font& font, const sf::Vector2f&
mousePosition, float zoom) const {
    sf::CircleShape shape(4 * zoom);
    shape.setFillColor(sf::Color::Blue);
    shape.setPosition(position);
    window.draw(shape);
```

```
    sf::Text text;
    text.setFont(font);
    text.setString(url);
    text.setCharacterSize(12);
    text.setFillColor(sf::Color::Black);
    text.setPosition(position.x + 5 * zoom, position.y);
    window.draw(text);
}
```

ThirdDepthNode

ThirdDepthNode.h:

```
#ifndef ThirdDepthNode_h
#define ThirdDepthNode_h

#include "PageNode.h"

class ThirdDepthNode : public PageNode {
public:
    ThirdDepthNode(const std::string& url);

    void draw(sf::RenderWindow& window, sf::Font& font, const sf::Vector2f& mousePosition,
float zoom) const override;
};

#endif /* ThirdDepthNode_h */
```

ThirdDepthNode.cpp:

```
#include "ThirdDepthNode.h"

ThirdDepthNode::ThirdDepthNode(const std::string& url)
    : PageNode(url, 3) {}

void ThirdDepthNode::draw(sf::RenderWindow& window, sf::Font& font, const sf::Vector2f&
mousePosition, float zoom) const {
    sf::CircleShape shape(2 * zoom);
    shape.setFillColor(sf::Color::Green);
    shape.setPosition(position);
    window.draw(shape);

    sf::Text text;
    text.setFont(font);
    text.setString(url);
    text.setCharacterSize(12);
    text.setFillColor(sf::Color::Black);
    text.setPosition(position.x + 3 * zoom, position.y);
    window.draw(text);
}
```

Drawing Nodes in main.cpp

The drawPageLinks function in main.cpp recursively draws each node and the links between them:

main.cpp:

```
void drawPageLinks(sf::RenderWindow& window, const PageNode* node, sf::Font& font, const
sf::Vector2f& mousePosition, float zoom) {
    node->draw(window, font, mousePosition, zoom);
    for (const auto& child : node->getChildren()) {
        sf::Vertex line[] = {
            sf::Vertex(node->getPosition(), sf::Color::Black),
            sf::Vertex(child->getPosition(), sf::Color::Black)
        };
        window.draw(line, 2, sf::Lines);
        drawPageLinks(window, child, font, mousePosition, zoom);
    }
}
```

- The function takes a reference to the SFML RenderWindow, the current node, font, mouse position, and zoom level.
- It calls the node's draw method to render it.
- It iterates over the node's children, drawing lines between the node and each child.
- It recursively calls itself for each child to draw the entire tree structure.

Integrating Drawing with Event Handling

The main event loop in main.cpp handles window events and triggers the drawing process:

```
while (window.isOpen()) {
    sf::Event event;
    while (window.pollEvent(event)) {
        if (event.type == sf::Event::Closed) {
            window.close();
        } else if (event.type == sf::Event::MouseMoved) {
            mousePosition = static_cast<sf::Vector2f>(sf::Mouse::getPosition(window));
        } else if (event.type == sf::Event::MouseWheelScrolled) {
            if (event.mouseWheelScroll.delta > 0) {
                zoomLevel *= 1.1f;
            } else {
                zoomLevel /= 1.1f;
            }
        }
    }

    window.clear(sf::Color::White);
    for (const auto& node : nestedLinks) {
        drawPageLinks(window, node, font, mousePosition, zoomLevel);
    }
    window.display();
```

```
}
```

- The event loop handles closing the window, updating the mouse position, and zooming in and out.
- After handling events, it clears the window.
- It calls drawPageLinks for each top-level node in nestedLinks to render the entire hierarchy.
- Finally, it displays the rendered frame.

Conclusion

The node drawing mechanism in this program involves the following steps:

- Defining Node Classes: Abstract base class PageNode and derived classes RootNode, FirstDepthNode, SecondDepthNode, and ThirdDepthNode.
- Implementing the draw Method: Each node type has its own rendering logic in its draw method.
- Drawing Nodes and Links: The drawPageLinks function recursively draws nodes and the lines connecting them.
- Handling Events: The main event loop processes user input and triggers the drawing process.

This structure ensures that each type of node is rendered appropriately and that the hierarchical structure is visually represented in the SFML window.