MATLAB Interface for RaycasterGL Rendering Engine (renderMex)

Version: 0.2.0 **Date:** April 1, 2025 **Author:** Jonathan Ladden

1. Introduction

This document describes the MATLAB interface for the "RaycasterGL" rendering engine. This engine, written in C++ and utilizing the Raylib library (via vcpkg, including GLFW), is exposed to MATLAB through a compiled **MEX function** (`renderMex.mexw64` on 64-bit Windows).

To simplify interaction with the underlying MEX function and provide a more MATLAB-idiomatic experience, a set of **MATLAB wrapper functions (`.m` files)** are provided. These wrappers handle calling the `renderMex` function with the correct command strings and arguments, provide input validation, offer help text, and enable MATLAB's standard autocomplete features.

Core Purpose: To allow MATLAB users to open a rendering window and draw basic 2D primitives (text, rectangles, lines) by calling simple MATLAB functions, leveraging the performance of the C++ backend.

Underlying Mechanism:

1. MATLAB calls a wrapper function (e.g., `renderDrawRect(...)`).
2. The wrapper function validates inputs and constructs a call to the MEX function, passing a command string and arguments (e.g., `renderMex('drawRect', x, y, w, h, color)`).
3. The compiled C++ code within `renderMex.mexw64` receives the command and arguments.
4. It parses the command and calls the appropriate function within the linked `RaycasterGL` static library.
5. The `RaycasterGL` library function calls Raylib/OpenGL functions to perform the rendering.

2. Setup and Prerequisites

Before using these functions, ensure the following requirements are met:

1. **MATLAB Installation:** A compatible version of MATLAB (likely R2019b or later for the `arguments`

block syntax, confirmed working with R2024b during testing).

2. **C++ Compiler:** A compatible C++ compiler must be installed and configured for MATLAB using the `mex -setup C++` command. Microsoft Visual Studio 2022 (Community, Professional, or Enterprise) was used during the MEX file creation.
3. **Compiled MEX File:** The compiled MEX file (`renderMex.mexw64` for 64-bit Windows) must be present and located either in MATLAB's Current Folder or in a directory on the MATLAB path.
4. **Wrapper `.m` Files:** All the provided wrapper function files (`renderInit.m` , `renderShutdown.m` , `renderDrawRect.m` , etc.) must be located either in MATLAB's Current Folder or in a directory on the MATLAB path.
5. **Runtime Dependencies:**
   - **C++ Runtime:** Since the MEX file and its linked static libraries (`RaycasterGL.lib` , `raylib.lib` , `glfw3.lib`) were compiled using the `/MD` flag (dynamic release CRT), the target machine must have the corresponding **Microsoft Visual C++ Redistributable for Visual Studio 2022** installed.
   - **System Libraries:** The underlying libraries rely on standard Windows libraries (OpenGL, GDI, User32, Kernel32, Shell32, WinMM), which are typically part of Windows itself.

3. Core Concepts

- **Initialization and Shutdown:** You *must* call `renderInit` before any drawing functions and `renderShutdown` when you are finished to properly manage the rendering window and resources.
- **Render Loop:** The typical usage involves a `while` loop that continues as long as `renderShouldClose` returns `false` .
- **Frame Structure:** Inside the loop, each frame should be structured as:
  i. `renderBeginFrame()` - Clears the screen and prepares for drawing.
  ii. Multiple calls to drawing functions (`renderDrawRect` , `renderDrawText` , etc.).
  iii. `renderEndFrame()` - Finalizes the frame and displays it on the window.
- **Coordinate System:** Assumed to be standard 2D screen coordinates where (0, 0) is the **top-left corner** of the window. The X-axis increases to the right, and the Y-axis increases downwards. Units are pixels.
- **Color Format:** Color arguments for drawing functions expect a **1x4 `uint8` row vector** representing `[Red, Green, Blue, Alpha]` , where each value ranges from 0 to 255. Alpha=255 is fully opaque, Alpha=0 is fully transparent (though transparency behavior depends on underlying Raylib settings, usually enabled by default).
- **Error Handling:** Wrapper functions include basic `try...catch` blocks. If the underlying `renderMex` call fails (e.g., due to incorrect arguments caught by the C++ code's

`mexErrMsgIdAndTxt`), a MATLAB warning will be issued. Fatal errors in the C++ code may crash MATLAB.

4. Function Reference

4.1. `renderInit`

- **Syntax:** `success = renderInit(width, height)`
- **Description:** Initializes the rendering engine and creates the output window with the specified dimensions. This function MUST be called successfully before any other rendering functions. The window title is currently hardcoded to "MATLAB Renderer" within the C++ MEX code.
- **Arguments:**
  - `width` : (Scalar, positive integer, `int32`) The desired width of the rendering window in pixels.
  - `height` : (Scalar, positive integer, `int32`) The desired height of the rendering window in pixels.
- **Return Values:**
  - `success` : (Scalar, `logical`) Returns `true` (1) if initialization was successful, `false` (0) otherwise.
- **Example Usage:**

```
success = renderInit(1280, 720);
if ~success
    error('Failed to initialize renderer!');
end
```

- **Notes:** Only call this function once unless `renderShutdown` has been called previously.

4.2. `renderShutdown`

- **Syntax:** `renderShutdown()`
- **Description:** Closes the rendering window and releases all resources allocated by the rendering engine (via Raylib/GLFW). Should be called when rendering is complete to clean up properly.
- **Arguments:** None.
- **Return Values:** None.
- **Example Usage:**

```
% (After render loop finishes)
renderShutdown();
```

- **Notes:** Always try to call this, even if errors occurred during rendering, to ensure the window closes.

4.3. `renderShouldClose`

- **Syntax:** `closeFlag = renderShouldClose()`
- **Description:** Checks if the user has requested to close the rendering window (e.g., by clicking the window's close button or pressing Alt+F4). This is typically used as the condition for the main render loop.
- **Arguments:** None.
- **Return Values:**
  - `closeFlag` : (Scalar, `logical`) Returns `true` (1) if the window should close, `false` (0) otherwise.
- **Example Usage:**

```
while ~renderShouldClose()
    % ... render frame ...
end
```

- **Notes:** If the underlying MEX call fails, this wrapper defaults to returning `true` to prevent infinite loops.

4.4. `renderBeginFrame`

- **Syntax:** `renderBeginFrame()`
- **Description:** Signals the start of drawing operations for a new frame. Internally, this typically calls Raylib's `BeginDrawing()` which clears the screen to a default background color (likely black, as set in the C++ code). All drawing commands for a frame should come after this call.
- **Arguments:** None.
- **Return Values:** None.
- **Example Usage:**

```
while ~renderShouldClose()
    renderBeginFrame();
    % ... drawing commands ...
    renderEndFrame();
```

```
end
```

4.5. `renderEndFrame`

- **Syntax:** `renderEndFrame()`
- **Description:** Signals the end of drawing operations for the current frame. Internally, this typically calls Raylib's `EndDrawing()` which performs necessary operations like swapping graphics buffers to display the newly drawn content on the window.
- **Arguments:** None.
- **Return Values:** None.
- **Example Usage:**

```
while ~renderShouldClose()
    renderBeginFrame();
    % ... drawing commands ...
    renderEndFrame(); % Display the frame
    pause(0.01);
end
```

4.6. `renderDrawRect`

- **Syntax:** `renderDrawRect(x, y, w, h, color)`
- **Description:** Draws a filled rectangle on the screen.
- **Arguments:**
  - `x` : (Scalar, numeric, `int32`) The X-coordinate of the top-left corner of the rectangle (pixels).
  - `y` : (Scalar, numeric, `int32`) The Y-coordinate of the top-left corner of the rectangle (pixels).
  - `w` : (Scalar, numeric, non-negative, `int32`) The width of the rectangle in pixels.
  - `h` : (Scalar, numeric, non-negative, `int32`) The height of the rectangle in pixels.
  - `color` : (1x4 `uint8` row vector) The color `[R, G, B, A]` for the rectangle (0-255 for each component).
- **Return Values:** None.
- **Example Usage:**

```
blueColor = uint8([0, 0, 255, 255]);
renderDrawRect(50, 100, 200, 80, blueColor);
```

- **Notes:** The underlying C++ function `DrawScreenRectangle` is called. Assumes standard top-left origin coordinate system.

4.7. `renderDrawText`

- **Syntax:** `renderDrawText(text, x, y, fontSize, color)`
- **Description:** Draws a text string on the screen using a default font (provided by Raylib).
- **Arguments:**
    - `text` : (String or Character vector) The text to draw.
    - `x` : (Scalar, numeric, `int32`) The X-coordinate of the top-left corner of the text's starting position (pixels).
    - `y` : (Scalar, numeric, `int32`) The Y-coordinate of the top-left corner of the text's starting position (pixels).
    - `fontSize` : (Scalar, positive integer, `int32`) The font size to use (approximate pixel height).
    - `color` : (1x4 `uint8` row vector) The color `[R, G, B, A]` for the text (0-255 for each component).
- **Return Values:** None.
- **Example Usage:**

```
statusText = 'Running...';
textColor = uint8([200, 200, 200, 255]); % Light gray
renderDrawText(statusText, 10, 500, 16, textColor);
```

- **Notes:** The underlying C++ function `DrawScreenText` is called. Font selection is not currently exposed via this interface; Raylib's default font is used. Text alignment is based on the top-left coordinate.

4.8. `renderLoadTexture`

- **Syntax:** `textureID = renderLoadTexture(filePath)`
- **Description:** Loads an image file from the specified path into GPU memory as a texture. Returns a handle (ID) needed for drawing functions like `renderDrawSprite` (if implemented).
- **Arguments:**
    - `filePath` : (String or Character vector) The full or relative path to the image file (e.g., PNG, JPG, BMP - supported formats depend on Raylib).
- **Return Values:**
    - `textureID` : (Scalar, numeric, `double`) A non-zero identifier for the loaded texture if successful. Returns `0` if the texture failed to load.
- **Example Usage:**

```
texPlayer = renderLoadTexture('assets/player_ship.png');
if texPlayer == 0
```

```
        warning('Failed to load player texture!');
    end
```

- **Notes:** Remember to unload textures using `renderUnloadTexture` when they are no longer needed to free up GPU memory.

4.9. `renderUnloadTexture`

- **Syntax:** `renderUnloadTexture(textureID)`
- **Description:** Unloads a texture from GPU memory using the ID obtained from `renderLoadTexture` .
- **Arguments:**
  - `textureID` : (Scalar, numeric, non-negative, `double`) The ID of the texture to unload. Passing 0 may be safe but typically does nothing.
- **Return Values:** None.
- **Example Usage:**

```
renderUnloadTexture(texPlayer); % Unload texture loaded previously
```

4.10. `renderDrawLine`

- **Syntax:** `renderDrawLine(x1, y1, x2, y2, color)`
- **Description:** Draws a straight line between two points (x1, y1) and (x2, y2).
- **Arguments:**
  - `x1` : (Scalar, numeric, `int32`) X-coordinate of the starting point.
  - `y1` : (Scalar, numeric, `int32`) Y-coordinate of the starting point.
  - `x2` : (Scalar, numeric, `int32`) X-coordinate of the ending point.
  - `y2` : (Scalar, numeric, `int32`) Y-coordinate of the ending point.
  - `color` : (1x4 `uint8` row vector) The color `[R, G, B, A]` for the line.
- **Return Values:** None.
- **Example Usage:**

```
whiteColor = uint8([255, 255, 255, 255]);
renderDrawLine(10, 10, 800, 500, whiteColor); % Draw diagonal white line
```

4.11. `renderGetScreenSize`

- **Syntax:** `[width, height] = renderGetScreenSize()`

- **Description:** Returns the current width and height of the rendering window managed by the engine. This might be useful if the window is resizeable (although resize handling isn't explicitly implemented in the current MEX interface).
- **Arguments:** None.
- **Return Values:**
  - `width` : (Scalar, numeric, `double`) The current width of the window in pixels. Returns 0 on failure.
  - `height` : (Scalar, numeric, `double`) The current height of the window in pixels. Returns 0 on failure.
- **Example Usage:**

  ```
  [currentW, currentH] = renderGetScreenSize();
  title(['Window: ', num2str(currentW), 'x', num2str(currentH)]); % Example updating MATLAB
  ```

5. Full Example Script

This script demonstrates initializing the renderer, running a basic render loop that draws text and a rectangle, displaying an FPS counter, and shutting down cleanly.

```
% Example Usage in MATLAB with FPS Counter:

try % Wrap the whole thing in try-catch for better error handling

    if renderInit(1024, 768) % Check return value of init
        disp('Renderer Initialized.');
    else
        error('Failed to initialize renderer. Check MEX file and dependencies.'); % Exit if in
    end

    % --- FPS Counter Variables ---
    fpsString = 'FPS: Calculating...'; % Placeholder text
    loopTic = tic; % Initialize timer for the *first* frame duration calculation

    % --- Main Render Loop ---
    while ~renderShouldClose()

        % --- Start of Frame ---
        renderBeginFrame(); % Tell C++ engine to clear screen etc.

        % --- Draw FPS Counter ---
        % Draw the FPS calculated from the *previous* frame iteration
```

```
            renderDrawText(fpsString, 10, 10, 20, uint8([0 255 0 255])); % Green text at top-left

            % --- Draw Your Content ---
            % Coordinates adjusted slightly so they don't overlap FPS counter
            renderDrawRect(150, 150, 100, 50, uint8([255 0 0 255])); % Red rectangle
            renderDrawText('Hello World!', 150, 220, 20, uint8([0 255 255 255])); % Cyan text
            % --- End Your Content ---

            % --- End of Frame ---
            renderEndFrame(); % Tell C++ engine to display the drawn frame

            % --- Calculate FPS for *Next* Frame ---
            % Measure time elapsed since the last time we started the timer
            frameTime = toc(loopTic);
            % Immediately restart the timer for the next frame measurement
            loopTic = tic;

            % Calculate FPS, avoid division by zero or near-zero
            if frameTime > 1e-6 % Check if frame time is reasonably positive
                fps = 1.0 / frameTime;
                fpsString = sprintf('FPS: %.1f', fps); % Format with 1 decimal place
            else
                % If frame time is too small, avoid displaying Inf or huge numbers
                % Keep previous value to avoid jarring display
                % fpsString = 'FPS: >1000'; % Alternative
            end
             % --- End FPS Calculation ---

            % --- Yield CPU ---
            % pause is important to prevent MATLAB hogging 100% CPU and
            % allow window events (like closing) to be processed smoothly.
            pause(0.001); % Pause for 1 millisecond

        end % --- End of Main Render Loop ---

        renderShutdown();
        disp('Render loop finished, renderer shutdown.');

    catch ME
        % Catch errors during initialization or the loop
        disp('------------------------------------');
        disp('ERROR during render loop or initialization:');
        % Display the full error report
        disp(ME.getReport('extended', 'hyperlinks','off'));
        disp('------------------------------------');
        % Attempt graceful shutdown even if an error occurred
        try
            disp('Attempting shutdown after error...');
            renderShutdown();
            disp('Renderer shutdown.');
        catch shutdownME
            disp('Shutdown also failed after error:');
            disp(shutdownME.message);
        end
    end % try-catch block
```