Detailed Report on the Hiking Simulator Assignment

Introduction

The goal of this assignment is to develop a 3D hiking simulator using OpenGL, where a hiker traverses a terrain following a predefined path derived from GPS data. The application involves rendering a terrain, loading and processing GPS data, converting geospatial coordinates to local coordinates, and animating a character along the path. This report provides a comprehensive explanation of the codebase, data processing steps, classes, shaders, and the mathematical conversions involved.

---

1. Overview of the Application Structure

The hiking simulator is composed of several key components:

- **Terrain Rendering**: Generating a 3D terrain from a heightmap image.
- **Data Processing**: Converting GPX (GPS Exchange Format) data into local XYZ coordinates compatible with the terrain.
- **Hiker Path Representation**: Loading and validating the hiker's path data.
- **Character Animation**: Animating a character model moving along the path.
- **Shaders**: Implementing vertex and fragment shaders for rendering the terrain, path, and skybox.
- **Camera Control**: Providing different camera modes to view the simulation.

---

2. Data Processing and Coordinate Conversion

2.1. GPX Data Extraction

The hiker's path is derived from a GPX file containing GPS track points. Each track point includes latitude (`lat`), longitude (`lon`), and elevation (`ele`).

2.2. Conversion to Local Coordinates

2.2.1. Need for Conversion

GPS coordinates (latitude and longitude) are spherical and not directly usable in a Cartesian coordinate system used for terrain rendering. Therefore, they need to be converted to a local Cartesian coordinate system to align with the terrain's coordinate system.

2.2.2. Local Tangent Plane Approximation

For small areas (like a hiking trail), the Earth's surface can be approximated as flat. A **local tangent plane** is established at a reference point (usually the starting point of the hike), and all other points are calculated relative to this point.

2.2.3. Conversion Formulas

The conversion from latitude and longitude differences to local X and Y coordinates in meters is performed using the following formulas:

- Variables:
  - $R = 6371000$ meters (average Earth radius)
  - $\Delta\phi = \text{radians}(\text{lat} - \text{lat}_{\text{ref}})$
  - $\Delta\lambda = \text{radians}(\text{lon} - \text{lon}_{\text{ref}})$
  - $\phi_{\text{mean}} = \text{radians}\left(\frac{\text{lat} + \text{lat}_{\text{ref}}}{2}\right)$
- Formulas:
  - $X = R \cdot \Delta\lambda \cdot \cos(\phi_{\text{mean}})$
  - $Y = R \cdot \Delta\phi$
  - $Z = \text{Elevation (ele)}$

2.2.4. Python Script for Conversion

python Copy code

```
import xml.etree.ElementTree as ET import math def gpx_to_local_xyz(gpx_file, txt_file): # Parse the GPX file tree = ET.parse(gpx_file) root = tree.getroot() # Namespace handling namespace = {'default': 'http://www.topografix.com/GPX/1/1'} # Get all track points trkpts = root.findall(".//default:trkpt", namespace) if not trkpts: print("No track points found in GPX file.") return # Reference point (first track point) lat_ref = float(trkpts[0].attrib['lat']) lon_ref = float(trkpts[0].attrib['lon']) # Open TXT file for writing with open(txt_file, 'w') as file: # Iterate through track points for trkpt in trkpts: lat = float(trkpt.attrib['lat']) lon = float(trkpt.attrib['lon']) ele = float(trkpt.find("default:ele", namespace).text) # Convert lat/lon to local X, Y coordinates x, y = latlon_to_local(lat_ref, lon_ref, lat, lon) z = ele # Use elevation as Z # Write to TXT file file.write(f"{x} {y} {z}\n") # Space-separated values def latlon_to_local(lat_ref, lon_ref, lat, lon): R = 6371000 # Earth radius in meters delta_lat = math.radians(lat - lat_ref) delta_lon = math.radians(lon - lon_ref) mean_lat = math.radians((lat + lat_ref) / 2.0) x = R * delta_lon * math.cos(mean_lat) y = R *
```

```
delta_lat return x, y # Usage example gpx_file = 'path/to/your.gpx' # Replace with your GPX file path txt_file = 'path/to/your_output.txt' # Replace with your output TXT file path gpx_to_local_xyz(gpx_file, txt_file)
```

2.2.5. Data Preparation for the Application

- **Formatting**: The resulting TXT file contains space-separated values without a header, as expected by the application.
- **Sample Data**:

```
diff
```

Copy code

```
0.0 0.0 311.6 -1.225832464862802 1.89031375292544 311.6 -2.492525076072911
3.7806275074310545 311.6 -3.7592167360826605 5.670941260356495 311.6
-13.238943252280198 19.903891869689115 311.8 ...
```

---

3. Classes and Their Functionalities

3.1. Terrain Class

Purpose

- To generate and render a 3D terrain based on a heightmap image.
- To provide height information at specific positions for collision detection and path validation.

Key Methods

- `loadHeightmap(const std::string& heightmapFile)` : Loads the heightmap image and extracts height data.
- `loadTexture(const std::string& textureFile)` : Loads the texture image for the terrain.
- `getHeightAtPosition(float x, float z) const` : Returns the height of the terrain at a given (x, z) position.
- `render(...)` : Renders the terrain using OpenGL.
- `cleanup()` : Cleans up OpenGL resources associated with the terrain.

Mesh Generation

- **Vertices**: Created based on the heightmap data, with positions, normals, and texture coordinates.
- **Indices**: Used for efficient rendering using triangle strips.

Scaling Factors

- **Horizontal Scale** (`horizontalScale`): Adjusts the size of the terrain in the X and Z directions.
- **Height Scale** (`heightScale`): Adjusts the elevation of the terrain in the Y direction.

Shader Usage

- The terrain uses a shader (`terrainShader`) that handles lighting, texturing, and other visual effects.

3.2. Hiker Class

Purpose

- Represents the hiker moving along the path on the terrain.
- Handles the loading, validation, and traversal of the path data.

Key Methods

- `Hiker(const std::string& pathFile)` : Constructor that initializes the hiker with a path file.
- `loadPathData(const Terrain& terrain)` : Loads the path data from the file and validates it against the terrain.
- `updatePosition(float deltaTime, const Terrain& terrain)` : Updates the hiker's position along the path based on elapsed time.
- `renderPath(...)` : Renders the path as a line on the terrain.
- `moveForward(float deltaTime)` : Moves the hiker forward along the path.
- `moveBackward(float deltaTime)` : Moves the hiker backward along the path.
- `resetPath()` : Resets the hiker's position to the start of the path.
- `getPosition() const` : Returns the current position of the hiker.
- `getPathPoints() const` : Returns the path points for use by other classes.

Path Validation

- Adjusts the path points to ensure they are within the terrain boundaries.
- Sets the Y coordinate of each path point to match the terrain's height at that location.

Movement Logic

- Uses **linear interpolation** between path points to determine the hiker's position.
- Calculates the current segment and interpolation factor based on the `currentDistance` along the path.
- Supports moving forward and backward along the path.

3.3. AnimatedCharacter Class

- Similar to the Hiker class but handles character animation along the path.
- Uses the same path points for movement.
- Integrates with character models and animations (not detailed in this report).

3.4. Skybox Class

- Implements a skybox that surrounds the scene to simulate the sky.
- Uses a cube map texture to render the sky in all directions.
- Provides a singleton instance for global access.

3.5. Lighting Class

- Manages lighting parameters for the scene.
- Sets up light positions, colors, and intensities for shaders to use.

3.6. HikingSimulator Class

Purpose

- Acts as the main controller of the application.
- Initializes all components, processes input, updates the simulation, and handles rendering.

Key Methods

- `initialize()` : Loads resources, initializes components, and sets up the simulation.
- `processInput(GLFWwindow* window)` : Handles keyboard and mouse input for camera control and hiker movement.
- `updateViewMatrix()` : Updates the camera's view matrix based on the current camera mode.
- `render(float deltaTime)` : Renders the entire scene, including the terrain, hiker, skybox, and other effects.
- `cleanup()` : Cleans up all resources before the application exits.

Camera Modes

- **Overview**: A fixed camera showing the entire terrain.
- **Follow**: Camera follows the hiker from behind.
- **First-Person**: Camera positioned at the hiker's viewpoint.

---

4. Shaders

4.1. Shader Class

- Manages the compilation and linking of vertex and fragment shaders.
- Provides methods to set uniform variables in the shaders.

4.2. Terrain Shader

- **Vertex Shader**: Processes vertex positions, normals, and texture coordinates. Applies transformations for positioning in the world.
- **Fragment Shader**: Handles lighting calculations, applies textures, and outputs the final color.

4.3. Path Shader

- Used to render the hiker's path as a line.
- Supports setting the path color and height offset to prevent z-fighting with the terrain.

4.4. Skybox Shader

- Specialized shader for rendering the skybox without being affected by lighting or depth testing.

---

5. Application Flow

1. **Initialization**
   - Load the terrain heightmap and texture.
   - Initialize the hiker with the path data.
   - Set up the shaders and other components.
   - Prepare the view and projection matrices.
2. **Main Loop**
   - **Input Processing**: Handle user input for camera control and hiker movement.
   - **Update**: Update the hiker's position along the path.
   - **Render**:
     - Clear the screen and set up depth testing.
     - Render the skybox.
     - Render the terrain.
     - Render the hiker's path.
     - Render the animated character.

- Render any additional effects.

3. **Cleanup**
   - Release all resources and terminate the application gracefully.

---

6. Detailed Explanation of Key Code Sections

6.1. Hiker's `loadPathData` Method

cpp Copy code

```cpp
bool Hiker::loadPathData(const Terrain& terrain) { std::ifstream file(pathFile); if (!file.is_open()) { std::cerr << "ERROR::HIKER::FAILED_TO_OPEN_PATH_FILE: " << pathFile << std::endl; return false; } float x, y, z; pathPoints.clear(); // Load path points from file while (file >> x >> y >> z) { pathPoints.emplace_back(x, y, z); } file.close(); if (pathPoints.empty()) { std::cerr << "ERROR::HIKER::NO_PATH_POINTS_LOADED" << std::endl; return false; } // Validate and adjust the path points against the terrain validatePath(terrain); // Calculate segment distances and total path length calculateSegmentDistances(); position = pathPoints[0]; currentDistance = 0.0f; currentSegmentIndex = 0; // Setup OpenGL buffers for rendering the path setupPathVAO(); return true; }
```

Explanation:

- Opens the path data file and reads the points into `pathPoints`.
- Calls `validatePath` to ensure the points are within terrain boundaries and to adjust their Y values based on the terrain height.
- Calculates distances between segments to facilitate movement along the path.
- Sets up the OpenGL buffers needed to render the path as a line strip.

6.2. Hiker's `updatePosition` Method

cpp Copy code

```cpp
void Hiker::updatePosition(float deltaTime, const Terrain& terrain) { // Advance along the path based on speed and deltaTime float distanceToMove = speed * deltaTime; currentDistance += distanceToMove; if (currentDistance >= totalPathLength) { currentDistance = 0.0f; // Loop back to start currentSegmentIndex = 0; } // Find the segment of the path we're currently on while (currentSegmentIndex < segmentDistances.size() - 1 && currentDistance > segmentDistances[currentSegmentIndex + 1]) { currentSegmentIndex++; } // Calculate the interpolation factor 't' for the current
```

```
segment float segmentStartDistance = segmentDistances[currentSegmentIndex]; float segmentEndDistance = segmentDistances[currentSegmentIndex + 1]; float segmentLength = segmentEndDistance - segmentStartDistance; float t = (currentDistance - segmentStartDistance) / segmentLength; // Interpolate between the start and end points of the segment glm::vec3 startPoint = pathPoints[currentSegmentIndex]; glm::vec3 endPoint = pathPoints[currentSegmentIndex + 1]; glm::vec3 interpolatedPosition = glm::mix(startPoint, endPoint, t); // Update the hiker's position position = interpolatedPosition; }
```

Explanation:

- Advances the `currentDistance` based on the hiker's speed and `deltaTime`.
- Loops back to the start when the end of the path is reached.
- Determines the current segment and calculates the interpolation factor `t`.
- Uses linear interpolation to find the hiker's position along the segment.
- Updates the hiker's position accordingly.

6.3. Rendering the Terrain

cpp Copy code

```
void Terrain::render(const glm::mat4& model, const glm::mat4& view, const glm::mat4& projection, const glm::vec3& cameraPosition) { if (!terrainShader) { std::cerr << "ERROR: Terrain shader not set." << std::endl; return; } terrainShader->use(); // Set uniforms terrainShader->setMat4("model", model); terrainShader->setMat4("view", view); terrainShader->setMat4("projection", projection); terrainShader->setVec3("viewPos", cameraPosition); // Bind texture glActiveTexture(GL_TEXTURE0); glBindTexture(GL_TEXTURE_2D, textureID); terrainShader->setInt("terrainTexture", 0); // Draw the terrain glBindVertexArray(terrainVAO); glDrawElements(GL_TRIANGLES, static_cast<GLsizei>(indices.size()), GL_UNSIGNED_INT, 0); glBindVertexArray(0); }
```

Explanation:

- Uses the terrain shader to render the terrain mesh.
- Sets the necessary uniforms for the shader, including transformation matrices and camera position.
- Binds the terrain texture.
- Draws the terrain using indexed drawing.

6.4. Processing Input in HikingSimulator

cpp Copy code

```
void HikingSimulator::processInput(GLFWwindow* window) { float currentTime = static_cast<float>(glfwGetTime()); float deltaTime = currentTime - lastFrameTime; lastFrameTime = currentTime; // Camera mode toggles if (glfwGetKey(window, GLFW_KEY_1) == GLFW_PRESS) { cameraMode = CameraMode::OVERVIEW; isMouseEnabled = false; updateViewMatrix(); } else if (glfwGetKey(window, GLFW_KEY_2) == GLFW_PRESS) { cameraMode = CameraMode::FOLLOW; isMouseEnabled = true; updateViewMatrix(); } else if (glfwGetKey(window, GLFW_KEY_3) == GLFW_PRESS) { cameraMode = CameraMode::FIRST_PERSON; isMouseEnabled = true; updateViewMatrix(); } // Movement controls if (cameraMode != CameraMode::OVERVIEW) { if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS) { hiker.moveForward(deltaTime); } if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS) { hiker.moveBackward(deltaTime); } } // Reset path if (glfwGetKey(window, GLFW_KEY_R) == GLFW_PRESS) { hiker.resetPath(); } updateViewMatrix(); }
```

Explanation:

- Handles keyboard input to switch camera modes and control the hiker's movement.
- Updates the view matrix when necessary.

---

7. Conclusion

This assignment involved creating a complex 3D application that simulates a hiker moving along a path on a terrain. Key challenges included processing GPS data, converting it to a suitable coordinate system, and ensuring that all components (terrain, hiker, camera) align correctly.

Through careful data processing and the use of appropriate mathematical models, the hiker's path was successfully integrated into the simulation. The classes were designed to handle specific responsibilities, promoting modularity and maintainability.

---

8. References

- **OpenGL Documentation**: For understanding rendering techniques and shader programming.
- **GPX Format Specification**: For parsing GPS data.
- **Mathematical References**: For coordinate conversion formulas.

Appendix

A. Important Formulas

- Latitude and Longitude to Local X, Y Conversion:

  $$\Delta\phi = \text{radians}(\text{lat} - \text{lat}_{\text{ref}})$$

  $$\Delta\lambda = \text{radians}(\text{lon} - \text{lon}_{\text{ref}})$$

  $$\phi_{\text{mean}} = \text{radians}\left(\frac{\text{lat} + \text{lat}_{\text{ref}}}{2}\right)$$

  $$X = R \cdot \Delta\lambda \cdot \cos(\phi_{\text{mean}})$$

  $$Y = R \cdot \Delta\phi$$

- Linear Interpolation:

  $$\text{Interpolated Position} = (1 - t) \cdot \text{Start Point} + t \cdot \text{End Point}$$

B. Sample Shader Code

Vertex Shader (terrainVert.glsl):

glsl Copy code

```
#version 330 core layout(location = 0) in vec3 aPos; layout(location = 1) in vec3 aNormal; layout(location = 2) in vec2 aTexCoord; out vec3 FragPos; out vec3 Normal; out vec2 TexCoord; uniform mat4 model; uniform mat4 view; uniform mat4 projection; void main() { FragPos = vec3(model * vec4(aPos, 1.0)); Normal = mat3(transpose(inverse(model))) * aNormal; TexCoord = aTexCoord; gl_Position = projection * view * vec4(FragPos, 1.0); }
```

Fragment Shader (terrainFrag.glsl):

glsl Copy code

```
#version 330 core out vec4 FragColor; in vec3 FragPos; in vec3 Normal; in vec2 TexCoord; struct Light { vec3 position; vec3 color; vec3 ambient; vec3 diffuse; vec3 specular; }; uniform Light light; uniform vec3 viewPos; uniform sampler2D terrainTexture; void main() { // Ambient vec3 ambient = light.ambient * texture(terrainTexture, TexCoord).rgb; // Diffuse vec3 norm = normalize(Normal); vec3 lightDir = normalize(light.position - FragPos); float diff = max(dot(norm, lightDir), 0.0); vec3 diffuse = light.diffuse * diff * texture(terrainTexture, TexCoord).rgb; // Specular vec3 viewDir = normalize(viewPos - FragPos); vec3 reflectDir = reflect(-lightDir, norm); float spec = pow(max(dot(viewDir,
```

```
reflectDir), 0.0), 32); vec3 specular = light.specular * spec; vec3 result = ambient + diffuse + specular; FragColor = vec4(result, 1.0); }
```

C. Notes on Scaling and Alignment

- **Consistency**: Ensure that scaling factors are consistently applied across the terrain and hiker's path.
- **Origin Alignment**: Both the terrain and the hiker's path should use the same origin point for their coordinate systems.
- **Units**: All measurements should be in the same units (e.g., meters) to prevent discrepancies.