3D Hiking Simulator using OpenGL and C++

Sumaia Jahan Brinti
Department of Applied Computer Science
DTE-3605-1 Virtual reality, Graphics and Animation

December 2024

Abstract

The 3D Hiking Simulator is a C++ and OpenGL-based application that shows a hiker moving through a mountain terrain in three dimensions, using appropriate libraries such as GLFW, GLEW, and GLM for graphics and input handling. This report describes the design, methodology, results, and analysis of the simulator using the IMRaD structure. The simulator illustrates an interesting way of perceiving the combination of interactive visuals integrated with simulation techniques, therefore, making it suitable for educational and recreational purposes.

Introduction

The main goal is to create a simulator, a real-world with natural scene such as mountains where the hiker moves from start to end point by drawing a line along the route where need to build a 3D hiking environment. Data from the terrain and hiking routes were provided as input for guide development.

This report follows the IMRaD structure and documents the program's features, and the learning process involved in its development. It also details the use of AI in programming and problem solving throughout the project.

Related Study

Most of the code is written in C++, with GLSL for shaders. This is intended to be a cross-platform project.

Environment Setup **IDE:** XCode
Libraries:

- `GLFW (Graphics Library Framework)`
  - Cross-platform window and OpenGL context creation
  - Handling of keyboard, mouse, and joystick input
  - Multi-monitor support
  - Support for multiple video modes
  - simplifies the process of creating and managing windows, OpenGL contexts, and handling user input across different operating systems like Windows, macOS, Linux, and various Unix-like systems
- `GLEW (OpenGL Extension Wrangler Library)`.
  - is a cross-platform open-source C/C++ extension loading library for OpenGL
  - provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform
  - GLEW offers several key benefits:
  - exposes OpenGL core and extension functionality in a single header file .
  - simplifies the process of accessing modern OpenGL features and extensions across different platforms

– manages the loading of OpenGL function pointers, allowing developers to use advanced OpenGL functionality that may not be directly exposed by the operating system

– eliminates the need for manually querying and loading OpenGL extensions, which can be a tedious and error-prone process []itemize

– `GLM (OpenGL Mathematics)`

  * Provides essential mathematical operations for computer graphics, including vector and matrix operations
  * Cross-platform compatibility, supporting major compilers like GCC, LLVM, and Visual Studio
  * No external dependencies, making it easy to integrate into projects
  * Includes core `GLSL` features and additional extensions for quaternions, transformations, splines, and more
  * Offers functions that replicate deprecated `OpenGL` operations like `glRotate` and `gluLookAt`

– **STB Image** is a popular single-header library for loading and writing image files in C/C++ created by Sean T. Barrett. STB Image is widely used due to its simplicity, ease of integration, and support for multiple image formats, making it a popular choice for graphics programming and game development

Learning Materials: LearnOpenGL.com is extensively used for understanding terrain rendering, texture mapping, and shader programming. Additionally, YouTube and GitHub provided valuable resources for understanding how the code should function to achieve the expected output. **Resources:**Textures Sourced from Given Data

Methods

The following is the directory structure for the 3D Hiking Simulator project:

```
triangle/
|-- resources/
|   |-- terrainhikingdata/ % Resources to be loaded
|   |   |-- afternoon_run/
|   |   |-- colorsdata
|   |   |-- graydata
|   |   `-- hiker_path
|   |-- shaders/    % Shaders handling for each featur
|   |   |-- hikerFrag
|   |   |-- hikerVert
|   |   |-- skyboxFrag
|   |   |-- skyboxVert
|   |   |-- snowFrag
|   |   |-- snowVert
|   |   |-- terrainFrag
|   |   `-- terrainVert
|   `-- textures/
|       `-- skybox// % Skybox images
|       |-- right.png, bottom.png, left.png
|       |-- top.png, front.png, back.png

|-- camera.h
|-- camera.cpp  % Camera control
|-- cameraFrag
|-- cameraVert
|-- cpuheight.h
|-- hiker.h
|-- hiker.cpp   % Hiker path
|-- hikingSimulator.h
|-- hikingSimulator.cpp % Hiking simulation
|-- main.cpp    % Main Program
|-- SeasonalEffect.h
|-- SeasonalEffect.cpp
|-- shader.h
|-- shader.cpp   % Shaders handling
|-- Skybox.h
|-- Skybox.cpp  % Skybox rendering
|-- stb_image.h
|-- stb_image.cpp
|-- terrain.h
`-- terrain.cpp % terrain Generation
```

Main Features

– **Rendering Terrain using Height Maps** When terrain (without any caves or overhangs) is rendered, a mesh can be perturbed on the basis of a height map. The height map is a grayscale image with the texel value corresponding to the distance from which a vertex should move along its normal. In order to achieve highly detailed terrain it is necessary to have a high resolution mesh. I generated a mesh that matches the resolution of the given image. The `stbi_load()` method is a function that loads an image file into memory. It sets the variables `width` and `height` to the dimensions of the image, while `data` contains the pixel data in an array of size:

$$\texttt{width} \times \texttt{height} \times nChannels$$

Here:

* `width` is the number of pixels in the horizontal direction.
* `height` is the number of pixels in the vertical direction.
* `nChannels` represents the number of color channels (e.g., 3 for RGB, 4 for RGBA).

My mesh will then be an array consisting of width x height vertices. I use a Vertex Buffer Object (VBO) to specify the vertex data of the mesh. My mesh will be centered at the origin, lie in the XZ-plane, and have a width x height. Each vertex will be one unit apart in the model space. The vertex will then be displayed along the surface normal (the Y axis) based on the corresponding location from the height map. The following image helps visualize how the mesh will be built up by a set of vertices.

`loadTerrainData()` Loads terrain height data from a grayscale image file (heightmap) using the `stb_image` library.

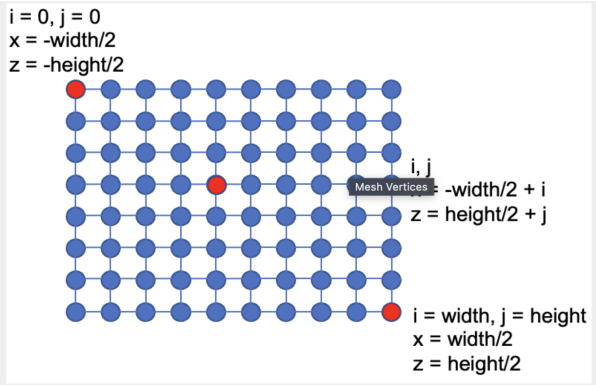

Figure 1: Mesh Visualization

INFO: Number of terrain vertices: 69300
INFO: Number of terrain indices: 412566

– **Camera Setup** When we're talking about camera/view space we're talking about all the vertex coordinates as seen from the camera's perspective as the origin of the scene: the view matrix transforms all the world coordinates into view coordinates that are relative to the camera's position and direction. To define a camera we need its position in world space, the direction it's looking at, a vector pointing to the right and a vector pointing upwards from the camera. This uses GLM's `lookAt` function to create a view matrix based on the camera's position, where it's looking (Position + Front), and its up vector. Keyboard Input Handling The `ProcessKeyboard()` function handles camera movement based on keyboard input: It adjusts the camera's position based on the input direction and the time elapsed since the last frame (deltaTime) for smooth movement.

– **Hiking Simulator** Here terrain Heightmap is loaded via graydata.png which scales hiker's path to align with the terrain dimensions. It loads and validates shaders for the hiker path visualization.

* **Dynamic Camera Movement** Users

can move the camera freely to explore terrain in detail.
    - **Scalable Terrain Rendering:** Adjusts heightmap scaling for realistic terrain proportions.
    - **Shader Management:** Modular shader loading for terrain and path rendering.
- **Path Visualization** The Hiker class implementation focuses on loading, rendering, and updating the position of a simulated hiker along a predefined path, while interacting with the terrain. Initializes the Hiker object with default values and the path file name. Then it updates the horizontal and vertical scaling factors to match the terrain's dimensions with `setScales` method. The `loadPathData()` reads path data from a file and maps it onto the terrain. `glm::mix)` enables smooth transitions between path points.The path is rendered as a continuous `GL_LINE_STRIP`.
- **Shaders** The Shader class is a utility for loading, compiling, linking, and managing OpenGL shaders. Shaders are essential for rendering graphics in OpenGL, and this class simplifies the process of working with vertex and fragment shaders.
    - Reads shader source code from external files specified by paths (vertexPath and fragmentPath).
    - The `compileShader()` method compiles the source code for vertex and fragment shaders.
    - Compiled shaders are linked into a shader program using OpenGL's `glCreateProgram()` and `glLinkProgram()`.
    - Uniform variables in shaders can be set dynamically using methods such as `setMat4`, `setVec3`, `setFloat`, and `setInt`.
    - Uniform locations are cached using the `getUniformLocation()` method to optimize performance.

Results

After successful completion of the program the output looks like below

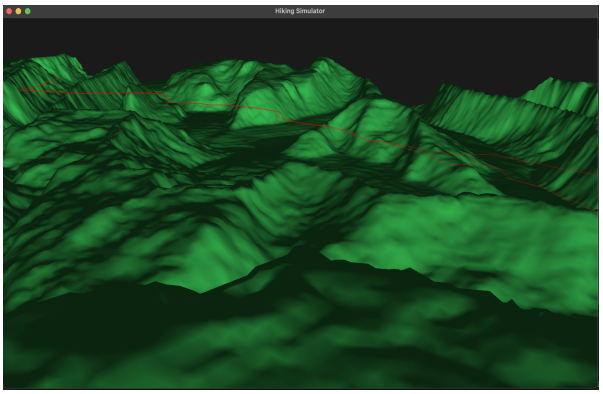

Figure 2: Hiking Simulator

0.1 Challenges

0.1.1 Camera Position and View Frustum

Issue: The camera's far plane was set too close, causing parts of the terrain to be clipped and rendered invisible.

Evidence: The terrain spans a large area from coordinates (0, 0, 0) to approximately (10060, height, 4960) units. However, the camera's far plane was initially set to 1000.0f, which is insufficient to encompass the entire terrain. This caused the distant parts of the terrain to fall outside the view frustum and remain invisible.

Solution: The far plane of the projection matrix was increased to a value that adequately covers the entire terrain. After adjusting the far plane to a larger value (e.g., 15000.0f), the entire terrain became visible within the camera's view frustum. Additionally, to optimize performance,

near and far plane values were fine-tuned to balance rendering efficiency and depth precision.

0.1.2 Shader Compilation Errors

Issue: Shaders may not be compiling correctly, but the error handling is insufficient to catch this.
Evidence: The Shader class lacks proper error logging, and the return statements on shader load failures are commented out.
Solution: Implemented detailed error logging in the Shader class and ensured that the program exits or handles the error if shaders fail to compile.

0.1.3 Rendering a Large Terrain:

Issue: Rendering over 11 million indices exceeds the GPU's capabilities, leading to rendering failures.
Evidence: The log message "INFO: Number of terrain indices: 11957406" indicates a very large number of indices, making it impractical for rendering on most hardware.
Solution: A smaller heightmap was used for testing, which significantly reduced the number of vertices and indices. This allows for efficient rendering and debugging during development. Future implementations will incorporate Level of Detail (LOD) techniques to dynamically adjust mesh resolution based on camera distance.

0.1.4 Rendering Skybox:

Issue: The skybox was incorrectly rendered at the bottom of the scene instead of enveloping the environment as intended.
Evidence: Visual inspection during testing revealed the misplacement of the skybox at the base of the view, which obstructed other scene elements.
Solution: The skybox was temporarily disabled while debugging the issue.

AI Usage

The success of this project was greatly supported by AI tools, which played a key role in organizing content, generating report ideas, and assisting with technical debugging. This collaboration between AI assistance and manual programming boosted productivity and ensured the documentation was clear and well-structured. AI also helped identify and resolve shader compilation issues and provided suggestions for improving terrain rendering logic.

Future Improvements

Improvements to Consider:

- **Camera Enhancements:** Add mouse controls for a smoother camera movement experience.
- **Optimization:** Implement Level of Detail (LOD) for terrain rendering to reduce performance overhead on large terrains.
- **Lighting Improvements:** Introduce dynamic sunlight effects or shadow mapping.

Conclusion

The 3D Hiking Simulator demonstrates the effective use of OpenGL and C++ to create an interactive, visually engaging environment that simulates a hiker navigating through detailed terrain. By integrating advanced techniques such as heightmap-based terrain rendering, shader programming, and dynamic camera movement, the project highlights the potential of real-time graphics for educational and recreational applications.

Challenges encountered, including GPU limitations, shader compilation errors, and camera frustum adjustments, were systematically addressed, leading to a deeper understanding of OpenGL programming and rendering optimization. The use of AI tools and modern libraries like GLFW, GLEW, and GLM significantly enhanced the development process by simplifying complex tasks and enabling efficient debugging. Overall, this project serves as a foundation for exploring advanced graphics programming concepts and their practical applications in immersive environments.

References

[1] cppget.org. glfw. `https://cppget.org/glfw?f=full&q=library`.

[2] glew.sourceforge. Glew. `http://glew.sourceforge.net/`.

[3] glfw.org. Glfw. `https://www.glfw.org/docs/latest/`.

[4] " LearnOpenGL.com Joey de Vries, "LearnOpenGL. Learnopengl. `https://learnopengl.com/`.

[5] " LearnOpenGL.com Joey de Vries, "LearnOpenGL. Learnopengl. `https://learnopengl.com/Guest-Articles/2021/Tessellation/Height-map`.

[6] OGLDEV. Terrain rendering. `https://www.youtube.com/watch?v=xoqESu9iOUE&t=136s`.

[7] reddit. what$_i s_g lew_a nd_g lf w$..

[4] [3] [2] [6] [5] [7] [1]