A类大作业实验报告

冯志远 2024311588

A.4 实验

实验内容

4. 使用半边结构实现网格加密算法-Loop scheme（OpenGL 20 分）
    - **1)** 实现Loop细分算法，参考以下链接中的方法：
      http://multires.caltech.edu/pubs/sig99notes.pdf 第46页，3.1节
      或者直接参考附件“sig99notes.pdf”。
    - **2)** 选择网格加密选项，使用半边结构管理网格数据。
    - **3)** 编写C++程序，使用OpenGL进行网格的渲染与细分，确保程序能够动态增加细分级别。
    - **4)** 提供可执行程序，通过 `make run` 命令直接运行，观察网格细分效果。
    - **5)** 实现用户交互，通过键盘输入调整细分级别。

实验环境

- **编程语言**：C++
- **开发环境**：Visual Studio Code
- **依赖库**：GLFW、GLAD、GLM
- **运行平台**：Windows

实验步骤

1. 环境配置

- 安装并配置OpenGL开发环境，确保能够正确链接GLFW、GLAD和GLM库。
- 在项目目录中包含必要的资源文件，本实验用了 `cow.obj` 模型文件，就是斯坦福牛，用于网格加载。
- 通过 `make run` 命令能够编译并运行程序。

2. 半边结构设计

- **半边数据结构**：
  - 定义半边（Half-Edge）的结构体，包含起点、终点、相邻半边等信息，用于高效管理网格拓扑。
  - 定义索引对（IndexPair2）和索引三元组（IndexPair3）结构体，用于边的管理和哈希计算。
- **哈希函数**：
  - 实现索引对的哈希函数（ `IndexPair2Hash` ），确保在 `unordered_map` 中能够高效查找边信息。
- **边顶点管理**：
  - 通过 `addEdgeVertex` 函数管理边上的新顶点，确保每条边只被细分一次，避免重复插入。

3. Loop细分算法实现

- **细分步骤**：
  i. **顶点复制**：将原网格的顶点复制到新网格中。
  ii. **边点插入**：对每条边，根据Loop细分规则计算新的边点位置，并将其添加到新网格中。
  iii. **面拆分**：将每个原始三角形拆分为四个新的三角形，使用新插入的边点。
  iv. **顶点位置调整**：根据Loop细分的权重规则，调整原始顶点的位置，使细分后的网格更加平滑。
- **算法实现**：
  - 通过 `LoopSubdivideNative` 函数实现细分算法，处理每个三角形面片，插入边点并生成新面。
  - 使用阿基米德螺线公式和权重系数，确保细分后的网格保持光滑性。

4. 网格加载与渲染

- **模型加载**：
  - 使用自定义的 `Mesh` 类加载 `cow.obj` 模型文件，初始化顶点和面数据。
- **着色器设计**：
  - **顶点着色器**：
    - 实现MVP矩阵变换，对顶点位置进行投影、视图和模型变换。
    - 传递顶点颜色和法线信息到片段着色器。
  - **片段着色器**：
    - 实现基础的光照计算，增强网格的视觉效果。
    - 支持线框模式渲染，便于观察细分效果。
- **渲染设置**：
  - 启用深度测试和背面剔除，确保渲染的正确性。
  - 设置相机位置和视角，调整投影矩阵以获得最佳视图。

5. 用户交互与动态细分

- **输入处理**：
  - 实现 `handleInputEvents` 函数，监听键盘输入：
    - **A键**：增加细分级别，应用Loop细分算法。
    - **S键**：减少细分级别，回退到上一级网格。
- **细分控制**：
  - 使用 `subdivisionLevel` 变量控制当前的细分级别。
  - 动态添加细分后的网格到 `meshList` 中，确保每次细分基于最新的网格数据。
- **渲染循环**：
  - 在主渲染循环中，根据当前的细分级别绘制对应的网格。
  - 使用线框模式渲染，方便观察细分效果。

6. 资源清理

- **资源释放**：
  - 程序退出前，释放所有OpenGL资源，包括顶点数组对象（VAO）、顶点缓冲对象（VBO）和着色器程序。
  - 调用 `glfwTerminate` 函数，确保正确关闭GLFW窗口和上下文。

实验结果

1. **程序运行**：通过 `make run` 命令成功编译并运行程序，打开一个800x600像素的窗口。
2. **网格渲染**：初始显示加载的 `cow.obj` 模型，采用线框模式，展示原始网格结构。
3. **细分效果**：
   - 按下**A键**，程序应用Loop细分算法，网格细分一级，显示更高分辨率的网格结构。
   - 多次按下**A键**，网格细分多级，逐渐显示更为光滑和复杂的网格形态。
   - 按下**S键**，细分级别减少，回退到较低分辨率的网格。
4. **用户交互**：细分和回退操作响应迅速，实时更新网格显示，确保良好的用户体验。
5. **资源管理**：程序关闭后，所有资源被正确释放，无内存泄漏或未释放资源的问题。

实验总结

通过本实验，我成功实现了基于半边结构的Loop细分算法，并在OpenGL环境下进行渲染与展示。实验过程中，掌握了以下技能：

1. **半边结构管理**：学习了半边数据结构在网格拓扑管理中的应用，理解了边与面的关系。
2. **细分算法实现**：深入理解了Loop细分算法的细节，掌握了顶点插入和位置调整的权重计算。
3. **OpenGL渲染**：提升了使用OpenGL进行复杂网格渲染的能力，熟悉了着色器编写与矩阵变换。
4. **动态交互设计**：实现了用户通过键盘输入动态调整细分级别，增强了程序的交互性。
5. **资源管理**：确保了程序的资源正确管理与释放，避免了潜在的内存泄漏问题。

在实验过程中，解决了半边结构在边点插入中的边界处理问题，优化了细分算法的效率。此外，通过调试着色器和渲染管线，提升了对图形渲染流程的理解。整体而言，本实验不仅加深了对网格细分算法的理解，还提升了在实际项目中应用数据结构和图形渲染技术的能力。

算法部分简要介绍

Loop细分算法是由Charles Loop提出的一种用于三角形网格的细分方案，属于顶点插入类型。其主要步骤包括：

1. **边点插入**：对每条边插入一个新的边点，位置由周围原网格顶点的加权平均确定。
2. **面拆分**：将每个原始三角形拆分为四个新的三角形，使用新插入的边点。
3. **顶点位置调整**：根据周围顶点的位置，调整原始顶点的位置，以确保细分后的网格平滑。

细分规则基于三向四次盒样条，保证了无限细分后的表面光滑性。具体权重如下：

- **内部边**：新边点的位置由边上两个端点各占3/8，对应两个对面顶点各占1/8。
- **边界边**：新边点的位置由边上两个端点各占1/2。
- **内部顶点**：原顶点的位置由自身和邻接顶点按特定权重调整。
- **边界顶点**：原顶点的位置由自身和边界邻接顶点按特定权重调整。

通过上述规则，Loop细分算法能够有效地提升网格的细节和光滑度。

A.6 实验

实验内容

6. 实现以下材料中的 Terrain Engine。（OpenGL 20 分）
    - **1)** 材料见附件 TerrainEngine 文件夹。
    - **2)** 摄像机坐标系与全局坐标系之间的变换；

- **3)** 海面的波浪效果；
- **4)** 地形的读取、绘制及纹理贴图；
- **5)** 天空和地形的倒影效果。

实验报告

实验环境

- **编程语言**：C++
- **开发环境**：Visual Studio Code
- **依赖库**：GLFW、GLAD、GLM、stb_image
- **运行平台**：Windows

实验步骤

1. 环境配置

- 安装并配置OpenGL开发环境，确保正确链接GLFW、GLAD、GLM和stb_image库。
- 在项目目录中包含必要的资源文件，如 `skybox` 纹理、 `heightmap.bmp` 高度图、地形和水面纹理等。
- 配置编译系统，使得通过 `make run` 命令能够编译并运行程序。

2. Shader设计与加载

- **顶点着色器（Vertex Shader）**：
  - 实现MVP矩阵变换，对顶点位置进行投影、视图和模型变换。
  - 传递顶点颜色、法线和纹理坐标到片段着色器。
- **片段着色器（Fragment Shader）**：
  - 实现纹理映射，应用不同的纹理到地形和水面。
  - 实现基础的光照计算，增强视觉效果。
  - 实现水面的透明度和波浪效果。
- **天空盒着色器**：
  - 实现天空盒的渲染，确保天空盒始终位于相机周围。
  - 处理天空盒的纹理采样和环境映射。
- **加载与编译**：
  - 使用自定义的 `Shader` 类加载、编译并链接各个着色器程序。

- 确保所有着色器程序能够正确工作，并输出调试信息。

3. 纹理加载与处理

- **加载纹理**：
  - 使用 `stb_image` 库加载天空盒、水面、地形和细节纹理。
  - 设置纹理的采样参数，如环绕模式（`GL_CLAMP_TO_EDGE`、`GL_REPEAT`）和过滤方式（`GL_LINEAR`、`GL_LINEAR_MIPMAP_LINEAR`）。
  - 生成多级纹理（Mipmap）以优化渲染性能和质量。
- **纹理绑定**：
  - 将加载的纹理绑定到相应的纹理单元（如 `GL_TEXTURE0`、`GL_TEXTURE1` 等）。
  - 在着色器中设置纹理采样器，确保正确绑定和使用纹理。

4. 地形加载与生成

- **高度图处理**：
  - 使用 `stb_image` 加载高度图文件 `heightmap.bmp`，获取每个像素的高度值。
  - 根据高度图生成地形的顶点位置，计算每个顶点的X、Y、Z坐标。
  - 根据高度图生成地形的索引数据，构建三角形网格。
- **OBJ文件生成**：
  - 将生成的顶点和索引数据写入OBJ文件 `land.obj`，用于后续的网格加载。
- **网格加载**：
  - 使用自定义的 `Mesh` 类加载 `land.obj` 文件，初始化地形的顶点和面数据。
  - 计算每个顶点的纹理坐标，根据地形的坐标映射纹理。
  - 设置顶点数组对象（VAO）和顶点缓冲对象（VBO），准备渲染。

5. 摄像机与视图设置

- **摄像机类**：
  - 使用自定义的 `Camera` 类管理摄像机的位置、方向和视角。
  - 实现摄像机的移动、旋转和缩放功能，通过键盘和鼠标输入控制。
- **坐标变换**：
  - 计算视图矩阵和投影矩阵，将摄像机坐标系与全局坐标系进行变换。
  - 更新MVP矩阵，确保渲染的物体正确显示在视野范围内。
- **用户交互**：

- 实现键盘输入处理，支持W、A、S、D、空格和左Shift键控制摄像机的移动。
- 实现鼠标移动和滚轮缩放，控制摄像机的旋转和视野范围。

6. 水面波浪效果实现

- **波浪动画**：
  - 在水面顶点着色器中实现波浪效果，使用正弦函数动态调整顶点的Y坐标。
  - 通过时间变量控制波浪的移动和高度，实现动态波动。
- **透明度与混合**：
  - 设置水面的透明度，使用Alpha混合实现真实的水面效果。
  - 启用OpenGL的混合功能，设置适当的混合函数（ `GL_SRC_ALPHA`，`GL_ONE_MINUS_SRC_ALPHA` ）。

7. 天空盒与倒影效果实现

- **天空盒渲染**：
  - 使用专门的VAO和VBO渲染天空盒，确保其始终围绕摄像机位置。
  - 关闭深度写入（ `glDepthFunc(GL_ALWAYS)` ），确保天空盒不被其他物体遮挡。
- **倒影效果**：
  - 实现地形和天空盒的倒影，通过镜像变换生成地形的镜像。
  - 使用自定义的着色器处理倒影效果，确保倒影与实际地形的动态一致。

8. 主渲染循环与优化

- **渲染顺序**：
  i. 清空颜色和深度缓冲区。
  ii. 计算并传递MVP矩阵到各个着色器。
  iii. 渲染地形，应用纹理贴图和倒影效果。
  iv. 渲染水面，应用波浪动画和透明度。
  v. 渲染天空盒，确保其正确显示在背景中。
- **帧率控制**：
  - 设置目标FPS（如30 FPS），通过时间控制确保渲染循环的稳定性。
  - 使用 `glfwPollEvents` 处理用户输入和窗口事件。
- **资源管理**：
  - 在程序退出前，释放所有OpenGL资源，包括VAO、VBO和着色器程序。

- 调用 `glfwTerminate` 正确关闭GLFW窗口和上下文。

9. 调试与优化

- **调试信息输出**：
  - 在加载纹理、着色器和网格时输出调试信息，确保各个步骤正确执行。
  - 检查OpenGL错误，确保渲染过程中无错误发生。
- **性能优化**：
  - 优化纹理加载和绑定，减少不必要的状态切换。
  - 优化波浪动画的计算，确保实时性和流畅性。
  - 使用多级纹理（Mipmap）提高渲染质量和性能。

实验结果

1. **程序运行**：通过 `make run` 命令成功编译并运行程序，打开一个800x600像素的窗口。
2. **地形渲染**：加载并正确显示基于高度图生成的地形，应用地面和细节纹理，展示出丰富的地形细节。
3. **水面效果**：
   - 实现动态波浪效果，水面随时间波动，呈现自然的水流动感。
   - 水面的透明度和反射效果真实，增强了整体视觉效果。
4. **天空盒渲染**：天空盒正确渲染，覆盖整个背景，随着摄像机移动保持一致。
5. **倒影效果**：
   - 地形和天空盒的倒影效果自然，随着摄像机视角变化实时更新。
   - 倒影与实际地形相匹配，增强了场景的真实感。
6. **用户交互**：摄像机的移动和旋转响应迅速，通过键盘和鼠标控制，提供流畅的浏览体验。
7. **资源管理**：程序关闭后，所有资源被正确释放，无内存泄漏或未释放资源的问题。

实验总结

通过本实验，我成功实现了一个功能全面的Terrain Engine，涵盖了地形生成、纹理贴图、波浪动画、天空盒渲染和倒影效果。实验过程中，掌握了以下技能：

1. **OpenGL渲染流程**：深入理解了OpenGL的渲染管线，包括VAO、VBO的使用，着色器编写与调试。
2. **纹理处理**：学习了多种纹理加载和处理方法，熟悉了纹理的环绕模式和过滤方式的设置。

3. **高度图应用**：掌握了基于高度图生成地形网格的技术，能够将2D高度图转换为3D地形模型。
4. **动态动画实现**：通过波浪动画和倒影效果，学习了如何在渲染过程中实现动态视觉效果。
5. **用户交互设计**：实现了摄像机的多维度控制，提升了程序的交互性和用户体验。
6. **资源管理与优化**：确保了程序的资源正确管理与释放，优化了渲染性能，避免了潜在的性能瓶颈。

在实验过程中，解决了地形网格生成中的索引计算问题，优化了水面波浪效果的实现方法，并通过调试倒影效果提升了整体的视觉真实感。通过此次实验，不仅加深了对计算机图形学中地形渲染和动态效果实现的理解，还提升了在实际项目中应用OpenGL和相关技术的能力。

算法部分简要介绍

Terrain Engine 的实现涉及多个关键算法和技术，主要包括地形生成、波浪动画和倒影效果的实现。

1. 地形生成算法

- **高度图处理**：
  - 使用高度图（ `heightmap.bmp` ）的灰度值表示地形的高度信息。每个像素的灰度值被映射到地形的Y坐标，生成3D顶点位置。
  - 通过遍历高度图的每个像素，计算对应的X、Y、Z坐标，构建地形的顶点数据。
- **网格生成**：
  - 根据高度图的尺寸，生成地形的三角形网格。每个像素点与其右侧和下方的像素点形成两个三角形，构建整个地形的面数据。
  - 使用索引缓冲区（EBO）优化渲染性能，避免重复顶点的绘制。

2. 波浪动画算法

- **顶点位移**：
  - 在水面的顶点着色器中，使用时间变量和正弦函数动态调整每个顶点的Y坐标，模拟波浪的起伏。
  - 公式示例： `y = amplitude * sin(frequency * (x + time))` ，其中 `amplitude` 控制波幅， `frequency` 控制波长， `time` 控制波浪的移动速度。
- **纹理偏移**：
  - 通过动态调整纹理坐标，实现水面纹理的流动效果。
  - 纹理偏移公式： `texCoords += vec2(xShift, yShift)` ，其中 `xShift` 和 `yShift` 随时间变化，模拟水流方向。

3. 倒影效果算法

- **镜像变换**：
  - 通过镜像矩阵将地形和天空盒的顶点位置进行翻转，生成倒影。

- 镜像矩阵示例（沿Y轴翻转）：

```
glm::mat4 mirror_y({
    {1, 0, 0, 0},
    {0, -1, 0, 0},
    {0, 0, 1, 0},
    {0, 0, 0, 1}
});
```

- **渲染顺序**：
  - 首先渲染地形的倒影，确保倒影位于水面之下。
  - 然后渲染实际地形和水面，覆盖倒影部分，增强视觉层次感。
- **透明度与混合**：
  - 在渲染倒影时，应用适当的透明度和混合模式，使倒影看起来更加自然。
  - 使用Alpha混合函数 `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`，控制倒影的透明度。

4. 摄像机坐标变换算法

- **视图矩阵计算**：
  - 使用摄像机的位置、前向方向和上方向，计算视图矩阵，定义摄像机在世界中的位置和朝向。
  - 公式示例： `glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp)`
- **投影矩阵计算**：
  - 使用透视投影，定义视野范围和近远裁剪面，计算投影矩阵。
  - 公式示例： `glm::perspective(glm::radians(fov), aspectRatio, nearPlane, farPlane)`
- **MVP矩阵组合**：
  - 将模型矩阵、视图矩阵和投影矩阵相乘，生成最终的MVP矩阵，用于顶点位置的变换。
  - 公式示例： `glm::mat4 MVP = projection * view * model`

通过上述算法的结合，实现了一个功能全面且视觉效果丰富的Terrain Engine，展示了地形渲染、动态水面和倒影效果的综合应用。