CG大作业最终报告

游戏介绍

游戏简介

本次程序设计为一款猎人在森林中打猎的游戏，使用基本的Opengl工具，通过导入制作相关的素材，运用图形学的基本原理实现了基本的玩法，场景设计和动作操作。

玩法简介

- WASD控制人物移动
- 鼠标滑动控制视角旋转
- 鼠标左键开枪
- 鼠标右键抬枪
- 鼠标侧键开镜

附加说明

- 代码中应用的第三方库
  - Assimp 用于导入3D模型
  - STB 用于导入图片
  - GLM 用于游戏中涉及到的数学表达与计算
- 版本开发控制

  Github仓库 https://github.com/ZJU-CG-Hunter/CG_Hunter.git 最终版本位于 v2.0分支
- 项目资源文件结构
  - CG_Hunter
    - src 项目核心代码 (.cpp文件)
    - include 项目头文件以及第三方头文件 (.h文件)
    - lib 第三方静态库

- dll 第三方动态库
- resources 项目三维模型文件及相关文件
- x64 项目最终可执行文件

基本得分点

基本几何表达

游戏中的所有模型，如树木、猎人、猪等，都是由三角面构成的，我们实现了对于多面体模型的基本的建模表达。同时，很多基础图形的绘制也都涉及到基础几何的表达。

三维网格导入

我们使用blender完成对动物和人物的设计，并导出为fbx文件。在游戏中同样以fbx的形式导入，具体我们使用了Assimp库将设计好的模型载入，并读取所需的数据和结构，从而实现了模型的多样化。

在导入模型到游戏中时，我们给每个模型指定其所属的模型类型，同时设置了模型在地图中的位置、方向、初始运动状态等，还为模型生成了碰撞检测的盒子，以便于后续的检测。

材质纹理

在游戏中，我们实现了较为庞大的森林地图，在地图中也插入了许多其它模型。对于场景中的天空盒、树木、模型，都分别设置了对应的顶点着色器和片段着色器来完成对顶点的处理以及对材质的模拟。同时模型的纹理贴图也能一起完成。

几何变换

通过使用平移矩阵和旋转矩阵的乘法运算，我们可以在每一次渲染时对猎人、动物以及他们的肢体进行相应的几何变换，从而实现猎人和动物流畅的动作表现。使用平移矩阵与模型的位置相乘，还可以控制猎人在不平的地图随意走动，以及子弹在地图上的飞行。

基本光照

光照部分是我们自己调节和实现的，在游戏场景中我们添加了点光源和环境光源，同时实现了阴影增加了真实感。

具体实现时，在处理光照时采用了基本的冯氏光照模型，来计算游戏场景中的环境光、漫反射以及镜面光，模拟比较自然的游戏场景。

在处理阴影时，为了获得比较真实的效果，我们利用GLSL帧缓冲中的深度缓存信息来构建场景阴影。包含两个方面：第一，利用PCF方法缓和阴影锯齿感；第二，对深度施加偏移极大改进阴影失真。

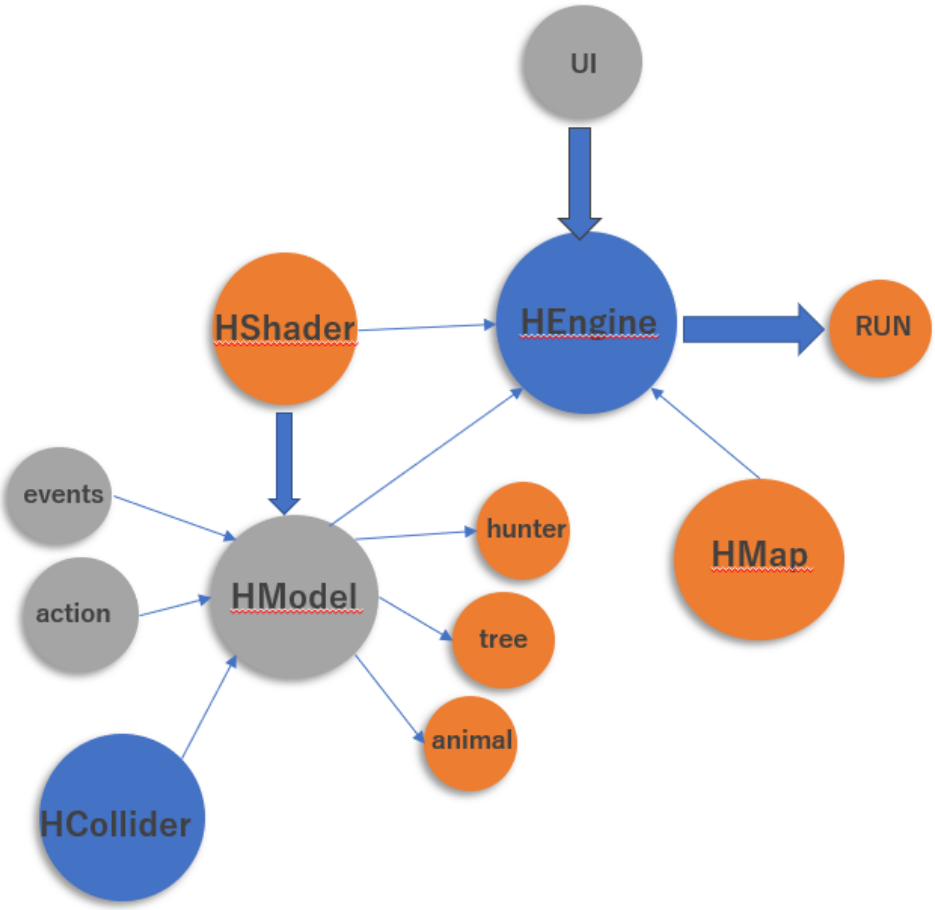
漫游

通过处理鼠标和键盘的输入，游戏中的角色可以实现视角的转换和漫游。视角转换在鼠标上下左右移动时处理，漫游在按下wasd键时处理。在触发相应事件时，我们对猎人的方向向量进行调整，使用平移矩阵来改变猎人的水平位置，同时修改相机的位置和方向，展现出朝向正确方向的画面。在每次循环中（每次渲染），我们根据猎人当前的位置坐标获得地图地形的当前高度，从而调整猎人的高度，使其可以紧贴地面，而不会漂浮在空中。

代码框架

框架逻辑

本项目中代码的主要框架逻辑如下图所示



- HEngine类，微小的引擎，主要负责封装GLFW初始化细节、提供UI接口、统一管理场景中的数据以及驱动整个游戏运行
  - 封装GLFW初始化细节

    在HEngine构造方法中会调用三个初始化方法，分别对应GLFW窗口初始化、STB库初始化以及类本身的初始化

    ```
    HEngine::HEngine(): _deltatime(0.0f), _lasttime(0.0f) ,
    _current_window(-1), _lastX(SCR_WIDTH / 2.0f), _lastY(SCR_HEIGHT /
    2.0f), _firstMouse(true) {
        ini_window_setting(); // 窗口初始化
        ini_stb_setting(); // stb库初始化
        ini_enging_setting(); //  HEngine类初始化
    }
    ```

  - 提供UI接口

    设置回调事件方法的第一个参数为由HEngine创建的窗口的序号，第二个参数为回调函数对应需要响应的函数指针

```
/* 缩放事件回调 */
void set_framebuffersize_callback(int window_index,
GLFWframebuffersizefun funptr);
/* 鼠标移动事件回调 */
void set_cursorpos_callback(int window_index, GLFWcursorposfun funptr);
/* 鼠标滚轮事件回调 */
void set_scroll_callback(int window_index, GLFWscrollfun funptr);
/* 鼠标按键事件回调 */
void set_mouse_button_callback(int window_index, GLFWmousebuttonfun
funptr);
```

- 统一管理场景中的数据

  HEngine类成员中包括了游戏中所有的模型HModel、游戏地图HMap、着色器HShader以及其他必要的需要维护的信息。通过调用HEngine类提供的方法接口，可以根据外部提供的参数（如文件相对路径、初始化必须参数等）更新上述数据

- 驱动整个游戏运行

  HEngine类中的run函数负责整个游戏场景的驱动，其每一次循环的步骤大致为

  - 处理用户UI
  - 处理碰撞检测
  - 更新时间
  - 通知模型更新数据
  - 渲染深度贴图
  - 清除颜色与深度缓存
  - 渲染游戏场景至缓存
  - 交换缓存显示游戏场景

```
void HEngine::run() {
    ini_render_setting();

    while (!glfwWindowShouldClose(_windows[_current_window])) {
        processInput();
        glfwPollEvents();

        /* Event */
        collision_detection();

        /* Process time */
        adjust_time();

        /* action */
        _hunter->Action(_map, _deltatime);
        _hunter->update_camera();

        for (unsigned int i = 0; i < _models.size(); i++)
            _models[i]->Action(_map, _deltatime);

        /* generate shadow */
        draw_shadow();

        /* clear */
        clear_buffer();
```

```
        /* render */
        _map->Draw();

        _hunter->Draw();

        for (unsigned int i = 0; i < _models.size(); i++)
            _models[i]->Draw();

        /* render skybox */
        _skybox[_current_skybox]->Draw();

        _hunter->DrawMagnifier();


        /* swap buffers and poll IO events */
        glfwSwapBuffers(_windows[_current_window]);


    }

    _windows.erase(_windows.begin()+_current_window);
}
```

- HModel类，游戏模型，负责单个模型的加载，变换、事件、渲染
  - 模型加载

    首先**调用第三方Assimp库**将fbx或obj文件加载为assimp风格的scene，根据scene将模型的顶点数据、骨骼数据、贴图数据加载入该模型中，并写入VBO、VAO中

    ```
    HModel::HModel(string const& path, bool gamma, int
    default_animation_index) : gammaCorrection(gamma),
    animation_index(default_animation_index)
    {
      genModelBuffer(); // 生成VAO VBO
      loadModel(path); // 根据文件路径加载模型
      genModelCollider(); // 生成模型碰撞盒子
    }
    ```

  - 模型变换

    HModel类本身兼具模板类的作用，其仅提供了基础的变换操作（平移、旋转、缩放），更个性化的变换（模型的行为）由其继承的类改写/增写，如HPig类、HBullet类、HHunter类

    如果该模型本身载入了骨骼数据，HModel类会根据其关键帧以及当前时间进行插值计算，从而完成模型的骨骼动作

  - 模型事件

    HModel提供了两类模型事件

    - Action： HEngine run方法中每轮循环都会主动调用所有模型的Action方法，其通用作用一般为更新骨骼数据、调整模型在地图上的高度，以下以HHunter类改写的Action方法为例（HModel本身的Action方法为空，仅用于被继承），

      ```
      void HHunter::Action(HMap* map, float duration_time) {
          switch (animation_index) {
      ```

```
        /* 根据当前动作更新响应数据 */
    }
    /* 更新骨骼变换 */
    UpdateBoneTransform();

    /* 更新模型高度，从而“踩”在地面上 */
    AdjustStepOnGround(map);

    /* 在地图中更新该模型的位置信息 */
    map->update_model(this);

    /* 更新摄像机（即更新view/perspective）*/
    update_camera();
}
```

- Event：“被动”调用的方法，碰撞后调用的事件，这里的代码逻辑为：HEngine每轮循环中调用碰撞检测方法 - > 遍历所有模型调用其自身的碰撞检测方法 - > 根据碰撞检测的结果调用模型Event方法，其传入的参数包含了该次碰撞的信息。
  - HEngine碰撞检测：遍历HEngine中所有模型进行碰撞检测

```
void HEngine::collision_detection() {
    /* 调用猎人的碰撞检测 */
    _hunter->collision_detection(_map);

    /* 调用模型的碰撞检测 */
    for (int i = 0; i < _models.size(); i++) {
        _models[i]->collision_detection(_map);
    }
}
```

  - HPig碰撞检测：

```
void HPig::collision_detection(HMap* _map) {
  vector<Model_Data> nearby;

  /* 若需要碰撞检测 */
  if (engine_detect_collision) {
    /* 通过地图获取该模型附近的模型 */
    nearby = _map->get_model_nearby(this, 5.0f);

    /* 获取碰撞检测结果 */
    Collision* collision_type = get_collide_type(this,
nearby[i]._model);

    /* 调用碰撞模型双方的事件函数 */
    this->Event(collision_type);
    nearby[i]._model->Event(collision_type);
  }
}
```

  - Event方法

    根据每个继承自HModel的Event方法不同，可以定义不同类型模型的行为不同

- 模型渲染

  模型渲染主要依托于模型初始顶点数据、变换数据以及绑定到该模型上的着色器，以下以抽象类HModel的渲染为例介绍模型渲染数据

  - Draw

    每一个HModel下有以树状结构存储的多张HMesh，HMesh类似与一个模型的某个组件

```
void HModel::Draw()
{
  /* 使用着色器 */
  shader->use();

  /* 更新着色器中Uniform缓冲块信息 */
  unsigned int Matrix_index = glGetUniformBlockIndex(shader->ID,
"Matrices");
  glUniformBlockBinding(shader->ID, Matrix_index, binding_point);
  glBindBuffer(GL_UNIFORM_BUFFER, matrix_buffer_id);
  unsigned int buffer_offset = 0;

  /* Bind WVP*/

  /* Bind bones */
  for (unsigned int i = 0; i < bone_map.size(); i++)
    BindUniformData(buffer_offset, &bones[i].bone_transform);
  glBindBuffer(GL_UNIFORM_BUFFER, 0);

  for (unsigned int i = 0; i < meshes.size(); i++) {
    meshes[i].Draw(shader);
  }
}
```

  - Model Shader

```
#version 460 core
layout (location = 0) in vec3  aPos;
layout (location = 1) in vec3  aNormal;
layout (location = 2) in vec2  aTexCoords;
layout (location = 3) in vec3  aTangent;
layout (location = 4) in vec3  aBitangent;
layout (location = 5) in ivec4 aBoneIDs;
layout (location = 6) in vec4  aWeights;
layout (location = 7) in int   aNumOfBones;


const int MAX_BONES = 1000;

out VS_OUT {
  out vec3 Normal;
  out vec3 Position;
  out vec2 TexCoords;
  out vec3 FragPos;
  out vec4 FragPosLightSpace;
}vs_out;
```

```
layout (std140) uniform Matrices
{
  mat4 model;
  mat4 projection;
  mat4 view;
  mat4 bones[MAX_BONES];
};

uniform mat4 lightSpaceMatrix;

void main()
{
    /* 生成骨骼变换矩阵 */
    mat4 BoneTransform = mat4(1.0f);
    if(aNumOfBones > 0){
      BoneTransform = mat4(0.0f);
      for(int i = 0; i < aNumOfBones; i++){
        BoneTransform += bones[aBoneIDs[i]] * aWeights[i];
      }
    }
    /* 计算法向量矩阵，位置向量，贴图坐标向量，光线视角位置向量*/
    vs_out.Normal = mat3(transpose(inverse(model * BoneTransform)))
* aNormal;
    vs_out.Position = vec3(model* BoneTransform * vec4(aPos, 1.0));
    vs_out.TexCoords = aTexCoords;
    vs_out.FragPosLightSpace = lightSpaceMatrix *
vec4(vs_out.FragPos, 1.0);

    gl_Position = projection * view * model * BoneTransform *
vec4(aPos, 1.0);
}
```

闪光点

动作表现

使用blend自己制作了多样化的动作，比如猎人的静止和奔跑，同时也有开枪的动作，这些动作没有通过动捕而是自己调整的。

具体的实现就是通过将模型手动进行绑骨操作，在调整关节摆好关键帧的动作，其中中间的动作帧通过blender能够进行线型差值来填补大部分的帧。而表现在代码时仍然是读取关键帧，结合利用glfw现有库进行线型差值来实现实时的动作过程的渲染

代码实现过程中，需要对不同的动作进行序号标识，在读取不同的输入时，执行相应编号的动作。难点在于对于同一个整体文件的解析，如何界定不同的动作时，进行了大量的尝试进行动作的对应，这和文件的写入顺序有关系，有时候导入时对于代码编写者是很难知道的。

基本上全部的过程都是自己实现，对于动作之间的差值虽然调用了部分库，但是思路仍然是属于我们自己的。因为常规的导入很多个帧也是可以实现的，这个思路也是我们之前有过的，但是那将大量的消耗内存，考虑到效果问题我们放弃了。虽然动作的真实度受到引擎素材内存等的限制，同时没有设计完全和真实世界一样的动作，但是相比于一般的模型既包括平移操作要生动很多。

开镜效果

通过运转相机的位置，同时通过自己绘制实现了瞄准状态，具体表现为，打开瞄准镜会，视野将会拉近，同时面前将会被瞄准镜覆盖。

具体的实现包括三个部分，一就是视角的转换，通过将识别到用户的瞄准镜输入（鼠标右键），将相机转换到枪的上方的位置，同时前侧上的方向向量保持不变这样就能实现相机的切换，代码上在HEngine中保存了多个相机，只要相应切换就能实现。

二就是子弹的行驶方向，为了简单化，同时认为子弹的速度很快，代码中只考虑了子弹的速度，和固定的方向，而忽略了重力和阻力对于子弹的影响，同时我们使用到了子弹的实体，那么就是通过碰撞检测来检测是否射中。

因此有了第三个问题，就是如何在很快的速度下实现碰撞检测，测试初期很多子弹由于太快而检测失败，后修改了子弹OBB盒生成逻辑，使用一段长条形盒子来显示轨迹，检测成功率提升至100%。

实时碰撞检测

全部自己编写代码实现了一个完整的碰撞检测系统，基本的逻辑就是用长方体来代替光滑曲面，但是其中的逻辑和实现更加的复杂。

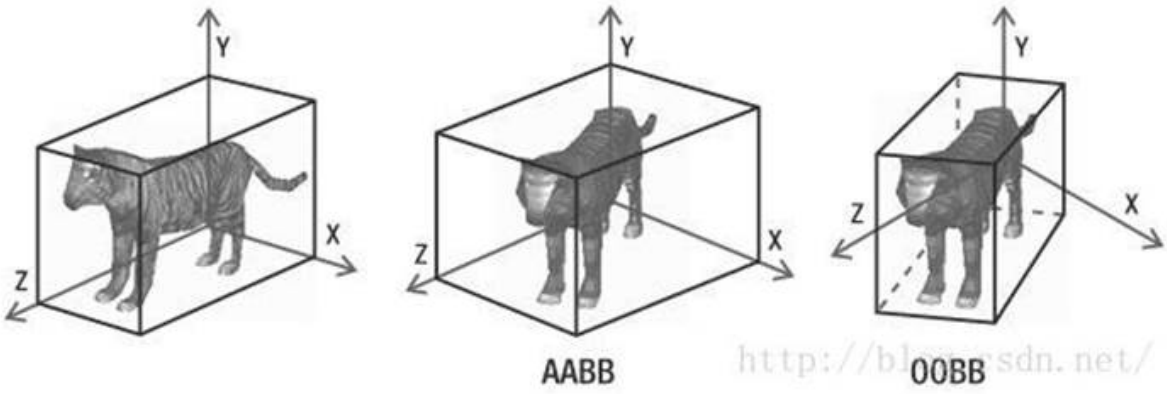
首先就是一般的3D模型的构架，一般的模型是有很多不同的Mesh的，比如头，手臂等。所以自然而然想到了一下的逻辑：为整个模型生成一个大盒子的同时，在模型不同部分生成小盒子，先检测到大盒子碰撞后，再检测小盒子之间的碰撞，这样实现了较为精细的碰撞检测，同时检测是几帧检测一次，基本上就是实时检测。

生成长方体的算法和长方体之间的检测也是整个系统的算法部分的重点。

- **自动生成OBB盒**

通过PCA分析主成分，具体就是计算协方差矩阵的特征向量后，再通过进行点积操作计算盒子的范围，计算初始的八个顶点。

这样的生成的盒子是紧贴着模型的，和AABB盒有本质上的区别，生成是在导入模型的时候而就生成盒子后续通过变换矩阵重复使用。



- **分离轴检测**

对于OBB盒检测采用分离轴方法，思想是找到一个可能的平面能够将这两个模型分开，那么他们就是分离的，相反所有的可能都显示碰撞，那就是碰撞了。

对于长方体可能性的平面是15个，其中很多只要沿着面的法向量进行投影，再判断投影的情况。这样三维投影到二维状态，二维再投影降至一维区间的比较，简单但是有效。