Project: Black Mamba

计算机图形学课程大项目

成员：薛辰立 3220102215

吴海槟 3220106042

李政达 3220102159

日期：2024 年 12 月 29 日

目录

Part I 游戏介绍 1
一、游戏背景 1
二、游戏玩法 1
Part II 架构设计 2
一、游戏对象 6
二、窗体系统 7
Part III 渲染系统 10
一、资源管理 10
（一）纹理资源 13
（二）顶点数据 16
（三）着色器资源 18
二、渲染流程 20
（一）着色器 21
（二）变换 22
（三）相机 23
（四）光照 25
（五）渲染实现 27
三、材质 32
（一）冯氏光照模型 34
（二）天空盒 41
（三）透明材质 45
（四）地形 46
（五）爆炸效果 51
Part IV 物理系统 54
一、物理组件 54
二、碰撞检测 57
（一）碰撞箱 57
（二）碰撞检测算法 57
三、力学模拟 58

（一）刚体 58
（二）质心定理 58
（三）物体运动定律 58
（四）碰撞 58
四、Jolt Physics[1] 59
（一）物理系统初始化 59
（二）物体放置与移除 62
（三）物理系统更新 63
Part V 音效系统 64
一、音频资源 65
二、音频组件 66
三、系统实现 67
Part VI 游戏实现 70
一、相机控制 70
二、模型导入 75
三、飞行控制 78
四、场景构建 81
Appendix: 环境配置 85
Bibliography 89

第一部分

游戏介绍

一、游戏背景

本游戏是一款飞行模拟器，在游戏中，你将扮演一名飞行员，驾驶飞机在天空中飞行。游戏中的飞机是根据真实飞机的力学模型构建的，玩家操作精密地绑定到飞机的力学控制面板上，通过操纵这些控制面板，玩家可以控制飞机的飞行姿态，从而实现飞机的飞行。游戏灵感来源[2]

二、游戏玩法

游戏中，玩家可以通过鼠标移动来控制视角旋转，通过键盘来操作飞机的飞行，`W`和`S`控制飞机机头的抬起和下降，`A`和`D`控制飞机的自身的转动，`Space`键控制飞机的加速。玩家可以通过这些操作来控制飞机的飞行，飞机在落地时，如果与地面的撞击过于强烈，飞机会损坏，玩家的飞行任务也会失败。

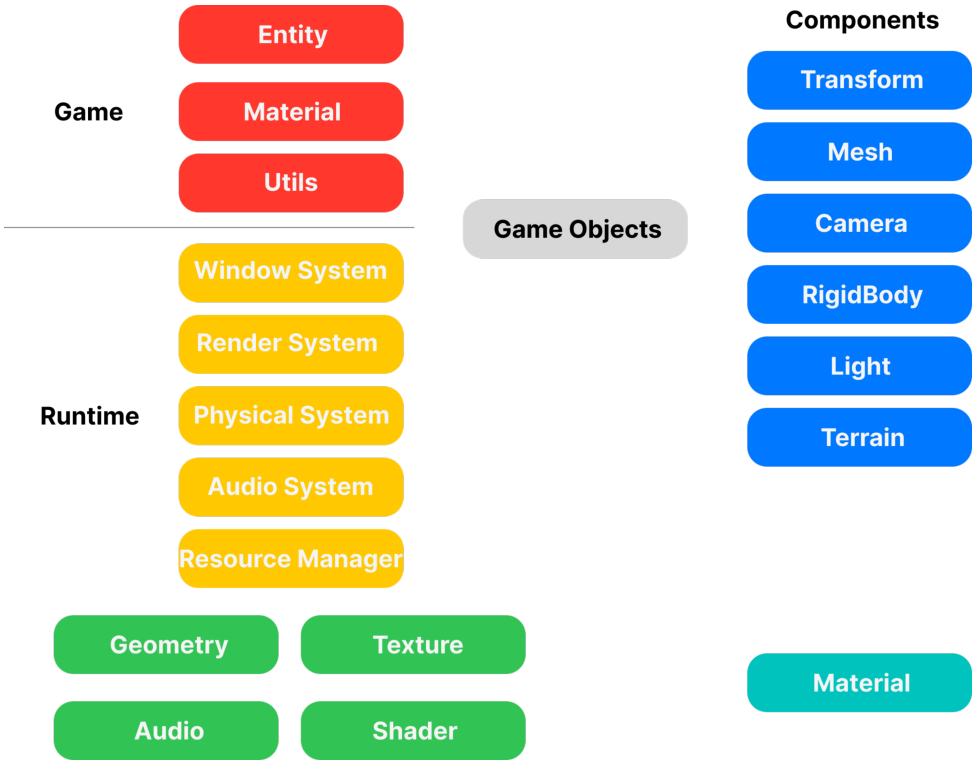
此外，玩家还可以通过`1`和`2`来切换相机视角，`1`键切换到自由视角，`2`键切换到驾驶员视角。在自由视角下，玩家可以自由地控制视角的旋转，同时可以通过小键盘的$\uparrow, \downarrow, \leftarrow, \rightarrow$来控制相机的移动；而在驾驶员视角下，玩家可以清晰地看到飞机驾驶舱内的情况，体验飞行员的视角。

图 1：游戏运行截图，模型[3]

第二部分

架构设计

本项目参考 games104[4] 介绍的引擎架构[5] 进行开发，组成如下



游戏由底层的引擎运行时与上层的游戏实现层构成，其中

- ◇ 运行时层由窗体系统(Window System)，渲染系统(Render System)，物理系统(Physical System)，音效系统(Audio System)与资源管理系统(Resource Manager)构成
  - ▷ 资源管理系统负责统一创建和销毁游戏中的各类资产，如顶点数据(Geometry)，纹理数据(Texture)，着色器(Shader)与音频(Audio)
  - ▷ 认为材质 = 纹理 + 着色器，对其进行单独封装，向游戏实现层提供扩展接口
- ◇ 游戏中一切实体皆为游戏对象(Game Objects)，游戏对象通过携带组建(Component)与系统与其它游戏对象进行交互
  - ▷ 基础组件包括：变换(Transform)，网格体(Mesh)，相机(Camera)，刚体(RigidBody)，光照(Light)和地形(Terrain)

♢ 游戏层则对运行时层提供的基础功能进行更加上层的封装，如游戏实体构建、定制化材质设计、相机控制器实现、场景布置等

引擎声明如下

include/runtime/engine.h

```C++
class Engine {
public:
  using MainLoop = std::function<void()>;

  enum class State { IDLE, RUNNING, PAUSE, STOP };

  static Engine *getEngine();
  static void destroyEngine();

  bool init(EngineInfo info);
  void start();
  void stop();
  void pause();
  void resume();

  State getState() const { return _state; }
  WindowSystem *getWindowSystem() const { return _windowSystem; }
  PhysicalSystem *getPhysicalSystem() const { return _physicalSystem; }
  AudioSystem *getAudioSystem() const { return _audioSystem; }
  ResourceManager *getResourceManager() const { return _resourceManger; }

  void setScene(GameObject *scene) { _scene = scene; }
  void setMainLoop(const MainLoop &mainloop) { _mainloop = mainloop; }

private:
  static Engine *_engine;

  State _state{State::IDLE};

  WindowSystem *_windowSystem;
  RenderSystem *_renderSystem;
  PhysicalSystem *_physicalSystem;
  AudioSystem *_audioSystem;
  ResourceManager *_resourceManger;

  GameObject *_scene;
  MainLoop _mainloop;

  Engine();
  ~Engine();

  void dispatch(GameObject *root);
```

```
43    void tick();
44  };
```

♢ 引擎采用单例模式设计，确保全局只有唯一的引擎实例，通过私有构造函数实现

♢ 其中 Game Object 的根结点以 Scene(场景)的形式在引擎中进行维护

引擎采用面向 Game Object 的形式进行更新，第一阶段先深度遍历 Game Object 树，将所有游戏对象分发到各个系统中；第二阶段依次更新各系统，每个系统根据分发时记录的游戏对象进行更新，如图所示

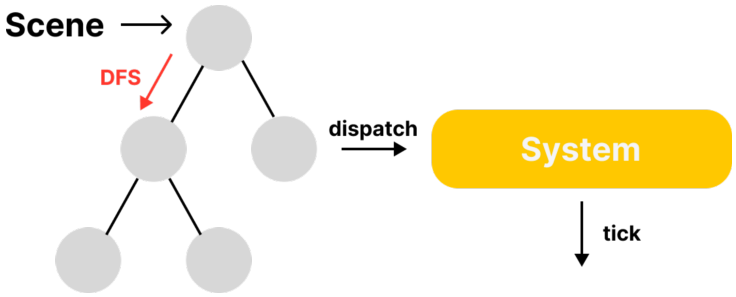


图 3：引擎更新逻辑

实现如下

runtime/engine.cpp C++

```cpp
void Engine::dispatch(GameObject *root) {
  if (root) {
    root->tick();
    _renderSystem->dispatch(root);
    _physicalSystem->dispatch(root);
    _audioSystem->dispatch(root);
    glm::mat4 rootModel = root->getComponent<TransformComponent>()->
    getModel();
    for (auto child : root->getChildren()) {
      child->getComponent<TransformComponent>()->setParentModel(rootModel);
      dispatch(child);
    }
  }
}

void Engine::tick(){
  float dt = 1.0f;
  _physicalSystem->tick(dt);
  _renderSystem->tick();
  _audioSystem->tick();
}
```

♢ 在分发过程中同时进行 Model Matrix 的下发，以维护父子游戏对象的变换继承关系

游戏层则在引擎层之上继续封装，声明如下

include/game/game.h

```cpp
class Game {
public:
  struct Resize {};
  struct KeyBoard {};
  struct Mouse {};
  struct Cursor {};

  static Game *getGame();

  void init(GameInfo info);
  void start();
  void exit();

  void bind(Resize, WindowSystem::ResizeCallback callback) {
    _engine->getWindowSystem()->addResizeCallback(callback);
  }
  void bind(KeyBoard, WindowSystem::KeyBoardCallback callback) {
    _engine->getWindowSystem()->addKeyBoardCallback(callback);
  }
  void bind(Mouse, WindowSystem::MouseCallback callback) {
    _engine->getWindowSystem()->addMouseCallback(callback);
  }
  void bind(Cursor, WindowSystem::CursorCallback callback) {
    _engine->getWindowSystem()->addCursorCallback(callback);
  }

  Engine *getEngine() { return _engine; }
  GameObject *getScene() { return _scene; }

private:
  static Game *_game;

  Engine *_engine{Engine::getEngine()};

  GameObject *_scene{nullptr};

  Game() = default;
  ~Game();

  void setupScene();
};
```

一、游戏对象

游戏对象即为游戏内一切实体，以树的形式进行管理。每个游戏对象负责维护父子关系，其中子物体继承父物体的所有变换；同时，游戏对象又是组件的载体，需维护自身具有的所有组件，并对外提供获取的接口。

游戏对象声明如下

include/runtime/framework/object/gameObject.h

```cpp
class GameObject {
public:
  using MainLoop = std::function<void()>;

  GameObject();
  ~GameObject();

  template <typename ComponentType>
  std::shared_ptr<ComponentType> getComponent() const {
    for (const auto &component : _components) {
      auto targetComponent =
          std::dynamic_pointer_cast<ComponentType>(component);
      if (targetComponent)
        return targetComponent;
    }
    return nullptr;
  }

  void addChild(GameObject *object);
  void removeChild(GameObject *object);

  void tick() {
    if (_tick)
      _tick();
  }

  void setTick(const MainLoop &tick) { _tick = tick; }

  void addComponent(std::shared_ptr<Component> component);
  void setComponent(std::shared_ptr<Component> component);

  GameObject *getParent() const { return _parent; }
  const std::vector<GameObject *> &getChildren() const { return _children; }

private:
  std::vector<std::shared_ptr<Component>> _components{};

  MainLoop _tick{};
```

```
39
40    GameObject *_parent{nullptr};
41    std::vector<GameObject *> _children{};
42  };
```

◇ 由于游戏对象具有复杂的层级关系，为了对其进行高效的内存管理，使用智能指针对组件进行维护，以兼容游戏对象共享组件的设计

◇ 使用 C++多态特性实现组件的统一管理

二、窗体系统

作为引擎实现的起点，窗体系统提供基本的窗体与事件管理功能。声明如下

include/runtime/framework/system/windowSystem.h C++

```
1   class WindowSystem {
2   public:
3     using ResizeCallback = std::function<void(int, int)>;
4     using KeyBoardCallback = std::function<void(int, int, int)>;
5     using MouseCallback = std::function<void(int, int, int)>;
6     using CursorCallback = std::function<void(double, double)>;
7
8     WindowSystem() = default;
9     ~WindowSystem();
10
11    bool init(WindowInfo info);
12    bool shouldClose();
13    void pollEvents();
14    void swapBuffers();
15
16    void addResizeCallback(ResizeCallback callback)
      { _resizeCallbacks.push_back(callback); };
17    void addKeyBoardCallback(KeyBoardCallback callback)
      { _keyBoardCallbacks.push_back(callback); }
18    void addMouseCallback(MouseCallback callback)
      { _mouseCallbacks.push_back(callback); }
19    void addCursorCallback(CursorCallback callback)
      { _cursorCallbacks.push_back(callback); }
20
21    int getWidth() const { return _width; }
22    int getHeight() const { return _height; }
23    float getAspect() const { return (float)_width / _height; }
24    void getCursorPosition(double &x, double &y) { glfwGetCursorPos(_window, &x,
      &y); }
25
```

```
26  private:
27    int _width, _height;
28    GLFWwindow *_window;
29
30    std::vector<ResizeCallback> _resizeCallbacks{};
31    std::vector<KeyBoardCallback> _keyBoardCallbacks{};
32    std::vector<MouseCallback> _mouseCallbacks{};
33    std::vector<CursorCallback> _cursorCallbacks{};
34
35    static void frameBufferSizeCallback(GLFWwindow *window, int width, int
      height) {
36      WindowSystem* app = (WindowSystem*)glfwGetWindowUserPointer(window);
37      if (app)
38        for (auto resizeCallback: app->_resizeCallbacks)
39          resizeCallback(width, height);
40    }
41
42    static void keyCallback(GLFWwindow *window, int key, int scancode, int
      action, int mods) {
43      WindowSystem* app = (WindowSystem*)glfwGetWindowUserPointer(window);
44      if (app)
45        for (auto keyBoardCallback: app->_keyBoardCallbacks)
46          keyBoardCallback(key, action, mods);
47    }
48
49    static void mouseCallback(GLFWwindow *window, int button, int action, int
      mods) {
50      WindowSystem* app = (WindowSystem*)glfwGetWindowUserPointer(window);
51      if (app)
52        for (auto mouseCallback: app->_mouseCallbacks)
53          mouseCallback(button, action, mods);
54    }
55
56    static void cursorCallback(GLFWwindow *window, double xpos, double ypos) {
57      WindowSystem* app = (WindowSystem*)glfwGetWindowUserPointer(window);
58      if (app)
59        for (auto cursorCallback: app->_cursorCallbacks)
60          cursorCallback(xpos, ypos);
61    }
62  };
```

◇ **回调函数注册逻辑：由于 GLFW 仅允许使用外部定义或静态成员函数进行回调注册，故在类中封装出静态回调函数，负责调用外部加入的回调函数；利用 GLFW 提供的 `glfwSetWindowUserPointer` 和 `glfwGetWindowUserPointer`，在静态函数中获取当前唯一的类对象的指针**

窗体初始化实现如下

runtime/framework/system/windowSystem.cpp

```cpp
bool WindowSystem::init(WindowInfo info) {
  _width = info.width, _height = info.height;

  glfwInit();
  glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
  glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 1);
  glfwWindowHint(GLFW_OPENGL_CORE_PROFILE, GLFW_OPENGL_CORE_PROFILE);

  _window = glfwCreateWindow(_width, _height, info.title, NULL, NULL);
  if (_window == nullptr) {
    Log("Fail to create window!");
    return false;
  }
  glfwMakeContextCurrent(_window);

  if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)) {
    Log("Fail to initialize GLAD!");
    return false;
  }

  glfwSetFramebufferSizeCallback(_window, frameBufferSizeCallback);
  glfwSetKeyCallback(_window, keyCallback);
  glfwSetMouseButtonCallback(_window, mouseCallback);
  glfwSetCursorPosCallback(_window, cursorCallback);
  glfwSetWindowUserPointer(_window, this);

  return true;
}
```

◇ 为了兼容 Apple silicon, 采用 OpenGL 4.1 core profile 版本

◇ 使用 GLAD 进行算子选择

第三部分

渲染系统

一、资源管理

由于游戏中的资源种类繁多，且存在大量重复与需要写入 GPU 的资源(这些资源需要合理地从 GPU 上释放)，因此需要一个专门的系统——资源管理系统来管理这些资源。资源管理系统的主要功能是加载、管理和释放资源，以及提供资源的查询接口，其内部实现是很多个资源池，按照资源类型进行分类，每种资源类型都有一个对应的资源池。

资源管理系统声明如下

include/runtime/resource/resourceManager.h

```cpp
class ResourceManager {
  friend class Engine;

public:
  ~ResourceManager();

  Texture *loadTexture(const std::string &filePath);
  Texture *loadTexture(const std::vector<std::string> &filePaths);
  Texture *loadTexture(const std::string &filePath, unsigned char *dataIn,
                       uint32_t widthIn, uint32_t heightIn);

  Shader *loadShader(const std::string &vertexPath,
                     const std::string &fragmentPath);
  Shader *loadShader(const std::string &vertexPath,
                     const std::string &geometryPath,
                     const std::string &fragmentPath);
  Shader *loadShader(const std::string &vertexPath,
                     const std::string &tessCtrlPath,
                     const std::string &tessEvalPath,
                     const std::string &fragmentPath);

  Geometry *createBoxGeometry(float size);
  Geometry *createPlaneGeometry(float width, float height);

  Geometry *loadGeometry(const std::vector<glm::vec3> &vertices,
                         const std::vector<glm::vec2> &uvs,
```

```
27                          const std::vector<glm::vec3> &normals,
28                          const std::vector<unsigned int> &indices);
29    Geometry *loadGeometry(const std::vector<glm::vec3> &vertices,
30                          const std::vector<glm::vec2> &uvs);
31
32    Audio *loadAudio(const std::string &filePath);
33
34    void pauseAllAudios();
35    void startAllAudios();
36
37  private:
38    std::unordered_map<std::string, Texture *> _textureMap{};
39    std::unordered_map<std::string, Shader *> _shaderMap{};
40    std::unordered_map<std::string, Audio *> _audioMap{};
41
42    std::vector<Geometry *> _geometryList{};
43
44    ResourceManager() = default;
45  };
```

◇ 资源管理系统负责纹理、着色器、顶点(几何)数据和音频数据的管理(音频数据将在音效系统进行介绍)

◇ 资源池使用字符串哈希实现，以资源路径作为索引构建哈希表，保证同一资产不会发生重复导入，从而降低内存占用

◇ 对各类数据对象的管理，通过私有数据对象的构造函数实现；各类数据对象声明对`ResourceManager`的友元关系，从而在资源管理器中实现统一创建与销毁

资源统一销毁实现如下

runtime/resource/resourceManager.cpp C++

```
1   ResourceManager::~ResourceManager() {
2     for (auto it = _textureMap.begin(); it != _textureMap.end(); it++)
3       delete it->second;
4     _textureMap.clear();
5     for (auto it = _shaderMap.begin(); it != _shaderMap.end(); it++)
6       delete it->second;
7     _shaderMap.clear();
8     for (auto it = _geometryList.begin(); it != _geometryList.end(); it++)
9       delete *it;
10    _geometryList.clear();
11    for (auto it = _audioMap.begin(); it != _audioMap.end(); it++)
12      delete it->second;
13    _audioMap.clear();
14  }
```

由于在 4.3 版本的 OpenGL 以前，并不提供错误回调，需要在每次调用 OpenGL API 时手动检错，实现如下

include/common/macro.h

```cpp
#define Log(format, ...)                                    \
    printf("\33[1;35m[%s,%d,%s] " format "\33[0m\n",    \
        __FILE__, __LINE__, __func__, ## __VA_ARGS__)

#define Err(format, ...) {                                  \
    printf("\33[1;31m[%s,%d,%s] " format "\33[0m\n",    \
        __FILE__, __LINE__, __func__, ## __VA_ARGS__);  \
    while(1);                                               \
}

void checkGLError(const char *file, int line, const char *func);

#ifdef DEBUG
#define GL_CALL(func)                                                   \
  func;                                                                 \
  checkGLError(__FILE__, __LINE__, __func__);
#else
#define GL_CALL(func) func;
#endif
```

common/checkError.cpp

```cpp
void checkGLError(const char *file, int line, const char *func) {
  GLenum errorCode = glGetError();
  std::string error = "";
  if (errorCode != GL_NO_ERROR) {
    switch (errorCode) {
    case GL_INVALID_ENUM:
      error = "INVALID_ENUM";
      break;
    case GL_INVALID_VALUE:
      error = "INVALID_VALUE";
      break;
    case GL_INVALID_OPERATION:
      error = "INVALID_OPERATION";
      break;
    case GL_OUT_OF_MEMORY:
      error = "OUT_OF_MEMORY";
      break;
    default:
      error = "UNKNOWN";
      break;
    }
    Err("OpenGL error: %s, file: %s, line: %d, func: %s", error.c_str(), file,
        line, func);
  }
}
```

渲染过程中涉及的如下三种资源：顶点数据、纹理数据与着色器。

（一）纹理资源

纹理资源是游戏中最常用的资源之一，用于渲染模型的表面，其在渲染时会绑定到一个纹理单元上参与运算，其主要属性有宽度，高度，纹理单元序号等，其定义如下：

include/runtime/resource/texture/texture.h

```cpp
class Texture {
    friend class ResourceManager;

public:
    int getWidth() const { return _width; };
    int getHeight() const { return _height; };
    GLuint getTextureID() const { return _textureID; };
    unsigned int getUnit() const { return _unit; };

    void setUnit(unsigned int unit) { _unit = unit; };
    void bind() const;

private:
    int _width{0};
    int _height{0};
    GLuint _textureID{0};
    unsigned int _unit{0};
    unsigned int _textureTarget{GL_TEXTURE_2D};

    Texture(const std::string& path, unsigned int unit);
    Texture(const std::vector<std::string>& paths, unsigned int unit);
    Texture(unsigned char* dataIn, uint32_t widthIn, uint32_t heightIn,
    unsigned int unit);
    ~Texture();

};
```

其在初始化时，通过传入纹理的路径或者数据，来创建一个纹理对象，其内部会调用 OpenGL 的 API 来生成纹理对象与 Mipmap，并将纹理数据写入 GPU 中。在渲染时，通过调用 bind 函数来绑定纹理到指定的纹理单元上。这些实现代码如下：

runtime/resource/texture/texture.cpp

```cpp
void Texture::bind() const{
    GL_CALL(glActiveTexture(GL_TEXTURE0 + _unit));
    GL_CALL(glBindTexture(_textureTarget, _textureID));
}

Texture::Texture(const std::string& path, unsigned int unit) : _unit(unit){

    int _nrChannels;
```

```
9       stbi_set_flip_vertically_on_load(true);
        unsigned char *data = stbi_load(path.c_str(), &_width, &_height,
10      &_nrChannels, STBI_rgb_alpha);
11
12      if(!data){
13          Log("Failed to load texture: %s", path.c_str());
14          stbi_image_free(data);
15      }
16
17      GL_CALL(glGenTextures(1, &_textureID));
18      GL_CALL(glActiveTexture(GL_TEXTURE0 + _unit));
19      GL_CALL(glBindTexture(GL_TEXTURE_2D, _textureID));
20
        GL_CALL(glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, _width, _height, 0,
21      GL_RGBA, GL_UNSIGNED_BYTE, data));
22      GL_CALL(glGenerateMipmap(GL_TEXTURE_2D));
23      stbi_image_free(data);
24
25      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
26      GL_NEAREST_MIPMAP_LINEAR);
27
        GL_CALL(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
28      GL_NEAREST));
        GL_CALL(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
29      GL_NEAREST));
30
31  }
32
33  // skybox: right left top bottom back front
    Texture::Texture(const std::vector<std::string>& paths, unsigned int unit) :
34  _unit(unit){
35      _textureTarget = GL_TEXTURE_CUBE_MAP;
36
37      stbi_set_flip_vertically_on_load(false);
38
39      GL_CALL(glGenTextures(1, &_textureID));
40      GL_CALL(glActiveTexture(GL_TEXTURE0 + _unit));
41      GL_CALL(glBindTexture(GL_TEXTURE_CUBE_MAP, _textureID));
42
43      int width, height, nrChannels;
44      for (unsigned int i = 0; i < paths.size(); i++){
            unsigned char *data = stbi_load(paths[i].c_str(), &width, &height,
45          &nrChannels, STBI_rgb_alpha);
46          if (data){
```

```
            GL_CALL(glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0,
47          GL_RGBA, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, data));
48          stbi_image_free(data);
49      } else {
50          Log("Failed to load texture: %s", paths[i].c_str());
51          stbi_image_free(data);
52      }
53  }
54
55  GL_CALL(glTexParameteri(_textureTarget, GL_TEXTURE_WRAP_S, GL_REPEAT));
56  GL_CALL(glTexParameteri(_textureTarget, GL_TEXTURE_WRAP_T, GL_REPEAT));
57  GL_CALL(glTexParameteri(_textureTarget, GL_TEXTURE_WRAP_R, GL_REPEAT));
58
59  GL_CALL(glTexParameteri(_textureTarget, GL_TEXTURE_MIN_FILTER,
    GL_NEAREST));
60  GL_CALL(glTexParameteri(_textureTarget, GL_TEXTURE_MAG_FILTER,
    GL_NEAREST));
61 }
62
63 Texture::Texture(unsigned char* dataIn, uint32_t widthIn, uint32_t heightIn,
  unsigned int unit) : _unit(unit), _width(widthIn), _height(heightIn){
64  int _nrChannels;
65  stbi_set_flip_vertically_on_load(true);
66
67  uint32_t dataInSize = !heightIn ? widthIn : (widthIn * heightIn * 4);
68  unsigned char *data = stbi_load_from_memory(dataIn, dataInSize, &_width,
    &_height, &_nrChannels, STBI_rgb_alpha);
69
70  if(!data){
71      Err("Failed to load texture from memory");
72      stbi_image_free(data);
73  }
74
75  GL_CALL(glGenTextures(1, &_textureID));
76  GL_CALL(glActiveTexture(GL_TEXTURE0 + _unit));
77  GL_CALL(glBindTexture(GL_TEXTURE_2D, _textureID));
78
79  GL_CALL(glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, _width, _height, 0,
    GL_RGBA, GL_UNSIGNED_BYTE, data));
80  GL_CALL(glGenerateMipmap(GL_TEXTURE_2D));
81  stbi_image_free(data);
82
83  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
84  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
    GL_NEAREST_MIPMAP_LINEAR);
85
```

```
86      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
87      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
88  }
```

◇ 使用 stb/image[6] 读取图片数据

◇ 对于 Cubemap, 需按照右、左、上、下、后、前的顺序载入 6 张贴图

◇ 为了防止远处贴图出现明显失真，使用 Mipmap 构建多级采样，并设置纹理过滤类型为 `GL_NEAREST_MIPMAP_LINEAR`，在相邻 Mipmap 层级间进行插值

（二）顶点数据

几何体资源是游戏中的另一种常用资源, 用于渲染模型的形状, 其在渲染时会将所需数据按照对应格式传入 GPU 参与运算，其主要属性有 VAO（储存数据格式），VBO（储存顶点, uv, 法线等信息, 供着色器以 attribute 数据形式进行调用）和 EBO（顶点索引信息）等, 其声明如下:

include/runtime/resource/geometry/geometry.h

```C++
class Geometry {
  friend class ResourceManager;

public:
  GLuint getVao() const { return _vao; }
  GLuint getPosVbo() const { return _posVbo; }
  GLuint getUvVbo() const { return _uvVbo; }
  GLuint getNormalVbo() const { return _normalVbo; }
  GLuint getEbo() const { return _ebo; }
  uint32_t getNumIndices() const { return _numIndices; }

private:
  GLuint _vao{0};
  GLuint _posVbo{0};
  GLuint _uvVbo{0};
  GLuint _normalVbo{0};
  GLuint _ebo{0};

  uint32_t _numIndices{0};

  Geometry();
  Geometry(const std::vector<glm::vec3> &vertices,
           const std::vector<glm::vec2> &uvs,
           const std::vector<glm::vec3> &normals,
           const std::vector<unsigned int> &indices);
  Geometry(const std::vector<glm::vec3> &vertices,
           const std::vector<glm::vec2> &uvs);
  ~Geometry();

  template <typename T>
```

```
31    void uploadBuffer(GLuint &buffer, const std::vector<T> &data,
32                      GLuint attributeIndex, GLint componentCount);
33    };
```

其中 VAO, VBO, EBO 的创建，实现如下

runtime/resource/geometry/geometry.cpp C++

```
1   template <typename T>
2   void Geometry::uploadBuffer(GLuint &buffer, const std::vector<T> &data,
3                               GLuint attributeIndex, GLint componentCount) {
4     GL_CALL(glGenBuffers(1, &buffer));
5     GL_CALL(glBindBuffer(GL_ARRAY_BUFFER, buffer));
6     GL_CALL(glBufferData(GL_ARRAY_BUFFER, data.size() * sizeof(T), data.data(),
7                          GL_STATIC_DRAW));
8     GL_CALL(glEnableVertexAttribArray(attributeIndex));
9     GL_CALL(glVertexAttribPointer(attributeIndex, componentCount, GL_FLOAT,
10                                  GL_FALSE, componentCount * sizeof(float),
11                                  (void *)0));
12  }
13
14  Geometry::Geometry(const std::vector<glm::vec3> &vertices,
15                     const std::vector<glm::vec2> &uvs,
16                     const std::vector<glm::vec3> &normals,
17                     const std::vector<unsigned int> &indices) {
18    _numIndices = indices.size();
19
20    glGenVertexArrays(1, &_vao);
21    glBindVertexArray(_vao);
22
23    uploadBuffer(_posVbo, vertices, 0, 3);
24    uploadBuffer(_uvVbo, uvs, 1, 2);
25    uploadBuffer(_normalVbo, normals, 2, 3);
26
27    glGenBuffers(1, &_ebo);
28    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, _ebo);
29    glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(unsigned int),
30                 &indices[0], GL_STATIC_DRAW);
31
32    glBindVertexArray(0);
33  }
```

♢ 使用 single buffer 模式管理各类顶点数据

♢ 使用模版统一不同数据类型 VBO 的创建

♢ 不考虑顶点数据在运行时的动态变化，使用`GL_STATIC_DRAW`开辟 GPU 侧静态缓冲区

（三）着色器资源

着色器资源是管线中的一个重要组成部分，用于控制渲染的流程，其储存着代码片段，在渲染时传入 GPU 进行编译和链接，需要维护着色器的 ID，在析构时，需要根据着色器 ID 释放着色器资源，其声明如下：

include/runtime/resource/shader/shader.h

```cpp
class Shader {
  friend class ResourceManager;

public:
  void begin() const;
  void end() const;

  GLuint getProgram() const { return _program; }

  void setUniform(const std::string &name, float value) const;
  void setUniform(const std::string &name, int value) const;
  void setUniform(const std::string &name, glm::vec3 value) const;
  void setUniform(const std::string &name, glm::mat3 value) const;
  void setUniform(const std::string &name, glm::mat4 value) const;

private:
  Shader(const std::string &vertexPath, const std::string &fragmentPath);
  Shader(const std::string &vertexPath, const std::string &geometryPath, const std::string &fragmentPath);
  Shader(const std::string &vertexPath, const std::string &tessCtrlPath, const std::string &tessEvalPath, const std::string &fragmentPath);
  ~Shader();

  enum class ErrorType { COMPILE, LINK };

  GLuint _program{0};

  static void checkShaderErrors(GLuint target, ErrorType type);

  GLuint compileShader(const std::string &path, GLenum shaderType);
  void linkProgram(const std::vector<GLuint> &shaders);
};
```

◇ 着色器准备流程为：从文件中读取着色器代码，编译，链接

◇ 由于在 OpenGL 可编程管线中，几何着色器、镶嵌评估/控制着色器为可选着色器，为了兼容不同的着色器组合，在链接着色器时，使用可变长数组，同时提供多种着色器生成接口

◇ 使用函数重载统一不同类型的`Uniform`变量的设置

◇ 由于着色器准备过程中涉及编译链接，需为其提供完备的错误检查

其中着色器资源的准备，实现如下(以最简单的顶点+片元着色器为例)

runtime/resource/shader/shader.cpp C++

```cpp
Shader::Shader(const std::string &vertexPath, const std::string &fragmentPath)
{
  GLuint vertex = compileShader(vertexPath, GL_VERTEX_SHADER);
  GLuint fragment = compileShader(fragmentPath, GL_FRAGMENT_SHADER);
  linkProgram({vertex, fragment});
  glDeleteShader(vertex);
  glDeleteShader(fragment);
}

void Shader::checkShaderErrors(GLuint target, ErrorType type) {
  int success = 0;
  int infoLogLength = 0;

  if (type == ErrorType::COMPILE) {
    glGetShaderiv(target, GL_COMPILE_STATUS, &success);
    if (!success) {
      glGetShaderiv(target, GL_INFO_LOG_LENGTH, &infoLogLength);
      std::vector<char> infoLog(infoLogLength);
      glGetShaderInfoLog(target, infoLogLength, NULL, infoLog.data());
      Err("Shader compile error: %s", infoLog.data());
    }
  } else if (type == ErrorType::LINK) {
    glGetProgramiv(target, GL_LINK_STATUS, &success);
    if (!success) {
      glGetProgramiv(target, GL_INFO_LOG_LENGTH, &infoLogLength);
      std::vector<char> infoLog(infoLogLength);
      glGetProgramInfoLog(target, infoLogLength, NULL, infoLog.data());
      Err("Shader link error: %s", infoLog.data());
    }
  }
}

GLuint Shader::compileShader(const std::string &path, GLenum shaderType) {
  std::ifstream file;
  std::string code;
  file.exceptions(std::ifstream::failbit | std::ifstream::badbit);
  try {
    file.open(path.c_str());
    std::stringstream stream;
    stream << file.rdbuf();
    file.close();
    code = stream.str();
  } catch (std::ifstream::failure &e) {
    Err("Shader file error: %s", e.what());
  }
```

```
45    GLuint shader = glCreateShader(shaderType);
46    const char *source = code.c_str();
47    glShaderSource(shader, 1, &source, NULL);
48    glCompileShader(shader);
49    checkShaderErrors(shader, ErrorType::COMPILE);
50    return shader;
51  }
52
53  void Shader::linkProgram(const std::vector<GLuint> &shaders) {
54    _program = glCreateProgram();
55    for (auto s : shaders) {
56      glAttachShader(_program, s);
57    }
58    glLinkProgram(_program);
59    checkShaderErrors(_program, ErrorType::LINK);
60  }
```

二、渲染流程

OpenGL 的渲染管线为顶点着色、图元装配、剪裁剔除、光栅化、片元着色、测试与混合。具体到实现上，即先遍历所有需要渲染的物体，设置渲染状态，然后绑定着色器与纹理单元，设置着色器中的 Uniform 变量，最后绑定 VAO，下发 DrawCall.

渲染过程中涉及的数据及其来源如下图所示

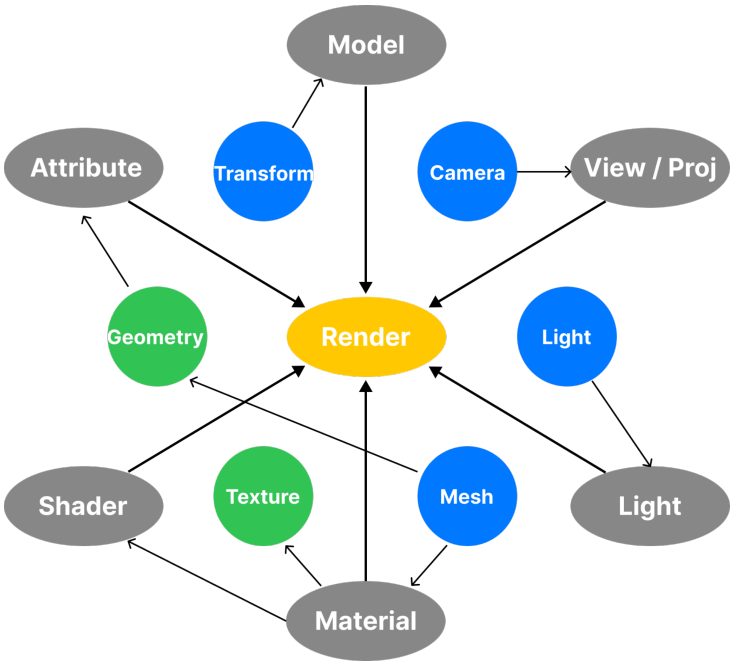


图 4：渲染过程涉及的数据

◇ **Mesh = Geometry + Material，即网格体组件提供渲染所需的几何形体与材质信息**

♢ Material = Texture + Uniform(Texture 等), 即 Material 维护在 GPU 上实际运行的程序和其所需要的与材质本身相关的数据

♢ Transform 组件的目的是提供顶点着色器的 ModelMatrix，供顶点着色器使用

♢ Camera 组件的目的是提供顶点着色器的 ViewMatrix 和 ProjectionMatrix, 供顶点着色器使用；此外，对于光照计算所需的相机位置，则可有相机的 Transform 组件提供

♢ Light 组件的目的是提供 Shader 中与光照相关的 Uniform 变量的值

渲染本质上是要汇聚这些信息，然后对每个物体执行渲染流程，完成图形的渲染。

(一) 着色器

着色器作为渲染过程的核心，一切数据都是围绕着色器进行设置，如 VAO, 本质上是提供着色器的属性数据，而其它数据则是为 Uniform 变量提供数据。

最简单的着色器由顶点着色器和片元着色器构成。以下是最简单的白模的着色器, 用作默认材质的着色器

assets/shaders/white/white.vert vert

```vert
#version 410 core

layout(location = 0) in vec3 aPosition;
layout(location = 1) in vec2 aUv;
layout(location = 2) in vec3 aNormal;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
  gl_Position = projection * view * model * vec4(aPosition, 1.0);
}
```

♢ 对于顶点数据，在 VAO 中已经描述了位置信息、UV 信息和法线信息

♢ 顶点着色器负责对顶点位置进行 MVP 变换，先将坐标转换为齐次坐标，最后输出到内置变量 `gl_Position` 中

assets/shaders/white/white.frag frag

```frag
#version 410 core

out vec4 FragColor;

uniform vec3 color;

void main() {
    FragColor = vec4(color, 1.0);
}
```

（二）变换

Transform 组件对物体的变换信息进行维护，目的是提供物体的模型矩阵用于渲染、为物理系统提供位置信息等。

仿照 Unity[7],游戏对象默认携带 Transform 组件。声明、实现如下

include/runtime/framework/component/transform/transform.h C++

```cpp
class TransformComponent : public Component {
public:
  TransformComponent() = default;
  ~TransformComponent() override{};

  glm::mat4 getModel() const;
  glm::vec3 getPositionLocal() const { return _position; }
  glm::vec3 getPositionWorld() const { return getModel() * glm::vec4(0.0f, 0.0f, 0.0f, 1.0f); }
  glm::vec3 getForwardVec() const {
    return  glm::normalize(glm::vec3(glm::vec4{_forwardVec,0.0f}  *
    glm::yawPitchRoll(glm::radians(_angle.y), glm::radians(_angle.x), glm::radians(-_angle.z)) ));
  }
  glm::vec3 getUpVec() const {
    return  glm::normalize(glm::vec3(glm::vec4{_upVec,0.0f}  *
    glm::yawPitchRoll(glm::radians(_angle.y), glm::radians(_angle.x), glm::radians(-_angle.z)) ));
  }
  glm::vec3 getRightVec() const {
    return  glm::normalize(glm::vec3(glm::vec4{_rightVec,0.0f}  *
    glm::yawPitchRoll(glm::radians(_angle.y), glm::radians(_angle.x), glm::radians(-_angle.z)) ));
  }

  glm::vec3 getAngle() const { return _angle; }
  glm::vec3 getScale() const { return _scale; }
  glm::mat4 getParentModel() const { return _parentModel; }

  void setPositionLocal(const glm::vec3 &position) { _position = position; }
  void setAngle(const glm::vec3 &angle) { _angle = angle; }
  void setScale(const glm::vec3 &scale) { _scale = scale; }
  void setParentModel(const glm::mat4 &model) { _parentModel = model; }

private:
  glm::vec3 _position{glm::zero<float>()};
  glm::vec3 _angle{glm::zero<float>()};
  glm::vec3 _scale{glm::one<float>()};

```

```
    glm::vec3 _forwardVec{0.0f, 0.0f, 1.0f};
    glm::vec3 _upVec{0.0f, 1.0f, 0.0f};
    glm::vec3 _rightVec{-1.0f, 0.0f, 0.0f};

    glm::mat4 _parentModel{glm::one<float>()};
};
```

runtime/framework/component/transform/transform.cpp C++

```
glm::mat4 TransformComponent::getModel() const {
    glm::mat4 model{1.0f};
    model = glm::scale(model, _scale);
    model = glm::rotate(model, glm::radians(_angle.x), glm::vec3(1.0f, 0.0f, 0.0f));
    model = glm::rotate(model, glm::radians(_angle.y), glm::vec3(0.0f, 1.0f, 0.0f));
    model = glm::rotate(model, glm::radians(_angle.z), glm::vec3(0.0f, 0.0f, 1.0f));
    return _parentModel * glm::translate(glm::mat4(1.0f), _position) * model;
}
```

◇ `getPositionLocal`，`getPositionWorld` 用于获取物体的局部坐标与全局坐标

◇ `getForwardVec`，`getUpVec`，`getRightVec` 用于确定物体的朝向，用于物理系统

在分发过程中，对父节点的 ModelView 信息进行维护，具体见 Listing 1.

渲染时，通过获取游戏对象的 Transform 组件，得到相应的 ModelView.

（三）相机

Camera 组件目的是提供 View 与 Projection Matrix, 根据投影方式的不同，分为正交投影和透视投影两种。

声明、实现如下

include/runtime/framework/component/camera/camera.h C++

```
class CameraComponent : public Component {
public:
    enum class Type { Ortho, Perspective, Invalid };

    CameraComponent(float left, float right, float top, float bottom, float near, float far);
    CameraComponent(float fovy, float aspect, float near, float far);

    ~CameraComponent() override{};

    bool isMain() const { return _isMain; }
    void pick() { _isMain = true; }
    void put() { _isMain = false; }
```

```

    glm::mat4 getView(glm::vec3 position) const;
    glm::mat4 getProjection() const;

    const glm::vec3 &getUpVec() const { return _upVec; }
    const glm::vec3 &getRightVec() const { return _rightVec; }

    void setUpVec(const glm::vec3 &up) { _upVec = up; }
    void setRightVec(const glm::vec3 &right) { _rightVec = right; }

private:
    glm::vec3 _upVec{0.0f, 1.0f, 0.0f}, _rightVec{1.0f, 0.0f, 0.0f};
    Type _type{Type::Invalid};

    bool _isMain{true};

    float _top, _bottom, _left, _right, _near, _far, _fovy, _aspect;
};
```

runtime/framework/component/camera/camera.cpp C++

```
CameraComponent::CameraComponent(float left, float right, float top, float bottom, float near, float far) {
    _type = Type::Ortho;
    _left = left, _right = right, _top = top, _bottom = bottom, _near = near, _far = far;
}

CameraComponent::CameraComponent(float fovy, float aspect, float near, float far) {
    _type = Type::Perspective;
    _fovy = fovy, _aspect = aspect, _near = near, _far = far;
}

glm::mat4 CameraComponent::getView(glm::vec3 position) const {
    glm::vec3 front = glm::cross(_upVec, _rightVec);
    return glm::lookAt(position, position + front, _upVec);
}

glm::mat4 CameraComponent::getProjection() const {
    if (_type == Type::Ortho)
        return glm::ortho(_left, _right, _bottom, _top);
    else if (_type == Type::Perspective)
        return glm::perspective(glm::radians(_fovy), _aspect, _near, _far);
    Log("Unknown camera type!");
    return glm::mat4(glm::one<float>());
}
```

◇ **通过重载构造函数，实现不同类型相机组件的构造**

（四）光照

Light 组件用于记录不同光源所需的信息，包括环境光、平行光、点光源、探照灯四种类型，声明如下

include/runtime/framework/component/light/light.h

```cpp
class LightComponent : public Component {
public:
  enum class Type { Directional, Point, Spot, Ambient, Invalid };

  LightComponent(const glm::vec3 &color) : _color(color), _type(Type::Ambient) {}
  LightComponent(const glm::vec3 &color, const glm::vec3 &direction,
        float specularIntensity)
      : _color(color), _direction(direction),
        _specularIntensity(specularIntensity), _type(Type::Directional) {}
  LightComponent(const glm::vec3 &color, float specularIntensity, float k2,
  float k1,
        float kc)
      : _color(color), _specularIntensity(specularIntensity), _k2(k2),
      _k1(k1),
        _kc(kc), _type(Type::Point) {}
  LightComponent(const glm::vec3 &color, const glm::vec3 &direction,
        float specularIntensity, float inner, float outer)
      : _color(color), _direction(direction),
        _specularIntensity(specularIntensity), _inner(inner), _outer(outer),
        _type(Type::Spot) {}
  ~LightComponent() override{};

  Type getType() const { return _type; }
  glm::vec3 getColor() const { return _color; }
  glm::vec3 getDirection() const { return _direction; }
  float getSpecularIntensity() const { return _specularIntensity; }
  float getK2() const { return _k2; }
  float getK1() const { return _k1; }
  float getKc() const { return _kc; }
  float getInner() const { return _inner; }
  float getOuter() const { return _outer; }

private:
  glm::vec3 _color, _direction;
  float _specularIntensity, _k2, _k1, _kc, _inner, _outer;

  Type _type{Type::Invalid};
};
```

其中光源声明如下

include/runtime/framework/component/light/light.h

```cpp
struct DirectionalLight {
  glm::vec3 color;
  glm::vec3 direction;
  float specularIntensity;
};

struct PointLight {
  glm::vec3 position;
  glm::vec3 color;
  float specularIntensity;
  float k2;
  float k1;
  float kc;
};

struct SpotLight {
  glm::vec3 position;
  glm::vec3 color;
  glm::vec3 direction;
  float specularIntensity;
  float inner;
  float outer;
};

struct AmbientLight {
  glm::vec3 color;
};
```

与片元着色器中的声明保持一致

assets/shaders/phong/phong.frag

```frag
struct DirectionalLight {
  vec3 color;
  vec3 direction;
  float specularIntensity;
};

struct PointLight {
  vec3 position;
  vec3 color;
  float specularIntensity;
  float k2;
  float k1;
  float kc;
};

```

```
16 struct SpotLight {
17   vec3 position;
18   vec3 color;
19   vec3 direction;
20   float specularIntensity;
21   float inner;
22   float outer;
23 };
24
25 struct AmbientLight {
26   vec3 color;
27 };
```

（五）渲染实现

渲染系统只需按照上述流程收集所有信息，再对每个物体进行绘制即可。

实现如下：

include/runtime/framework/system/renderSystem.h C++

```
1  struct ModelInfo {
2    glm::mat4 model;
3  };
4
5  struct CameraInfo {
6    glm::vec3 position;
7    glm::mat4 view;
8    glm::mat4 project;
9  };
10
11 struct LightInfo {
12   std::vector<DirectionalLight> directionalLights{};
13   std::vector<PointLight> pointLights{};
14   std::vector<SpotLight> spotLights{};
15   std::vector<AmbientLight> ambientLights{};
16 };
17
18 struct RenderInfo {
19   ModelInfo &modelInfo;
20   CameraInfo &cameraInfo;
21   LightInfo &lightInfo;
22 };
23
24 class RenderSystem {
25 public:
26   RenderSystem() = default;
27   ~RenderSystem() = default;
```

```
28
29    void dispatch(GameObject *object);
30    void tick();
31
32  private:
33    std::vector<GameObject *> _meshes[2], _lights, _cameras;
34
35    void initState();
36    void setDepthState(Material *material);
37    void setBlendState(Material *material);
38
39    void clear();
40  };
```

◇ 声明模型信息、相机信息与光照信息结构体，便于传参

```
1    void RenderSystem::dispatch(GameObject *object) {
2      if (object->getComponent<MeshComponent>())
3        _meshes[object->getComponent<MeshComponent>()->getMaterial()->getBlend()]
4            .push_back(object);
5      if (object->getComponent<LightComponent>())
6        _lights.push_back(object);
7      if (object->getComponent<CameraComponent>())
8        _cameras.push_back(object);
9      if (object->getComponent<TerrainComponent>())
10       _meshes[0].push_back(object);
11   }
12
13   void RenderSystem::tick() {
14     initState();
15     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
16
17     GameObject *mainCamera = nullptr;
18     if (_cameras.empty())
19       Err("No camera found!");
20     for (auto camera : _cameras) {
21       if (camera->getComponent<CameraComponent>()->isMain()) {
22         mainCamera = camera;
23         break;
24       }
25     }
26     if (!mainCamera) {
27       Log("No main camera found! Use default.");
28       _cameras[0]->getComponent<CameraComponent>()->pick();
29       mainCamera = _cameras[0];
30     }
31     std::shared_ptr<CameraComponent> mainCameraComp =
32         mainCamera->getComponent<CameraComponent>();
```

```
33      std::shared_ptr<TransformComponent> mainCameraTransfrom =
34          mainCamera->getComponent<TransformComponent>();
35      glm::vec3 mainCameraPosition = mainCameraTransfrom->getPositionWorld();
36      CameraInfo cameraInfo{.position = mainCameraPosition,
37                            .view = mainCameraComp->getView(mainCameraPosition),
38                            .project = mainCameraComp->getProjection()};
39
40      LightInfo lightInfo;
41      for (auto light : _lights) {
42        std::shared_ptr<LightComponent> lightComp =
43            light->getComponent<LightComponent>();
44        switch (lightComp->getType()) {
45        case LightComponent::Type::Directional:
46          lightInfo.directionalLights.emplace_back(DirectionalLight{
47              .color = lightComp->getColor(),
48              .direction = lightComp->getDirection(),
49              .specularIntensity = lightComp->getSpecularIntensity()});
50          break;
51        case LightComponent::Type::Point:
52          lightInfo.pointLights.emplace_back(PointLight{
53              .position =
54                  light->getComponent<TransformComponent>()->getPositionWorld(),
55              .color = lightComp->getColor(),
56              .specularIntensity = lightComp->getSpecularIntensity(),
57              .k2 = lightComp->getK2(),
58              .k1 = lightComp->getK1(),
59              .kc = lightComp->getKc()});
60          break;
61        case LightComponent::Type::Spot:
62          lightInfo.spotLights.emplace_back(SpotLight{
63              .position =
64                  light->getComponent<TransformComponent>()->getPositionWorld(),
65              .color = lightComp->getColor(),
66              .direction = lightComp->getDirection(),
67              .specularIntensity = lightComp->getSpecularIntensity(),
68              .inner = lightComp->getInner(),
69              .outer = lightComp->getOuter()});
70          break;
71        case LightComponent::Type::Ambient:
72          lightInfo.ambientLights.emplace_back(
73              AmbientLight{.color = lightComp->getColor()});
74          break;
75        default:
76          Log("Unknow light type. Abort.");
77          break;
78        }
79      }
```

```
80
81      std::sort(
82          _meshes[1].begin(), _meshes[1].end(),
83          [&cameraInfo](const GameObject *a, const GameObject *b) {
84            return (cameraInfo.view *
85                    glm::vec4(
86                        a->getComponent<TransformComponent>()->
                        getPositionWorld(),
87                        1.0))
88                       .z <
89                   (cameraInfo.view *
90                    glm::vec4(
91                        b->getComponent<TransformComponent>()->
                        getPositionWorld(),
92                        1.0))
93                       .z;
94          });
95
96      for (auto meshes : _meshes)
97        for (auto mesh : meshes) {
98          std::shared_ptr<MeshComponent> meshComp =
99              mesh->getComponent<MeshComponent>();
100         Geometry *geometry = meshComp->getGeometry();
101         Material *material = meshComp->getMaterial();
102         ModelInfo modelInfo{
103             .model = mesh->getComponent<TransformComponent>()->getModel()};
104
105         setDepthState(material);
106         setBlendState(material);
107
108         material->apply(RenderInfo{.modelInfo = modelInfo,
109                                    .cameraInfo = cameraInfo,
110                                    .lightInfo = lightInfo});
111         GL_CALL(glBindVertexArray(geometry->getVao()));
112         if (mesh->getComponent<TerrainComponent>()) {
113           int rez = mesh->getComponent<TerrainComponent>()->getRez();
114           GL_CALL(glPatchParameteri(GL_PATCH_VERTICES, 4));
115           GL_CALL(glDrawArrays(GL_PATCHES, 0, 4 * rez * rez));
116         } else {
117           GL_CALL(glDrawElements(GL_TRIANGLES, geometry->getNumIndices(),
118                                  GL_UNSIGNED_INT, 0));
119         }
120         material->finish();
121       }
122     clear();
123   }
124
```

```cpp
void RenderSystem::clear() {
  _meshes[0].clear();
  _meshes[1].clear();
  _lights.clear();
  _cameras.clear();
}

void RenderSystem::initState() {
  glEnable(GL_DEPTH_TEST);
  glDepthFunc(GL_LEQUAL);
  glDepthMask(GL_TRUE);
  glDisable(GL_BLEND);
}

void RenderSystem::setDepthState(Material *material) {
  if (material->getDepthTest()) {
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(material->getDepthFunc());
  } else
    glDisable(GL_DEPTH_TEST);

  if (material->getDepthWrite())
    glDepthMask(GL_TRUE);
  else
    glDepthMask(GL_FALSE);
}

void RenderSystem::setBlendState(Material *material) {
  if (material->getBlend()) {
    glEnable(GL_BLEND);
    glBlendFunc(material->getSFactor(), material->getDFactor());
  } else
    glDisable(GL_BLEND);
}
```

◇ 分发阶段，需对物体进行分类，确定网格体、光照与相机

◇ 更新阶段，由于考虑到透明材质与地形的渲染，需对基础的渲染流程进行调整

▷ 基础流程：设定渲染状态，收集信息，绑定着色器，施用材质，绑定 VAO，调用 DrawCall

▷ 由于透明材质的实现，本质上是利用了颜色混合功能，故需修改渲染顺序。对于渲染系统而言，游戏对象乱序分发，原先基于深度缓冲的绘制算法无法保证透明材质能混合到正确的颜色；若透明材质先于不透明材质而绘制，则由于透明材质不进行深度写入，若不透明材质的深度值小于透明材质，则会将透明物体遮蔽；故正确的渲染流程为：先绘制不透明物体，再按距离相机由远及近的顺序依次绘制透明物体，以确保颜色能正确混合。上述实现中，不透明物体被放置在 `_mesh[0]`，透明物体被放置在 `_mesh[1]`；在获取到相机信息后，即可计算出与相机的距离(变换到相机系，取 z 分量)并排序

- 对于地形数据，由于绘制的补丁顶点数不同，且地形绘制并不依赖于 EBO，故需对地形进行特殊判断，调用地形绘制的 DrawCall

此处的渲染流程亦可扩展，若需进行阴影渲染等后处理，可以为其添加帧缓冲，并渲染多个 pass，如在前一次渲染的颜色附件、深度模版附件上进行采样等。由于时间关系，此处并未做过多处理，若后续有机会再引入阴影系统的处理。

三、材质

CPU 端准备了复杂且精美的美术模型，而实时、高级、细腻的特效则依赖于 GPU 端的着色器，如天空盒、地形生成、爆炸效果等。GPU 端的程序(着色器)及其数据（材质）被封装为 Material，声明如下

include/runtime/resource/material/material.h

```cpp
class Material {
public:
  virtual ~Material() = 0;

  Material() = default;
  Material(const Material& other) {
    _shader = other._shader;
    _depthTest = other._depthTest;
    _depthFunc = other._depthFunc;
    _depthWrite = other._depthWrite;
    _blend = other._blend;
    _sFactor = other._sFactor;
    _dFactor = other._dFactor;
  }

  virtual Material *clone() const = 0;
  virtual void apply(const RenderInfo &info) = 0;

  bool getDepthTest() const { return _depthTest; }
  GLenum getDepthFunc() const { return _depthFunc; }
  bool getDepthWrite() const { return _depthWrite; }
  bool getBlend() const { return _blend; }
  unsigned int getSFactor() const { return _sFactor; }
  unsigned int getDFactor() const { return _dFactor; }

  virtual void setDiffuse(Texture *diffuse){};

  void finish() {
    if (_shader)
      _shader->end();
  };
```

```
32
33  protected:
34    Shader *_shader = nullptr;
35
36    // Depth test
37    bool _depthTest = true;
38    GLenum _depthFunc = GL_LEQUAL;
39    bool _depthWrite = true;
40
41    // Color blending
42    bool _blend = false;
43    unsigned int _sFactor = GL_SRC_ALPHA;
44    unsigned int _dFactor = GL_ONE_MINUS_SRC_ALPHA;
45  };
```

◇ 为了利用C++多态特性，提供`apply`作为虚方法，在渲染时通过`apply`接口来调用各种不同材质统一设置着色器参数

◇ 由于模型载入时，需对材质进行深拷贝，提供`clone`方法实现材质拷贝

这样，添加新材质时，只需重载这些虚方法即可。以白模为例

include/runtime/resource/material/whiteMaterial.h

```C++
class WhiteMaterial : public Material {
public:
  WhiteMaterial();
  WhiteMaterial(const WhiteMaterial &other) : Material(other) {
    _color = other._color;
  }

  ~WhiteMaterial() override{};
  void setColor(const glm::vec3 &color);

  WhiteMaterial* clone() const override { return new WhiteMaterial(*this); }
  void apply(const RenderInfo &info) override;

private:
  glm::vec3 _color = glm::vec3(1.0f, 1.0f, 1.0f);
};
```

实现如下：

```C++
WhiteMaterial::WhiteMaterial(){
    ResourceManager *resourceManager = Engine::getEngine()->
    getResourceManager();
    _shader = resourceManager->loadShader("assets/shaders/white/white.vert",
    "assets/shaders/white/white.frag");
}

void WhiteMaterial::setColor(const glm::vec3& color){
```

```
7       _color = color;
8   }
9
10  void WhiteMaterial::apply(const RenderInfo &info){
11      _shader->begin();
12
13      _shader->setUniform("model",info.modelInfo.model);
14      _shader->setUniform("view",info.cameraInfo.view);
15      _shader->setUniform("projection",info.cameraInfo.project);
16
17      _shader->setUniform("color", _color);
18  }
```

♢ 由于`RenderInfo`里面已经汇集了全部信息，只需给对应的 Uniform 变量赋值即可

其着色器见 Listing 2.

（一）冯氏光照模型

冯氏模型(Phong Material)作为最经典的光照模型，可以非常简单的实现较为真实的光照

$$I = I_{\text{ambient}} + I_{\text{diffuse}} + I_{\text{specular}}$$

故只需在着色器中累计不同光的贡献即可。着色器实现如下

assets/shaders/phong/phong.vert vert

```
1   #version 410 core
2
3   layout(location = 0) in vec3 aPosition;
4   layout(location = 1) in vec2 aUv;
5   layout(location = 2) in vec3 aNormal;
6
7   out vec2 uv;
8   out vec3 normal;
9   out vec3 worldPosition;
10
11  out VS_OUT {
12    vec2 uv;
13    vec3 normal;
14    vec3 worldPosition;
15  } vs_out;
16
17  uniform mat4 model;
18  uniform mat4 view;
19  uniform mat4 projection;
20  uniform mat3 normalMatrix;
21
22  void main() {
23    vec4 transformPosition = model * vec4(aPosition, 1.0);
24
```

```
25    gl_Position = projection * view * transformPosition;
26    uv = aUv;
27    normal = normalMatrix * aNormal;
28    worldPosition = transformPosition.xyz;
29    vs_out.uv = aUv;
30    vs_out.normal = aNormal;
31    vs_out.worldPosition = worldPosition;
32  }
```

◇ 为了复用顶点着色器，为其设置两组输出变量。利用着色器可存在冗余输出的特性，其中`VS_OUT`为提供给几何着色器的输出，其它输出为提供给片元着色器的输出，从而实现这份顶点着色器编译后即可以只与片元着色器进行链接，也可以与几何着色器、片元着色器一同链接

◇ NormalMatrix 为法线矩阵，用于对法线进行变换，其大小为$(\text{Model}^{-1})^{\top}$. 由于 GPU 端求逆阵计算量大，并行性差(需使用高斯消元法转化为 TRSM 求解，或直接暴力求解)，且每个顶点都需计算，浪费 GPU 算力，故在 CPU 端计算一次后直接下发计算结果

assets/shaders/phong/phong.frag frag

```
1   #version 410 core
2
3   #define MAX_LIGHT_NUM 5
4
5   in vec2 uv;
6   in vec3 normal;
7   in vec3 worldPosition;
8
9   out vec4 FragColor;
10
11  struct DirectionalLight {
12    vec3 color;
13    vec3 direction;
14    float specularIntensity;
15  };
16
17  struct PointLight {
18    vec3 position;
19    vec3 color;
20    float specularIntensity;
21    float k2;
22    float k1;
23    float kc;
24  };
25
26  struct SpotLight {
27    vec3 position;
28    vec3 color;
```

```
29    vec3 direction;
30    float specularIntensity;
31    float inner;
32    float outer;
33  };
34
35  struct AmbientLight {
36    vec3 color;
37  };
38
39  uniform sampler2D diffuse;
40  uniform float shiness;
41
42  uniform vec3 cameraPosition;
43
44  uniform int numDirectionalLight;
45  uniform DirectionalLight directionalLight[MAX_LIGHT_NUM];
46
47  uniform int numPointLight;
48  uniform PointLight pointLight[MAX_LIGHT_NUM];
49
50  uniform int numSpotLight;
51  uniform SpotLight spotLight[MAX_LIGHT_NUM];
52
53  uniform AmbientLight ambientLight;
54
55  uniform sampler2D alphaMap;
56  uniform float useAlphaMap;
57
58  vec3 calDiffuse(vec3 lightColor, vec3 objectColor, vec3 lightDir, vec3
    normal) {
59    float diffuse = clamp(dot(-lightDir, normal), 0.0, 1.0);
60    return lightColor * diffuse * objectColor;
61  }
62
63  vec3 calSpecular(vec3 lightColor, vec3 lightDir, vec3 normal, vec3 viewDir,
64                   float intensity) {
65    vec3 lightReflect = normalize(reflect(lightDir, normal));
66    float specular = clamp(dot(lightReflect, -viewDir), 0.0, 1.0);
67    specular = pow(specular, shiness);
68    float back = step(0.0, dot(-lightDir, normal));
69    return lightColor * specular * intensity * back;
70  }
71
72  vec3 calDirectionalLight(DirectionalLight light, vec3 normal, vec3 viewDir) {
73    vec3 objectColor = texture(diffuse, uv).xyz;
74    vec3 lightDir = normalize(light.direction);
```

```
75    vec3 diffuseColor = calDiffuse(light.color, objectColor, lightDir, normal);
76    vec3 specularColor = calSpecular(light.color, lightDir, normal, viewDir,
77                                     light.specularIntensity);
78    return diffuseColor + specularColor;
79  }
80
81  vec3 calPointLight(PointLight light, vec3 normal, vec3 viewDir) {
82    vec3 objectColor = texture(diffuse, uv).xyz;
83    vec3 lightDir = normalize(worldPosition - light.position);
84    float dist = length(worldPosition - light.position);
85    float attenuation =
86        1.0 / (light.k2 * dist * dist + light.k1 * dist + light.kc);
87    vec3 diffuseColor = calDiffuse(light.color, objectColor, lightDir, normal);
88    vec3 specularColor = calSpecular(light.color, lightDir, normal, viewDir,
89                                     light.specularIntensity);
90    return (diffuseColor + specularColor) * attenuation;
91  }
92
93  vec3 calSpotLight(SpotLight light, vec3 normal, vec3 viewDir) {
94    vec3 objectColor = texture(diffuse, uv).xyz;
95    vec3 lightDir = normalize(worldPosition - light.position);
96    vec3 targetDir = normalize(light.direction);
97    float cGamma = dot(lightDir, targetDir);
98    float intensity =
99        clamp((cGamma - light.outer) / (light.inner - light.outer), 0.0, 1.0);
100   vec3 diffuseColor = calDiffuse(light.color, objectColor, lightDir, normal);
101   vec3 specularColor = calSpecular(light.color, lightDir, normal, viewDir,
102                                    light.specularIntensity);
103   return (diffuseColor + specularColor) * intensity;
104 }
105
106 vec3 calAmbientLight(AmbientLight light) {
107   return light.color * texture(diffuse, uv).xyz;
108 }
109
110 void main() {
111   vec3 normalN = normalize(normal);
112   vec3 viewDir = normalize(worldPosition - cameraPosition);
113
114   vec3 result = vec3(0.0);
115
116   for (int i = 0; i < numDirectionalLight; i++)
117     result += calDirectionalLight(directionalLight[i], normalN, viewDir);
118   for (int i = 0; i < numPointLight; i++)
119     result += calPointLight(pointLight[i], normalN, viewDir);
120   for (int i = 0; i < numSpotLight; i++)
121     result += calSpotLight(spotLight[i], normalN, viewDir);
```

```
122     result += calAmbientLight(ambientLight);
123
124     float alphaValue = mix(1.0, texture(alphaMap, uv).r, useAlphaMap);
125     FragColor = vec4(result, alphaValue);
126 }
```

◇ 各种光照计算公式如下

▷ 漫反射计算公式：$I_{\text{diffuse}} = k_d \cdot (L \cdot N) \cdot I_L$

使用从漫反射贴图采样得到的物体颜色作为漫反射系数k_d，L 是光线方向，N 是法线方向，使用光源颜色代替光源强度I_L

▷ 镜面反射计算公式：$I_{\text{specular}} = k_s \cdot (R \cdot V)^n \cdot I_L$

其中，k_s 是镜面反射系数，R 是反射光方向，V 是视线方向，n 是高光指数

▷ 环境光计算公式：$I_{\text{ambient}} = k_a \cdot I_a$

其中，k_a 是环境光系数，使用环境光颜色代替环境光强度I_a

▷ 平行光计算公式：$I = I_{\text{diffuse}} + I_{\text{specular}}$

▷ 点光源计算公式：$I = \frac{I_{\text{diffuse}} + I_{\text{specular}}}{k_c + k_l \cdot d + k_q \cdot d^2}$

其中，k_c 是常数衰减系数，k_1 是线性衰减系数，k_2 是二次衰减系数，d 是光源到点的距离

▷ 探照灯计算公式：$I = \frac{L \cdot D - \cos(\theta_{\text{outer}})}{\cos(\theta_{\text{inner}}) - \cos(\theta_{\text{outer}})} (I_{\text{diffuse}} + I_{\text{specular}})$

其中，D 是探照灯方向，θ_{inner} 是内锥角，θ_{outer} 是外锥角

◇ 为了实现着色器复用，在着色器中埋了透明材质的计算。默认情况下，`useAlphaMap` 恒为 0，只有在启用透明材质时才会参与混合

▷ 为了降低条件判断对 GPU 执行效率的影响，使用 `mix` 函数代替条件分支

使用类 `PhongMaterial` 继承抽象类 `Material`，实现如下

include/game/material/phongMaterial.h

```
1   class PhongMaterial : public Material {
2   public:
3     PhongMaterial();
4     PhongMaterial(const PhongMaterial &other) : Material(other) {
5       _diffuse = other._diffuse;
6       _shiness = other._shiness;
7     }
8     PhongMaterial(const Material &other) : Material(other) {
9       ResourceManager *resourceManager = Game::getGame()->getEngine()->
        getResourceManager();
10      _shader = resourceManager->loadShader(
11        "assets/shaders/phong/phong.vert", "assets/shaders/phong/phong.frag");
12      _diffuse = nullptr;
13      _shiness = 32.0f;
14    }
```

```

  PhongMaterial* clone() const override { return new PhongMaterial(*this); }
  void apply(const RenderInfo &info) override;

  void setDiffuse(Texture *diffuse) override { _diffuse = diffuse; }
  void setShiness(float shiness) { _shiness = shiness; }

private:
  Texture *_diffuse{nullptr};
  float _shiness{32.0f};
};
```

◇ 从 `Material` 拷贝构造出 `PhongMaterial` 时，需为其添加着色器

game/material/phongMaterial.cpp C++

```
PhongMaterial::PhongMaterial() {
  ResourceManager *resourceManager = Game::getGame()->getEngine()->getResourceManager();
  _shader = resourceManager->loadShader(
      "assets/shaders/phong/phong.vert", "assets/shaders/phong/phong.frag");
}

void PhongMaterial::apply(const RenderInfo &info) {
  _shader->begin();

  _shader->setUniform("model", info.modelInfo.model);
  _shader->setUniform("view", info.cameraInfo.view);
  _shader->setUniform("projection", info.cameraInfo.project);
  _shader->setUniform("normalMatrix", glm::mat3(glm::transpose(glm::inverse(info.modelInfo.model))));

  _diffuse->bind();
  _shader->setUniform("diffuse", (int)_diffuse->getUnit());
  _shader->setUniform("shiness", _shiness);
  _shader->setUniform("cameraPosition", info.cameraInfo.position);

  _shader->setUniform("numDirectionalLight", int(info.lightInfo.directionalLights.size()));
  for (int i = 0; i < info.lightInfo.directionalLights.size(); i++) {
    auto &dl = info.lightInfo.directionalLights[i];
    std::string prefix = "directionalLight[" + std::to_string(i) + "]";
    _shader->setUniform(prefix + ".color", dl.color);
    _shader->setUniform(prefix + ".direction", dl.direction);
    _shader->setUniform(prefix + ".specularIntensity", dl.specularIntensity);
  }

```

```
    _shader->setUniform("numPointLight",
29  int(info.lightInfo.pointLights.size()));
30  for (int i = 0; i < info.lightInfo.pointLights.size(); i++) {
31    auto &pl = info.lightInfo.pointLights[i];
32    std::string prefix = "pointLight[" + std::to_string(i) + "]";
33    _shader->setUniform(prefix + ".position", pl.position);
34    _shader->setUniform(prefix + ".color", pl.color);
35    _shader->setUniform(prefix + ".specularIntensity", pl.specularIntensity);
36    _shader->setUniform(prefix + ".k2", pl.k2);
37    _shader->setUniform(prefix + ".k1", pl.k1);
38    _shader->setUniform(prefix + ".kc", pl.kc);
39  }
40
41  _shader->setUniform("numSpotLight", int(info.lightInfo.spotLights.size()));
42  for (int i = 0; i < info.lightInfo.spotLights.size(); i++) {
43    auto &sl = info.lightInfo.spotLights[i];
44    std::string prefix = "spotLight[" + std::to_string(i) + "]";
45    _shader->setUniform(prefix + ".position", sl.position);
46    _shader->setUniform(prefix + ".color", sl.color);
47    _shader->setUniform(prefix + ".direction", sl.direction);
48    _shader->setUniform(prefix + ".specularIntensity", sl.specularIntensity);
49    _shader->setUniform(prefix + ".inner", sl.inner);
50    _shader->setUniform(prefix + ".outer", sl.outer);
51  }
52
    _shader->setUniform("ambientLight.color",
53  info.lightInfo.ambientLights[0].color);
54
55  _shader->setUniform("useAlphaMap", .0f);
56 }
```

由于冯氏模型实现了基本的光照模型，故后续部分材质直接在`PhongMaterial`上派生，继承关系如下

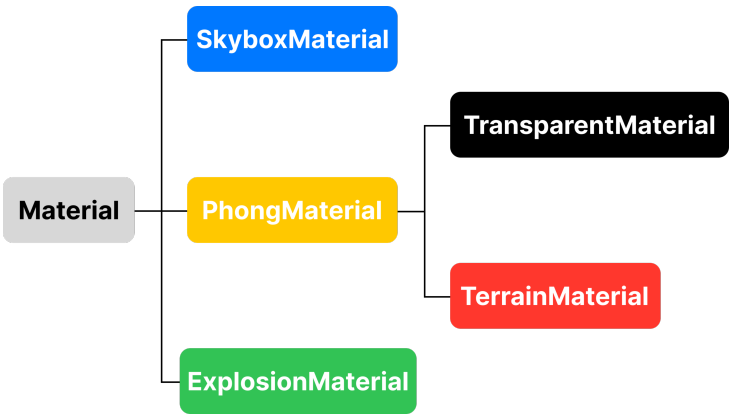


图 5：材质继承关系

凡是共用着色器的材质都可采用继承关系实现。

（二）天空盒

天空盒即为蒙在头顶、跟随相机移动的立方体，通过对 CubeMap 进行采样实现。

首先，需准备天空盒的顶点数据

```C++
Geometry *ResourceManager::createBoxGeometry(float size) {
  std::vector<glm::vec3> vertices = {glm::vec3(size / 2, size / 2, size / 2),
                                     glm::vec3(-size / 2, size / 2, size / 2),
                                     glm::vec3(-size / 2, size / 2, -size / 2),
                                     glm::vec3(size / 2, size / 2, -size / 2),

                                     glm::vec3(size / 2, -size / 2, size / 2),
                                     glm::vec3(-size / 2, -size / 2, size / 2),
                                     glm::vec3(-size / 2, -size / 2, -size /2),
                                     glm::vec3(size / 2, -size / 2, -size / 2),

                                     glm::vec3(size / 2, size / 2, size / 2),
                                     glm::vec3(-size / 2, size / 2, size / 2),
                                     glm::vec3(-size / 2, -size / 2, size / 2),
                                     glm::vec3(size / 2, -size / 2, size / 2),

                                     glm::vec3(size / 2, size / 2, -size / 2),
                                     glm::vec3(-size / 2, size / 2, -size / 2),
                                     glm::vec3(-size / 2, -size / 2, -size /2),
                                     glm::vec3(size / 2, -size / 2, -size / 2),

                                     glm::vec3(size / 2, size / 2, size / 2),
                                     glm::vec3(size / 2, size / 2, -size / 2),
                                     glm::vec3(size / 2, -size / 2, -size / 2),
                                     glm::vec3(size / 2, -size / 2, size / 2),

                                     glm::vec3(-size / 2, size / 2, size / 2),
                                     glm::vec3(-size / 2, size / 2, -size / 2),
                                     glm::vec3(-size / 2, -size / 2, -size /2),
                                     glm::vec3(-size / 2, -size / 2, size /2)};

  // six faces, each face has 4 vertices
  std::vector<glm::vec2> uvs = {
      glm::vec2(0.0f, 1.0f), glm::vec2(1.0f, 1.0f),
      glm::vec2(1.0f, 0.0f), glm::vec2(0.0f, 0.0f),

      glm::vec2(0.0f, 1.0f), glm::vec2(1.0f, 1.0f),
      glm::vec2(1.0f, 0.0f), glm::vec2(0.0f, 0.0f),

      glm::vec2(0.0f, 1.0f), glm::vec2(1.0f, 1.0f),
      glm::vec2(1.0f, 0.0f), glm::vec2(0.0f, 0.0f),

      glm::vec2(0.0f, 1.0f), glm::vec2(1.0f, 1.0f),
```

```
        glm::vec2(1.0f, 0.0f), glm::vec2(0.0f, 0.0f),

        glm::vec2(0.0f, 1.0f), glm::vec2(1.0f, 1.0f),
        glm::vec2(1.0f, 0.0f), glm::vec2(0.0f, 0.0f),

        glm::vec2(0.0f, 1.0f), glm::vec2(1.0f, 1.0f),
        glm::vec2(1.0f, 0.0f), glm::vec2(0.0f, 0.0f),
    };

    std::vector<glm::vec3> normals = {
        glm::vec3(0.0f, 1.0f, 0.0f),  glm::vec3(0.0f, 1.0f, 0.0f),
        glm::vec3(0.0f, 1.0f, 0.0f),  glm::vec3(0.0f, 1.0f, 0.0f),

        glm::vec3(0.0f, -1.0f, 0.0f), glm::vec3(0.0f, -1.0f, 0.0f),
        glm::vec3(0.0f, -1.0f, 0.0f), glm::vec3(0.0f, -1.0f, 0.0f),

        glm::vec3(0.0f, 0.0f, 1.0f),  glm::vec3(0.0f, 0.0f, 1.0f),
        glm::vec3(0.0f, 0.0f, 1.0f),  glm::vec3(0.0f, 0.0f, 1.0f),

        glm::vec3(0.0f, 0.0f, -1.0f), glm::vec3(0.0f, 0.0f, -1.0f),
        glm::vec3(0.0f, 0.0f, -1.0f), glm::vec3(0.0f, 0.0f, -1.0f),

        glm::vec3(1.0f, 0.0f, 0.0f),  glm::vec3(1.0f, 0.0f, 0.0f),
        glm::vec3(1.0f, 0.0f, 0.0f),  glm::vec3(1.0f, 0.0f, 0.0f),

        glm::vec3(-1.0f, 0.0f, 0.0f), glm::vec3(-1.0f, 0.0f, 0.0f),
        glm::vec3(-1.0f, 0.0f, 0.0f), glm::vec3(-1.0f, 0.0f, 0.0f)};

    std::vector<unsigned int> indices = {
        0,  1,  2,  0,  2,  3,  4,  6,  5,  4,  7,  6,  8,  9,  10, 8,  10, 11,
        12, 14, 13, 12, 15, 14, 16, 18, 17, 16, 19, 18, 20, 21, 22, 20, 22, 23};

    Geometry *_geometry = new Geometry(vertices, uvs, normals, indices);

    _geometryList.push_back(_geometry);

    return _geometry;
}
```

然后准备天空盒着色器

assets/shaders/skybox/skybox.vert vert

```
#version 410 core

layout(location = 0) in vec3 aPosition;
layout(location = 1) in vec2 aUv;
layout(location = 2) in vec3 aNormal;

```

```
7   out vec3 uvw;
8
9   uniform mat4 model;
10  uniform mat4 view;
11  uniform mat4 projection;
12
13  void main() {
14    gl_Position = projection * view * model * vec4(aPosition, 1.0);
15    gl_Position = gl_Position.xyww;
16    uvw = aPosition;
17  }
```

◇ 为了保证天空盒始终显示在画布最后方，使用`gl_Position = gl_Position.xyww`将天空盒深度变为1，即始终位于最后方；同时，设置默认深度测试模式为`GL_LEQUAL`，防止画布底色遮掩天空盒

assets/shaders/skybox/skybox.frag frag

```
1  #version 410 core
2  out vec4 FragColor;
3
4  in vec3 uvw;
5
6  uniform samplerCube sampler;
7
8  void main() { FragColor = texture(sampler, uvw); }
```

封装相应的材质类

include/game/material/skyboxMaterial.h C++

```
1   class SkyboxMaterial : public Material {
2   public:
3     SkyboxMaterial();
4     SkyboxMaterial(const SkyboxMaterial &other) : Material(other) {
5       _diffuse = other._diffuse;
6     }
7
8     SkyboxMaterial* clone() const override { return new SkyboxMaterial(*this); }
9     void apply(const RenderInfo &info) override;
10
11    void setDiffuse(Texture *diffuse) override { _diffuse = diffuse; }
12
13  private:
14    Texture *_diffuse{nullptr};
15  };
```

game/material/skyboxMaterial.cpp C++

```
1   SkyboxMaterial::SkyboxMaterial() {
2     ResourceManager* resourceManager = Game::getGame()->getEngine()->
      getResourceManager();
```

```
    _shader = resourceManager->loadShader(
        "assets/shaders/skybox/skybox.vert", "assets/shaders/skybox/
        skybox.frag");

    _depthWrite = false;
  }

  void SkyboxMaterial::apply(const RenderInfo &info) {
    _shader->begin();

    glm::mat4 model = info.modelInfo.model;
    model[0] = glm::vec4(1.0f, 0.0f, 0.0f, model[0].w);
    model[1] = glm::vec4(0.0f, 1.0f, 0.0f, model[1].w);
    model[2] = glm::vec4(0.0f, 0.0f, 1.0f, model[2].w);
    _shader->setUniform("model", model);
    _shader->setUniform("view",info.cameraInfo.view);
    _shader->setUniform("projection",info.cameraInfo.project);

    _diffuse->bind();
    _shader->setUniform("sampler", (int)_diffuse->getUnit());
  }
```

由于天空盒要跟随相机，在游戏实现层封装天空盒对象

include/game/entity/skybox.h C++

```
  class Skybox {
  public:
    Skybox(const std::vector<std::string> &paths);

    void bind(GameObject *camera);

    GameObject *getSkybox() const { return _skybox; }

  private:
    GameObject *_skybox{nullptr};
  };
```

game/entity/skybox.cpp C++

```
  Skybox::Skybox(const std::vector<std::string> &paths) {
    ResourceManager *resourceManager =
        Game::getGame()->getEngine()->getResourceManager();
    _skybox = new GameObject();
    Geometry *box = resourceManager->createBoxGeometry(1.0f);
    SkyboxMaterial *material = new SkyboxMaterial();
    material->setDiffuse(resourceManager->loadTexture(paths));
    _skybox->addComponent(std::make_shared<MeshComponent>(box, material));
  }

  void Skybox::bind(GameObject *camera) {
```

```
12    if (_skybox->getParent())
13      _skybox->getParent()->removeChild(_skybox);
14    camera->addChild(_skybox);
15  }
```

（三）透明材质

由于透明材质和冯氏材质共用着色器，直接继承 `PhongMaterial` 类即可，实现如下

include/game/material/transparentMaterial.h C++

```
1   class TransparentMaterial : public PhongMaterial {
2   public:
3     TransparentMaterial() : PhongMaterial() {
4       _blend = true;
5       _depthWrite = false;
6     };
7     TransparentMaterial(const TransparentMaterial &other) : PhongMaterial(other)
      {
8       _alphaMap = other._alphaMap;
9     }
10    TransparentMaterial(const PhongMaterial &other) : PhongMaterial(other) {
11      _blend = true;
12      _depthWrite = false;
13      _alphaMap = nullptr;
14    }
15    TransparentMaterial(const Material &other) : PhongMaterial(other) {
16      _blend = true;
17      _depthWrite = false;
18      _alphaMap = nullptr;
19    }
20
21    ~TransparentMaterial() override {}
22
23    TransparentMaterial *clone() const override {
24      return new TransparentMaterial(*this);
25    }
26
27    void apply(const RenderInfo &info) override;
28
29    void setDiffuse(Texture *diffuse) override {
30      PhongMaterial::setDiffuse(diffuse);
31    }
32    void setAlphaMap(Texture *alphaMap) {
33      _alphaMap = alphaMap;
34      _alphaMap->setUnit(1);
35    }
```

```
36
37  private:
38    Texture *_alphaMap{nullptr};
39  };
```

♢ 将透明蒙版放在 1 号纹理单元

♢ 透明材质需开启颜色混合功能，并关闭深度检测，从而实现“透明”

```C++
1  void TransparentMaterial::apply(const RenderInfo &info) {
2    PhongMaterial::apply(info);
3
4     _alphaMap->bind();
5    _shader->setUniform("alphaMap", (int)_alphaMap->getUnit());
6    _shader->setUniform("useAlphaMap", 1.0f);
7  }
```

♢ 直接调用 `PhongMaterial` 的 `apply` 方法，然后再开启透明功能，并绑定透明蒙版

（四）地形

参考[8] 中实现，使用镶嵌着色器生成地形，实现如下

assets/shaders/terrain/terrain.vert

```vert
1  #version 410 core
2
3  layout (location = 0) in vec3 aPos;
4  layout (location = 1) in vec2 aTex;
5
6  out vec2 TexCoord;
7
8  void main() {
9    gl_Position = vec4(aPos, 1.0);
10   TexCoord = aTex;
11 }
```

assets/shaders/terrain/terrain.tesc

```tesc
1  #version 410 core
2
3  layout(vertices = 4) out;
4
5  in vec2 TexCoord[];
6  out vec2 TextureCoord[];
7
8  uniform mat4 model;
9  uniform mat4 view;
10 uniform mat4 projection;
11
12 void main() {
```

```
13    gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
14    TextureCoord[gl_InvocationID] = TexCoord[gl_InvocationID];
15
16    if (gl_InvocationID == 0) {
17      const int MIN_TESS_LEVEL = 4;
18      const int MAX_TESS_LEVEL = 64;
19      const float MIN_DISTANCE = 20;
20      const float MAX_DISTANCE = 800;
21
22      vec4 eyeSpacePos00 = view * model * gl_in[0].gl_Position;
23      vec4 eyeSpacePos01 = view * model * gl_in[1].gl_Position;
24      vec4 eyeSpacePos10 = view * model * gl_in[2].gl_Position;
25      vec4 eyeSpacePos11 = view * model * gl_in[3].gl_Position;
26
27      float distance00 = clamp((abs(eyeSpacePos00.z) - MIN_DISTANCE) /
        (MAX_DISTANCE - MIN_DISTANCE), 0.0, 1.0);
28      float distance01 = clamp((abs(eyeSpacePos01.z) - MIN_DISTANCE) /
        (MAX_DISTANCE - MIN_DISTANCE), 0.0, 1.0);
29      float distance10 = clamp((abs(eyeSpacePos10.z) - MIN_DISTANCE) /
        (MAX_DISTANCE - MIN_DISTANCE), 0.0, 1.0);
30      float distance11 = clamp((abs(eyeSpacePos11.z) - MIN_DISTANCE) /
        (MAX_DISTANCE - MIN_DISTANCE), 0.0, 1.0);
31
32      float tessLevel0 = mix(MAX_TESS_LEVEL, MIN_TESS_LEVEL, min(distance10,
        distance00));
33      float tessLevel1 = mix(MAX_TESS_LEVEL, MIN_TESS_LEVEL, min(distance00,
        distance01));
34      float tessLevel2 = mix(MAX_TESS_LEVEL, MIN_TESS_LEVEL, min(distance01,
        distance11));
35      float tessLevel3 = mix(MAX_TESS_LEVEL, MIN_TESS_LEVEL, min(distance11,
        distance10));
36
37      gl_TessLevelOuter[0] = tessLevel0;
38      gl_TessLevelOuter[1] = tessLevel1;
39      gl_TessLevelOuter[2] = tessLevel2;
40      gl_TessLevelOuter[3] = tessLevel3;
41
42      gl_TessLevelInner[0] = max(tessLevel1, tessLevel3);
43      gl_TessLevelInner[1] = max(tessLevel0, tessLevel2);
44    }
45  }
```

◇ **通过计算补丁到相机的距离，动态确定细分级数，从而实现类似 Mipmap 的效果，在减轻 GPU 压力的同时避免过采样问题**

assets/shaders/terrain/terrain.tese tese

```
#version 410 core

layout(quads, fractional_odd_spacing, ccw) in;

uniform sampler2D heightMap;
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
uniform mat3 normalMatrix; // unused

in vec2 TextureCoord[];

out vec2 uv;
out vec3 normal;
out vec3 worldPosition;

void main() {
  float u = gl_TessCoord.x;
  float v = gl_TessCoord.y;

  vec2 t00 = TextureCoord[0];
  vec2 t01 = TextureCoord[1];
  vec2 t10 = TextureCoord[2];
  vec2 t11 = TextureCoord[3];
  vec2 t0 = (t01 - t00) * u + t00;
  vec2 t1 = (t11 - t10) * u + t10;
  vec2 texCoord = (t1 - t0) * v + t0;
  float height = texture(heightMap, texCoord).y * 64.0 - 32.0;

  vec4 p00 = gl_in[0].gl_Position;
  vec4 p01 = gl_in[1].gl_Position;
  vec4 p10 = gl_in[2].gl_Position;
  vec4 p11 = gl_in[3].gl_Position;
  vec4 uVec = p01 - p00;
  vec4 vVec = p10 - p00;
  vec4 normalN = normalize(vec4(cross(vVec.xyz, uVec.xyz), 0));
  vec4 p0 = (p01 - p00) * u + p00;
  vec4 p1 = (p11 - p10) * u + p10;
  vec4 p = (p1 - p0) * v + p0;
  p += normalN * height;

  uv = texCoord;
  normal = normalN.xyz;
  worldPosition = (model * p).xyz;
  gl_Position = projection * view * model * p;
}
```

♢ 手动对纹理坐标、高度进行双线性插值，得到当前采样点的属性数据

♢ 通过对高度图进行采样，生成地形；由于高度图像素值范围$[0,1]$，需对计算结果进行放缩处理

♢ 为了能与冯氏模型片元着色器进行对接，声明 Uniform 变量 `normalMatrix`，防止 `SetUniform` 时出错

为了使地形响应光照，直接将这些着色器与冯氏模型片元着色器 Listing 3 进行链接，故实现材质时可直接继承 `PhongMaterial`

include/game/material/terrainMaterial.h

```cpp
class TerrainMaterial : public PhongMaterial {
public:
  TerrainMaterial() : PhongMaterial() { prepareShader(); }
  TerrainMaterial(const TerrainMaterial &other) : PhongMaterial(other) {
    _heightMap = other._heightMap;
  }
  TerrainMaterial(const PhongMaterial &other) : PhongMaterial(other) {
    prepareShader();
    _heightMap = nullptr;
  }
  TerrainMaterial(const Material &other) : PhongMaterial(other) {
    prepareShader();
    _heightMap = nullptr;
  }

  ~TerrainMaterial() override {}

  TerrainMaterial *clone() const override { return new
  TerrainMaterial(*this); }
  void apply(const RenderInfo &info) override;

  void setDiffuse(Texture *diffuse) override {
    PhongMaterial::setDiffuse(diffuse);
  }
  void setHeightMap(Texture *heightMap) {
    _heightMap = heightMap;
    _heightMap->setUnit(1);
  }

private:
  Texture *_heightMap{nullptr};

  void prepareShader();
};
```

♢ 将高度图放在 1 号纹理单元

game/material/terrainMaterial.cpp

```cpp
void TerrainMaterial::apply(const RenderInfo &info) {
  PhongMaterial::apply(info);

  _heightMap->bind();
  _shader->setUniform("heightMap", (int)_heightMap->getUnit());
}

void TerrainMaterial::prepareShader() {
  ResourceManager *resourceManager =
      Game::getGame()->getEngine()->getResourceManager();
  _shader = resourceManager->loadShader("assets/shaders/terrain/terrain.vert",
                                        "assets/shaders/terrain/terrain.tesc",
                                        "assets/shaders/terrain/terrain.tese",
                                        "assets/shaders/phong/phong.frag");
}
```

由于地形初始化时，需根据分辨率准备网格数据，在游戏实现层对地形对象进行封装

include/game/entity/terrain.h

```cpp
class Terrain {
public:
  Terrain(float width, float height, int rez, int repeat, TerrainMaterial
  *material);
  ~Terrain();

  GameObject *getTerrain() const { return _terrain; }

private:
  GameObject *_terrain{nullptr};
};
```

game/entity/terrain.cpp

```cpp
Terrain::Terrain(float width, float height, int rez, int repeat,
                 TerrainMaterial *material) {
  ResourceManager *resourceManager =
      Game::getGame()->getEngine()->getResourceManager();

  std::vector<glm::vec3> vertices;
  std::vector<glm::vec2> uvs;
  for (int i = 0; i <= rez - 1; i++) {
    for (int j = 0; j <= rez - 1; j++) {
      vertices.push_back(glm::vec3(-width / 2.0f + width * i / (float)rez,
      0.0f, -height / 2.0f + height * j / (float)rez));
      uvs.push_back(glm::vec2(repeat * i / (float)rez, repeat * j /
      (float)rez));
      vertices.push_back(glm::vec3(-width / 2.0f + width * (i + 1) /
      (float)rez, 0.0f, -height / 2.0f + height * j / (float)rez));
```

```
        uvs.push_back(glm::vec2(repeat * (i + 1) / (float)rez, repeat * j /
13      (float)rez));
14      vertices.push_back(
15          glm::vec3(-width / 2.0f + width * i / (float)rez, 0.0f,
16                    -height / 2.0f + height * (j + 1) / (float)rez));
        uvs.push_back(glm::vec2(repeat * i / (float)rez, repeat * (j + 1) /
17      (float)rez));
18      vertices.push_back(
19          glm::vec3(-width / 2.0f + width * (i + 1) / (float)rez, 0.0f,
20                    -height / 2.0f + height * (j + 1) / (float)rez));
        uvs.push_back(glm::vec2(repeat * (i + 1) / (float)rez, repeat * (j +
21      1) / (float)rez));
22    }
23  }
24
25  _terrain = new GameObject();
26  Geometry *terrainGeometry =
27      resourceManager->loadGeometry(std::move(vertices), std::move(uvs));
28  std::shared_ptr<MeshComponent> meshComp =
29      std::make_shared<MeshComponent>(terrainGeometry, material);
30  std::shared_ptr<TerrainComponent> terrainComp =
31      std::make_shared<TerrainComponent>(rez);
32  _terrain->addComponent(meshComp);
33  _terrain->addComponent(terrainComp);
34 }
35
36 Terrain::~Terrain() {
37   if (_terrain) {
38     delete _terrain;
39     _terrain = nullptr;
40   }
41 }
```

◇ **根据地形分辨率，从** $\left[-\frac{\text{width}}{2}, \frac{\text{width}}{2}\right] \times \left[-\frac{\text{height}}{2}, \frac{\text{height}}{2}\right]$ **生成地形**

◇ **当地形过大时，若仍选取四角 uv 为** $(0,0),(0,1),(1,0),(1,1)$ **会导致地形变得过于平坦失真，通过修改纹理坐标实现地形重复密铺**

（五）爆炸效果

使用几何着色器实现爆炸效果，参考[9]，实现如下

shaders/effect/explode.geom geom

```
1  #version 410 core
2
3  layout(triangles) in;
4  layout(triangle_strip, max_vertices = 3) out;
```

```
5
6   in VS_OUT {
7     vec2 uv;
8     vec3 normal;
9     vec3 worldPosition;
10  } gs_in[];
11
12  out vec2 uv;
13  out vec3 normal;
14  out vec3 worldPosition;
15
16  uniform float time;
17
18  vec3 getNormal() {
19    vec3 a = vec3(gl_in[0].gl_Position) - vec3(gl_in[1].gl_Position);
20    vec3 b = vec3(gl_in[2].gl_Position) - vec3(gl_in[1].gl_Position);
21    return normalize(cross(a, b));
22  }
23
24  vec4 explode(vec4 position, vec3 normal) {
25    vec3 direction = normal * time;
26    return position + vec4(direction, 0.0);
27  }
28
29  void main() {
30    vec3 normalN = getNormal();
31
32    gl_Position = explode(gl_in[0].gl_Position, normalN);
33    uv = gs_in[0].uv;
34    normal = gs_in[0].normal;
35    worldPosition = gs_in[0].worldPosition;
36    EmitVertex();
37    gl_Position = explode(gl_in[1].gl_Position, normalN);
38    uv = gs_in[1].uv;
39    normal = gs_in[1].normal;
40    worldPosition = gs_in[1].worldPosition;
41    EmitVertex();
42    gl_Position = explode(gl_in[2].gl_Position, normalN);
43    uv = gs_in[2].uv;
44    normal = gs_in[2].normal;
45    worldPosition = gs_in[2].worldPosition;
46    EmitVertex();
47    EndPrimitive();
48  }
```

◇ **通过叉积得到每个三角面片的法向，让每个三角面的顶点朝着法向量方向偏移，从而实现爆炸效果**

顶点和片元着色器都复用冯氏模型。材质继承`PhongMaterial`，实现如下

include/game/material/explosionMaterial.h

```cpp
class ExplosionMaterial : public PhongMaterial {
public:
  ExplosionMaterial() : PhongMaterial() { prepareShader(); }
  ExplosionMaterial(const ExplosionMaterial &other) : PhongMaterial(other) {
    _time = other._time;
  }
  ExplosionMaterial(const PhongMaterial &other) : PhongMaterial(other) {
    prepareShader();
    _time = .0f;
  }
  ExplosionMaterial(const Material &other) : PhongMaterial(other) {
    prepareShader();
    _time = .0f;
  }
  ~ExplosionMaterial() override {}

  ExplosionMaterial *clone() const override {
    return new ExplosionMaterial(*this);
  }
  void apply(const RenderInfo &info) override;

  void setTime(float time) { _time = time; }

private:
  float _time{0.0f};

  void prepareShader();
};
```

game/material/explosionMaterial.cpp

```cpp
void ExplosionMaterial::apply(const RenderInfo &info) {
  PhongMaterial::apply(info);

  _shader->setUniform("time", _time);
}

void ExplosionMaterial::prepareShader() {
  ResourceManager *resourceManager =
      Game::getGame()->getEngine()->getResourceManager();
  _shader = resourceManager->loadShader("assets/shaders/phong/phong.vert",
                                        "assets/shaders/effect/explode.geom",
                                        "assets/shaders/phong/phong.frag");
}
```

这样，只需设置着色器的时间变量便可实现爆炸效果。

第四部分

物理系统

物理系统是游戏引擎中的重要组成部分，用于模拟物体的运动和碰撞效果。每次物理系统调用 tick 时，根据其在物理世界中的位置、速度、加速度等参数来更新物体的位置和速度，从而模拟物体的运动。物理系统还可以检测物体之间的碰撞，当两个物体发生碰撞时，会产生相互作用力，从而改变物体的运动状态。

一、物理组件

物理组件是游戏中用于模拟物体运动和碰撞效果的组件，其主要功能是设置物体的位置、速度、碰撞箱、物理 id(0xffffffff 为无效值)等参数。本项目中，物理组件实现了 Box，Sphere 和 Capsule 三种碰撞箱以及自定义碰撞箱的初始化。物理组件声明如下：

include/runtime/framework/component/physics/rigidBody.h C++

```
1  class RigidBodyComponent : public Component {
2  public:
3    RigidBodyComponent(JPH::EMotionType motionType, JPH::ObjectLayer layer,
     JPH::ShapeRefC shape)
4        : _motionType(motionType), _layer(layer), _shape(shape) {};
5    RigidBodyComponent(JPH::EMotionType motionType, JPH::ObjectLayer layer,
     uint32_t x, uint32_t y, uint32_t z, float density);
6    RigidBodyComponent(JPH::EMotionType motionType, JPH::ObjectLayer layer,
     float radius, float density);
7    RigidBodyComponent(JPH::EMotionType motionType, JPH::ObjectLayer layer,
     float radius, float halfHeight, float density);
8
9    ~RigidBodyComponent() override{Engine::getEngine()→getPhysicalSystem()→
     unregisterComponent(_id);};
10
11   uint32_t getId() const { return _id; }
12   float getDensity() const {return _density;}
13   float getLiftCoefficient() const {return _liftCoefficient;}
14   JPH::EMotionType getMotionType() const { return _motionType; }
15   JPH::ObjectLayer getLayer() const { return _layer; }
16   JPH::ShapeRefC getShape() const { return _shape; }
```

```
17    glm::vec3 getForce() const { return _force; }
18    glm::vec3 getTorque() const { return _torque; }
19    glm::vec3 getLinearVelocity() const { return _linearVelocity; }
20    float getMaxLinearVelocity() const { return _maxLinearVelocity; }
21    float getMaxAngularVelocity() const { return _maxAngularVelocity; }
22    bool isCollide() const { return _isCollide; }
23
24    void setId(uint32_t id) { _id = id; }
25    void setDensity(float density) { _density = density; }
26    void setLiftCoefficient(float liftCoefficient) { _liftCoefficient =
      liftCoefficient; }
27    void setMotionType(JPH::EMotionType type) { _motionType = type; }
28    void setLayer(JPH::ObjectLayer layer) { this->_layer = layer; }
29    void setShape(JPH::Shape* shape) { this->_shape = shape; }
30    void setForce(glm::vec3 force) { _force = force; }
31    void setTorque(glm::vec3 torque) { _torque = torque; }
32    void setLinearVelocity(glm::vec3 linearVelocity) { _linearVelocity =
      linearVelocity; }
33    void setMaxLinearVelocity(float maxLinearVelocity) { _maxLinearVelocity =
      maxLinearVelocity; }
34    void setMaxAngularVelocity(float maxAngularVelocity) { _maxAngularVelocity =
      maxAngularVelocity; }
35    void setCollide(bool isCollide) { _isCollide = isCollide; }
36
37  private:
38    uint32_t _id {0xffffffff};
39
40    float _density {1.0f};
41    float _liftCoefficient {0.5f};
42    JPH::EMotionType _motionType = JPH::EMotionType::Static;
43    JPH::ObjectLayer _layer       = Layers::STATIC;
44    JPH::ShapeRefC _shape {nullptr};
45
46    glm::vec3 _force {0.0f, 0.0f, 0.0f};
47    glm::vec3 _torque {0.0f, 0.0f, 0.0f};
48    glm::vec3 _linearVelocity {0.0f, 0.0f, 0.0f};
49    float _maxLinearVelocity {500.0f};
50    float _maxAngularVelocity {0.25f * JPH::JPH_PI};
51
52    bool _isCollide {false};
53  };
```

♢ 使用 Jolt Physics 库[1] 作为力学模拟的后端

其初始化时，可以通过传入不同的参数来创建不同形状的碰撞箱，这些函数的实现如下：

runtime/framework/component/physics/rigidBody.cpp

```cpp
// for boxs
RigidBodyComponent::RigidBodyComponent(JPH::EMotionType motionType,
                                       JPH::ObjectLayer layer, uint32_t x,
                                       uint32_t y, uint32_t z, float density)
    : _motionType(motionType), _layer(layer), _density(density) {
  JPH::BoxShapeSettings bodyShapeSettings(JPH::Vec3(0.5 * x, 0.5 * y, 0.5 *
  z));
  bodyShapeSettings.SetDensity(_density);

  JPH::ShapeSettings::ShapeResult bodyShapeResult =
  bodyShapeSettings.Create();
  _shape = bodyShapeResult.Get();
}

// for spheres
RigidBodyComponent::RigidBodyComponent(JPH::EMotionType motionType,
                                       JPH::ObjectLayer layer, float radius,
                                       float density)
    : _motionType(motionType), _layer(layer), _density(density) {
  JPH::SphereShapeSettings bodyShapeSettings(radius);
  bodyShapeSettings.SetDensity(_density);

  JPH::ShapeSettings::ShapeResult bodyShapeResult =
  bodyShapeSettings.Create();
  _shape = bodyShapeResult.Get();
}

// for capsules
RigidBodyComponent::RigidBodyComponent(JPH::EMotionType motionType,
                                       JPH::ObjectLayer layer, float radius,
                                       float halfHeight, float density)
    : _motionType(motionType), _layer(layer), _density(density) {
  JPH::CapsuleShapeSettings bodyShapeSettings(halfHeight, radius);
  bodyShapeSettings.SetDensity(_density);

  JPH::ShapeSettings::ShapeResult bodyShapeResult =
  bodyShapeSettings.Create();
  _shape = bodyShapeResult.Get();
};
```

二、碰撞检测

（一）碰撞箱

物理系统中，碰撞箱是一个用于检测碰撞的简单几何体，例如立方体、球体、胶囊体等，其形状和大小可以根据需要进行调整。

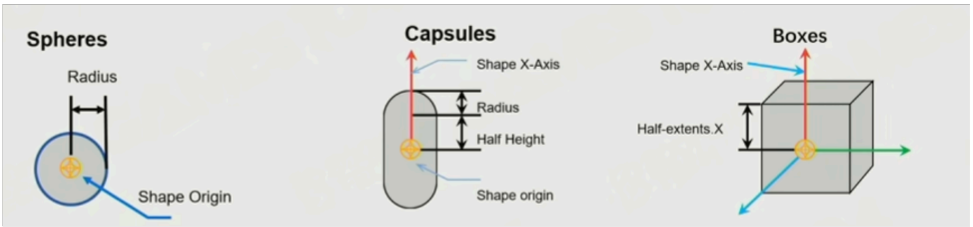


图 6：碰撞箱，图片来源[4]

（二）碰撞检测算法

Board-Phase

此阶段主要是为了减少碰撞检测的次数，将所有的碰撞箱分为不同的组，每个组中的碰撞箱都是不会发生碰撞的，这样就可以减少碰撞检测的次数。接着，通过 AABB 算法对每个组中的碰撞箱进行碰撞检测。AABB 算法是一种简单的碰撞检测算法，以碰撞箱的包围盒为基础，通过检测包围盒之间是否相交来判断碰撞箱是否发生碰撞。

以二维碰撞箱为例，碰撞箱的包围盒是一个矩形，其左上角坐标为(x_1, y_1)，右下角坐标为(x_2, y_2). 如果两个碰撞箱的包围盒相交，则说明两个碰撞箱可能发生碰撞，接下来进入 Narrow-Phase 阶段进行精确的碰撞检测。

Narrow-Phase

在 Narrow-Phase 阶段，对 Board-Phase 阶段中可能发生碰撞的碰撞箱进行精确的碰撞检测。对于不同形状的碰撞箱，可以采用不同的碰撞检测算法，例如，对于球体和球体之间的碰撞检测，可以采用球体之间的距离公式来判断是否发生碰撞；对于球体和立方体之间的碰撞检测，可以采用球心到立方体表面的距离来判断是否发生碰撞。这些碰撞的检测算法较为直白，运算量较大但可以获得精细的参数。

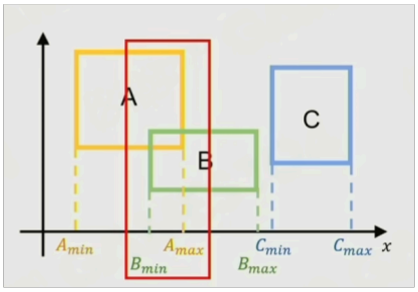


图 7：碰撞检测算法，图片来源[4]

三、力学模拟

（一）刚体

在物理学中，刚体是指形状不变的物体，即物体的形状和大小在运动过程中不会发生变化。在游戏中，刚体是一个用于模拟物体运动的基本单位，其运动状态由位置、速度、角度和角速度等参数来描述。

（二）质心定理

质心定理是刚体运动学中的一个重要定理，刚体的运动可以看作是质心的平动和绕质心的转动的叠加。

$$\boldsymbol{v}_i = \boldsymbol{v}_{\mathrm{CM}} + \boldsymbol{\omega} \times \boldsymbol{r}_i$$

在游戏中，质心定理可以用来简化刚体的运动模拟，将刚体的运动分解为质心的平动和绕质心的转动两个部分，从而简化运动方程的求解。

（三）物体运动定律

物体运动状态的改变是由外力引起的，一个力作用在物体上时，首先会对物体位置的运动产生作用，其等效于对物体的质心施加一个力，从而改变物体的位置；其次会对物体的角度运动产生作用，其等效于对物体的质心施加一个力矩，从而改变物体的角度。

$$\boldsymbol{F}_{\mathrm{ext}} = M\frac{d^2\boldsymbol{R}_{\mathrm{CM}}}{dt^2}$$

$$\boldsymbol{M}_{\mathrm{ext}} = \frac{d(\boldsymbol{I}\boldsymbol{\omega})}{dt}$$

结合质心定理，可以得到物体的运动方程，从而模拟物体的运动。

（四）碰撞

在游戏中，当两个物体发生碰撞时，两个物体之间会产生相互作用力，从而改变物体的运动状态。相互作用力的大小和方向可以根据碰撞箱的形状和碰撞箱之间的相对位置来计算，从而模拟物体的碰撞效果。

四、Jolt Physics[1]

JoltPhysics 是一款基于 C++的物理引擎，其主要功能是模拟物体的运动和碰撞效果。JoltPhysics 提供了丰富的 API 接口，可以方便地创建物体、设置物体的运动参数、检测碰撞等。JoltPhysics 还提供了丰富的碰撞检测算法和碰撞效果，可以满足不同类型游戏的需求。

（一）物理系统初始化

在使用 JoltPhysics 时，需要初始化物理系统，包括创建物理世界、设置重力加速度等。此时注意，初始化时需要实现`BPLayerInterfaceImpl`类，其中需要自定义`Layers`层和`BroadPhaseLayers`层(`Layers`用于精确控制物体之间的碰撞规则，允许物体之间的碰撞选择性发生，`BroadPhaseLayers`主要用于碰撞检测的初步筛选，帮助快速排除不可能发生碰撞的物体，从而加速模拟过程)，以及实现`MyObjectVsBroadPhaseLayerFilter`类和`MyObjectLayerPairFilter`类，用于控制物体之间的可碰撞关系。

其声明与实现如下：

include/runtime/framework/system/jolt/utils.h

```C++
namespace Layers {
static constexpr uint8_t STATIC = 0;
static constexpr uint8_t MOVING = 1;
static constexpr uint8_t NUM_LAYERS = 2;
}; // namespace Layers

namespace BroadPhaseLayers {
static constexpr JPH::BroadPhaseLayer NON_MOVING(0);
static constexpr JPH::BroadPhaseLayer MOVING(1);
static constexpr uint32_t NUM_LAYERS(2);
}; // namespace BroadPhaseLayers

class BPLayerInterfaceImpl final : public JPH::BroadPhaseLayerInterface {
public:
  BPLayerInterfaceImpl();

  uint32_t GetNumBroadPhaseLayers() const override {
    return BroadPhaseLayers::NUM_LAYERS;
  }

  const char *
  GetBroadPhaseLayerName(JPH::BroadPhaseLayer inLayer) const override;

  JPH::BroadPhaseLayer
  GetBroadPhaseLayer(JPH::ObjectLayer inLayer) const override {
    return _object_to_broad_phase[inLayer];
```

```
27    }
28
29  private:
30    JPH::BroadPhaseLayer _object_to_broad_phase[Layers::NUM_LAYERS];
31  };
32
33  class MyObjectVsBroadPhaseLayerFilter
34      : public JPH::ObjectVsBroadPhaseLayerFilter {
35  public:
36    bool ShouldCollide(JPH::ObjectLayer inLayer1,
37                       JPH::BroadPhaseLayer inLayer2) const override;
38  };
39
40  class MyObjectLayerPairFilter : public JPH::ObjectLayerPairFilter {
41  public:
42    bool ShouldCollide(JPH::ObjectLayer inLayer1,
43                       JPH::ObjectLayer inLayer2) const override;
44  };
```

runtime/framework/system/jolt/utils.cpp C++

```
1   BPLayerInterfaceImpl::BPLayerInterfaceImpl() {
2     // Create a mapping table from object to broad phase layer
3     _object_to_broad_phase[Layers::STATIC] = BroadPhaseLayers::NON_MOVING;
4     _object_to_broad_phase[Layers::MOVING] = BroadPhaseLayers::MOVING;
5   }
6
7   const char *BPLayerInterfaceImpl::GetBroadPhaseLayerName(
8       JPH::BroadPhaseLayer inLayer) const {
9     switch ((JPH::BroadPhaseLayer::Type)inLayer) {
10    case (JPH::BroadPhaseLayer::Type)BroadPhaseLayers::NON_MOVING:
11      return "NON_MOVING";
12    case (JPH::BroadPhaseLayer::Type)BroadPhaseLayers::MOVING:
13      return "MOVING";
14    default:
15      return "INVALID";
16    }
17  }
18
19  bool MyObjectVsBroadPhaseLayerFilter::ShouldCollide(
20      JPH::ObjectLayer inLayer1, JPH::BroadPhaseLayer inLayer2) const {
21    switch (inLayer1) {
22    case Layers::STATIC:
23      return inLayer2 == BroadPhaseLayers::MOVING;
24    case Layers::MOVING:
25      return inLayer2 == BroadPhaseLayers::NON_MOVING ||
26             inLayer2 == BroadPhaseLayers::MOVING;
27    default:
28      return false;
```

```
  }
}

bool MyObjectLayerPairFilter::ShouldCollide(JPH::ObjectLayer inObject1,
                                            JPH::ObjectLayer inObject2) const
                                            {
  switch (inObject1) {
  case Layers::STATIC:
    return inObject2 == Layers::MOVING;
  case Layers::MOVING:
    return inObject2 == Layers::STATIC || inObject2 == Layers::MOVING;
  default:
    return false;
  }
}
```

实现基本类后，其初始化代码如下：

runtime/framework/system/physicalSystem.cpp C++

```cpp
bool PhysicalSystem::init(PhysicsInfo info) {
  // init physics config
  _config = info;

  // init jolt physics
  JPH::RegisterDefaultAllocator();
  JPH::Factory::sInstance = new JPH::Factory();
  JPH::RegisterTypes();

  _joltPhysics._joltPhysicsSystem = new JPH::PhysicsSystem();
  _joltPhysics._joltBroadPhaseLayerInterface = new BPLayerInterfaceImpl();

  _joltPhysics._joltJobSystem = new JPH::JobSystemThreadPool(
      _config.maxJobCount, _config.maxBarrierCount,
      static_cast<int>(_config.maxConcurrentJobCount));

  // 16M temp memory
  _joltPhysics._tempAllocator = new JPH::TempAllocatorImpl(16 * 1024 * 1024);

  _joltPhysics._objectVsBroadPhaseFilter =
      new MyObjectVsBroadPhaseLayerFilter();
  _joltPhysics._objectLayerPairFilter = new MyObjectLayerPairFilter();
  _joltPhysics._joltPhysicsSystem->Init(
      _config.maxBodyCount, _config.bodyMutexCount, _config.maxBodyPairs,
      _config.maxContactConstraints,
      *(_joltPhysics._joltBroadPhaseLayerInterface),
      *(_joltPhysics._objectVsBroadPhaseFilter),
      *(_joltPhysics._objectLayerPairFilter));

```

```
    _joltPhysics._contactListener = new MyContactListener();
    _joltPhysics._joltPhysicsSystem->SetContactListener(
        _joltPhysics._contactListener);

    _joltPhysics._joltPhysicsSystem->SetGravity(
        {_config.gravity.x, _config.gravity.y, _config.gravity.z});

    return true;
}
```

物理系统初始化完成后，就可以开始模拟物体的运动和碰撞效果了。

（二）物体放置与移除

物体放置是指在物理世界中放置物体，并设置对应的位置、速度、角度等参数，同时注意设置物体的碰撞箱、质量和惯性等参数。在 JoltPhysics 中，通过 `BodyInterface` 设置，其代码如下：

runtime/framework/system/physicalSystem.cpp C++

```
uint32_t PhysicalSystem::createRigidBody(GameObject *object) {
    std::shared_ptr<RigidBodyComponent> _rigidBody =
        object->getComponent<RigidBodyComponent>();
    std::shared_ptr<TransformComponent> _transform =
        object->getComponent<TransformComponent>();

    glm::vec3 position = _transform->getPositionWorld();
    glm::vec3 scale = {1.0f, 1.0f, 1.0f};
    glm::vec3 angle = _transform->getAngle();

    JPH::ShapeRefC shape = _rigidBody->getShape();
    JPH::EMotionType motionType = _rigidBody->getMotionType();
    JPH::ObjectLayer layer = _rigidBody->getLayer();

    JPH::BodyCreationSettings settings(
        shape, toVec3(position), toQuat(toRotation(angle)), motionType, layer);
    settings.mApplyGyroscopicForce = true;
    settings.mMaxLinearVelocity = 10000.0;
    settings.mLinearDamping = 0.1;
    settings.mAngularDamping = 0.1;

    JPH::BodyInterface &bodyInterface =
        _joltPhysics._joltPhysicsSystem->GetBodyInterface();
    JPH::Body *body = bodyInterface.CreateBody(settings);
    bodyInterface.AddBody(body->GetID(), JPH::EActivation::Activate);
    _joltPhysics._contactListener->registerBody(
        body->GetID().GetIndexAndSequenceNumber());

    uint32_t id = body->GetID().GetIndexAndSequenceNumber();
```

```
30    _rigidBody->setId(id);
31
32    return id;
33  }
```

物体的移除也是类似的，通过`BodyInterface`的`RemoveBody`函数来移除物体，其代码如下：

runtime/framework/system/physicalSystem.cpp

```cpp
void PhysicalSystem::destroyRigidBody(uint32_t ridigbodyId) {
  JPH::BodyInterface &bodyInterface =
      _joltPhysics._joltPhysicsSystem->GetBodyInterface();
  bodyInterface.RemoveBody(JPH::BodyID(ridigbodyId));
  // bodyInterface.DestroyBody(JPH::BodyID(ridigbodyId));
}
```

♢ 在最新版本的Jolt中，`RemoveBody`后无法立即调用`DestroyBody`，由于此处未做精细的线程管理，跳过了`DestroyBody`的调用，可能会造成一定的内存泄漏和资源占用

（三）物理系统更新

每一帧中，调用物理系统的`tick`函数，根据物体的位置、速度、加速度等参数来更新物体的位置和速度，从而模拟物体的运动。在更新物理系统时，首先，根据物体的位置、速度、加速度等参数来更新物体的位置和速度，从而模拟物体的运动；其次，物理系统会检测物体之间的碰撞，当两个物体发生碰撞时，会产生相互作用力，从而改变物体的运动状态。

物理系统更新的代码如下：

runtime/framework/system/physicalSystem.cpp

```cpp
void PhysicalSystem::update(float dt) {
  // update physics
  _joltPhysics._joltPhysicsSystem->Update(dt);
}
```

物理系统更新完成后，就可以模拟出物体的运动和碰撞效果了。

第五部分

音效系统

音效是营造沉浸式游戏体验不可或缺的重要成分。通过声音的大小、频率变化，可以从侧面反映出游戏对象的位置与运行状态。

OpenAL[10] 是一款跨平台的 3D 音频 API，其接口设计上借鉴了 OpenGL 的实现标准，具备缓冲区、绑定等概念，非常适合为游戏提供音效系统开发支持。

整体音效系统以组件——系统形式实现，通过在游戏对象上挂载音频组件，在系统更新时检测音频播放状态，并对状态做出更新。

由于 OpenAL 提供了 OpenGL 类似的接口，同样为其实现检错

include/common/macro.h

```cpp
#ifdef DEBUG
#define AL_CALL(func)                                                          \
  func;                                                                        \
  checkALError(__FILE__, __LINE__, __func__);
#else
#define AL_CALL(func) func;
#endif
```

common/checkError.cpp

```cpp
void checkALError(const char *file, int line, const char *func) {
  ALenum err = alGetError();
  if (err != AL_NO_ERROR) {
    std::string errorStr = "AL_UNKNOWN_ERROR";
    switch (err) {
    case AL_INVALID_NAME:
      errorStr = "AL_INVALID_NAME";
      break;
    case AL_INVALID_ENUM:
      errorStr = "AL_INVALID_ENUM";
      break;
    case AL_INVALID_VALUE:
      errorStr = "AL_INVALID_VALUE";
      break;
    case AL_INVALID_OPERATION:
      errorStr = "AL_INVALID_OPERATION";
      break;
```

```
18      case AL_OUT_OF_MEMORY:
19        errorStr = "AL_OUT_OF_MEMORY";
20        break;
21      }
22      Err("OpenAL error: %s, file: %s, line: %d, func: %s", errorStr.c_str(),
23          file, line, func);
24    }
25  }
```

一、音频资源

由于 OpenAL 本身不具备音频载入功能，使用 stb/vorbis[6] 实现音频数据的载入。

首先定义音频资源，声明如下

include/runtime/resource/audio/audio.h C++

```
1   class Audio {
2     friend class ResourceManager;
3
4   public:
5     ALuint getAudioID() const { return _audioID; }
6
7   private:
8     ALuint _audioID{0};
9     ALuint _buffer{0};
10
11    Audio(const std::string &path);
12    ~Audio();
13  };
```

在构造函数中实现音频资源的载入

runtime/resource/audio/audio.cpp C++

```
1   Audio::Audio(const std::string &path) {
2     int channels = 0;
3     int sampleRate = 0;
4     short *data = nullptr;
5     int samples =
6         stb_vorbis_decode_filename(path.c_str(), &channels, &sampleRate, &data);
7     if (samples < 0) {
8       Log("Failed to load audio: %s", path.c_str());
9     } else {
10      AL_CALL(alGenBuffers(1, &_buffer));
11      ALenum format = (channels == 1) ? AL_FORMAT_MONO16 : AL_FORMAT_STEREO16;
12      AL_CALL(alBufferData(_buffer, format, data,
13                           samples * channels * sizeof(short), sampleRate));
14      AL_CALL(alGenSources(1, &_audioID));
15      AL_CALL(alSourcei(_audioID, AL_BUFFER, _buffer));
```

```
16      free(data);
17    }
18  }
```

二、音频组件

音频组件声明如下

include/runtime/framework/component/audio/audio.h C++

```
1   class AudioComponent : public Component {
2   public:
3     enum class Mode { SINGLE, REPEAT_SINGEL, REPEAT_LIST, INVALID };
4
5     AudioComponent() = default;
6     ~AudioComponent() override {}
7
8     void start() { _isPlaying = true; }
9     void stop() { _isPlaying = false; }
10    void append(Audio *audio);
11
12    void tick();
13
14    Audio *getAudio() const { return _playlist[_current]; }
15
16    void setMode(Mode mode) { _mode = mode; }
17
18  private:
19    Mode _mode{Mode::INVALID};
20
21    std::vector<Audio *> _playlist{};
22    int _current{0};
23    bool _isPlaying{false};
24
25    int getNext();
26  };
```

◇ 在组件内部维护音频播放列表

◇ 提供 3 种音频播放模式：单曲，单曲循环，列表循环

其更新逻辑实现如下，根据音频播放模式决定下一首播放的曲目

runtime/framework/component/audio/audio.cpp C++

```
1   void AudioComponent::tick() {
2     ALint state;
3     Audio *audio = _playlist[_current];
4     alGetSourcei(audio->getAudioID(), AL_SOURCE_STATE, &state);
5     if ((state == AL_PLAYING && _isPlaying) ||
```

```
6         (state ≠ AL_PLAYING && !_isPlaying))
7       return;
8     if (!_isPlaying)
9       alSourcePause(audio→getAudioID());
10    else if (state == AL_PAUSED || state == AL_INITIAL)
11      alSourcePlay(audio→getAudioID());
12    else if (_mode == Mode::SINGLE)
13      _isPlaying = false;
14    else
15      alSourcePlay(_playlist[getNext()]→getAudioID());
16  }
17
18  int AudioComponent::getNext() {
19    if (_mode == Mode::REPEAT_SINGEL)
20      return _current;
21    if (_mode == Mode::REPEAT_LIST)
22      return (_current + 1) % _playlist.size();
23    Log("Unknown audio mode.");
24    return 0;
25  }
```

◇ 无需每次都进行音频播放状态更新，通过维护期望播放状态`_isPlaying`，通过与实际播放状态`state`进行比较，若一致则不进行更新，否则判断是否播放下一曲，决定下一曲是哪一曲

三、系统实现

音效系统声明如下

include/runtime/framework/system/audioSystem.h C++

```
1   class AudioSystem {
2   public:
3     AudioSystem() = default;
4     ~AudioSystem() = default;
5
6
7     bool init();
8     void dispatch(GameObject *object);
9     void tick();
10
11  private:
12    std::vector<GameObject *> _audios;
13    GameObject* _listener;
14
15    void clear();
16  };
```

♢ OpenAL 提供 Listener 概念，用于设定用户位置；为保证音画一致，在分发时，直接将当前的主相机设为 Listener

音效系统的初始化实现如下

runtime/framework/system/audioSystem.cpp

```cpp
bool AudioSystem::init() {
  ALCdevice *device = alcOpenDevice(NULL);
  ALCcontext *context = alcCreateContext(device, NULL);
  alcMakeContextCurrent(context);
  return context != nullptr;
}
```

♢ 先对音频设备进行初始化，再对其进行绑定

♢ 处于简化，此处并未考虑音频播放设备的切换

更新逻辑实现如下

runtime/framework/system/audioSystem.cpp

```cpp
void AudioSystem::dispatch(GameObject *object) {
  if (object->getComponent<AudioComponent>())
    _audios.push_back(object);
  if (object->getComponent<CameraComponent>() &&
      object->getComponent<CameraComponent>()->isMain())
    _listener = object;
}

void AudioSystem::tick() {
  for (auto audio : _audios) {
    std::shared_ptr<AudioComponent> audioComp =
        audio->getComponent<AudioComponent>();
    audioComp->tick();
    glm::vec3 pos =
        audio->getComponent<TransformComponent>()->getPositionWorld();
    alSource3f(audioComp->getAudio()->getAudioID(), AL_POSITION, pos.x, pos.y,
               pos.z);
    std::shared_ptr<RigidBodyComponent> rigidbodyComp =
        audio->getComponent<RigidBodyComponent>();
    if (rigidbodyComp) {
      glm::vec3 v = rigidbodyComp->getLinearVelocity();
      alSource3f(audioComp->getAudio()->getAudioID(), AL_VELOCITY, v.x, v.y,
                 v.z);
    } else {
      alSource3f(audioComp->getAudio()->getAudioID(), AL_VELOCITY, 0, 0, 0);
    }
  }
  glm::vec3 pos =
      _listener->getComponent<TransformComponent>()->getPositionWorld();
  alListener3f(AL_POSITION, pos.x, pos.y, pos.z);
  std::shared_ptr<RigidBodyComponent> rigidbodyComp =
```

```
32        _listener->getComponent<RigidBodyComponent>();
33    if (rigidbodyComp) {
34      glm::vec3 v = rigidbodyComp->getLinearVelocity();
35      alListener3f(AL_VELOCITY, v.x, v.y, v.z);
36    } else {
37      alListener3f(AL_VELOCITY, 0.0f, 0.0f, 0.0f);
38    }
39    clear();
40  }
```

◇ **分发阶段，音效系统捕获所有具有音频组件的游戏对象，同时寻找主相机，并设置为聆听者**

◇ **更新阶段，对音频组件进行更新，并设置其位置、速度等参数**

第六部分

游戏实现

一、相机控制

相机组件仅仅实现了获取 ViewMatrix 的功能，并没有实现相机对事件的响应。为了实现相机的控制，对相机进行封装，认为相机是一个具有相机组件的游戏对象，并通过向游戏注册回调的方式实现相机的控制。

为了实现多视角的观察，实现自由相机和第一人称相机两种不同的相机类型。

游戏实现层相机声明如下

include/game/entity/camera.h

```cpp
class Camera {
public:
  enum class Type { Free, FirstPersion, Invalid };

  Camera(std::shared_ptr<CameraComponent> cameraComp, Type type);
  ~Camera();

  void enable();
  void disable();

  GameObject *getCamera() const { return _camera; }

  void setSpeed(float speed) { _speed = speed; }

private:
  GameObject *_camera{nullptr};

  std::shared_ptr<TransformComponent> _transformComp;
  std::shared_ptr<CameraComponent> _cameraComp;

  Type _type{Type::Invalid};

  std::map<int, bool> _keyMap;
  float _speed{1.0f};

  bool _leftMouseDown, _rightMouseDown, _middleMouseDown;
```

```
27    float _currentX, _currentY;
28    float _sensitivity{0.2f};
29
30    float _pitch{0.0f};
31    float _yaw{0.0f};
32
33    void onKey(int key, int action, int mods);
34    void onMouse(int button, int action, float xpos, float ypos);
35    void onCursor(double xpos, double ypos);
36
37    void tick();
38    void updatePosition();
39    void pitch(float angle);
40    void yaw(float angle);
41  };
```

◇ **使用 Key map 方式处理按键，在事件回调中，仅维护当前按下的按键状态；在更新逻辑中，读取 Key map，响应键盘事件**

相机实现如下

game/entity/camera.cpp C++

```
1   Camera::Camera(std::shared_ptr<CameraComponent> cameraComp, Type type)
2       : _type(type) {
3     _camera = new GameObject();
4     _transformComp = _camera->getComponent<TransformComponent>();
5     _cameraComp = cameraComp;
6     _camera->addComponent(_cameraComp);
7
8     Game::getGame()->bind(
9         Game::KeyBoard{},
10        [this](int key, int action, int mods) { onKey(key, action, mods); });
11    Game::getGame()->bind(Game::Mouse{}, [this](int button, int action,
12                                                int mods) {
13      double xpos = 0, ypos = 0;
14      Game::getGame()->getEngine()->getWindowSystem()->getCursorPosition(xpos,
15                                                                         ypos);
16      onMouse(button, action, xpos, ypos);
17    });
18    Game::getGame()->bind(Game::Cursor{}, [this](double xpos, double ypos) {
19      onCursor(xpos, ypos);
20      _currentX = xpos, _currentY = ypos;
21    });
22    _camera->setTick([this]() { tick(); });
23  }
24
25  Camera::~Camera() {
26    if (_camera) {
```

```
27      delete _camera;
28      _camera = nullptr;
29    }
30  }
31
32  void Camera::enable() {
33    if (_camera)
34      _camera->getComponent<CameraComponent>()->pick();
35  }
36
37  void Camera::disable() {
38    if (_camera)
39      _camera->getComponent<CameraComponent>()->put();
40  }
41
42  void Camera::onKey(int key, int action, int mods) {
43    _keyMap[key] = action == GLFW_PRESS  || action == GLFW_REPEAT;
44  }
45
46  void Camera::onCursor(double xpos, double ypos) {
47    if (_type == Type::Free || _type == Type::FirstPersion) {
48      float dx = (xpos - _currentX) * _sensitivity,
49            dy = (ypos - _currentY) * _sensitivity;
50      pitch(-dy);
51      yaw(-dx);
52    }
53  }
54
55  void Camera::onMouse(int button, int action, float xpos, float ypos) {
56    bool pressed = action == GLFW_PRESS;
57    if (pressed)
58      _currentX = xpos, _currentY = ypos;
59    switch (button) {
60    case GLFW_MOUSE_BUTTON_LEFT:
61      _leftMouseDown = pressed;
62      break;
63    case GLFW_MOUSE_BUTTON_RIGHT:
64      _rightMouseDown = pressed;
65      break;
66    case GLFW_MOUSE_BUTTON_MIDDLE:
67      _middleMouseDown = pressed;
68      break;
69    }
70  }
71
72  void Camera::updatePosition() {
73    if (_type == Type::Free) {
```

```
74      glm::vec3 direction{0.0f};
75      glm::vec3 front =
76          glm::cross(_cameraComp->getUpVec(), _cameraComp->getRightVec());
77      glm::vec3 right = _cameraComp->getRightVec();
78      if (_keyMap[GLFW_KEY_UP])
79        direction += front;
80      if (_keyMap[GLFW_KEY_DOWN])
81        direction -= front;
82      if (_keyMap[GLFW_KEY_LEFT])
83        direction -= right;
84      if (_keyMap[GLFW_KEY_RIGHT])
85        direction += right;
86      if (glm::length(direction)) {
87        direction = glm::normalize(direction);
88        _transformComp->setPositionLocal(_transformComp->getPositionLocal() +
89                                         direction * _speed);
90      }
91    }
92  }
93
94  void Camera::tick() {
95    updatePosition();
96    if (_type == Type::FirstPersion) {
97      if (_camera && _camera->getParent()) {
98        glm::mat4 model = _camera->getParent()->getComponent<TransformComponent>()->getModel();
99        glm::vec3 pos, euler, scl;
100       Utils::decompose(model, pos, euler, scl);
101       _camera->getComponent<TransformComponent>()->setAngle(euler);
102     }
103   }
104 }
105
106 void Camera::pitch(float angle) {
107   if (_type == Type::Free) {
108     if (_pitch + angle > 89.0f || _pitch + angle < -89.0f)
109       return;
110     _pitch += angle;
111     glm::mat4 rotation =
112         glm::rotate(glm::mat4(glm::one<float>()), glm::radians(angle),
113                     _cameraComp->getRightVec());
114     _cameraComp->setUpVec(glm::vec4(_cameraComp->getUpVec(), 0.0f) * rotation);
115   }
116   if(_type == Type::FirstPersion) {
117     glm::vec3 position, eulerAngle, scale;
```

```
    Utils::decompose(_camera->getComponent<TransformComponent>()->
118 getParentModel(), position, eulerAngle, scale);
    if (_pitch + angle > fmin(70.0f + eulerAngle.x, 89.0f) || _pitch + angle
119 < fmax(-30.0f + eulerAngle.x, -89.0f))
120   return;
121 _pitch += angle;
122 glm::mat4 rotation =
123     glm::rotate(glm::mat4(glm::one<float>()), glm::radians(angle),
124                 _cameraComp->getRightVec());
    _cameraComp->setUpVec(glm::vec4(_cameraComp->getUpVec(), 0.0f) *
125 rotation);
126 }
127 }
128
129 void Camera::yaw(float angle) {
130 if (_type == Type::Free) {
131 glm::mat4 rotation =
132     glm::rotate(glm::mat4(glm::one<float>()), glm::radians(angle),
133                 glm::vec3(0.0f, 1.0f, 0.0f));
    _cameraComp->setUpVec(glm::vec4(_cameraComp->getUpVec(), 0.0f) *
134 rotation);
135 _cameraComp->setRightVec(glm::vec4(_cameraComp->getRightVec(), 0.0f) *
136                          rotation);
137 }
138 if (_type == Type::FirstPersion) {
139 if(_yaw + angle > 70.0f || _yaw + angle < -70.0f)
140   return;
141 _yaw += angle;
142 glm::mat4 rotation =
143     glm::rotate(glm::mat4(glm::one<float>()), glm::radians(angle),
144                 glm::vec3(0.0f, 1.0f, 0.0f));
    _cameraComp->setUpVec(glm::vec4(_cameraComp->getUpVec(), 0.0f) *
145 rotation);
146 _cameraComp->setRightVec(glm::vec4(_cameraComp->getRightVec(), 0.0f) *
147                          rotation);
148 }
149 }
```

◇ 由于相机组件分为透视和正交投影相机，为了降低设计复杂度，在构造相机对象时。要求从外部传入相机组件

◇ 根据不同相机的特点，对相机系的基(up, right)进行更新；为了防止视角倒转，对俯仰角范围进行限制

◇ 为了防止因视角限制，在飞机机头拉高时看不到玻璃，对于第一人称相机，动态获取父亲的ModelView, 并分解得到俯仰角，实时调整视角限制

二、模型导入

为了丰富游戏体验，需要对外部的 3D 模型导入提供支持。由于模型文件中 数据多以层次结构存在，正好与游戏对象的树状结构形成对应。

同时，为了实现模型文件中变换矩阵的表示与 Transform 组件中的变换表达之间的转化，需实现工具函数对矩阵进行分解。

由于实现了透明材质，利用材质的继承关系和 C++的多态特性，可以根据模型每个部分含有的贴图情况，动态适配适合的材质。

模型导入实现如下

game/utils/utils.cpp C++

```cpp
namespace Utils {
GameObject *loadModel(const std::string &path, Material &material) {
  GameObject *root = new GameObject();

  Assimp::Importer importer;
  const aiScene *scene =
      importer.ReadFile(path, aiProcess_Triangulate | aiProcess_GenNormals);
  if (!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE ||
      !scene->mRootNode) {
    Err("Assimp error: %s", importer.GetErrorString());
    return nullptr;
  }

  std::size_t lastIndex = path.find_last_of('/');
  auto rootPath = path.substr(0, lastIndex + 1);

  parse(scene->mRootNode, scene, root, rootPath, material);

  return root;
}

void parse(aiNode *ainode, const aiScene *scene, GameObject *parent,
           const std::string &rootPath, Material &material) {
  GameObject *gameObject = new GameObject();

  glm::mat4 localMatrix = getMat4f(ainode->mTransformation);
  glm::vec3 position, enlerAngle, scale;
  decompose(localMatrix, position, enlerAngle, scale);

  gameObject->getComponent<TransformComponent>()->setPositionLocal(position);
  gameObject->getComponent<TransformComponent>()->setAngle(enlerAngle);
  gameObject->getComponent<TransformComponent>()->setScale(scale);

  parent->addChild(gameObject);

```

```
36    for (std::size_t i = 0; i < ainode->mNumMeshes; i++) {
37      aiMesh *aimesh = scene->mMeshes[ainode->mMeshes[i]];
38      Material *childMat = material.clone();
39      auto mesh = parseMesh(aimesh, scene, rootPath, *childMat);
40      auto _object = new GameObject();
41      _object->addComponent(mesh);
42      gameObject->addChild(_object);
43    }
44
45    for (std::size_t i = 0; i < ainode->mNumChildren; i++) {
46      Material *childMat = material.clone();
47      parse(ainode->mChildren[i], scene, gameObject, rootPath, *childMat);
48    }
49  }
50
51  std::shared_ptr<MeshComponent> parseMesh(aiMesh *aimesh, const aiScene
    *scene,
52                                           const std::string &rootPath,
53                                           Material &material) {
54    ResourceManager *resourceManager =
55        Game::getGame()->getEngine()->getResourceManager();
56
57    std::vector<glm::vec3> vertices;
58    std::vector<glm::vec2> uvs;
59    std::vector<glm::vec3> normals;
60    std::vector<unsigned int> indices;
61
62    for (std::size_t i = 0; i < aimesh->mNumVertices; i++) {
63      vertices.push_back(glm::vec3(aimesh->mVertices[i].x, aimesh->
      mVertices[i].y,
64                                   aimesh->mVertices[i].z));
65      normals.push_back(glm::vec3(aimesh->mNormals[i].x, aimesh->mNormals[i].y,
66                                  aimesh->mNormals[i].z));
67
68      if (aimesh->mTextureCoords[0]) {
69        uvs.push_back(glm::vec2(aimesh->mTextureCoords[0][i].x,
70                                aimesh->mTextureCoords[0][i].y));
71      } else {
72        uvs.push_back(glm::vec2(0.0f, 0.0f));
73      }
74    }
75
76    for (std::size_t i = 0; i < aimesh->mNumFaces; i++) {
77      aiFace face = aimesh->mFaces[i];
78      for (std::size_t j = 0; j < face.mNumIndices; j++) {
79        indices.push_back(face.mIndices[j]);
80      }
```

```
81      }
82
83      auto geometry =
84          resourceManager->loadGeometry(vertices, uvs, normals, indices);
85
86      Material *finalMaterial = &material;
87
88      if (aimesh->mMaterialIndex >= 0) {
89        aiMaterial *aiMat = scene->mMaterials[aimesh->mMaterialIndex];
90        material.setDiffuse(parseTexture(aiMat, aiTextureType_DIFFUSE, scene,
          rootPath));
91        aiString aipath;
92        if (aiMat->Get(AI_MATKEY_TEXTURE(aiTextureType_OPACITY, 0), aipath) ==
          AI_SUCCESS) {
93          TransparentMaterial *transMaterial = nullptr;
94          PhongMaterial *phongMaterial = dynamic_cast<PhongMaterial *>
          (&material);
95          if (phongMaterial) {
96            transMaterial = new TransparentMaterial(*phongMaterial);
97          } else {
98            transMaterial = new TransparentMaterial(material);
99          }
100         transMaterial->setAlphaMap(parseTexture(aiMat, aiTextureType_OPACITY,
          scene, rootPath));
101         finalMaterial = transMaterial;
102       }
103     } else {
104       material.setDiffuse(
105           resourceManager->loadTexture("assets/textures/default.jpg"));
106     }
107     return std::make_shared<MeshComponent>(geometry, finalMaterial);
108   }
109
110   Texture *parseTexture(aiMaterial *aimat, aiTextureType type,
111                         const aiScene *scene, const std::string &rootPath) {
112     ResourceManager *resourceManager =
113         Game::getGame()->getEngine()->getResourceManager();
114     aiString aipath;
115     aimat->Get(AI_MATKEY_TEXTURE(type, 0), aipath);
116     if (aipath.length == 0) {
117       return resourceManager->loadTexture("assets/textures/default.jpg");
118     }
119     const aiTexture *aitexture = scene->GetEmbeddedTexture(aipath.C_Str());
120     if (aitexture) {
121       unsigned char *dataIn =
122           reinterpret_cast<unsigned char *>(aitexture->pcData);
123       uint32_t widthIn = aitexture->mWidth;
```

```
124       uint32_t heightIn = aitexture->mHeight;
125       return resourceManager->loadTexture(aipath.C_Str(), dataIn, widthIn,
126                                           heightIn);
127     } else {
128       std::string path = rootPath + aipath.C_Str();
129       return resourceManager->loadTexture(path);
130     }
131   }
132
133   glm::mat4 getMat4f(aiMatrix4x4 value) {
134     return glm::mat4(
135         value.a1, value.b1, value.c1, value.d1,
136         value.a2, value.b2, value.c2, value.d2,
137         value.a3, value.b3, value.c3, value.d3,
138         value.a4, value.b4, value.c4, value.d4
139     );
140   }
141
142   void decompose(glm::mat4 matrix, glm::vec3 &position, glm::vec3 &eulerAngle,
143                  glm::vec3 &scale) {
144     glm::quat quaternion;
145     glm::vec3 skew;
146     glm::vec4 perspective;
147
148     glm::decompose(matrix, scale, quaternion, position, skew, perspective);
149
150     glm::mat4 rotation = glm::toMat4(quaternion);
151     glm::extractEulerAngleXYZ(rotation, eulerAngle.x, eulerAngle.y,
      eulerAngle.z);
152
153     eulerAngle.x = glm::degrees(eulerAngle.x);
154     eulerAngle.y = glm::degrees(eulerAngle.y);
155     eulerAngle.z = glm::degrees(eulerAngle.z);
156   }
157   } // namespace Utils
```

◇ 使用 Assimp[11] 实现模型文件的读取

◇ 将模型读取为三角面，以适配渲染过程时的 DrawCall 类型

◇ 对于漫反射贴图缺失的部分，为其贴上默认贴图

三、飞行控制

飞机在飞行时，需要考虑到重力、空气阻力、升力等因素。首先，飞机自身有一个引擎的动力，推动飞机向机头方向运动，其大小固定，对飞机产生固定大小的加速度；同时，飞机的机翼产

生升力，该升力的方向朝向飞机模型的上方，其大小与飞机速度的平方成正比；此外，飞机的机身受到空气阻力的影响，空气阻力的方向与飞机的运动方向相反，大小也与速度成正比；最后，飞机可以通过控制升降舵和方向舵，给飞机自身施加转向力矩，控制飞机的滚转、俯仰和偏航。

玩家类（即飞机）的声明如下：

include/game/entity/player.h C++

```cpp
class Player {
public:
  using ExplosionFunc = std::function<float(float)>;

  Player(GameObject *player, const glm::vec3 &position, const glm::vec3 &angle,
         const float &scale, const glm::vec3 &boxSize, const float &mass,
         const float &liftCoefficient, const float &maxAcceleration,
         const float &maxAngularAcceleration);
  ~Player();

  GameObject *getPlayer() const { return _player; }

  void setLiftCoefficient(const float &liftCoefficient) {
    _liftCoefficient = liftCoefficient;
  }
  void setMaxAcceleration(const float &maxAcceleration) {
    _maxAcceleration = maxAcceleration;
  }
  void setMaxAngularAcceleration(const float &maxAngularAcceleration) {
    _maxAngularAcceleration = maxAngularAcceleration;
  }

private:
  GameObject *_player{nullptr};

  glm::vec3 _position;
  glm::vec3 _angle;
  float _scale;

  glm::vec3 _boxSize;
  float _mass;
  float _liftCoefficient;
  float _maxAcceleration;
  float _maxAngularAcceleration;

  void tick();
};
```

其受力模型的实现如下：

game/entity/player.cpp C++

```
void Player::tick() {
  if (_onExplode) {
    if (glfwGetTime() ≥ _explodeEnd) {
      _player→getParent()→removeChild(_player);
      return;
    }
    _explode(_player, _explosionFunc(glfwGetTime() - _explodeStart));
    return;
  }

  if(_canExplode && _player→getComponent<RigidBodyComponent>()->getLinearVelocity().y < -0.6f && _player->getComponent<RigidBodyComponent>()→isCollide()){
    explode();
    _canExplode = false;
  }

  glm::vec3 forward = _player→getComponent<TransformComponent>()→getForwardVec();
  glm::vec3 right = _player→getComponent<TransformComponent>()→getRightVec();
  glm::vec3 up = _player→getComponent<TransformComponent>()→getUpVec();

  // set force
  glm::vec3 engineForce = glm::vec3{0.0f};
  if (_keyMap[GLFW_KEY_SPACE]) {
    engineForce += forward * _maxAcceleration * _mass;
  }

  glm::vec3 liftForce = glm::vec3{0.0f};
  glm::vec3 linearVelocity =
      _player→getComponent<RigidBodyComponent>()→getLinearVelocity();
  liftForce = static_cast<float>(std::pow(_liftCoefficient, 2)) *
              glm::length(linearVelocity) * up;

  if(glm::length(liftForce) > glm::length(engineForce) * 0.5f)
    liftForce = glm::normalize(liftForce) * glm::length(engineForce) * 0.5f;

  _player→getComponent<RigidBodyComponent>()→setForce(engineForce +
                                                      liftForce);

  // set torque
  glm::vec3 torque = glm::vec3{0.0f};
  if (_keyMap[GLFW_KEY_W]) {
    torque +=
        right * _maxAngularAcceleration * (1.0f / 12.0f) *
```

```
            static_cast<float>(std::pow(_boxSize.y, 2) + std::pow(_boxSize.z, 2))
43          * _mass;
44      }
45      if (_keyMap[GLFW_KEY_S]) {
46        torque +=
47            -right * _maxAngularAcceleration * (1.0f / 12.0f) *
            static_cast<float>(std::pow(_boxSize.y, 2) + std::pow(_boxSize.z, 2))
48          * _mass;
49      }
50      if (_keyMap[GLFW_KEY_A]) {
51        torque +=
52            -forward * _maxAngularAcceleration * (1.0f / 12.0f) *
            static_cast<float>(std::pow(_boxSize.x, 2) + std::pow(_boxSize.y, 2))
53          * _mass;
54      }
55      if (_keyMap[GLFW_KEY_D]) {
56        torque +=
57            forward * _maxAngularAcceleration * (1.0f / 12.0f) *
            static_cast<float>(std::pow(_boxSize.x, 2) + std::pow(_boxSize.y, 2))
58          * _mass;
59      }
60
61      _player->getComponent<RigidBodyComponent>()->setTorque(torque);
62  }
```

四、场景构建

游戏场景十分简单，具有 6 个游戏对象：场景、自由相机、第一人称相机、玩家、天空盒、地形。

由于天空盒实现的原理是在相机外围蒙上立方体，故让天空盒始终作为主相机的子节点，并设置本地位移为 0，从而实现天空盒对相机的追随。

对于第一人称相机，为了沉浸式体验开飞机的感觉，需让主相机成为模型的子节点。

对于视角切换，将其注册为键盘回调即可。

实现如下

ame/game.cpp C++

```
1   void Game::setupScene() {
2     _scene = new GameObject();
3
4     Camera *camera = new Camera(
5         std::make_shared<CameraComponent>(
6             45.0f, _engine->getWindowSystem()->getAspect(), 0.1f, 10000.0f),
```

```
7           Camera::Type::Free);
8
9       camera->setSpeed(0.3f);
10
11      _scene->addComponent(
12          std::make_shared<LightComponent>(glm::vec3(1.0f, 1.0f, 1.0f)));
13
14      Skybox *skybox = new Skybox(
15          {"assets/textures/skybox/right.jpg", "assets/textures/skybox/left.jpg",
16           "assets/textures/skybox/top.jpg", "assets/textures/skybox/bottom.jpg",
             "assets/textures/skybox/back.jpg", "assets/textures/skybox/
17           front.jpg"});
18
19      skybox->bind(camera->getCamera());
20
21      _scene->addChild(camera->getCamera());
22
23      ExplosionMaterial *explosionMat = new ExplosionMaterial();
24      GameObject *model =
25          Utils::loadModel("assets/models/fighter/fighter.obj", *explosionMat);
26
27      Player *player = new Player(model, glm::vec3(0.0f, 3.0f, 0.0f),
28                                  glm::vec3(0.5f, 0.0f, 0.5f), 1.0f,
29                                  glm::vec3(1.0f), 1.0f, 0.7f, 10.0f, 0.02f);
30
31      GameObject* light = new GameObject();
        light->addComponent(std::make_shared<LightComponent>(glm::vec3(0.9f, 0.54f,
32      0.0f), glm::normalize(glm::vec3(0.0f, -0.2f, -1.0f)), 0.3, 45, 60));
        light->getComponent<TransformComponent>()->setPositionLocal(glm::vec3(0.0f,
33      -1.0f, 5.0f));
34      player->getPlayer()->addChild(light);
35
36      std::shared_ptr<AudioComponent> audioComp =
37          std::make_shared<AudioComponent>();
38      audioComp->append(
39          _engine->getResourceManager()->loadAudio("assets/audios/engine.ogg"));
40      audioComp->start();
41      audioComp->setMode(AudioComponent::Mode::REPEAT_SINGEL);
42      player->getPlayer()->addComponent(audioComp);
43
44      Camera *fCamera = new Camera(
45          std::make_shared<CameraComponent>(
46              60.0f, _engine->getWindowSystem()->getAspect(), 0.1f, 10000.0f),
47          Camera::Type::FirstPersion);
48      fCamera->disable();
49      fCamera->getCamera()->getComponent<TransformComponent>()->setPositionLocal(
50          glm::vec3(0.0f, 1.75f, 4.15f));
```

```
51      fCamera->getCamera()->getComponent<CameraComponent>()->setRightVec(
52          glm::vec3(-1.0f, 0.0f, 0.0f));
53      fCamera->getCamera()->getComponent<CameraComponent>()->setUpVec(
54          glm::normalize(glm::vec3(0.0f, 1.0f, 0.5f)));
55
56      player->getPlayer()->addChild(fCamera->getCamera());
57
58      Game::getGame()->bind(KeyBoard{}, [player, camera, fCamera,
59                                         skybox](int key, int action, int mods) {
60        if (key == GLFW_KEY_2 && action == GLFW_PRESS) {
61          camera->disable();
62          fCamera->enable();
63          skybox->bind(fCamera->getCamera());
64        }
65        if ((key == GLFW_KEY_1 && action == GLFW_PRESS)) {
66          fCamera->disable();
67          camera->enable();
68          skybox->bind(camera->getCamera());
69          camera->getCamera()->getComponent<TransformComponent>()-
            >setPositionLocal(
70              glm::inverse(camera->getCamera()
71                               ->getComponent<TransformComponent>()
72                               ->getParentModel()) *
73              glm::vec4((player->getPlayer()
74                             ->getComponent<TransformComponent>()
75                             ->getPositionWorld() +
76                         glm::vec3(0.0f, 5.0f, 0.0f)),
77                        1.0f));
78        }
79      });
80      skybox->getSkybox()->setTick([camera, player, skybox]() {
81        if (!player->getPlayer()->getParent())
82          skybox->bind(camera->getCamera());
83      });
84
85      TerrainMaterial *terrainMat = new TerrainMaterial();
86      terrainMat->setDiffuse(_engine->getResourceManager()->loadTexture(
87          "assets/textures/terrain/diffuse.jpg"));
88      terrainMat->setHeightMap(_engine->getResourceManager()->loadTexture(
89          "assets/textures/terrain/heightMap.png"));
90      Terrain *terrain = new Terrain(10000.0f, 10000.0f, 200, 100, terrainMat);
91      terrain->getTerrain()->addComponent(std::make_shared<RigidBodyComponent>(
            JPH::EMotionType::Static, Layers::STATIC, 10000.0f, 1.f, 10000.0f,
92          1.0f));
93
94      _scene->addChild(player->getPlayer());
95      _scene->addChild(terrain->getTerrain());
```

```
96
97      player->setExplosionFunc([](float x) -> float { return 10 * log(x + 1); });
98
99      _engine->setMainLoop([] {});
100 }
```

◇ 值得注意的是，在飞机爆炸销毁后，飞机将从场景的子节点列表中移除，从而不再参与渲染流程；而与此同时，作为飞机子节点的相机、作为相机子节点的天空盒也会被从游戏对象树中移除，即此时会陷入主相机缺失的场景。渲染系统会感知到主相机的缺失，并自动启用副相机，但天空盒并不会随之切换，因为游戏对象是否为天空盒对渲染系统而言不可见。故需为天空盒注册回调，动态检测飞机是否被移出游戏对象树，一旦发现飞机已被销毁，自动绑定到自由相机上，防止在爆炸瞬间天空盒出现渲染异常

Appendix

环境配置

使用 CMake 进行管理，项目结构如下

```
.
├── CMakeLists.txt
├── common
│   └── checkError.cpp
├── game
│   ├── entity
│   │   ├── camera.cpp
│   │   ├── player.cpp
│   │   ├── skybox.cpp
│   │   └── terrain.cpp
│   ├── game.cpp
│   ├── material
│   │   ├── explosionMaterial.cpp
│   │   ├── phongMaterial.cpp
│   │   ├── skyboxMaterial.cpp
│   │   ├── terrainMaterial.cpp
│   │   └── transparentMaterial.cpp
│   └── utils
│       └── utils.cpp
├── include
│   ├── common
│   │   ├── common.h
│   │   └── macro.h
│   ├── game
│   │   ├── entity
│   │   │   ├── camera.h
│   │   │   ├── player.h
│   │   │   ├── skybox.h
│   │   │   └── terrain.h
│   │   ├── game.h
│   │   ├── material
│   │   │   ├── explosionMaterial.h
│   │   │   ├── phongMaterial.h
│   │   │   ├── skyboxMaterial.h
│   │   │   ├── terrainMaterial.h
```

```
│   │   │   └── transparentMaterial.h
│   │   └── utils
│   │       └── utils.h
│   └── runtime
│       ├── engine.h
│       ├── framework
│       │   ├── component
│       │   │   ├── audio
│       │   │   │   └── audio.h
│       │   │   ├── camera
│       │   │   │   └── camera.h
│       │   │   ├── component.h
│       │   │   ├── light
│       │   │   │   └── light.h
│       │   │   ├── mesh
│       │   │   │   └── mesh.h
│       │   │   ├── physics
│       │   │   │   └── rigidBody.h
│       │   │   ├── terrain
│       │   │   │   └── terrain.h
│       │   │   └── transform
│       │   │       └── transform.h
│       │   ├── object
│       │   │   └── gameObject.h
│       │   └── system
│       │       ├── audioSystem.h
│       │       ├── jolt
│       │       │   └── utils.h
│       │       ├── physicalSystem.h
│       │       ├── renderSystem.h
│       │       └── windowSystem.h
│       └── resource
│           ├── audio
│           │   └── audio.h
│           ├── geometry
│           │   └── geometry.h
│           ├── material
│           │   ├── material.h
│           │   └── whiteMaterial.h
│           ├── resourceManager.h
│           ├── shader
│           │   └── shader.h
│           └── texture
│               └── texture.h
├── main.cpp
└── runtime
    ├── engine.cpp
```

```
    ├── framework
    │   ├── component
    │   │   ├── audio
    │   │   │   └── audio.cpp
    │   │   ├── camera
    │   │   │   └── camera.cpp
    │   │   ├── physics
    │   │   │   └── rigidBody.cpp
    │   │   └── transform
    │   │       └── transform.cpp
    │   ├── object
    │   │   └── gameObject.cpp
    │   └── system
    │       ├── audioSystem.cpp
    │       ├── jolt
    │       │   └── utils.cpp
    │       ├── physicalSystem.cpp
    │       ├── renderSystem.cpp
    │       └── windowSystem.cpp
    └── resource
        ├── audio
        │   └── audio.cpp
        ├── geometry
        │   └── geometry.cpp
        ├── material
        │   └── whiteMaterial.cpp
        ├── resourceManager.cpp
        ├── shader
        │   └── shader.cpp
        └── texture
            └── texture.cpp
```

`CMakeLists` 内容如下

CMakeLists.txt CMake

```cmake
cmake_minimum_required(VERSION 3.12)

project(BlackMamba)

set(CMAKE_CXX_STANDARD 17)
set(BUILD_SHARED_LIBS OFF)

file(GLOB ASSETS "./assets")
file(COPY ${ASSETS} DESTINATION ${CMAKE_BINARY_DIR})

add_definitions(-DDEBUG)
add_compile_options(-Wno-unknown-pragmas)
```

```
add_subdirectory(thirdParty)
add_subdirectory(src)
```

src/CMakeLists.txt CMake

```
file(GLOB_RECURSE MAIN_SOURCES ./ *.cpp)

set(SRC_INCLUDE_DIR ${PROJECT_SOURCE_DIR}/src/include)

set(EXECUTABLE_OUTPUT_PATH ${PROJECT_BINARY_DIR})
add_executable(main ${MAIN_SOURCES})

target_include_directories(main PUBLIC
  ${SRC_INCLUDE_DIR}
  ${CMAKE_SOURCE_DIR}/thirdParty/glfw/include
  ${CMAKE_SOURCE_DIR}/thirdParty/glad/include
  ${CMAKE_SOURCE_DIR}/thirdParty/glm
  ${CMAKE_SOURCE_DIR}/thirdParty/imgui
  ${CMAKE_SOURCE_DIR}/thirdParty/stb
  ${CMAKE_SOURCE_DIR}/thirdParty/assimp/include
  ${CMAKE_SOURCE_DIR}/thirdParty/JoltPhysics
  ${CMAKE_SOURCE_DIR}/thirdParty/openal/include
)

target_link_libraries(main
  glfw
  glad
  glm
  imgui
  stb
  assimp
  Jolt
  OpenAL
)
```

thirdParty/CMakeLists.txt CMake

```
set(THIRD_PARTY_LIBRARY_PATH ${CMAKE_SOURCE_DIR}/thirdParty)

add_subdirectory(${THIRD_PARTY_LIBRARY_PATH}/glfw)
add_subdirectory(${THIRD_PARTY_LIBRARY_PATH}/glad)
add_subdirectory(${THIRD_PARTY_LIBRARY_PATH}/glm)
include(${THIRD_PARTY_LIBRARY_PATH}/imgui.cmake)
include(${THIRD_PARTY_LIBRARY_PATH}/stb.cmake)
add_subdirectory(${THIRD_PARTY_LIBRARY_PATH}/assimp)
add_subdirectory(${THIRD_PARTY_LIBRARY_PATH}/JoltPhysics/Build)

set(LIBTYPE STATIC)
add_subdirectory(${THIRD_PARTY_LIBRARY_PATH}/openal)
```

Bibliography

[1] J. Rouwe, "Jolt Physics: Collision Detection." [Online]. Available: https://jrouwe.github.io/JoltPhysics/index.html#collision-detection

[2] CrazyGames, " 3D Flight Simulator." [Online]. Available: https://www.crazygames.com/game/3d-flight-simulator

[3] CGTrader, "F104G Starfighter Fighter Jet." [Online]. Available: https://www.cgtrader.com/free-3d-models/aircraft/jet/f104g-starfighter-fighter-jet

[4] BoomingTech, " Games 104." [Online]. Available: https://games104.boomingtech.com/

[5] BoomingTech, " Piccolo." [Online]. Available: https://github.com/BoomingTech/Piccolo

[6] Nothings, "stb single-file public domain libraries for C/C++." [Online]. Available: https://github.com/nothings/stb

[7] U. Technologies, "Unity Real-Time Development Platform." [Online]. Available: https://unity.com/

[8] LearnOpenGL, "Tessellation." [Online]. Available: https://learnopengl.com/Guest-Articles/2021/Tessellation/Tessellation

[9] LearnOpenGL, "Geometry Shader." [Online]. Available: https://learnopengl.com/Advanced-OpenGL/Geometry-Shader

[10] Kcat, "OpenAL Soft." [Online]. Available: https://github.com/kcat/openal-soft

[11] Assimp, "Open Asset Import Library (assimp)." [Online]. Available: https://github.com/assimp/assimp