GAMES104 现代游戏引擎：从入门到实践

作业3 动画系统与物理系统

一、前言

经过了 4 节游戏动画与物理相关的课程之后，同学们应已对游戏的动画系统与物理系统有了初步的认识，对其大致流程和功能的理解也初具雏形。相信现在有的同学已经跃跃欲试，想在我们的开源小引擎 Pilot 中亲自实验实现动画与物理功能，做出富有表现力的动画物理效果。然而从零开始的动画与物理的实现复杂繁琐而且难以调试。

本次作业将为同学们做好所有前期准备，搭好框架，提供功能的入口，而让同学们实现具体的代码，希望带领同学们初探动画系统与物理系统。

而对于已经对动画与物理有一定经验的同学，也可以自行阅读理解 Pilot 中的相关代码。

二、本次作业具体内容

1. 在 Pilot 小引擎代码中找到 pilot/engine/source/runtime/function/animation/pose.cpp，找到blend函数补充代码，实现动画融合。
2. 在 Pilot 小引擎代码中找到 pilot/engine/source/runtime/function/animation/animation_FSM.cpp，找到update函数补充代码，实现机器人走跑跳状态机。
3. 在 Pilot 小引擎代码中找到 pilot/engine/source/runtime/function/controller/character_controller.cpp，找到move函数修改代码，利用SceneQuery实现具有相对真实物理表现的character controller：如可跳至平台上，跳起后空中碰到墙壁可以落回地面，前进碰到墙壁可以自动调整位移方向等。

三、代码框架说明

第一部分

1. 每个拥有骨骼动画的物体都拥有一个AnimationComponent，它存储了每一个骨骼动画所需的数据及动画当前的状态。
    - m_animation_res存储了骨骼及动画资源
    - m_animation_fsm是一个动画状态机实例
    - 每一帧AnimationComponent的tick函数将会被调用，它是该动画组件运算的入口
    - 外部系统可以通过updateSignal传递一些信号

```
REFLECTION_TYPE(AnimationComponent)
CLASS(AnimationComponent : public Component, WhiteListFields)
{
    REFLECTION_BODY(AnimationComponent)

public:
    AnimationComponent() = default;

    void postLoadResource(std::weak_ptr<GObject> parent_object) override;

    void tick(float delta_time) override;
```

```
    const AnimationResult& getResult() const;
    void                   animateBasicClip(float ratio, BasicClip*
basic_clip);
    void                   blend(float desired_ratio, BlendState*
blend_state);
    void                   blend1D(float desired_ratio, BlendSpace1D*
blend_state);
    template<typename T>
    void updateSignal(const std::string& key, const T& value);

protected:
    META(Enable)
    AnimationComponentRes m_animation_res;

    Skeleton m_skeleton;
    AnimationResult       m_animation_result;
    AnimationFSM          m_animation_fsm;
    json11::Json::object  m_signal;
    float                 m_ratio {0};
};
```

2. AnimationComponentRes

- m_skeleton_file_path指定骨骼资源
- m_clips指定所有的动画ClipBase资源

```
REFLECTION_TYPE(AnimationComponentRes)
CLASS(AnimationComponentRes, WhiteListFields)
{
    REFLECTION_BODY(AnimationComponentRes);

public:

    META(Enable)
    std::string m_skeleton_file_path;
    META(Enable)
    std::vector<Reflection::ReflectionPtr<ClipBase>> m_clips;
};
```

3. ClipBase

- ClipBase是一个拥有名字，长度的抽象类，它有BasicClip、BlendState、BlendSpace1D三个子类

```
REFLECTION_TYPE(ClipBase)
CLASS(ClipBase, Fields)
{
    REFLECTION_BODY(ClipBase);

public:
    std::string             m_name;
    virtual ~ClipBase() = default;
    virtual float getLength() const { return 0; }
};
```

- BasicClip是普通的动画clip资源，它指定了一个普通动画的clip文件路径，与骨骼的映射关系，clip的长度等

```
    REFLECTION_TYPE(BasicClip)
    CLASS(BasicClip : public ClipBase, Fields)
    {
        REFLECTION_BODY(BasicClip);

    public:
        std::string m_clip_file_path;
        float       m_clip_file_length;
        std::string m_anim_skel_map_path;
        virtual ~BasicClip() override {}
        virtual float getLength() const override
        {
            return m_clip_file_length;
        }
    };
```

- BlendState是基本的混合动画资源，它指定了一系列普通动画clip，以及他们的混合权重

```
    REFLECTION_TYPE(BlendState)
    CLASS(BlendState : public ClipBase, Fields)
    {
        REFLECTION_BODY(BlendState);

    public:
        int                      m_clip_count;
        std::vector<std::string> m_blend_clip_file_path;
        std::vector<float>       m_blend_clip_file_length;
        std::vector<std::string> m_blend_anim_skel_map_path;
        std::vector<float>       m_blend_weight;
        std::vector<std::string> m_blend_mask_file_path;
        std::vector<float>       m_blend_ratio;
        virtual ~BlendState() override {}
        virtual float getLength() const override;
    };
```

- BlendSpace1D是一维的混合空间，继承于BlendState，它指定了一系列普通动画clip，并通过一个命名float变量在运行时计算clip的混合权重

```
REFLECTION_TYPE(BlendSpace1D)
CLASS(BlendSpace1D : public BlendState, Fields)
{
    REFLECTION_BODY(BlendSpace1D);

public:
    std::string m_key;

    std::vector<double> m_values;

    virtual ~BlendSpace1D()override {}
};
```

4. AnimationFSM是一个简易的状态机

```
class AnimationFSM
{
    enum class States
    {
        _idle,
        _walk_start,
        _walk_run,
        _walk_stop,
        _jump_start_from_idle,
        _jump_loop_from_idle,
        _jump_end_from_idle,
        _jump_start_from_walk_run,
        _jump_loop_from_walk_run,
        _jump_end_from_walk_run,
        _count
    };

    States m_state {States::_idle};

public:
    AnimationFSM();
    bool   update(const json11::Json::object& signals);
    std::string getCurrentClipBaseName();
};
```

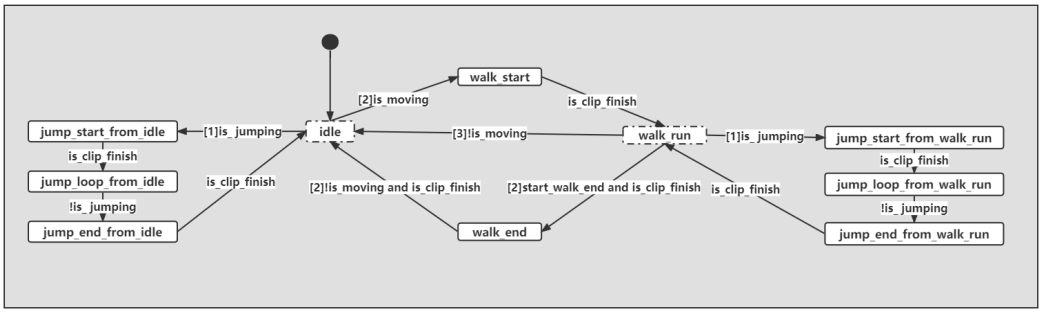
- AnimationComponent将会调用状态机的update函数，将接收集到的信号传递给状态机
- 状态机根据signals以及update函数内部的逻辑更新当前状态m_state，update函数已经留空

```
bool AnimationFSM::update(const json11::Json::object& signals)
{
    States last_state     = m_state;
    bool   is_clip_finish = tryGetBool(signals, "clip_finish", false);
    bool   is_jumping     = tryGetBool(signals, "jumping", false);
    float  speed          = tryGetFloat(signals, "speed", 0);
    bool   is_moving      = speed > 0.01f;
    bool   start_walk_end = false;

    switch (m_state)
    {
        case States::_idle:
             /**** [0] ****/
            break;
        case States::_walk_start:
             /**** [1] ****/
            break;
        case States::_walk_run:
             /**** [2] ****/
            break;
        case States::_walk_stop:
             /**** [3] ****/
            break;
        case States::_jump_start_from_idle:
```

```
                /**** [4] ****/
            break;
        case States::_jump_loop_from_idle:
                /**** [5] ****/
            break;
        case States::_jump_end_from_idle:
                /**** [6] ****/
            break;
        case States::_jump_start_from_walk_run:
                /**** [7] ****/
            break;
        case States::_jump_loop_from_walk_run:
                /**** [8] ****/
            break;
        case States::_jump_end_from_walk_run:
                /**** [9] ****/
            break;
        default:
            break;
    }
    return last_state != m_state;
}
```

- AnimationComponent将会调用getClipBaseName获得需要播放的ClipBase的名字(它可能是BasicClip、BlendState、BlendSpace1D)并将计算该ClipBase的结果应用到物体的骨骼。完成update函数后跳跃将看到动画切换
- 根据状态机示意图 （其中边说明中的[x]代表优先级，数字越小优先级越高） 完成代码。



第二部分

1. 如果当前AnimationComponent的ClipBase是一个BlendState，它将通过blend函数计算结果（结果是一个AnimationPose）
    - 它首先计算出所有BlendState列表里的所有Clip的当前的Pose，然后通过AnimationPose::blend将这些结果Pose混合到一个pose上

```
 void AnimationComponent::blend(float desired_ratio, BlendState*
blend_state)
 {
     for (auto& ratio : blend_state->m_blend_ratio)
     {
```

```
            ratio = desired_ratio;
        }
        auto blendStateData =
AnimationManager::getBlendStateWithClipData(*blend_state);
        std::vector<AnimationPose> poses;
        for (int i = 0; i < blendStateData.m_clip_count; i++)
        {
            AnimationPose pose(blendStateData.m_blend_clip[i],
                               blendStateData.m_blend_weight[i],
                               blendStateData.m_blend_ratio[i],
                               blendStateData.m_blend_anim_skel_map[i]);
            m_skeleton.resetSkeleton();
            m_skeleton.applyAdditivePose(pose);
            m_skeleton.extractPose(pose);
            poses.push_back(pose);
        }
        for (int i = 1; i < blendStateData.m_clip_count; i++)
        {
            for (auto& pose : poses[i].m_weight.m_blend_weight)
            {
                pose = blend_state->m_blend_weight[i];
            }
            poses[0].blend(poses[i]);
        }

        m_skeleton.applyPose(poses[0]);
        m_animation_result = m_skeleton.outputAnimationResult();
    }
```

2. AnimationPose

    - AnimationPose存储骨骼Transform数组及对应的权重（通过下标对应）

```
REFLECTION_TYPE(BoneBlendWeight)
CLASS(BoneBlendWeight, Fields)
{
    REFLECTION_BODY(BoneBlendWeight);

public:
    ...
    std::vector<float> m_blend_weight;
    ...
};
class AnimationPose
{
    static void extractFromClip (std::vector<Transform>&bones, const
AnimationClip& clip, float ratio);
public:
    std::vector<Transform> m_bone_poses;
    BoneBlendWeight        m_weight;
    AnimationPose();
    AnimationPose(const AnimationClip& clip, float ratio, const
AnimSkelMap& animSkelMap);
    AnimationPose(const AnimationClip& clip, const BoneBlendWeight&
weight, float ratio);
```

```
    AnimationPose(const AnimationClip& clip, const BoneBlendWeight&
weight, float ratio, const AnimSkelMap& animSkelMap);
    void blend(const AnimationPose& pose);
};
```

- AnimationPose::blend函数已经挖空，完成函数可以看到走到跑过渡的效果（可以根据注释的提示，也可以不理会注释）

```
void AnimationPose::blend(const AnimationPose& pose)
{
    for (int i = 0; i < m_bone_poses.size(); i++)
    {
        auto& bone_trans_one = m_bone_poses[i];
        const auto& bone_trans_two      = pose.m_bone_poses[i];

        // float sum_weight =
        // if (sum_weight != 0)
        {
            // float cur_weight =
            // m_weight.m_blend_weight[i] =
            // bone_trans_one.m_position  =
            // bone_trans_one.m_scale     =
            // bone_trans_one.m_rotation  =
        }
    }
}
```

- Transform的定义

```
REFLECTION_TYPE(Transform)
CLASS(Transform, Fields)
{
    REFLECTION_BODY(Transform);

public:
    Vector3    m_position {Vector3::ZERO};
    Vector3    m_scale {Vector3::UNIT_SCALE};
    Quaternion m_rotation {Quaternion::IDENTITY};

    Transform() = default;
    Transform(const Vector3& position, const Quaternion& rotation, const
Vector3& scale) :
        m_position {position}, m_scale {scale}, m_rotation {rotation}
    {}

    Matrix4x4 getMatrix() const;
};
```

- 可能会用到的函数

```
class Vector3
{
    ...
    static Vector3 lerp(const Vector3& lhs, const Vector3& rhs, float
alpha);
    ...
}

class Quaternion
{
    ...
    static Quaternion nLerp(float t, const Quaternion& kp, const
Quaternion& kq, bool shortest_path = false);
    ...
}
```

第三部分

1. 玩家控制角色运动涉及功能层中input、motor、character controller三个系统中的逻辑。玩家按下ASWD（前后左右移动）、Shift（跑）、空格键（跳）后，Input系统对应处理产生游戏指令m_game_command。

```
void InputSystem::onKeyInGameMode(int key, int scancode, int action, int
mods)
{
    ...
    if (action == GLFW_PRESS)
    {
        switch (key)
        {
            ...
            case GLFW_KEY_A:
                m_game_command |= (unsigned int)GameCommand::left;
                break;
            case GLFW_KEY_S:
                m_game_command |= (unsigned int)GameCommand::backward;
                break;
            case GLFW_KEY_W:
                m_game_command |= (unsigned int)GameCommand::forward;
                break;
            case GLFW_KEY_D:
                m_game_command |= (unsigned int)GameCommand::right;
                break;
            ...
        }
    }
    ...
}
```

2. motor系统根据各游戏指令属性，计算出motor层逻辑期望的运动位移m_desired_displacement。具体计算流程分散在motor_component.cpp的若干函数中。motor系统分别计算水平向速度、竖直方向速度、水平移动方向、逻辑期望位移，最终计算出移动位置。同学们可以阅读motor_component.cpp相关函数代码以理解跳跃状态相关逻辑。

```
void MotorComponent::tickPlayerMotor(float delta_time)
{
    ...

    unsigned int command = g_runtime_global_context.m_input_system-
>getGameCommand();

    if (command >= (unsigned int)GameCommand::invalid)
        return;

    // 计算水平方向速度
    calculatedDesiredHorizontalMoveSpeed(command, delta_time);
    // 计算竖直方向速度
    calculatedDesiredVerticalMoveSpeed(command, delta_time);
    // 计算水平移动方向
    calculatedDesiredMoveDirection(command, transform_component-
>getRotation());
    // 计算逻辑期望位移
    calculateDesiredDisplacement(delta_time);
    // 计算最终移动位置
    calculateTargetPosition(transform_component->getPosition());

    transform_component->setPosition(m_target_position);

    ...
}
```

3. character controller收到motor的逻辑期望位移，在物理场景中进行场景请求计算出符合物理规律的实际运动后的位置。目前实现的版本实现了竖直方向上的检测，可以跳至平台上。character controller通常的实现方式是分离水平和竖直方向上的检测pass来分阶段处理。controller内部会维护touch_ground等状态，使motor层能够根据controller在物理场景中的结果改变跳跃状态等。

   同学们需要改写此函数利用射线检测或形状扫描等场景请求来实现如下功能：

   - 水平移动时可以被墙壁挡住
   - 跳起后空中碰到墙壁可以落回地面（目前如果在下落阶段碰到墙壁会卡在墙上）
   - 感兴趣的同学可以继续尝试实现水平移动碰到墙面不仅仅是阻挡前进而且可以自动修正运动方向、auto-stepping或者沿斜面下滑等行为，不作为打分作业内容

```
Vector3 CharacterController::move(const Vector3& current_position, const
Vector3& displacement)
    {
        std::shared_ptr<PhysicsScene> physics_scene =
            g_runtime_global_context.m_world_manager-
>getCurrentActivePhysicsScene().lock();
        ASSERT(physics_scene);

        std::vector<PhysicsHitInfo> hits;

        Transform world_transform = Transform(
            current_position + 0.1f * Vector3::UNIT_Z,
            Quaternion::IDENTITY,
            Vector3::UNIT_SCALE);

        Vector3 vertical_displacement   = displacement.z * Vector3::UNIT_Z;
```

```
        Vector3 horizontal_displacement = Vector3(displacement.x,
displacement.y, 0.f);

        Vector3 vertical_direction   =
vertical_displacement.normalisedCopy();
        Vector3 horizontal_direction =
horizontal_displacement.normalisedCopy();

        Vector3 final_position = current_position;

        m_is_touch_ground = physics_scene->sweep(
            m_rigidbody_shape,
            world_transform.getMatrix(),
            Vector3::NEGATIVE_UNIT_Z,
            0.105f,
            hits);

        hits.clear();

        world_transform.m_position -= 0.1f * Vector3::UNIT_Z;

        // vertical pass
        if (physics_scene->sweep(
            m_rigidbody_shape,
            world_transform.getMatrix(),
            vertical_direction,
            vertical_displacement.length(),
            hits))
        {
            final_position += hits[0].hit_distance * vertical_direction;
        }
        else
        {
            final_position += vertical_displacement;
        }

        hits.clear();

        // homework: side pass
        //if (physics_scene->sweep(
        //    m_rigidbody_shape,
        //    /**** [0] ****/,
        //    /**** [1] ****/,
        //    /**** [2] ****/,
        //    hits))
        //{
        //    final_position += /**** [3] ****/;
        //}
        //else
        {
            final_position += horizontal_displacement;
        }

        return final_position;
    }
```

4. 射线检测接口定义以及参考用法如下

```
/// cast a ray and find the hits
/// @ray_origin: origin of ray
/// @ray_direction: ray direction
/// @ray_length: ray length, anything beyond this length will not be
reported as a hit
/// @out_hits: the found hits, sorted by distance
/// @return: true if any hits found, else false
bool raycast(Vector3 ray_origin,
             Vector3 ray_direction,
             float ray_length,
             std::vector<PhysicsHitInfo>& out_hits);

/// example:
std::shared_ptr<PhysicsScene> physics_scene =
g_global_runtime_context.m_world_manager->getCurrentPhysicsScene().lock();
std::vector<PhysicsHitInfo> hits;
bool is_hitted = physics_scene->raycast(start_position,
                                        ray_direction,
                                        ray_length,
                                        hits);
```

5. 形状扫描接口定义以及参考用法如下

```
/// cast a shape and find the hits
/// @shape: the casted rigidbody shape
/// @shape_transform: the initial global transform of the casted shape
/// @sweep_direction: sweep direction
/// @sweep_length: sweep length, anything beyond this length will not be
reported as a hit
/// @out_hits: the found hits, sorted by distance
/// @return: true if any hits found, else false
bool sweep(const RigidBodyShape&        shape,
           const Matrix4x4&             shape_transform,
           Vector3                      sweep_direction,
           float                        sweep_length,
           std::vector<PhysicsHitInfo>& out_hits);

/// example:
std::shared_ptr<PhysicsScene> physics_scene =
g_global_runtime_context.m_world_manager->getCurrentPhysicsScene().lock();
std::vector<PhysicsHitInfo> hits;
bool is_hitted = physics_scene->sweep(rigidbody_shape,
                                      world_transform.getMatrix(),
                                      sweep_direction,
                                      sweep_length,
                                      hits);
```

6. 形状求交接口定义以及参考用法如下

```
/// overlap test
/// @shape: rigidbody shape
/// @global_transform: testing shape transform
/// @return: true if overlapped with any rigidbodies
bool isOverlap(const RigidBodyShape& shape, const Matrix4x4&
global_transform);

/// example:
std::shared_ptr<PhysicsScene> physics_scene =
g_global_runtime_context.m_world_manager->getCurrentPhysicsScene().lock();
bool is_overlapped = physics_scene->isOverlap(rigidbody_shape,
                                              world_transform.getMatrix());
```

四、作业提交说明

作业根据评分细则分部给分。想要完成作业同学们需得到60分，这样便视为通过作业。

作业提交格式：

提交作业为一个压缩文件，命名为 Games104_homework3.zip 或 Games104_homework3.rar，其中内容为：

1. 一个文件夹runtime，里面是 engine/source/runtime 目录下的代码；
2. Games104_homework3_report.doc 或 Games104_homework3_report.docx 或 Games104_homework3_report.pdf 格式的报告文档。

   报告内容包含以下几点（若只完成部分题目，可以只写对应部分的报告）：
   - 状态机update函数代码的截图及实现思路讲解
   - AnimationPose::blend代码截图及实现思路讲解
   - CharacterController::move代码截图及实现思路讲解
3. 小白人移动及与周围环境交互的视频Games104_homework3_video.mp4，其中内容为（分别对应三题，若只完成部分题目，可以只录制对应部分的视频）：
   - 小白人跳跃并成功播放对应动画（包含起跳、腾空、落地）
   - 小白人加速移动并成功播放对应动画（从走路过渡到跑步）
   - 体现真实物理表现：如可跳至平台上，跳起后空中碰到墙壁可以落回地面，前进碰到墙壁可以自动调整位移方向往侧边移动

打分细则：

1. [25 分] 正确实现状态机update，通过编译并实现跳跃动画的状态切换。
2. [25 分] 正确实现blend，通过编译并成功实现。
3. [50 分] 修改move，通过编译并成功体现一些真实物理表现，完成对应功能可获得对应分数
   1. [25分]水平移动时可以被墙壁挡住
   2. [25分]跳起后空中碰到墙壁可以落回地面