实 验 报 告 二

（2017-2018 学年第二学期）

3D 动画游戏设计算法

实验题目：Ogre Animation

目录：

1. 分析 Ogre Animation 的部分类
2. 相关类有：

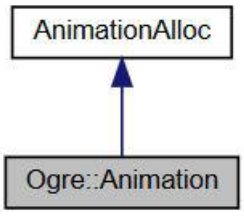
实验报告：

1. Introduction：

OGRE 版本：1.10

前言：动画是游戏引擎里面的重要模块，而 OGRE 是支持多种形式的动画的。OGRE 默认支持的有四种动画：1.骨骼动画;2.节点(Node)动画;3.顶点动画;4.数值(Numeric)动画。其中骨骼动画也是一种节点动画（这其实非常的自然，因为骨骼动画里面的关节的组织方式与场景图非常类似，都是用树组织的节点层次），所以 Ogre::Bone 也是继承自 Ogre::Node。那么这里就分析一下 **OGRE 动画系统的一些基础设施与大致结构**。

2. Ogre::Animation




```
//This class defines the interface for a sequence of animation, whether that be animation of a mesh, a path along a spline, or possibly more than one type of animation in one. An animation is made up of many 'tracks', which are the more specific types of animation.
//You should not create these animations directly. They will be created via a parent object which owns the animation, e.g. Skeleton.
```

这个类是几种动画的接口类，也可以看作是动画轨道(Animation Track)的集合，因为一个“动画”可能会包含很多片段。而且在前言里面提到的几种动画都由 Animation 类来管理。

它的 public 接口大致可以分为几类：

```
XXXTrack* createXXXTrack(...);
XXXTrack* getXXXTrack(...);
XXXTrack* destroyXXXTrack(...);
void apply(....);
```

一些 protected 的成员变量：

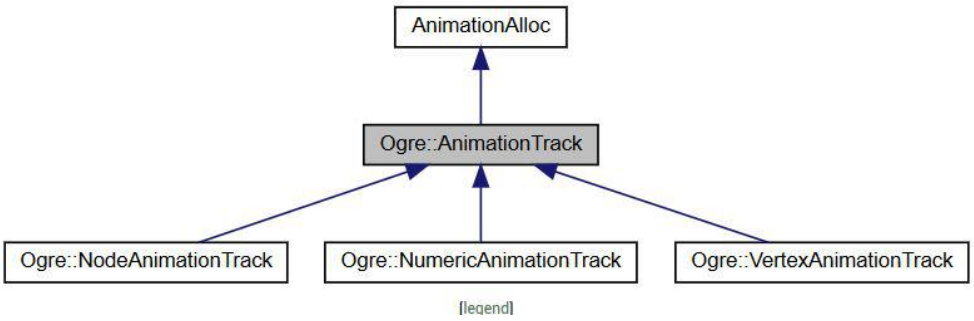
```
/// Node tracks, indexed by handle
NodeTrackList mNodeTrackList;
/// Numeric tracks, indexed by handle
NumericTrackList mNumericTrackList;
/// Vertex tracks, indexed by handle
VertexTrackList mVertexTrackList;
```

还有其他一些关于某个动画集合的全局信息的 get/set。

那么 Create/get/destroy Track 就很明显了，就是对动画轨道的生存周期管理基本操作，可见他是作为一个管理类而存在的，主要管理三种（不包括派生自节点动画的骨骼动画）动画的轨迹。然后它是**播放动画**的关键，用重载过的 apply() 来把特定类型的动画片段施加到特定的对象上。

```
void apply(Real timePos, Real weight = 1.0, Real scale = 1.0f);
void applyToNode(Node* node, Real timePos, Real weight = 1.0, Real scale = 1.0f);
void apply(Skeleton* skeleton, Real timePos, Real weight = 1.0, Real scale = 1.0f);
void apply(Skeleton* skeleton, Real timePos, float weight,
     const AnimationState::BoneBlendMask* blendMask, Real scale);
void apply(Entity* entity, Real timePos, Real weight, bool software,
     bool hardware);
void applyToAnimable(const AnimableValuePtr& anim, Real timePos, Real weight = 1.0, Real scale = 1.0f);
void applyToVertexData(VertexData* data, Real timePos, Real weight = 1.0);
```

3. Ogre::AnimationTrack



//A 'track' in an animation sequence, i.e. a sequence of keyframes which affect a certain type of animable object.

//This class is intended as a base for more complete classes which will actually animate specific types of object, e.g. a bone in a skeleton to affect skeletal animation. An animation will likely include multiple tracks each of which can be made up of many KeyFrame instances. Note that the use of tracks allows each animable object to have it's own number of keyframes, i.e. you do not have to have the maximum number of keyframes for all animable objects just to cope with the most animated one.

//Since the most common animable object is a Node, there are options in this class for associating the track with a Node which will receive keyframe updates automatically when the 'apply' method is called.

+-//By default rotation is done using shortest-path algorithm. It is possible to change this behaviour using setUseShortestRotationPath() method.

Ogre::Animation 管理了三个 AnimationTrack 的 list，那么这里就说一下它的元素 AnimationTrack。一个

AnimationTrack 就是一个动画片段，而动画片段又是由一系列的关键帧（Keyframe）组成。关键帧是动画时间轴上描述一些关键的空间节点信息的基本单元。

但是其实这个类只是作为更加详细的实现而做的铺垫，这在上面的类图也说明了，Ogre::AnimationTrack 也算式一个接口类，它是要被具体动画类型的动画轨道类（Ogre::NodeAnimationTrack, Ogre::NumericAnimationTrack, Ogre::VertexAnimationTrack）继承并重写相应虚函数才能够有相应的意义。

```
    protected:
        typedef vector<KeyFrame*>::type KeyFrameList;
        KeyFrameList mKeyFrames;
        Animation* mParent;
```

从 Ogre::AnimationTrack 的 public 成员函数和部分成员来看，它主要就是管理了一系列的 KeyFrame（关键帧）。对于关键帧的生存周期管理的接口就比较的 trivial 这里不作太多解析。但是可以留意到有一个接口函数：

```
        virtual void apply(const TimeIndex& timeIndex, Real weight = 1.0, Real scale = 1.0f) = 0;
```

继续分析下去，看到具体的动画轨迹类，可以发现最重要的动画相关的几何变换就是在与 apply()相关的函数里面实现的。拿 Ogre::NodeAnimationTrack 作为例子：

```
    void Ogre::NodeAnimationTrack::apply(const TimeIndex& timeIndex, Real weight, Real scale)
    {
        applyToNode(mTargetNode, timeIndex, weight, scale);
    }

    void Ogre::NodeAnimationTrack::applyToNode(Node* node, const TimeIndex& timeIndex, Real weight,
        Real scl)
    {
        // Nothing to do if no keyframes or zero weight or no node
        if (mKeyFrames.empty() || !weight || !node)
            return;

        TransformKeyFrame kf(0, timeIndex.getTimePos());
        getInterpolatedKeyFrame(timeIndex, &kf);

        // add to existing. Weights are not relative, but treated as absolute multipliers for the animation
        Vector3 translate = kf.getTranslate() * weight * scl;
        node->translate(translate);

        // interpolate between no-rotation and full rotation, to point 'weight', so 0 = no rotate, 1 = full
        Quaternion rotate;
        Animation::RotationInterpolationMode rim =
            mParent->getRotationInterpolationMode();
        if (rim == Animation::RIM_LINEAR)
        {
            rotate = Quaternion::nlerp(weight, Quaternion::IDENTITY, kf.getRotation(),
mUseShortestRotationPath);
        }
        else //if (rim == Animation::RIM_SPHERICAL)
        {
```

```
            rotate = Quaternion::Slerp(weight, Quaternion::IDENTITY, kf.getRotation(),
mUseShortestRotationPath);
        }
        node->rotate(rotate);

        Vector3 scale = kf.getScale();
        // Not sure how to modify scale for cumulative anims... leave it alone
        //scale = ((Vector3::UNIT_SCALE - kf.getScale()) * weight) + Vector3::UNIT_SCALE;
        if (scale != Vector3::UNIT_SCALE)
        {
            if (scl != 1.0f)
                scale = Vector3::UNIT_SCALE + (scale - Vector3::UNIT_SCALE) * scl;
            else if (weight != 1.0f)
                scale = Vector3::UNIT_SCALE + (scale - Vector3::UNIT_SCALE) * weight;
        }
        node->scale(scale);
      }
```

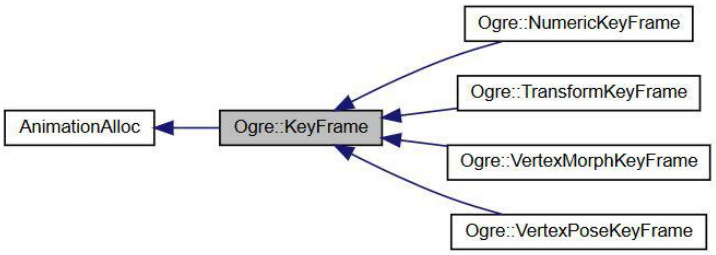
可以看到 Ogre::NodeAnimationTrack::applyToNode(...)里面就是干动画的几何变换这种底层工作：先获取一个“变换关键帧”（储存着几何变换变化信息的关键帧），根据这个关键帧的信息，再根据对应的旋转插值方法（线性插值、球面插值 SLERP），按关键帧与外部给定的混合权重来给节点施加动画。

Ogre::NumericAnimationTrack, Ogre::VertexAnimationTrack 的 apply()也有各自的具体实现，在此不展开。

顺带一提一些可选的插值方式：

```
        /** The types of animation interpolation available. */
        enum InterpolationMode
        {
            IM_LINEAR,//线性插值
            IM_SPLINE//样条插值
        };
        /** The types of rotational interpolation available. */
        enum RotationInterpolationMode
        {
            RIM_LINEAR,//线性插值
            RIM_SPHERICAL//球面线性插值
        };
```

4. Ogre::KeyFrame




```
//A key frame in an animation sequence defined by an AnimationTrack.
// This class can be used as a basis for all kinds of key frames. The unifying principle is that multiple KeyFrames define an animation sequence, with the exact state of the animation being an  interpolation between these key frames.
```

其实关键帧类本身的结构比较简单，相当于是一个容器类一样，最重要的变换工作是之前讲的 AnimationTrack 里面实现的。

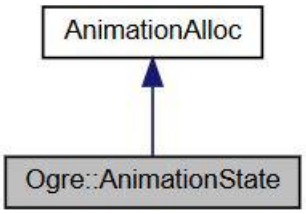
KeyFrame 基类：

```
public:
    /** Default constructor, you should not call this but use AnimationTrack::createKeyFrame instead. */
    KeyFrame(const AnimationTrack* parent, Real time);
    virtual ~KeyFrame() {}
    /** Gets the time of this keyframe in the animation sequence. */
    Real getTime(void) const { return mTime; }
    /** Clone a keyframe (internal use only) */
    virtual KeyFrame* _clone(AnimationTrack* newParent) const;
protected:
    Real mTime;
    const AnimationTrack* mParentTrack;
```

基类就比较简单，只定义了少许的接口。准确的说，不同的关键帧就只有“时间”是公有的属性，具体的关键帧属性还要靠派生类来实现。但是具体的、继承自 Ogre::KeyFrame 的关键帧就有如下几种：

1. Ogre::NumericKeyFrame：储存某个时间点上的某一个数值
2. Ogre::TransformKeyFrame：储存某个时间点上的一个完整的变换。
   （包含平移 Vector3 mTranslate;缩放 Vector3 mScale;旋转（四元数） Quaternion mRotate;）
3. Ogre::VertexPoseKeyFrame：储存某个时间点上的 Mesh::Pose（网格的姿态，也就是所有顶点的 offset 的集合）
4. Ogre::VertexMorphKeyFrame：储存某个时间点上的顶点变形(vertex morph)动画的数据。也就是会存下所有顶点在某一个时间点上的绝对坐标。

5. Ogre::AnimationState




```
//Represents the state of an animation and the weight of its influence.
//Other classes can hold instances of this class to store the state of any animations they are using.
```

Ogre::AnimationState 是用来储存动画的状态、影响权重的，其他类可以创建一个 AnimationState 的实例来储存相应的动画状态。用 Visual Studio 的查找引用发现在很多地方都实例化了 AnimationState。

Ogre::AnimationState 的成员变量：

```
protected:
    /// The blend mask (containing per bone weights)
    BoneBlendMask* mBlendMask;
    String mAnimationName;
    AnimationStateSet* mParent;
    Real mTimePos;
    Real mLength;
    Real mWeight;
    bool mEnabled;
    bool mLoop;
```

End.