本页由胡四娃译自 官方教程

动画(animation)教程

综述

这篇教程只涉及了orx中最基本的动画使用。

所有的动画都存储在一个directed graph（有向图）中。

该图定义了动画间所有可能的切换方式。动画通过一个唯一的字符串来引用。所有的切换和动画都是通过配置文件来创建的。 当一个动画被请求的时候，引擎会计算从当前动画到请求动画之间的链路 如果这个链路存在，它会自动执行。用户将通过事件被告知动画何时开始、停止、删节或者循环。 如果我们不能具体制定任何目标动画，引擎就会很自然的沿着属性中定义的线路（走下去）。 也有一个方法来越过这个寻路过程并且迅速的指向一个动画。

详细说明

通常，我们先载入config file(配置文件)，创建一个viewport,创建一个clock(时钟)并且注册Update(更新)函数，最后创建一个主对象。 请从之前的教程中获得更多的信息。

现在我们开始从代码入手，我们将会从本页的底部看到数据是如何组织的。 在Update函数中，当输入GoLeft激活的时候会触发WalkLeft动画GoRight激活的时候会触发WalkRight函数. 当没有激活态的输入时，我们会移除目标动画，让这个图保持一个自然的状态

```
if(orxInput_IsActive("GoRight"))
{
  orxObject_SetTargetAnim(pstSoldier, "WalkRight");
}
else if(orxInput_IsActive("GoLeft"))
{
  orxObject_SetTargetAnim(pstSoldier, "WalkLeft");
}
else
{
  orxObject_SetTargetAnim(pstSoldier, orxNULL);
}
```

就是这样！如何从任意当前动画切换到目标动画将会通过这个矢量图来计算。如果需要切换，他们将会自动播放。 注意：有很多的函数可以用高级的方法来控制动画，但是99%的时候，这两个函数是最常用的orxObject_SetCurrentAnim() 和 orxObject_SetTargetAnim()

让我们来看一下，动画是如何通知我们发生了什么的（比如，就像同步语音一样）。 首先，我们要向动画事件注册回调函数。

```
orxEvent_AddHandler(orxEVENT_TYPE_ANIM, EventHandler);
```

好了！让我们看下现在可以做什么了。 我们说我们想要打印出对象中哪个动画被播放、停止、剪切或者

循环。需要写一下的回调函数。

```
orxSTATUS orxFASTCALL EventHandler(const orxEVENT *_pstEvent)
{
orxANIM_EVENT_PAYLOAD *pstPayload;

pstPayload = (orxANIM_EVENT_PAYLOAD *)_pstEvent->pstPayload;

switch(_pstEvent->eID)
{
  case orxANIM_EVENT_START:
    orxLOG("Animation <%s>@<%s> has started!", pstPayload->zAnimName,
orxObject_GetName(orxOBJECT(_pstEvent->hRecipient)));
    break;

  case orxANIM_EVENT_STOP:
    orxLOG("Animation <%s>@<%s> has stoped!", pstPayload->zAnimName,
orxObject_GetName(orxOBJECT(_pstEvent->hRecipient)));
    break;

  case orxANIM_EVENT_CUT:
    orxLOG("Animation <%s>@<%s> has been cut!", pstPayload->zAnimName,
orxObject_GetName(orxOBJECT(_pstEvent->hRecipient)));
    break;

  case orxANIM_EVENT_LOOP:
    orxLOG("Animation <%s>@<%s> has looped!", pstPayload->zAnimName,
orxObject_GetName(orxOBJECT(_pstEvent->hRecipient)));
    break;
  }

  return orxSTATUS_SUCCESS;
}
```

先得到了事件的payload指针，因为我们只是在这里传递动画事件，所以我们可以安全的将payload 转化为orxANIM_EVENT_PAYLOAD类型，它在 orxAnim.h中定义。 如果我们在不同的事件（译者注：原文是even 根据上下文推断是作者拼写错误）类型中调用了同一个回调函数，我们首先将会查看是否得到了一个动画事件，可以这样做：

```
if(_pstEvent->eType == orxEVENT_TYPE_ANIM)
```

最后，事件接收者(_pstEvent→hRecipient)通常是播放动画的那个对象。将其用宏orxOBJECT()来转化为orOBJECT类型的对象。

现在让我们来看一眼数据方面的东西吧。 首先，我们需要定义一个动画集，它将会包含指定对象的动画的整个矢量图。 动画集在不会再内存中重复，并且它与矢量图相对应的多有动画和链路。 在上面这个例子中，我们又4个动画和10条可以用来切换的链路。

```
[AnimSet]
AnimationList = IdleRight#WalkRight#IdleLeft#WalkLeft
```

```
LinkList =
IdleRightLoop#IdleRight2Left#IdleRight2WalkRight#WalkRightLoop#WalkRight2IdleRight#IdleLeftLoop#IdleLeft2Right#IdleLeft2WalkLeft#WalkLeftLoop#WalkLeft2IdleLeft
```

现在我们来开始定义动画！ 在这之前，为了减少文章篇幅，我们将要使用orx 配置文件的集成特性。 先锚点的位置定义一项。 也许你可能在对象教程中看到了锚点的相关知识，锚点的位置信息将会匹配世界中的对象。如果没有确定的话，将会把左上角做为默认值。 锚点可以通过语义关键字来确定，如 top, bottom, center, left and right也可以通过实际的值来确定。

```
[Pivot]
Pivot = (15.0, 31.0, 0.0)
```

现在，我们来定义从锚点继承过来的图像对象。在我们这个例子中，它是一个位图,，包含了对象中所有的帧。因此基本的属性就是 位图文件的名字和一帧的大小。

```
[FullGraphic@Pivot]
Texture     = ../../data/anim/soldier_full.png
TextureSize = (32, 32, 0)
```

创建帧的准备工作都做好了。 让我们定义所有都是right-oriented的动画。一共6个。

```
[AnimRight1@FullGraphic]
TextureCorner = (0, 0, 0)

[AnimRight2@FullGraphic]
TextureCorner = (0, 32, 0)

[AnimRight3@FullGraphic]
TextureCorner = (0, 64, 0)

[AnimRight4@FullGraphic]
TextureCorner = (32, 0, 0)

[AnimRight5@FullGraphic]
TextureCorner = (32, 32, 0)

[AnimRight6@FullGraphic]
TextureCorner = (32, 64, 0)
```

看到了吧，他们全都继承于FullGraphic唯一能区分他们的属性就是TextureCorner. 好，我们已经定义完了所有的图形对象（他们载入的时候会转变为orxGraphic结构），下面定义动画本身。让我们从ideright动画开始说起，它包含一个单帧并持续0. 1秒。

```
[IdleRight]
KeyData1      = AnimRight6
KeyDuration1  = 0.1
```

太简单了，来尝试下第二个：

```
[WalkRight]
```

```
DefaultKeyDuration  = 0.1
KeyData1            = AnimRight1
KeyData2            = AnimRight2
KeyData3            = AnimRight3
KeyData4            = AnimRight4
KeyData5            = AnimRight5
KeyData6            = AnimRight6
```

当我们使用DefaultKeyDuration属性同时为所有的帧定义时并不是很难。我们可以像idleright动画中所做的那样，通过一个确定的键值来覆盖任意一帧。我们如法炮制做出left-oriented动画。通常我们使用翻转图形对象时，我们将会在代码运行中做这件事。但是那不是我们的目的！让我们来用与前面那个完全不同的方法来实现它！只有链路没有提到了让我们添上它。基本的链路结构非常简单，我们指定源动画和目的动画。

```
[IdleRightLoop]
Source      = IdleRight
Destination = IdleRight
```

这里，我们有跟之前一样的基本信息，但是多了一个immediate属性做为键值。这就是说，当我们处于IdleRight动画时，并且目标是 WalkRight,我们不必等到IdleRight完成，将直接完成这个动作，这就给了我们一个剪切动画的方法。 正如在代码中看到的一样。当我们已经开始行走的时候，没有显式的调用空闲动画，这是怎么做到的？看下从WalkRight到IdleRight的链路。

```
[IdleRight2WalkRight]
Source      = IdleRight
Destination = WalkRight
Property    = immediate
```

当我们再WalkRight状态并且移除了目标动画，引擎不得按照自然的路线走下去。这个意思是说，它会选取高优先级的链路。默认的优先级是8，它的范围是0到15. 在这里，优先级是9，也就是说当我们没有目标的时候，就会选取它。它将会带我们回到 IdleRight状态。这里也加了immdiate属性，这样，我们就不必等“走”这个循环完事再回到“空闲”

注意：这只是一个非常基本的图，用来阐述基本的动画切换过程，但是这个系统的扩展性很高。比如假设这样一个场景：你想从坐的状态变为走的状态，中间没有别的过度。随着游戏的开发，你可能觉得在这两个状态间加一个站立的状态会比较好。这时，你只需要再配置文件中添加这多出来的一步，而整个代码文件都不用更改。

资源

源代码：04_Anim.c

配置文件：04_Anim.ini