文档:教程:基础教程:基础教程四

出自 Ogre3D 开放资源地带

跳转到: 导航, 搜索

目录

先决条件

本教程假定你已经拥有了 c++程序设计的知识，并且已经安装和编译了一个 Ogre 的应用程序（如果你在设置你的应用程序中有困难，请参考 this guide 获得更详细的编译步骤）。这个教程同时也是建立在上一章基础上的，因此默认你已经了解了上个教程的内容。

介绍

这一章我们将介绍 Ogre 中最有用的构造：帧监听(FrameListener)。在本指南最后你将了解帧监听，怎样运用帧监听去实现一些要求每一帧更新的东西，怎样去用 Ogre 的无缓冲输入系统。

代码你都可以在这篇指南中找到。当你看完这篇教程以后，你应该试着慢慢的添加一些代码到你自己的工程里，然后看看结果。

从这开始

像上一个教程一样，我们将使用一个先前建立的代码作为我们出发的起点。在编译器中创建一个工程，添加如下的源代码：

```
#include "ExampleApplication.h"

class TutorialFrameListener : public ExampleFrameListener
{
public:
    TutorialFrameListener(RenderWindow* win, Camera* cam,
SceneManager *sceneMgr)
        : ExampleFrameListener(win, cam, false, false)
    {
    }

    bool frameStarted(const FrameEvent &evt)
    {
        return ExampleFrameListener::frameStarted(evt);
    }
protected:
    bool mMouseDown;       // Whether or not the left mouse button
was down last frame
    Real mToggle;          // The time left until next toggle
    Real mRotate;          // The rotate constant
    Real mMove;            // The movement constant
    SceneManager *mSceneMgr;   // The current SceneManager
    SceneNode *mCamNode;   // The SceneNode the camera is currently
attached to
};

class TutorialApplication : public ExampleApplication
{
public:
    TutorialApplication()
    {
    }

    ~TutorialApplication()
    {
    }
protected:
    void createCamera(void)
    {
    }

    void createScene(void)
    {
    }
```

```
    void createFrameListener(void)
    {
    }
};

#if OGRE_PLATFORM == PLATFORM_WIN32 || OGRE_PLATFORM ==
OGRE_PLATFORM_WIN32
#define WIN32_LEAN_AND_MEAN
#include "windows.h"

INT WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR strCmdLine, INT)
#else
int main(int argc, char **argv)
#endif
{
    // Create application object
    TutorialApplication app;

    try {
        app.go();
    } catch(Exception& e) {
#if OGRE_PLATFORM == PLATFORM_WIN32 || OGRE_PLATFORM ==
OGRE_PLATFORM_WIN32
        MessageBox(NULL, e.getFullDescription().c_str(), "An
exception has occurred!", MB_OK | MB_ICONERROR | MB_TASKMODAL);
#else
        fprintf(stderr, "An exception has occurred: %s\n",
                e.getFullDescription().c_str());
#endif
    }

    return 0;
}
```

如果你是在 Windows 下使用 OgreSDK 的，请确定添加“"[OgreSDK_DIRECTORY]\samples\include”目录到这个工程(ExampleApplication.h 文件所在的位置)除了标准包含以外。如果使用的是 Ogre 的源代码包，这个文件应该在“"[OgreSource_DIRECTORY]\Samples\Common\include”目录中。不要试着去运行程序，因为我们还没有定义键盘的行为。如果你有问题，请查看这个 Wiki 页获得设置你编译器的信息，如果仍然有问题请试着去看看帮助栏。

我们将在这章中定义程序控制器。

帧监听

介绍

在前面的指南中当我们添加代码到创建场景方法时候，我们仅仅考虑我们可以做什么。在 Ogre 中我们可以注册一个类去接收消息当一帧被渲染到屏幕之前和之后。FrameListener 接口定义两个函数：

```
bool frameStarted(const FrameEvent& evt)
bool frameEnded(const FrameEvent& evt)
```

Ogre 的主循环(Root::startRendering)类似这样：

1. Root object 调用 frameStarted 方法在所有已经注册的 FrameListeners 中。
2. Root object 渲染一帧。
3. Root object 调用 frameEnded 方法在所有已经注册的 FrameListeners 中。

这个循环直到有任意一个 FrameListener 的 frameStarted 或者 frameEnded 函数返回 false 时终止。这些函数返回值的主要意思是“保持渲染”。如果你返回否，这个程序将退出。FrameEvent 实体包含两个参数，但是只有 timeSinceLastFrame 在帧监听中是有用的。这个变量了解 frameStarted 或者 frameEnded 最近被激活时间。注意在 frameStarted 方法中，FrameEvent::timeSinceLastFrame 将包含 frameStarted 时间最近被激活的时间（而不是最近被激活的 frameEnded 方法）。

一个重要的概念你不能决定那个 FrameListener 被调用的次序。如果你需要确定 FrameListeners 被调用的准确顺序，你应该只注册一个 FrameListener，然后用适当的顺序来调用所有的物体。

你应该也会注意到主循环事实上做了三件事，在 frameEnded 方法和 frameStarted 方法被调用中间什么都没有发生，你甚至可以换着用。你的代码也可以随你放哪。你可以放到一个大的 frameStarted 或者 frameStarted 方法中，也可以插到两者之间。

注册一个帧监听

现在上面的代码是可以编译通过的，但是因为我们没有考虑在 ExampleApplication 和 createCamera 中创建帧监听，因此如果你运行这个程序你就无法结束她。在继续本指南其他内容之前我们将先解决这个问题。

找到 TutorialApplication::createCamera 方法添加如下代码：

```
        // create camera, but leave at default position
        mCamera = mSceneMgr->createCamera("PlayerCam");
        mCamera->setNearClipDistance(5);
```

除了最基本的之外我们什么都没有做。之所以我们要用 ExampleApplication 中的 createCamera 方法是因为 createCamera 方法可以用来移动相机和改变相机位置，这里我们详细讨论这个。

因为 Root 类是要每帧更新的，他也了解 FrameListeners。我们首先要创建 TutorialFrameListener 的实例，然后把他注册到 Root 对象中。代码如下：

```
        // Create the FrameListener
        mFrameListener = new TutorialFrameListener(mWindow, mCamera,
mSceneMgr);
        mRoot->addFrameListener(mFrameListener);
```

mRoot 和 mFrameListener 变量是在 ExampleApplication 类中定义的。addFrameListener 方法添加一个帧监听者，removeFrameListener 方法移除帧监听者（也就是说 FrameListener 就不用在更新了）。注意 add|removeFrameListener 方法仅仅是得到一个指向 FrameListener 的指针（也就是说 FrameListener 并没有一个名字你可以移除她）。这就意味着你需要对每一个 FrameListener 持有一个指针，让你随后可以移除他们。

这个 ExampleFrameListener（我们的 TutorialFrameListener 是从它继承来的），还提供了一个 showDebugOverlay(bool)方法，用来告诉 ExampleApplication 是否要在左下角显示帧率的提示框。我们会把它打开：

```
        // Show the frame stats overlay
        mFrameListener->showDebugOverlay(true);
```

确定你可以编译和运行这个应用程序再继续。

建立场景

介绍

在我们对代码的深入研究之前，我将简单介绍一下我将做什么，这样你可以了解当我们创建和添加一些东西到场景中去的目的。

我们将把一个物体(一个忍者)放到屏幕中，并在场景中加上点光源。当你点击鼠标左键，灯光会打开或关闭。按住鼠标右键，则开启“鼠标观察”模式（也就是你用摄像机四处观望）。我们将在场景里放置场景节点，来让作用于不同视口的摄像机附在上面。按下 1、2 键，以选择从哪一个视口来观看场景。

代码

找到 TutorialApplication::createScene 方法。首先我们需要把场景的环境光设置的很低。我们想要场景中物体在灯光关闭情况下仍然可见，而且我们仍然想做到在灯光开关时场景有所变化：

```
        mSceneMgr->setAmbientLight(ColourValue(0.25, 0.25, 0.25));
```

现在把一个忍者加到屏幕中去：

```
        Entity *ent = mSceneMgr->createEntity("Ninja", "ninja.mesh");
        SceneNode *node = mSceneMgr->getRootSceneNode()-
>createChildSceneNode("NinjaNode");
        node->attachObject(ent);
```

现在我们创建一个白色灯让后放到场景中，和忍者有一个相对小距离：

```
        Light *light = mSceneMgr->createLight("Light1");
        light->setType(Light::LT_POINT);
        light->setPosition(Vector3(250, 150, 250));
        light->setDiffuseColour(ColourValue::White);
        light->setSpecularColour(ColourValue::White);
```

现在需要创建场景节点供摄像机绑定：

```
        // Create the scene node
        node = mSceneMgr->getRootSceneNode()-
>createChildSceneNode("CamNode1", Vector3(-400, 200, 400));
        node->yaw(Degree(-45));
        node->attachObject(mCamera);

        // create the second camera node
        node = mSceneMgr->getRootSceneNode()-
>createChildSceneNode("CamNode2", Vector3(0, 200, 400));
```

现在我们在 TutorialApplication 类里做得差不多了。再到 TutorialFrameListener 里去...

帧监听指南

变量

在我们走得更远之前，还是先介绍一下 TutorialFrameListener 类里的一些变量：

```
bool mMouseDown;        // 鼠标左键是否在上一帧被按下
Real mToggle;           // 直到下一次触发的时间
Real mRotate;           // 滚动常量
Real mMove;             // 移动常量
SceneManager *mSceneMgr;   // 当前场景管理器
SceneNode *mCamNode;    // 当前摄像机所附在的场景节点
```

mSceneMgr 拥有一个目前 SceneManager 的指针，mCamNode 拥有目前 SceneNode 并且绑定着摄像机。mRotate 和 mMove 是旋转和移动的变量。如果你想移动或旋转的更快或更慢，改变这两个变量的大小就可以了。 另外两个变量（mToggle 和 mMouseDown）控制我们的输入。我们将在本章中使用“非缓冲”(unbuffered)鼠标和键盘输入(“缓冲”(buffered)输入将是下一章的议题。这意味着我们将在帧监听器查询鼠标和键盘状态时调用方法。

当我们试图用键盘来改变屏幕中物体的状态时，会遇到一些有趣的问题。如果知道一个键正被按下，我们会去处理它，但到了下一帧怎么办？我们还会看见它被按下，然后再重复地做操作吗？在一些情况（比如用方向键来移动），我们是这样做的。然而，我们想通过“T”键来控制灯的开关时，在第一帧里按下 T 键，拨动了灯的开关，但在下一帧 T 键仍然被按着，所以灯又打开了...如此返复，直到按键被释放。我们必须在帧与帧之间追踪这些按键状态，才能避免这个问题。我将介绍两种不同的方法来解决它。

mMouseDown 用来追踪鼠标在上一帧里是否也被按下（所以如果 mMouseDown 为真，在鼠标释放之前我们不会做同样的操作）。mToggle 指定了直到我们可以执行下一个操作的时间。即，当一个按键被按下去了，在 mToggle 指明的这段时间里不允许有其它动作发生。

构造函数

关于这个构造函数首先需要注意的是，我们要调用 ExampleFrameListener 的构造函数：首先我们要继承 ExampleFrameListener 的构造函数。

```
: ExampleFrameListener(win, cam, false, false)
```

一个需要注意的是，第三和第四个参数要置为 false。第三个参数指明是否使用带缓冲的键盘输入，第四个参数指明是否使用带缓冲的鼠标输入（我们在本课都不使用）。

在 TutorialFrameListener 的构造函数中，我们把所有的变量设置为默认值：

```
// 键盘和鼠标状态追踪
 mMouseDown = false;
mToggle = 0.0;

// Populate the camera and scene manager containers
```

```
mCamNode = cam->getParentSceneNode();
mSceneMgr = sceneMgr;

// 设置旋转和移动速度
mRotate = 0.13;
mMove = 250;
```

好了。mCamNode 变量被初始化成摄像机的任何当前父节点。

帧启动方法

现在，我们到了教程最核心的部分：在每一帧里执行动作。目前我们的 frameStarted 方法里有如下代码：

```
return ExampleFrameListener::frameStarted(evt);
```

这块代码非常重要。ExampleFrameListener::frameStarted 方法定义了许多行为（像所有的按键绑定、所有的摄像机移动等）。**清空 TutorialFrameListener::frameStarted 方法。**

开放输入系统(OIS)提供了三个主要的类来获得输入：Keyboard，Mouse，和 Joystick 。在教程里，我们只有涉及如何使用 Keyboard 和 Mouse 对象。倘若你对在 Ogre 里使用摇杆感兴趣，请查阅 Joystick 类。

我们使用无缓冲输入时，要做的第一件事情就是获取当前鼠标键盘的状态。为了达到目的，我们调用 Mouse 和 Keyboard 对象的 capture 方法。示例框架已经帮我们创建了这些对象，分别在 mMouse 和 mKeyboard 变量里。将以下代码添加到目前还是空的 TutorialFrameListener::frameStarted 成员方法里。

```
mMouse->capture();
mKeyboard->capture();
```

接下来，我们要保证当 Escape 键被按下时程序退出。我们通过调用 InputReader 的 isKeyDown 方法并指定一个 KeyCode，来检查一个按钮是否被按下。当 Escape 键被按下时，我们只要返回 false 到程序末尾。

```
if(mKeyboard->isKeyDown(OIS::KC_ESCAPE))
   return false;
```

为了能继续渲染，frameStarted 方法必须返回一个正的布尔值。我们将在方法最后加上这么一行。

```
return true;
```

所有我们将要讨论的代码都在最后的这一行"return true"之上。

我们使用 FrameListener 来做的第一个事情就是，让鼠标左键来控制灯的开关。通过调用 InputReader 的 getMouseButton 方法，并传入我们要查询的按钮，我们能够检查这个按钮是否被按下。通常 0 是鼠标左键，1 是右键，2 是中键。在某些系统里 1 是中键、2 是右键。如果按键不能很好的工作，可尝试这样设置：

```
        bool currMouse = mMouse-
>getMouseState().buttonDown(OIS::MB_Left);
```

如果鼠标被按下，currMouse 变量设为 true。现在我们拨动灯开关，依赖于 currMouse 是否为 true，同时在上一帧中鼠标没有被按下(因为我们只想每次按下鼠标时，只让灯开关被拨动一次)。同时注意 Light 类里的 setVisible 方法决定了这个对象实际上是否发光：

```
        if (currMouse && ! mMouseDown)
        {
            Light *light = mSceneMgr->getLight("Light1");
            light->setVisible(! light->isVisible());
        } // if
```

现在，我们把 mMouseDown 的值设置成与 currMouse 变量相同。下一帧里，它会告诉我们上次是否按下了鼠标按键。

```
        mMouseDown = currMouse;
```

编译运行程序。现在可以左击鼠标控制灯的开关！注意，因为我们不再调用 ExampleFrameListener 的 frameStarted 方法，我们不能转动摄像头（目前）。

这种保存上一次鼠标状态的方法非常好用，因为我们知道我们已经对为鼠标状态做了动作。

这样做的缺点就是你要为每一个被绑定动作的按键设置一个布尔变量。一种能避免这种情况的方法是，跟踪上一次点击任何按键的时刻，然后只允许经过了一段时间之后才触发事件。我们用 mToggle 变量来跟踪。如果 mToggle 大于 0，我们就不执行任何动作，如果 mToggle 小于 0，我们才执行动作。我们将用这个方法来进行下面两个按键的绑定。

首先我们要做的是拿 mToggle 变量减去从上一帧到现在所经历的时间。

```
        mToggle -= evt.timeSinceLastFrame;
```

现在我们更新了 mToggle，我们能操作它了。接下来我们将“1”键绑定为摄像机附在第一个场景节点上。在这之前，我们检查 mToggle 变量以保证它是小于 0 的：

```
        if ((mToggle < 0.0f ) && mKeyboard->isKeyDown(OIS::KC_1))
        {
```

现在我们需要设置 mToggle 变量，使得在下一个动作发生时至少相隔一秒钟：

```
            mToggle = 0.5f;
```

接下来，我们将摄像机从当前附属的节点取下来，然后让 mCamNode 变量含有"CamNode1"节点，再将摄像机放上去。

```
            mCamera->getParentSceneNode()->detachObject(mCamera);
            mCamNode = mSceneMgr->getSceneNode("CamNode1");
            mCamNode->attachObject(mCamera);
        }
```

当按下 2 键时，对于 CamNode2 我们也是这么干的。除了把 1 换成 2，if 改成 else if，其它代码都是相同的：

```
        else if ((mToggle < 0.0f) && mKeyboard->isKeyDown(OIS::KC_2))
        {
            mToggle = 0.5f;
            mCamera->getParentSceneNode()->detachObject(mCamera);
            mCamNode = mSceneMgr->getSceneNode("CamNode2");
            mCamNode->attachObject(mCamera);
        }
```

编译并运行。我们通过按 1、2 键，能够转换摄像机的视口。

我们的下一个目标就是，当用户按下方向键或 W、A、S、D 键时，进行 mCamNode 的平移。

与上面的代码不同，我们不必追踪上一次移动摄像机的时刻，因为在每一帧按键被按下时我们都要进行平移。这样一来我们的代码会相对简单，首先我们来创建一个 Vector3 用于保存平移的方位：

```
        Vector3 transVector = Vector3::ZERO;
```

好的，当按下 W 键或者上箭头时，我们希望前后移动（也就是 Z 轴，记住 Z 轴负的方向是指向屏幕里面的）：

```
        if (mKeyboard->isKeyDown(OIS::KC_UP) || mKeyboard-
>isKeyDown(OIS::KC_W))
            transVector.z -= mMove;
```

对于 S 键和下箭头键，我们基本上也是这么做的，只不过方向是往正 Z 轴：

```
    if (mKeyboard->isKeyDown(OIS::KC_DOWN) || mKeyboard-
>isKeyDown(OIS::KC_S))
        transVector.z += mMove;
```

对于左右方向移动的，我们是在 X 的正负方向进行的：

```
    if (mKeyboard->isKeyDown(OIS::KC_LEFT) || mKeyboard-
>isKeyDown(OIS::KC_A))
        transVector.x -= mMove;
    if (mKeyboard->isKeyDown(OIS::KC_RIGHT) || mKeyboard-
>isKeyDown(OIS::KC_D))
        transVector.x += mMove;
```

最后，我们还想沿着 Y 轴进行上下移动。我个人喜欢用 E 键或者 PageDown 键进行向下移动，用 Q 键或者 PageUp 键进行向上移动：

```
    if (mKeyboard->isKeyDown(OIS::KC_PGUP) || mKeyboard-
>isKeyDown(OIS::KC_Q))
        transVector.y += mMove;
    if (mKeyboard->isKeyDown(OIS::KC_PGDOWN) || mKeyboard-
>isKeyDown(OIS::KC_E))
        transVector.y -= mMove;
```

好了，我们的 transVector 变量保存了作用于摄像机场景节点的平移。这样做的第一个缺点是如果你旋转了场景节点，再作平移的时候我们的 x、y、z 坐标就不对了。为了修正它，我们必须把作用在场景节点的旋转，也作用于我们的平移。这实际上比听起来更简单。

为了表示旋转，Ogre 不是像一些图形引擎那样使用变换矩阵，而是使用四元组。四元组的数学意义是一个较难理解的四维线性代数。幸好，你不必去了解它的数字知识，了解如何使用就行了。非常简单，用四元组旋转一个向量只要将它们两个相乘即可。现在，我们希望将所有作用于场景节点的旋转，也作用在平移向量上。通过调用 SceneNode::getOrientation()，我们能得到这些旋转的四元组表示，然后用乘法让它作用在平移节点。

还有一个缺陷要引起注意的是，我们必须根据从上一帧到现在的时间，来对平移的大小进行缩放。否则，你的移动速度取决于应用程序的帧率。这确实不是我们想要的。这里有一个函数供我们来调用，可以避免平移摄像机时带来的问题:

```
     mCamNode-
>translate(transVector*evt.timeSinceLastFrame,Node::TS_LOCAL);
```

好了，我们来介绍一些新的东西。当你对一个节点进行平移，或者是绕某个轴进行旋转，你都能指定使用哪一个“变换空间”来移动它。一般你移动一个对

象时，不需要指定这个参数。它默认是 TS_PARENT，意思是这个对象使用的是父节点所在的变换空间。在这里，父节点是场景的根节点。当我们按下 W 键时（向前移动），我们走向 Z 轴负的方向。如果我们不在代码前面指明 TS_LOCAL，摄像机都会向全局负 Z 轴移动。然而，既然我们希望按下 W 键时摄像机往前走，我们就需要它往节点所面朝的方向走。所以，我们使用“本地”变换空间。

还有另一种方式我们也能实现（尽管不是怎么直接）。我们获取节点的朝向，一个四元组，用方向向量乘以它，能得到相同的结果。这完全是合法的:

```
        // Do not add this to the program
        mCamNode->translate(mCamNode-
>getOrientation()*transVector*evt.timeSinceLastFrame, Node::TS_WORLD);
```

这同样也在本地空间里进行移动。在这里，实际上没这样做的必要。Ogre 定义了三种变换空间：TS_LOCAL，TS_PARENT，和 TS_WORLD。也许存在其它的场合，你需要使用另外的向量空间来进行平移或旋转。若真是这样，你可以像上面的代码那样做。找到一个表示向量空间的四元组（或者你要操作的物体的朝向），用平移向量去乘它，得到一个正确的平移向量，然后用它在 TS_WORLD 空间里移动。目前应该不会遇到这样的情况，我们后面的教程也不会涉及这些内容。

好了，我们有了键盘控制的移动。我们还想实现一个鼠标效果，来控制我们观察的方向，但这只有在鼠标右键被按住的情况下。为了达到目的，我们首先要检查右键是否被按下去：

```
        if (mMouse->getMouseState().buttonDown(OIS::MB_Right))
        {
```

如果是，我们根据从上一帧开始鼠标移动的距离来控制摄像机的俯仰偏斜。为了达到目的，我们取得 X、Y 的相对改变，传到 pitch 和 yaw 函数里去：

```
            mCamNode->yaw(Degree(-mRotate * mMouse-
>getMouseState().X.rel), Node::TS_WORLD);
            mCamNode->pitch(Degree(-mRotate * mMouse-
>getMouseState().Y.rel), Node::TS_LOCAL);
        }
```

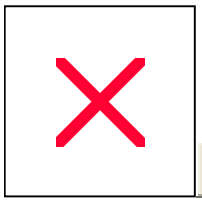
注意，我们使用 TS_WORLD 向量坐标来进行偏斜（如果不指明的话，旋转函数总是用 TS_LOCAL 作为默认值）。我们要保证物体随意的俯仰不影响它的偏斜，我们希望总是绕着固定的轴进行偏斜转动。如果我们把 yaw 设置成 TS_LOCAL，我们就能得到这样的：

http://www.idleengineer.net/images/beginner04_rot.png

编译程序并运行。

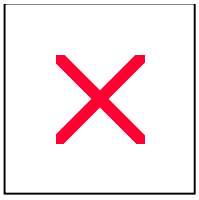
本课不是关于旋转和四元组的完整教程（那样的话会构成另一系列的教程）。下一课里，我们将使用带缓冲的输入，而不用在每一帧里都检查按键是否被按下。

原文
上一章节：基础教程三
下一章节：基础教程五
目录



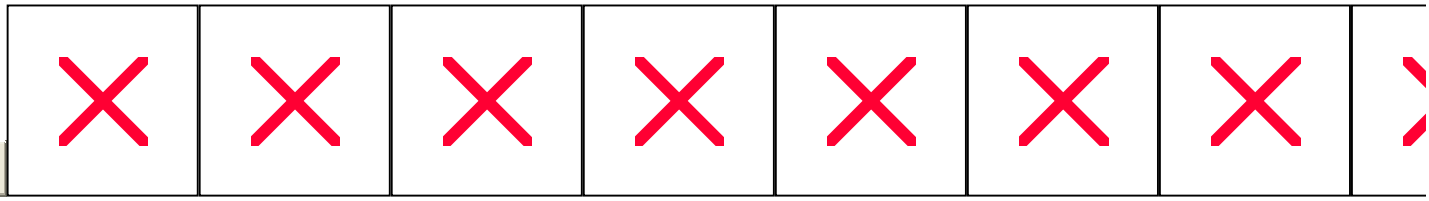
进入 搜索

Google **搜索**



our search terms

search form



earch

工具箱

- 链入页面
- 链出更改
- 特殊页面
- 可打印版
- 永久链接

Google Adsense

- 这页的最后修订在 2009 年 5 月 14 日（星期四）05:33。
- 本页面已经被浏览 1,724 次。
- 隐私政策
- 关于 Ogre3D 开放资源地带
- 沪 ICP 备 09049564 号