Event-Driven Design

A good game engine will provide a distinct layer of abstraction between input devices and the game code that the devises interact with. If this layer of abstraction is not present, the game code will be tightly coupled with the input device code. Ideally, we’d like to be able to separate these, allowing new input devices to be added or old ones replaced, without changing the game code. We could even replace human input with an AI controller. Event-driven design is a good method of achieving this separation. Instead of a system that constantly checks whether something has happened and then reacting (polling), we can fire an event *notifying* the system that something has happened so that it can react accordingly. The same “higher level” events can be fired by any input device (or an AI).

Event-driven design can be used in a wide variety of cases beyond input devices, allowing subsystems of a game engine to communicate with minimal knowledge of each-other’s operation (loosely coupled code). In this tutorial we’ll be focussing on using event-driven design to handle input effectively.

# Chase Camera Sample

A modified version of Microsoft’s ChaseCamera XNA sample is available on Moodle. **Download and extract the package from there.** This sample shows a simple spring-based camera that follows a ship as it moves around the world.

*Note: If you’d like to compare this with the original code sample, it can be found here (****do not*** *use the original sample when completing the tutorial):*

<http://xbox.create.msdn.com/en-US/education/catalog/sample/chasecamera>

**Compile and run the sample**.

You will find that a ship is just standing there motionless, waiting for your input. By the end of this tutorial, you’ll have the ship soaring.

# Handling Polling

Standard XNA input works by updating (polling) the state of the input devices (Mouse, Keyboard and Gamepad) during each *Update()*. We’ll create a class that will translate this polling behaviour into events if any changes are detected.

**Add a new C# class to the project (Add > Class). Call it *“InputListener.cs”* and give it the following implementation (replace the InputListener definition, but leave the surrounding code as it is):**

public class InputListener

{

// Current and previous keyboard states

private KeyboardState PrevKeyboardState { get; set; }

private KeyboardState CurrentKeyboardState { get; set; }

// List of keys to check for

public HashSet<Keys> KeyList;

public InputListener()

{

CurrentKeyboardState = Keyboard.GetState();

PrevKeyboardState = CurrentKeyboardState;

KeyList = new HashSet<Keys>();

}

public void AddKey(Keys key)

{

KeyList.Add(key);

}

public void Update()

{

PrevKeyboardState = CurrentKeyboardState;

CurrentKeyboardState = Keyboard.GetState();

FireKeyboardEvents();

}

private void FireKeyboardEvents()

{

}

}

**Add the following *using* statements at the top of the class:**

using Microsoft.Xna.Framework;

using Microsoft.Xna.Framework.Input;

This is a fairly straightforward class that simply updates two *KeyboardState* objects – the current state and the previous state of the keyboard. We’ll see how this is important soon.

**Note:** the class contains a *KeyList* member. XNA provides us with a *Keys* enum that maps numbers to all the keys on a keyboard. We could iterate through each value to check if the key was pressed or not using some C# magic:

foreach (Keys key in Enum.GetValues(typeof(Keys)))

{

// Check key states...

}

Bearing in mind that there are usually 256 values on a keyboard, it would be more efficient (and a lot easier to debug) if we were just to check keys that we know are being used in the game. This is where the *KeyList* collection and the *AddKey()* function comes in. We’ll be making use of the function in a later class, but first let’s make this class fire some events.

# Firing Events

Events notify an object that something has happened. C# simplifies this by using two keywords: *event* and *delegate*.

*Delegates* are similar to *function pointers* found in C++, and *events* are used to call one or more objects that implement these delegates. Each event can make use of the default C# delegate called *EventHandler* that handles event arguments. We can create custom event arguments by inheriting from the *EventArgs* class. The *KeyboardEventArgs* class has already been created for you. It’s a very simple class that contains the information needed by any delegate when a keyboard event is triggered. Now let’s add the appropriate event handlers to the *InputListener* class.

**Add the following fields to the *InputListener* class:**

//Keyboard event handlers

//key is down

public event EventHandler<KeyboardEventArgs> OnKeyDown = delegate { };

//key was up and is now down

public event EventHandler<KeyboardEventArgs> OnKeyPressed = delegate { };

//key was down and is now up

public event EventHandler<KeyboardEventArgs> OnKeyUp = delegate { };

Now we’ve got the event handlers in there, we can populate the *FireKeyboardEvents()* function.

**Add the following code to InputListener’s *FireKeyboardEvents()* function:**

private void FireKeyboardEvents()

{

// Check through each key in the key list

foreach (Keys key in KeyList)

{

// Is the key currently down?

if (CurrentKeyboardState.IsKeyDown(key))

{

// Fire the OnKeyDown event

if (OnKeyDown != null)

OnKeyDown(this, new KeyboardEventArgs(key, CurrentKeyboardState, PrevKeyboardState));

}

// Has the key been released? (Was down and is now up)

if (PrevKeyboardState.IsKeyDown(key) && CurrentKeyboardState.IsKeyUp(key))

{

// Fire the OnKeyUp event

if (OnKeyUp != null)

OnKeyUp(this, new KeyboardEventArgs(key, CurrentKeyboardState, PrevKeyboardState));

}

}

}

The previous keyboard state is used to check whether a button has been released. This is particularly useful for *toggling* actions.

You might notice that there isn’t any code to check for “pressed” states. **See if you can add this code yourself now**.

# Binding Input to a Game Event

Now that we’ve created something to send input events, we need something to *receive* those events. We’ll make a new class that will manage all the different keyboard bindings and fire the appropriate game actions when a key is pressed.

**Add a new C# class to the project (Add > Class). Call it *“CommandManager.cs”, t*hen add the following *using* statements at the top of the class (as you did above):**

using Microsoft.Xna.Framework;  
using Microsoft.Xna.Framework.Input;

**Add the following code:**

public class CommandManager

{

private InputListener m\_Input;

public CommandManager()

{

m\_Input = new InputListener();

// Register events with the input listener

m\_Input.OnKeyDown += this.OnKeyDown;

m\_Input.OnKeyPressed += this.OnKeyPressed;

m\_Input.OnKeyUp += this.OnKeyUp;

}

public void Update()

{

// Update polling input listener, everything else is handled by events

m\_Input.Update();

}

public void OnKeyDown(object sender, KeyboardEventArgs e)

{

}

public void OnKeyUp(object sender, KeyboardEventArgs e)

{

}

public void OnKeyPressed(object sender, KeyboardEventArgs e)

{

}

}

In the constructor, we’re initialising the *InputListener* class and registering the delegate functions *OnKeyDown, OnKeyUp* and *OnKeyPressed* with it. The default implementation for any delegate that has an *EventHandler* type is:

public void Delegate(object sender, EventArgs e);

Where the *sender* is the object that sent the event and *e* is any arguments the sender wishes to pass on. Other than that, the *CommandManager* class is pretty straightforward.

Now that we have something to receive the events, we need a method of executing game actions. We’d like to keep this as generic as possible so that different input devices can trigger the same action without having to change anything but the binding.

**Outside the *CommandManager* class (but still in the CommandManager.cs file and inside the namespace) add the line:**

public delegate void GameAction(eButtonState buttonState, Vector2 amount);

The first parameter of the delegate uses the *eButtonState* enum (provided in *ButtonState.cs*). Every button (on most input devices) can generate 3 possible events: Up, Down and Pressed. This parameter submits which event the button has generated, so that the function can make appropriate use of it.

The second parameter is useful for one-dimensional analogue inputs (e.g. triggers) and two dimensional inputs (such as thumbsticks or mice), providing the action with how much a trigger was pressed, or the direction a thumbstick was moved.

We can now complete our *CommandManager* by adding code to handle *GameAction*s. First, we will add a Dictionary class to store the bindings from key’s to the GameAction call-back functions. Look up the Dictionary class in the MSDN documentation if you’d like to know how it works.

**Add the following field to the CommandManager class:**

private Dictionary<Keys, GameAction> m\_KeyBindings = new Dictionary<Keys, GameAction>();

Next, we’ll need to add a method to the CommandManager that will allow us to store Dictionary entries, associating a particular key with a GameAction call-back function. Each key binding is added to a *Dictionary* so that we can look up the *GameAction* easily by making use of the *Keys* enum. We also add the key to the *InputListener* to make sure we check for it.

**To do this, add this method to the class:**

public void AddKeyboardBinding(Keys key, GameAction action)

{

// Add key to listen for when polling

m\_Input.AddKey(key);

// Add the binding to the command map

m\_KeyBindings.Add(key, action);

}

Finally, we need to complete the OnKeyDown method, so that it will trigger the call-back if one has been added.

**To do this, *add* the code in bold to fill out the *OnKeyDown()* method:**

public void OnKeyDown(object sender, KeyboardEventArgs e)

{

**GameAction action = m\_KeyBindings[e.Key];**

**if (action != null)**

**{**

**action(eButtonState.DOWN, new Vector2(1.0f));**

**}**

}

All the *OnKeyDown* function does is check whether there is an action for the key that triggered an event and, if there is, it fires it. Notice the parameters passed into the delegate.

**Now try to fill out the *OnKeyUp()* and *OnKeyPressed()* methods yourself. These will be very similar to the code for OnKeyDown above, but you will need to change the eButtonState in the method in each case.**

# Putting it all together – Adding Inputs to the Game

Almost there! We’ve got the foundations covered and we can finally add inputs to the game.

**Open up *ChaseCameraGame.cs* and we’ll add in our first input. First, add a *CommandManager* field to the class and initialize it in the constructor. Then, add a call to *CommandManager.Update()* at the top of the *ChaseCameraGame.Update()* function (do this before the *ship.Update()* call).**

We’ve now added the command manager, and the call to *Update()* means that it will be ready to fire keyboard events whenever they are received. Now we need to register some keys that we’re interested in and associate a call-back function with each. We’ll show how this is done by associating the Escape key with the *StopGame* call-back.

**Add the line in bold to *InitializeBindings() in ChaseCameraGame*:**

private void InitializeBindings()

{

**commandManager.AddKeyboardBinding(Keys.Escape, StopGame);**

}

That’s it! A single line and our game will now listen for the escape key.

**Compile and run the sample**. **When you press escape, the game should exit successfully.**

Our *GameAction* parameter here has been set to *StopGame*. This function has been provided in the class. Let’s take a look at the declaration:

public void StopGame(eButtonState buttonState, Vector2 amount)

Does this look familiar? It has the same *signature* as our *GameAction* delegate. As long as the function we want to execute has the same signature as the delegate, we can pass it in to the *AddKeyboardBinding()* function.

**Try adding a function that will enable/disable the camera spring behaviour and add a binding for it. The *ChasingGameCamera* class has a Boolean field called *cameraSpringEnabled*. You will need to add an EnableCameraSpring call-back function (similar to the one for StopGame above, but check for PRESSED rather than DOWN). Then add this to the InitializeBindings method, with an appropriate key association.**

**Test your code. If it is working correctly, the “Camera Spring” text should toggle between ON and OFF each time you press the key.**

Now let’s get the ship to move around.

**Add the following code into the *Ship* class (found in “*Ship.cs”*):**

public void Thrust(eButtonState buttonState, Vector2 amount)

{

if (buttonState == eButtonState.DOWN)

{

ThrustAmount = 1.0f;

}

else

{

ThrustAmount = 0.0f;

}

}

public void TurnLeft(eButtonState buttonState, Vector2 amount)

{

if (buttonState == eButtonState.DOWN)

{

RotationAmount.X = 1.0f;

}

}

public void TurnRight(eButtonState buttonState, Vector2 amount)

{

if (buttonState == eButtonState.DOWN)

{

RotationAmount.X = -1.0f;

}

}

These are the call-backs we’re going to use to control the ship.

**Add the bindings for each of these actions in *InitializeBindings()* in the *ChaseCameraGame* class. For example, the Thrust() call-back could have the binding:**

commandManager.AddKeyboardBinding(Keys.W, ship.Thrust);

**Run the sample now and you should be able to control the ship using the key bindings you’ve chosen.**

# Extra challenge: Add a Mouse Handler

Warning: the following is quite a “challenging” challenge. It will be excellent practice to attempt the code yourself, but if you get stuck, the completed code can be found in a zip file on Moodle (you could still follow the steps below to examine the changes).

**Examining the code, you will notice that *MouseEventArgs* has been provided for you. Try to use the knowledge gained from this tutorial to incorporate mouse events into the sample in a similar way to the use of *KeyboardEventArgs* for keyboard events. See if you can get the ship to thrust forward with the keyboard and steer it using the mouse X-position when the first mouse button is pressed.**

**Steps:**

1. **Add a new Dictionary to the Command Manager for MouseButtonBindings (call it *m\_MouseButtonBindings*). You can use the same GameAction class to respond to mouse actions. Use MouseButton in place of Keys in the dictionary.**
2. **Add a new AddMouseBinding method, with similar code to AddKeyboardBinding. You will need to use *m\_MouseButtonBindings* instead of *m\_KeyBindings*. You will also need to replace AddKey with AddButton (which we’ll add to the InputListener in the next step).**
3. **Add a method called “AddButton” to the InputListener. This should be similar to the AddKey method, but should add a Button to a ButtonList. You will also need to add the ButtonList member (which is similar to the KeyList.**
4. **Add an OnMouseButtonDown method to the CommandManager.**
   1. **See what code needs to be added by using “Edit > Find and Replace > Find in Files” to search through the code for instances of OnKeyDown.**
   2. **One of these will be to add an OnMouseButtonDown method to CommandManager. This should be similar to OnKeyDown, but replace the KeyboardEventArgs with MouseEventArgs. Use the m\_MouseButtonBindings member. You can get the current mouse position using e.CurrentState.x and e.CurrentState.y (use this in the Vector2 for the action).**
   3. **Once you’ve created the method OnMouseButtonDown, add it to the delegates in the CommandManager constructor. You will need to add the OnMouseButtonDown event handler to InputListener (similar to OnKeyDown).**
   4. **Search for OnKeyDown again. Another of the instances will be in the FireKeyboardEvents() method. Add a similar method called FireMouseEvents():**
      1. **Your FireMouseEvents() method will need to access a mouse button list. You should have already created the list in step 3 above. Now add the code for your ButtonList by finding places where KeyList is used.**
      2. **It will also need a CurrentMouseState and PrevMouseState. Add these to the InputListener in a similar way to the PrevKeyboardState is and CurrentKeyboardState. Add these to the InputListener constructor and the Update function.**
      3. **We will only be responding to the LeftMouseButton. You can iterate through the MouseButton list, but use an “if” statement to check for the left mouse button; and then use if (CurrentMouseState.LeftButton == ButtonState.Pressed) to check if we need to fire the action.**
      4. **Add FireMouseEvents() to the Update method of InputListener ;**
5. **Add an AddMouseBinding line to the InitializeBindings method, with a call-back to ship.MouseTurn for MouseButton.LEFT.**
6. **Add the ship.MouseTurn method to ship, similar to the other turn methods, but this time use the x dimension of the amount parameter to change the x rotation. The x position in the amount parameter will be in screen dimensions. You can normalise this using:**

graphicsDevice.Viewport.Width;

***Note: this is a slightly messy way of doing things. We’d prefer the ship actions to be less coupled to the mouse, so you might want to come back and re-write this with the normalisation performed in the generation of the action.***

# Final notes:

In this tutorial, we’ve seen how we can make use of C#’s events and delegates to create an event-driven architecture. This certainly isn’t limited to input devices though. An event-driven architecture has numerous applications, such as controlling NPCs and collision detection (which we will see next week). The tutorial code above is still a little bit more tied to the use of keys than it could be, and the code for the extra challenge is even more explicitly tied to the mouse. We’d really like to have game events that are higher level than this. You should have an idea of how to do this now, but don’t worry, we’ll come back to it in later lectures.

**You may want to explore this tutorial in more detail to see how XNA can be used to render 3D objects.**

**Also take a look at the original sample and note the filters that can be used to find further XNA samples.**