# Introduction

The components this guide covers are the Input Manager and Event Listener.

The Event Listener and Input Manager classes are set up initially with standard a keyboard, mouse, and XBox controller.

These classes use SDL2 for input extraction. Your project must be setup to run SDL2 with its dependencies to use this component.

The classes can very easily be expanded upon, to add more events and inputs. Adding custom inputs is not limited to SDL2.

The option of changing input extraction from SDL2 is also possible, though a new input would need to be provided for each Event you wanted to change.

The Command pattern is used to utilise custom commands.

Custom Commands are objects with an overloaded method to apply custom functions.

# How to use this component:

### **How to make a Custom Commands / How they work**

Custom Commands are object which take in a function pointer. [This allows you to encapsulate a request as an object, thereby letting users parameterize clients with different requests, queue or log requests, and support undoable operations](http://gameprogrammingpatterns.com/command.html) - gameprogrampatterns.com.

The command class is as follows:

class Command

{

public:

Command(std::function<void()> function) : m\_function(function) {}

Command(std::function<void()> function, EventListener::Type type) : m\_function(function), m\_type(type) { m\_functions.push\_back(m\_function); }

std::function<void()> m\_function;

std::vector<std::function<void()>> m\_functions;

virtual ~Command() {}

virtual void execute(){};

virtual void executePress() {};

virtual void executeRelease() {};

virtual void executeHold() {};

};

As you can see, it takes in a function pointer. This allows you to send in any function, which you then call when overriding the execute class. This is used for all inputs. You can send in a function, and a key event type, which can execute on a press, release, or a hold.

An override command object should look something like this for a custom event:

class OverRideCommand : public Command

{

public:

OverRideCommand(std::function<void()> function) : Command(function) {}

virtual void execute()

{

m\_function();

}

};

An override command object which takes in a type should look something like this.

class OverRideCommandType : public Command

{

public:

OverRideCommandType(std::function<void()> function, EventListener::Type type) : Command(function, type) {}

virtual void executePress()

{

for(int i = 0; m\_type == EventListener::Type::Press && i < m\_functions.size(); i++)

m\_functions[i]();

}

virtual void executeRelease()

{

for(int i = 0; m\_type == EventListener::Type::Release && i < m\_functions.size(); i++)

m\_functions[i]();

}

virtual void executeHold()

{

for(int i = 0; m\_type == EventListener::Type::Hold && i < m\_functions.size(); i++)

m\_functions[i]();

}

};

This will run the function you sent in to the constructor. However, you can add more inside the execute function for each object you make, and it will run when called.

You should make a custom command for every desired action, not every desired key. Since these are inherited, you can make them anywhere you have included FLInputManager.h.

A new command for the AddKey() function should look as follows:

EventListener::Event exampleEvent= EventListener::Event::SPACE;

Command\* example= new OverRideCommandType (std::bind(&ExampleClass::exampleFunction, exampleClassObject), EventListener::Type::Press);

inputManager->AddKey(exampleEvent, example, this);

this is an EventListener\* (See below).

### **How to use Custom Commands / The Input Manager**

First, add the FLInputManager .h and .cpp to your project.

In the .h of the classes you want to process inputs in (e.g. Game, menus), inherit from the EventListener class.

E.g. class Game : public EventListener

Add #include "FLinputManager.h"

Add InputManager\* inputManager = InputManager::getInstance();

Now, when you initialise your class, you are going to want to add your key, and specify your key command and function in the .cpp file initialisation.

inputManager->AddKey(EventListener::Event, Command\*, EventListener);

The listener is done for you. You just need to add a Command, which takes in a function pointer, and a key input type. This requires binding a class function, and a class object (see above).

(You can expand these events and command keys further from what is readily available if required.)

To process the input for the listener, you will need to add this to your update loop:

inputManager->ProcessInput();

This will execute commands based on desired key press when the dispatch event is called. This is all done internally.

The purpose of custom commands is you always call this key, but you can change the function of that key elsewhere, and it will work.

This can easily be applied to changing key bindings. This will allow you to use the same key across different classes, having the same, or completely different functions for each class.

### **How to make a Custom Events**

The Event Listener is as simple as:

class EventListener

{

public:

enum Event {

DesiredEvent1,

DesiredEvent2,

...

};

virtual void onEvent(Event)=0;

};

You can expand upon this as much as you want. The standard keyboard set up provided, is linked to characters, so if you want to hard code an enum value, you will need to make sure it is individual.

The following is how listeners are added and called, this does not need to be touched:

void InputManager::AddListener(EventListener::Event evt, EventListener \*listener)

{

if (listeners.find(evt) == listeners.end())

{

//event not in map add it.

listeners[evt] = new std::vector<EventListener\*>();

}

listeners[evt]->push\_back(listener);

}

//Send the events to interested objects

void InputManager::Dispatch(EventListener::Event evt)

{

if (listeners.find(evt) != listeners.end())

{

//go through all listeners for this event

for (auto const &listener : \*listeners[evt])

{

listener->onEvent(evt); //Call on event for the listener

}

}

}

From here you override the onEvent function, to listen for inputs, and produce the desired

List of Functions available in Input Manager

* InputManager::GetInstance(); (Used to set up object)
* ProcessInput(); (Used to update input)
* AddListener(EventListener::Event, EventListener\*);
* void AddKey(EventListener::Event, Command\*, EventListener\*);
* void ResetKey(EventListener::Event);
* void SetControllerButtonDelay(int delayInMilliseconds); (Set Delay for controller buttons)
* void SetControllerTriggerDelay(int delayInMilliseconds); (Set Delay for controller triggers)
* void SetStickDeadZone(int deadZone);
* Vector2f GetLeftStickVector();
* Vector2f GetLeftStickVectorNormal();
* float GetLeftStickAngle();
* float GetLeftTrigger();
* Vector2f GetRightStickVector();
* Vector2f GetRightStickVectorNormal();
* float GetRightStickAngle();
* float GetRightTrigger();
* void saveFile(); (Use when closing application)
* std::string GetTimeStamp(bool save);

# Demonstration of this component

This section describes the features of an application which demonstrates the feature set of this component.

## **Demo Game Feature**

* Press a key/ mouse button/ controller button
* Hold a key / controller button
* Release a key / mouse button/ controller button
* Switch between key bindings.
* Use of controller sticks / triggers.

## **Demo Game User Guide**

Press 1 to Bind keys and do the following:

* Press the W/A/S/D keys to move a player object Up/Left/Down/Right respectively.
* Press the right mouse button on the player to change its colour.
* Hold the D-Pad to move the player.

Press 2 to Bind Keys and do the following:

* Release the up/left/down/right keys to move in those directions
* Press the Left Mouse Button to reset the colour.
* Use the scroll to move the player up and down.

Press 3 to Bind sticks and triggers:

* Move Left stick to move the player after pressing V, press B to make the deadzone bigger.
* Move right stick to set the player’s rotation.
* Hold the Triggers to rotate the player each direction.

Press 0 to clear all commands.