

Gerard Marcos Freixas

---

# Input Combos

# INDEX

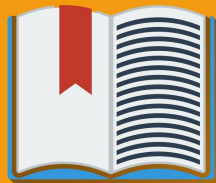
1. Introduction
  - 1.1. Presentation blocks
  - 1.2. What is a combo?
2. Block I: Deeping into the subject
  - 2.1. Combo differentiation
    - 2.1.1. Time dependency
    - 2.1.2. Immediate and holding input
    - 2.1.3. Degree of freedom
    - 2.1.4. Definiton & input scheme
3. Block II: Code structure
  - 3.1. Input storage
  - 3.2. Memory management
  - 3.3. The implementation in-depth
4. Code exercises
5. Extra concepts



# 1. INTRODUCTION

# 1.1. Presentation blocks

Block I: Theory



Block II: Code



### 1.2. What is a combo?

Definition by wikipedia:

**Combo:** set of actions performed in sequence, usually  
with strict timing limitations, that yield a significant benefit  
or advantage.



Do you **agree** or **disagree**?

## 2. BLOCK I: DEEPING INTO THE SUBJECT



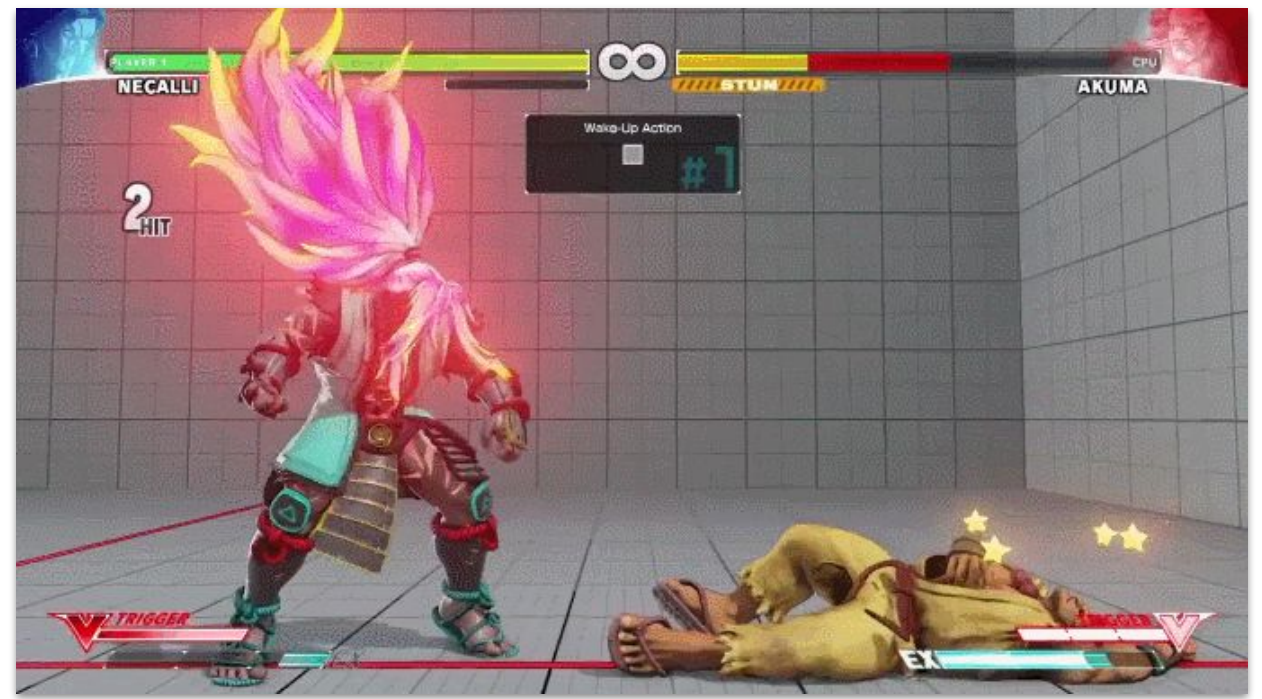
### 2.1. Combo differentiation

Notice the difference?

1)



2)





### 2.1. Combo differentiation

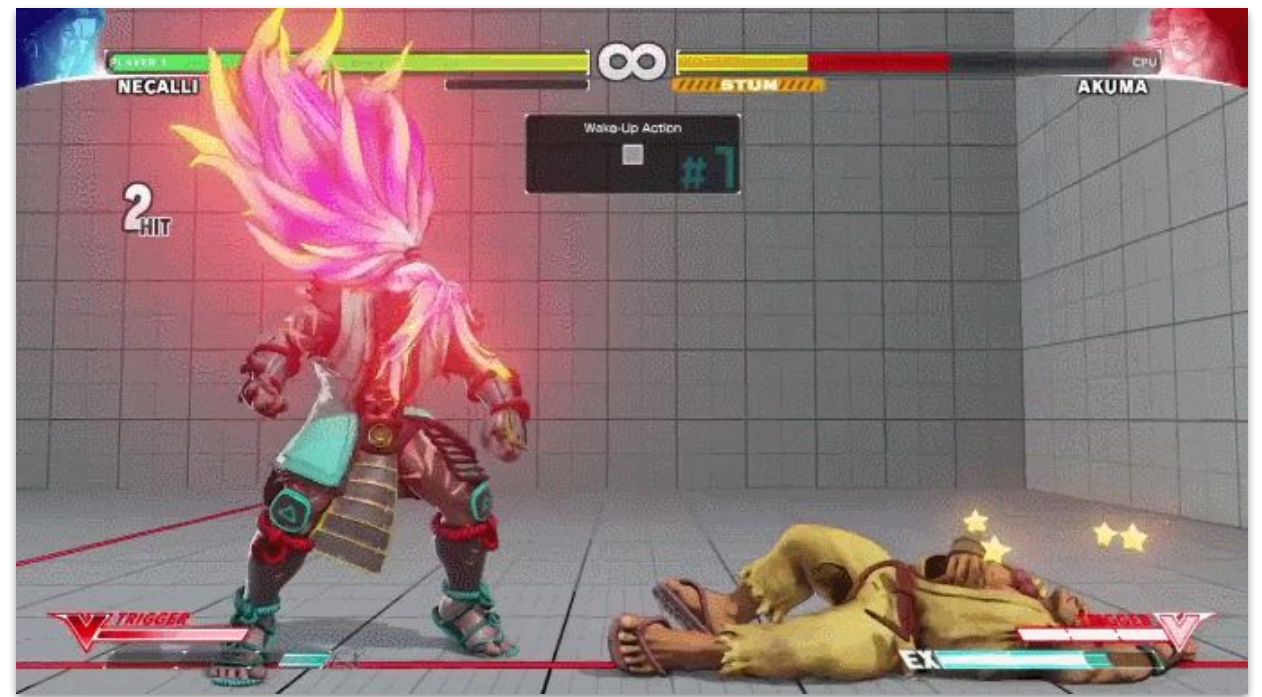
**First** main factor: the time between inputs



1) Use of **timing**: **NO**



2) Use of **timing**: **YES**





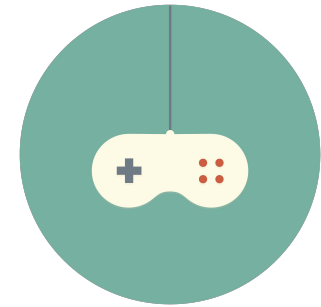
### 2.1.1. Time dependency

- Time-independent
  - **Single input**
- Time-dependent
  - **Single input**
  - **Parallel input**
  - **Single & Parallel input**



### 2.1.2. Immediate and holding input

**Second** main factor: the time pressing a key/button



- Button pressed

- Immediate



+ Frenzy, Action

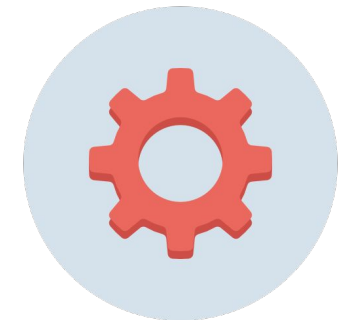
- Holding



- Frenzy, Action

## 2.1.3. Degree of freedom

**Third** main factor: the freedom of the inputs chain



- Input degrees of freedom

- **Defined**



**Determined chains**

**Checked chains**

- **Undefined**



~~**Determined chains**~~

~~**Checked chains**~~

### 2.1.4. Definition & input scheme (I)

#### The ideal definition:

**Combo**: set of actions performed in sequence, with or without strict timing limitations, that yield a significant benefit or advantage.



## 2.1.4. Definition & input scheme (II)

**Think about your type of input(s) you want in your videogame.**

### Combo

Time-independent

Time-dependent

Single input

Single input   Parallel input   S & P input

Immediate button pressed

Holding button pressed

Defined freedom

Undefined freedom

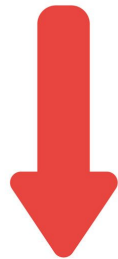


## 2. BLOCK II: CODE STRUCTURE

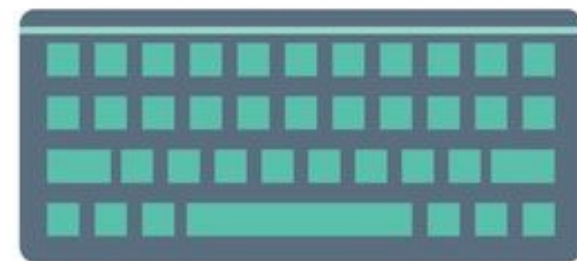


### 3.1. Input storage (I)

**Keep in mind:** we need to leave in evidence the buttons/keys we press!



Input tracking



### 3.1. Input storage (II)

#### Use of containers.

- **Vector** → Store the combos input events and the combos.
- **List** → Acts like a buffer. We store the actual user input events.
- **Queue** ✗
- **Tree** ✗

### 3.1. Input storage (III)

The combos need to be previously defined by us. Set up by xml.



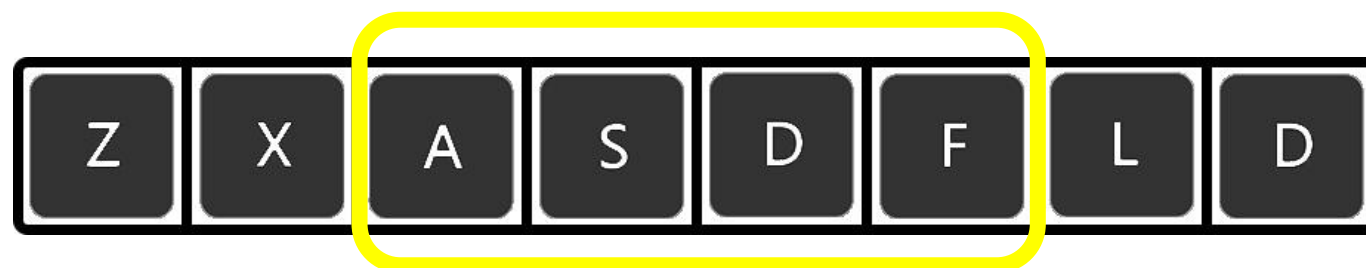
We will check if the array of the combo exists in our current buffer.

## 3.1. Input storage (IV)

Sample defined combo



Buffer #1



Buffer #2





## 3.2. Memory management (I)

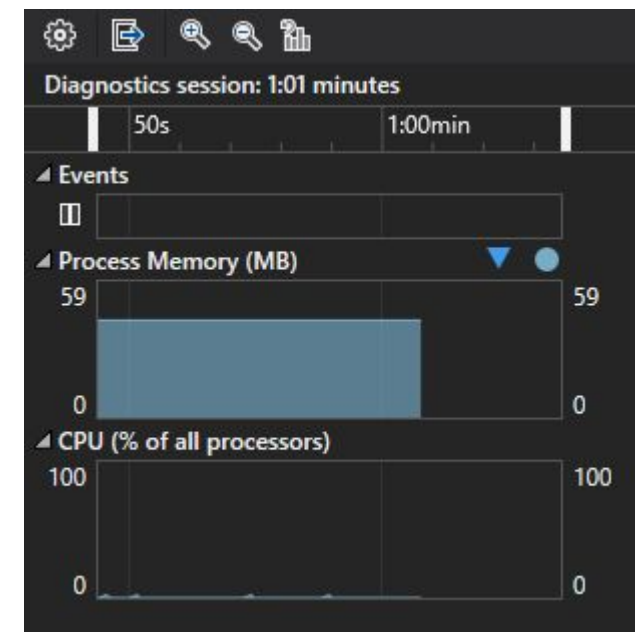


Note: dynamic memory!

We can't really save all the inputs during all the time.



We need some rules to know when to delete them.



## 3.2. Memory management (II)

**When we will delete the input events in the buffer?**

- When a single input event lasts more than the maximum time allowed.



**Delete this event**

- When the user completes a combo.



**Delete all the buffer elements**

### 3.3. The implementation in-depth (I)

**Application module**  **j1InputCombos**

**Main elements**

- `std::list<InputEvent*> user_input_events;`



Buffer

- `std::vector<InputCombo*> combos;`

# 3.3. The implementation in-depth (II)

## InputEvent class

### Main elements

- `j1PerfTimer timer;`



Crucial role in the class constructor!

- `CUSTOM_EVENT_TYPE type;`

```
InputEvent(CUSTOM_EVENT_TYPE type, bool use_of_timer = false);
```



Only use in the buffer

# 3.3. The implementation in-depth (III)

## ComboEvent class

### Main elements

- `std::vector<InputEvent*> vector_item;`

- `COMBO_NAME combo_name;`



## 4. CODE EXERCISES

```
if (we_code_it_ourselves)
{
    LOG("We learn effectively");
    learning.SetValue(100);
}
else
    learning.SetValue(10);
```

# TODO 1

**“Create another combo. Place any inputs you want! Guess what the attribute “name” of the combo should be. Investigate through the code for some clues. We can add another input\_event type called “kick”.”**

**Location:** Input\_Combos/Config/config.xml

# TODO 1 - Solution

```
<input_combos>
  <combo name="hadouken">
    <input_event type="down"/>
    <input_event type="right"/>
    <input_event type="punch"/>
  </combo>
  <combo name="shoryuken">
    <input_event type="left"/>
    <input_event type="up"/>
    <input_event type="down"/>
    <input_event type="kick"/>
  </combo>
</input_combos>
```

# TODO 2

**“We need to fill every combo creation. Think about where we can push every input event to the combo. Hint -> we need to call a function located in InputCombo.h/.cpp. Notice another hint in the code.”**

**Location:** Input\_Combos/Modules/j1InputCombos.cpp

# TODO 2 - Solution

```
for (pugi::xml_node inp = comb.child("input_event"); inp; inp = inp.next_sibling("input_event"))
{
    std::string event_name;
    event_name.assign(inp.attribute("type").as_string());

    InputEvent::CUSTOM_EVENT_TYPE event_type = InputEvent::CUSTOM_EVENT_TYPE::UNKNOWN;

    if (event_name == "left")
        event_type = InputEvent::CUSTOM_EVENT_TYPE::LEFT;
    else if (event_name == "up")
        event_type = InputEvent::CUSTOM_EVENT_TYPE::UP;
    else if (event_name == "right")
        event_type = InputEvent::CUSTOM_EVENT_TYPE::RIGHT;
    else if (event_name == "down")
        event_type = InputEvent::CUSTOM_EVENT_TYPE::DOWN;
    else if (event_name == "punch")
        event_type = InputEvent::CUSTOM_EVENT_TYPE::PUNCH;
    else if (event_name == "kick")
        event_type = InputEvent::CUSTOM_EVENT_TYPE::KICK;

    InputEvent* input_event = new InputEvent(event_type);
    combo->FillComboChain(input_event);
}
```



# TODO 3

**“We need to check and choose what keys we press (depending on the config.xml). If we press one of these keys, push this event to the user\_input\_events std::list. Hint -> Usage of ReturnEvent(parameter) function.”**

**Location:** Input\_Combos/Modules/j1InputCombos.cpp

# TODO 3 - Solution

```
if (App->input->GetKey(SDL_SCANCODE_LEFT) == KEY_DOWN)
{
    user_input_events.push_back(ReturnEvent(InputEvent::CUSTOM_EVENT_TYPE::LEFT));
}
if (App->input->GetKey(SDL_SCANCODE_UP) == KEY_DOWN)
{
    user_input_events.push_back(ReturnEvent(InputEvent::CUSTOM_EVENT_TYPE::UP));
}
if (App->input->GetKey(SDL_SCANCODE_RIGHT) == KEY_DOWN)
{
    user_input_events.push_back(ReturnEvent(InputEvent::CUSTOM_EVENT_TYPE::RIGHT));
}
if (App->input->GetKey(SDL_SCANCODE_DOWN) == KEY_DOWN)
{
    user_input_events.push_back(ReturnEvent(InputEvent::CUSTOM_EVENT_TYPE::DOWN));
}
if (App->input->GetKey(SDL_SCANCODE_Z) == KEY_DOWN)
{
    user_input_events.push_back(ReturnEvent(InputEvent::CUSTOM_EVENT_TYPE::PUNCH));
}
if (App->input->GetKey(SDL_SCANCODE_X) == KEY_DOWN)
{
    user_input_events.push_back(ReturnEvent(InputEvent::CUSTOM_EVENT_TYPE::KICK));
}
```

# TODO 4

**"First of all, uncomment the following block of code. This function deletes any input event that there is in the buffer Think about what should be the condition. Also, we need to place another line of code inside the condition block \*. Hint #1 -> Remember what I said about the buffer? (What was the condition to delete an element). Hint #2-> There is a macro you need somewhere..."**

**Location:** Input\_Combos/Modules/j1InputCombos.cpp

# TODO 4 - Solution

```
if (user_input_events.size() > 0)
{
    std::list<InputEvent*>::const_iterator item = user_input_events.begin();
    while (item != user_input_events.end())
    {
        if ((*item)->timer.ReadMs() > INPUT_MAX_TIME)
        {
            delete *item;
            item = user_input_events.erase(item);
        }
        else
        {
            ++item;
        }
    }
}
```

# TODO 5

**“Set the player animation. There are two lines of code you need to place somewhere here... No hints this time.”**

**Location:** Input\_Combos/Modules/j1InputCombos.cpp

# TODO 5 - Solution

```
if (user_input_events.size() > 0)
{
    for (std::vector<InputCombo*>::const_iterator item = combos.begin(); item != combos.end(); item++)
    {
        if ((*item)->CheckCommonInput(user_input_events))
        {
            InputCombo::COMBO_NAME temp_name = (*item)->combo_name;
            switch (temp_name)
            {
                case InputCombo::COMBO_NAME::HADOUKEN:
                    App->scene->SetPlayerAnimation(j1Scene::PLAYER_STATE::STATE_HADOUKEN);
                    break;
                case InputCombo::COMBO_NAME::SHORYUKEN:
                    App->scene->SetPlayerAnimation(j1Scene::PLAYER_STATE::STATE_SHORYUKEN);
                    break;
                default:
                    break;
            }

            ClearInputBuffer();

            break;
        }
    }
}
```

# TODO 6

**“Guess what we need to do here. Simple function call!”**

**Location:** Input\_Combos/Modules/j1InputCombos.cpp

# TODO 6 - Solution

```
ClearInputBuffer();
```



# Homework !

Investigate thoroughly all the code I implemented. See the relations between the scene and the module. Play with the classes I created, called:

- User\_Input/InputEvent.h
- User\_Input/InputCombo.h
- Modules/j1InputCombos.h



## 5. EXTRA CONCEPTS

# Animation canceling & actions or colliders (I)

In our code we can cancel our animations by using another one. Good example in League of Legends:



# Animation canceling & actions or colliders (II)

Animation cancelling is very rewarding but also very risky. Watch out for the actions and colliders!  
Another example:



# Improvements

- Controllers
- Colliders & actions
- Overall structure



Thank you for your  
attention