COMPLETE CONTROL

Streamlined Input for GameMaker 2022+
Version 1.0

Features Overview

- Create complex key configurations easily
- Support alternate keys for the same actions across keyboard, mouse, and gamepad
    - example: WASD, Arrows, dpad, and analog stick for movement
- Detect held, pressed, released, double tap, rapid fire, and more
- Use any input that GM Supports: gamepad, mouse buttons, analog sticks, and more
- Access controls easily without the need of a global controller object

Your First Complete Control Configuration

What You Are Probably Doing Now…

Let's quickly walk through a very basic example of how to get Complete Control working in your game.

Let's start with what should look pretty familiar. You've got a top down action game. You can use the arrow keys to move and the space bar to attack. You can also use WASD to move and the left mouse button to attack. Additionally, you support gamepads! So you can move with the analog stick as well as the dpad, and attack with the A button.
Whew, that sounds like a nightmare! Let's see what that might look like (for brevity's sake, I'm going to only show attack and left:

```
attack = keyboard_check_pressed(vk_space) ||
         mouse_check_button_pressed(mb_left) ||
         gamepad_button_check_pressed(0, gp_face1)

left = keyboard_check(vk_left) ||
       keyboard_check(ord("A")) ||
       gamepad_button_check(0, gp_padl) ||
       gamepad_axis_value(0, gp_axislh) < -.25
```

And that's before you even start worrying about rebindable controls! Yuck!

So let's use **Complete Control** to define two controller configs: one for gamepad and one for keyboard and mouse that supports all of the controls we wanted to support.

Define Your Keyboard and Gamepad Input Groups

First, let's set up our arrow keys and WASD. To do this we will use one of the built-in input constructors for defining a 4 directional input on the keyboard: ArrowKeys. Since we want to support two different ways to do the same thing (move in this case), we will create two ArrowKeys with our different sets of buttons and store them in an array:

```
moveKeyboard = [
  new ArrowKeys(ord("D"), ord("W"), ord("A"), ord("S")),
  new ArrowKeys(vk_right, vk_up, vk_left, vk_down)
]
```

When we create a new ArrowKeys struct, we just pass it the buttons we want to use for right, up, left, and down. The buttons are exactly what you would use for keyboard_check

Now let's do something similar for our gamepad controls. This time we'll use two different constructors: AnalogStick and DPad

```
moveGamepad = [
  new AnalogStick(gp_axislh, gp_axislv),
  new DPad(gp_padr, gp_padu, gp_padl, gp_padd)
]
```

The AnalogStick wants to know which two gp_axis you want to use: left horizontal and left vertical in our case. The DPad works exactly like the ArrowKeys: you just need to pass in the buttons you want to use for right, up, left, and down.

Finally, let's set up our attack buttons. We'll use 3 new constructors for the different input types: Key, MouseButton, and GamepadButton

```
21
22 attackKeyboard = [
23   new Key(vk_space),
24   new MouseButton(mb_left)
25 ]
26
27 attackGamepad = [
28   new GamepadButton(gp_face1)
29 ]
30
```

Build Your Config

Time to bring all this together. We need to take all of these different controls and "group" them logically under an Input. An Input represents a collection of buttons that all do the same "thing". In this example we have two inputs: Move and Attack. To build an Input is similar to what we just did before, we need to new up an Input struct and give it our buttons and wrap all of them in an array. This array represents our completed configuration.

```
30
31 config = [
32   new Input("move", InputType.directional, moveKeyboard, moveGamepad),
33   new Input("attack", InputType.button, attackKeyboard, attackGamepad),
34 ]
```

So for each **Input** we pass 4 arguments.

1. The first is the name. This is what we will be using to access this input later, so name it like you would a normal variable.
2. The type is either **InputType.directional** or **InputType.button**. This is mostly just to help **Complete Control** know if you've made a mistake or not.
3. Your array of buttons for the keyboard/mouse configuration.
4. Your array of buttons for the gamepad configuration.

With all that, we are ready to actually USE **Complete Control**. It's as easy as calling use_complete_control and passing your config as well as which controller slot we are listening to.

```
36 controls = use_complete_control(config, 0);
```

We'll use that controls variable whenever we want to do anything with our controls for this object. You can name it anyway you want (like input, keys, or whatever).

Reading Your Controls

Okay, yeah, that was a bit of an intense setup process, but I promise it will be worth it.

The next step is to update our controls. This is typically done in a step event.

```
1 controls.update();
```

Once you've called the update function, you are ready to read your controls!

```
3 moveDirection = controls.move.direction;
4 attack = controls.attack.pressed;
```

Since move is an **InputType.directional** it has a direction property that will quickly give you the direction the player is holding. Our attack input is an **InputType.button** so it has all the properties you would expect a button to have like pressed, held, released, and even doubleTapped. All of these available properties are detailed in a later section.

Summary

While the setup can be a bit overwhelming, once you get the hang of the pattern, it comes together very quickly, and using it almost couldn't be any easier. With a little practice, you'll soon be setting up complex configurations, alternate configurations, and maybe even allow for button rebinding! Everything is under your **COMPLETE CONTROL**!

Global Variable Defaults

At the top of use_complete_control are a number of global variables. These can be changed to whatever values you'd like, or you can leave them as is. Let's go through each and discuss what they are for.

`global.nullDirectionValue = undefined`

When using a directional input such as ArrowKeys or AnalogStick it can be important to know when no direction is currently being held. In these cases, the direction property will return this value. I recommend keeping this at undefined or a value less than 0 or greater than 360. You could get very bad results if you give it any number in the standard 0 - 360 range.

`global.defaultDoubleTap = 8`

This is the default delay for what counts as a double tap. You can customize this for each individual input if desired. The value represents how many frames can pass before the button is hit a second time to count as a doubleTap.

`global.defaultAnalogDeadzone = .25`

This is the default value for analog stick dead zones. You can customize this for each individual input if desired. The value represents how far the analog stick must be tilted before it is registered as being held at all. Making this a higher value will mean the analog stick must be held further to be detected.

`global.defaultButtonDeadzone = .5`

This is the default value for button dead zones. You can customize this for each individual input if desired. Some buttons support analog input; most notably the triggers on Xbox and Playstation controllers. This value will also be used when accessing any StickTilt input type. The value represents how far the button must be held before it is registered as being held or pressed at all. Making this a higher value will mean the button must be held further to be detected.

Controls Structure

`use_complete_control(config, [controllerSlot])`

```
36 controls = use_complete_control(config, 0);
```

This function returns a struct that is used to access any and all features of **Complete Control**.

Argument 0: an array of Input structs.

Argument 1: the controller slot to listen to. Default: -1 (no slot).

Once created, the Controls struct has some properties and functions for you to use.

`currentControllerType`

This property will contain a ControlType enum value. Either keyboard, mouse, or gamepad based on the last button pressed. This is helpful when you want to change button prompt icons or text based on the input the player is currently using. You often see this behavior on PC games when switching between Keyboard and Gamepad. More often than not, you'll only care if the value is currently ControlType.gamepad or not.

update( )

The most important function in the Controls structure. This must be called for the controls to be updated and read. I recommend calling it in the step or begin step events.

reset( )

This function can be called to reset any and all input back to default as if the controls had not been touched yet. This is primarily useful when you are using AI to update the controls and need to set everything back to default before updating the control state manually (covered in a future section).

Input Structure

```
30
31 config = [
32   new Input("move", InputType.directional, moveKeyboard, moveGamepad),
33   new Input("attack", InputType.button, attackKeyboard, attackGamepad),
34 ]
```

new Input(name, InputType, keyboardConfig, gamepadConfig)

This structure is used to define an input, typically named for an action that the input controls.

Argument 0: The name of your input as a string. This will be used to access the input's properties as if it was a normal variable on the struct. E.G. controls.attack.pressed. As such it is important to follow the same naming rules as any standard variable; case sensitive, no spaces or special characters.

Argument 1: The InputType; either directional or button. Represents the types of inputs collected in the config arrays passed in the next two arguments. This is primarily to help through helpful errors if you've done something **Complete Control** doesn't support.

Argument 2: An array of keyboard inputs. Supported types: ArrowKeys, Key, MouseButton, and MouseWheel. Do not combine ArrowKeys in the same array with any other supported type.

Argument 3: An array of gamepad inputs. Supported types: DPad, AnalogStick, GamepadButton, and StickTilt. Do not combine DPad or AnalogStick in the same array with any other supported type.

After you've defined your Controls struct by calling use_complete_control(), you'll be able to access any defined input by "dotting" into the controls struct and accessing it by the defined name.

```
moveDirection = controls.move.direction;
attack = controls.attack.pressed;
```

The properties under there will largely be determined by the type of input contained within. However there are some universal properties and functions you have access to.

`setRapidFire(minFrequency, maxFrequency, increment)`

This function can be set to enable rapid fire on an input.

Argument 0: The minimum number of steps between the button being reported as “pressed” when being held. If set to 5, for example, holding the button down will make the rapidPressed property report true a minimum of every 5 steps.

Argument 1: The maximum number of steps between the button being reported as “pressed” when being held. If set to 5, for example, holding the button down will make the rapidPressed property report true a maximum of every 5 steps.

Argument 2: Every time rapidPressed reports true, the number of steps until the next true will change based on this argument. Read below for a more detailed explanation of how to use this.

Rapid fire on a **Complete Control** Input can work in two different ways.

The first way is your standard rapid fire that hits the button every X frames at a consistent rhythm while held. To get this behavior, set Argument 0 and Argument 1 to the same value and Argument 2 to 0. An example of when you might use this behavior is in a space shoot ‘em up where you want holding the button down to press the fire button for the player.

The second way is to have the rate of button presses increase (or less commonly decrease) the longer the player holds the button down. To get this behavior, set Argument 0 and Argument 1 to the minimum and maximum press rate respectively and Argument 2 to the amount you want the length of time between each press to change.

For example, I may have a volume slider in my options menu. The longer the player holds down the same direction, I want to speed up how quickly the volume changes. To do this I would call:

```
changeVolume.setRapidFire(1, 30, 1)
```

As the player begins holding down the right button, it will take 30 steps to change the volume by 1. Then 29 steps to change it again, then 28 steps, then 27, until eventually changing it every single step.

Final note: setRapidFire can be called when initially creating the input itself like so:

```
config = [
  new Input("move", InputType.directional, moveKeyboard, moveGamepad)
  new Input("attack", InputType.button, attackKeyboard, attackGamepad).setRapidFire(10,10,0)
]
```

setDirection([direction], [percent])

It is possible to force a directional input type to report a specific direction being held. This is especially useful for using AI to control an object.

Argument 0: The direction to hold. Must be a value greater than or equal to 0 and less than 360. Defaults to the global.nullDirectionValue.

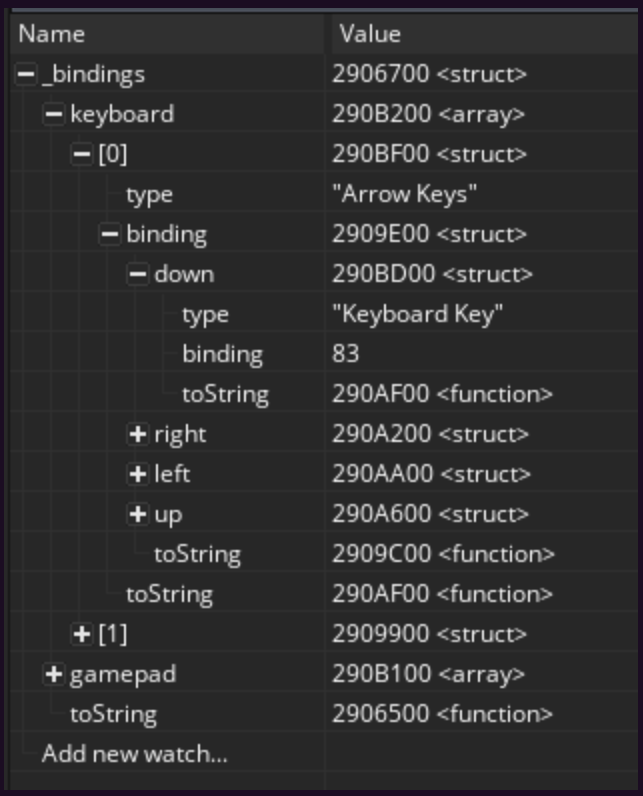
Argument 1: This argument represents how “far” the input is being held. Relevant for AI that can utilize analog movement. Defaults to 1, or 100%.

setHeld(), setPressed(), setReleased(), & setDoubleTapped()

These functions can force a button input type to report true for the relevant properties. Again, useful for AI that can control an object.

getBindings() *EXPERIMENTAL*

A utility function that is capable of returning the bindings for the Input. Not thoroughly tested, and likely needs some more features added to it, but this function will return a complicated structure detailing the bindings. It looks something like this:



Directional Inputs (InputType.directional)

When accessing any InputType.direction input, you'll have access to many properties to determine the state of the input. Let's go over them.

`any`

Contains true or false based on if any direction is currently held on this Input.

`hori`

Contains a value between 1 and -1. 1 is right, 0 is neither left nor right, and -1 is left. Will contain non-whole numbers of an AnalogStick type and will represent the raw value not adjusted for deadzone.

`vert`

Contains a value between 1 and -1. 1 is down, 0 is neither up nor down, and -1 is up. Will contain non-whole numbers of an AnalogStick type and will represent the raw value not adjusted for deadzone.

direction

Contains a value greater than or equal to 0 and less than 360 if any direction is held down. Will contain global.nullDirectionValue if no direction is held down.

percent

Contains the percentage that a direction is held on the Input. Should be either 0 or 1 in all cases except when an AnalogStick type is involved. In which case the percent is scaled by the deadzone value. If the deadzone is .25 and the analog stick is held 25% of the way, percent will be 0,

deadzone

Contains the currently configured deadzone percentage. Should be a value between 0 and 1. Only really relevant when an AnalogStick type is involved.

Furthermore, each Directional Input contains a property for right, up, left, and down. What that value looks like will depend on the type of input.

ArrowKeys

```
moveKeyboard = [
  new ArrowKeys(ord("D"), ord("W"), ord("A"), ord("S")),
  new ArrowKeys(vk_right, vk_up, vk_left, vk_down)
]
```

new ArrowKeys(right, up, left, down, [doubleTap], [deadZone])

Creates a new ArrowKeys struct. This struct should be passed to an Input as part of the **keyboard** bindings array argument for an InputType.directional input.

Argument 0: The keyboard key code for right.

Argument 1: The keyboard key code for up.

Argument 2: The keyboard key code for left.

Argument 3: The keyboard key code for down.

Argument 4: The currently configured number of steps a key must be tapped twice to count as a doubleTap. Default `global.defaultDoubleTap`

Argument 5: The deadzone for the keys. This may be completely useless? I honestly forgot why I had this at all, but I'm sure there's a good reason. You probably don't need to set it to anything other than the default, which is `global.defaultButtonDeadzone`.

`right, up, left, down`

`Each contains a Key struct with all relevant values.`

DPad

```
14 moveGamepad = [
15   new DPad(gp_padr, gp_padu, gp_padl, gp_padd)
16 ]
```

`new DPad(right, up, left, down, [doubleTap], [deadZone])`

Creates a new DPad struct. This struct should be passed to an Input as part of the **gamepad** bindings array argument for an InputType.directional input.

Argument 0: The gamepad button code for right.

Argument 1: The gamepad button code for up.

Argument 2: The gamepad button code for left.

Argument 3: The gamepad button code for down.

Argument 4: The currently configured number of steps a button must be tapped twice to count as a doubleTap. Default `global.defaultDoubleTap`

Argument 5: The deadzone for the buttons. This may be completely useless? I honestly forgot why I had this at all, but I'm sure there's a good reason. You probably don't need to set it to anything other than the default, which is `global.defaultButtonDeadzone`.

`right, up, left, down`

`Each contains a GamepadButton struct with all relevant values.`

AnalogStick

```
14  moveGamepad = [
15    new AnalogStick(gp_axislh, gp_axislv)
16  ]
```

`new AnalogStick(hori axis, vert axis, [cardinalSnap] [doubleTap], [deadZone])`

Creates a new DPad struct. This struct should be passed to an Input as part of the **gamepad** bindings array argument for an InputType.directional input.

Argument 0: The gp_axis code for left and right.

Argument 1: The gp_axis code for up and down.

Argument 2: Setting this argument to true will change the deadzone to be “plus shaped”, making it easier to hold a cardinal direction. Defaults to false.

Argument 3: The currently configured number of steps a direction must be tapped twice to count as a doubleTap. Default `global.defaultDoubleTap`

Argument 4: The deadzone for the stick. Default is `global.defaultButtonDeadzone`.

`right, up, left, down`

`Each contains a StickTilt struct with all relevant values.`

Button Inputs (InputType.button)

When accessing any InputType.button input, you’ll have access to many properties to determine the state of the input. Let’s go over them.

`button`

Contains the binding for the given input. This would be the value passed when creating the input type.

`doubleTapGap`

The currently configured number of steps a button must be tapped twice to count as a doubleTap.

deadzone

The currently configured deadzone for the button. Only relevant for StickTilt and GamepadButton bound to triggers.

pressed

Contains true if the button was pressed this step.

lastPressed

Contains a timer that tracks how many steps ago the button was pressed. Useful for input buffering.

rapidPressed

Contains true if the button is pressed or it is held and the rapid fire timer reports it should report pressed. Use this property to check the button state if you want to support rapid fire.

released

Contains true if the button was released this step.

releasedTimer

Contains a timer that tracks how many steps ago the button was released. Useful for input buffering.

held

Contains true if the button is currently held down.

heldTimer

Contains a timer that tracks how long the button has been held.

value

Contain the current value of the button. For digital buttons will contain 0 if not held and 1 if held. For StickTilt and a GamepadButton bound to a trigger, it will report the raw value between 0 and 1 for that button without taking deadzone into consideration.

any

Contains true if doubleTapped, held, pressed, or released is true.

Key

```
keyboardAttack = [
  new Key(vk_space)
]
```

`new Key(keyCode, [doubleTapGap])`

Creates a new Key struct. This struct should be passed to an Input as part of the **keyboard** bindings array argument for an InputType.button input.

Argument 0: The keyboard key code to bind the button to.

Argument 1: The currently configured number of steps a key must be tapped twice to count as a doubleTap. Default: `global.defaultDoubleTap`.

GamepadButton

```
gamepadAttack = [
  new GamepadButton(gp_face1)
]
```

`new GamepadButton(gamepadButtonCode, [doubleTapGap], [deadZone])`

Creates a new GamepadButton struct. This struct should be passed to an Input as part of the **gamepad** bindings array argument for an InputType.button input.

Argument 0: The gamepad button code to bind the button to.

Argument 1: The currently configured number of steps a button must be tapped twice to count as a doubleTap. Default: `global.defaultDoubleTap`.

Argument 2: The deadzone for triggers. Default is `global.defaultButtonDeadzone`.

MouseButton

```
gamepadAttack = [
  new MouseButton(mb_left)
]
```

`new MouseButton(keyCode, [doubleTapGap])`

Creates a new MouseButton struct. This struct should be passed to an Input as part of the **keyboard** bindings array argument for an InputType.button input.

Argument 0: The mouse button code to bind the button to.

Argument 1: The currently configured number of steps a key must be tapped twice to count as a doubleTap. Default: `global.defaultDoubleTap`.

MouseWheel

```
keyboardAttack = [
  new MouseWheel(mouse_wheel_down)
]
```

`new MouseWheel(mouseWheelFunction, [doubleTapGap])`

Creates a new MouseWheel struct. This struct should be passed to an Input as part of the **keyboard** bindings array argument for an InputType.button input.

Argument 0: Either mouse_wheel_up or mouse_wheel_down. No parentheses.

Argument 1: The currently configured number of steps the mouse wheel must be turned twice in the same direction to count as a doubleTap. Default: `global.defaultDoubleTap`.

StickTilt

```
gamepadAttack = [
  new StickTilt(gp_axislh, 1)
]
```

`new StickTilt(gp_axis, posOrNeg, [doubleTapGap], [deadZone])`

Creates a new StickTilt struct. This struct should be passed to an Input as part of the **gamepad** bindings array argument for an InputType.button input.

Argument 0: The gamepad axis code to bind to.

Argument 1: Whether the stick needs to be tilted in the positive or negative direction. -1 for left and 1 for right on a horizontal stick axis. -1 for up and 1 for down on a vertical stick axis.

Argument 2: The currently configured number of steps the stick must be tilted twice to count as a doubleTap. Default: `global.defaultDoubleTap`.

Argument 3: The deadzone for the stick. Default is `global.defaultButtonDeadzone`.