Wrote by squaresun

Rhythm-based control system

This plugin provides:

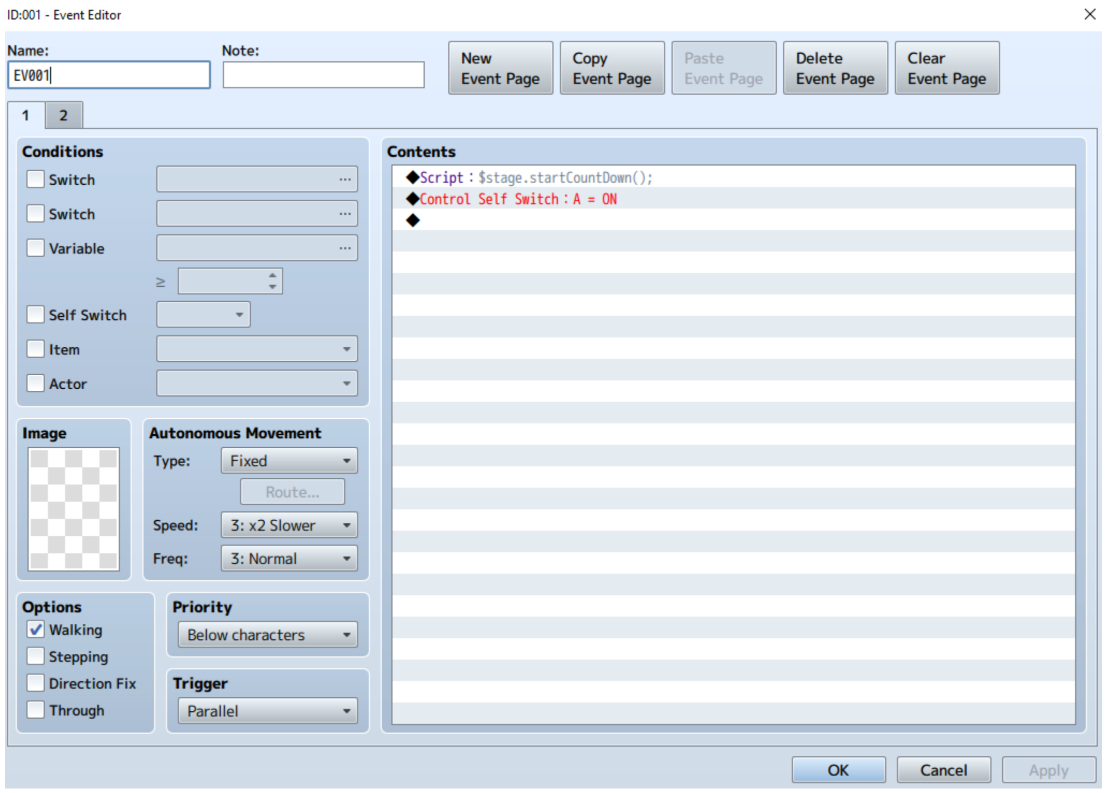
1. Measures of the user input, arrow keys, with the periodic rhythm, based on the BPM designated.
    - Player can't move after missed the beat.
2. A simple UI displaying the notes of beats.
3. Invoking functions while user input, in order to make events acting with beats.
4. Invoking functions with a sequence of beats defined, following the BPM.

This plugin **does not** provide**:**

1. The plugin command interface, which means in order to build your own games, you would need to write your own scripts, such as define the movement of events.
2. Pause handling, which means the system would be crashed if player pause the game.
3. Switching input system, which means player should input following the beats, in the whole game.

Quick Start

1. Import the scripts and pictures provided.
2. Add the plugins: Judge, StageManager, UIManager into the game.
3. Add a new event:



ID:001 - Event Editor

Name: EV001
Note:

Conditions
- [ ] Switch
- [ ] Switch
- [ ] Variable
- [ ] Self Switch
- [ ] Item
- [ ] Actor

Image

Autonomous Movement
Type: Fixed
Speed: 3: x2 Slower
Freq: 3: Normal

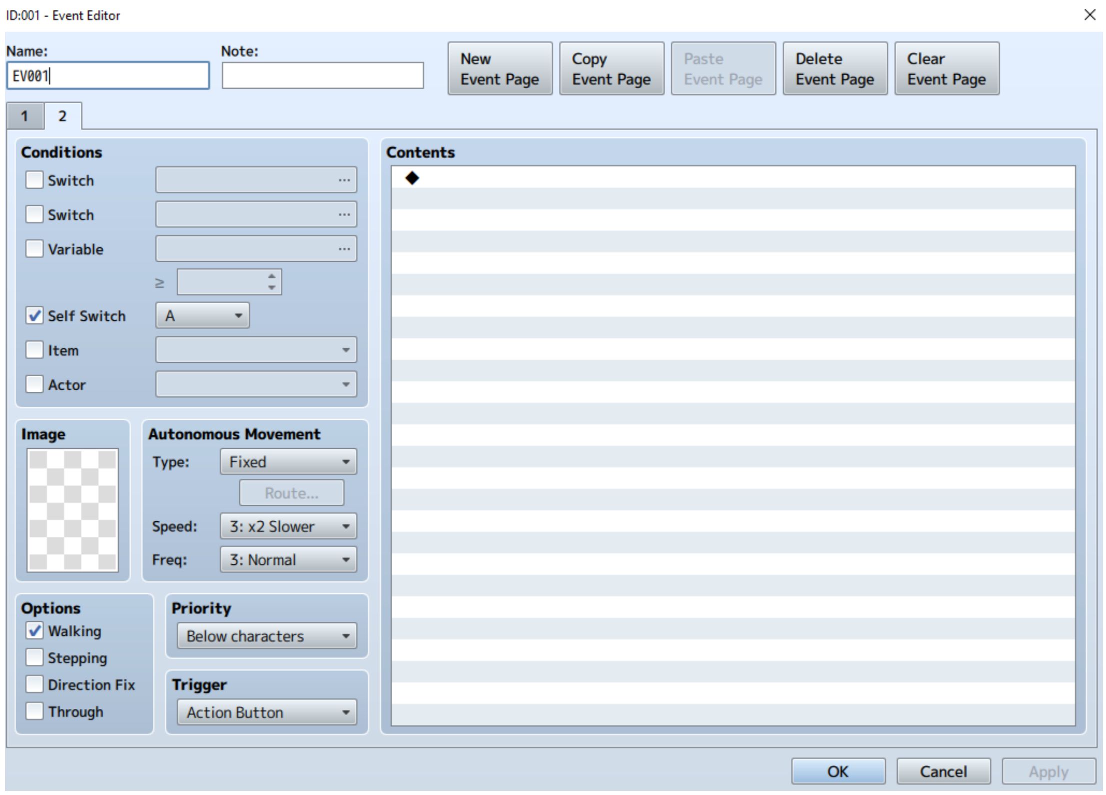
Options
- [x] Walking
- [ ] Stepping
- [ ] Direction Fix
- [ ] Through

Priority: Below characters

Trigger: Parallel

Contents

```
◆Script：$stage.startCountDown();
◆Control Self Switch：A = ON
◆
```



ID:001 - Event Editor

Name: EV001
Note:

Conditions
- [ ] Switch
- [ ] Switch
- [ ] Variable
- [x] Self Switch: A
- [ ] Item
- [ ] Actor

Image

Autonomous Movement
Type: Fixed
Speed: 3: x2 Slower
Freq: 3: Normal

Options
- [x] Walking
- [ ] Stepping
- [ ] Direction Fix
- [ ] Through

Priority: Below characters

Trigger: Action Button

Contents

```
◆
```

Judge

In general, the measure would be invoked by each time of user input and the miss checker. You should define the event or function that is going to be invoked after the measurement.

The measurement would be start once the function **Judge.prototype.startStopWatch()** called.

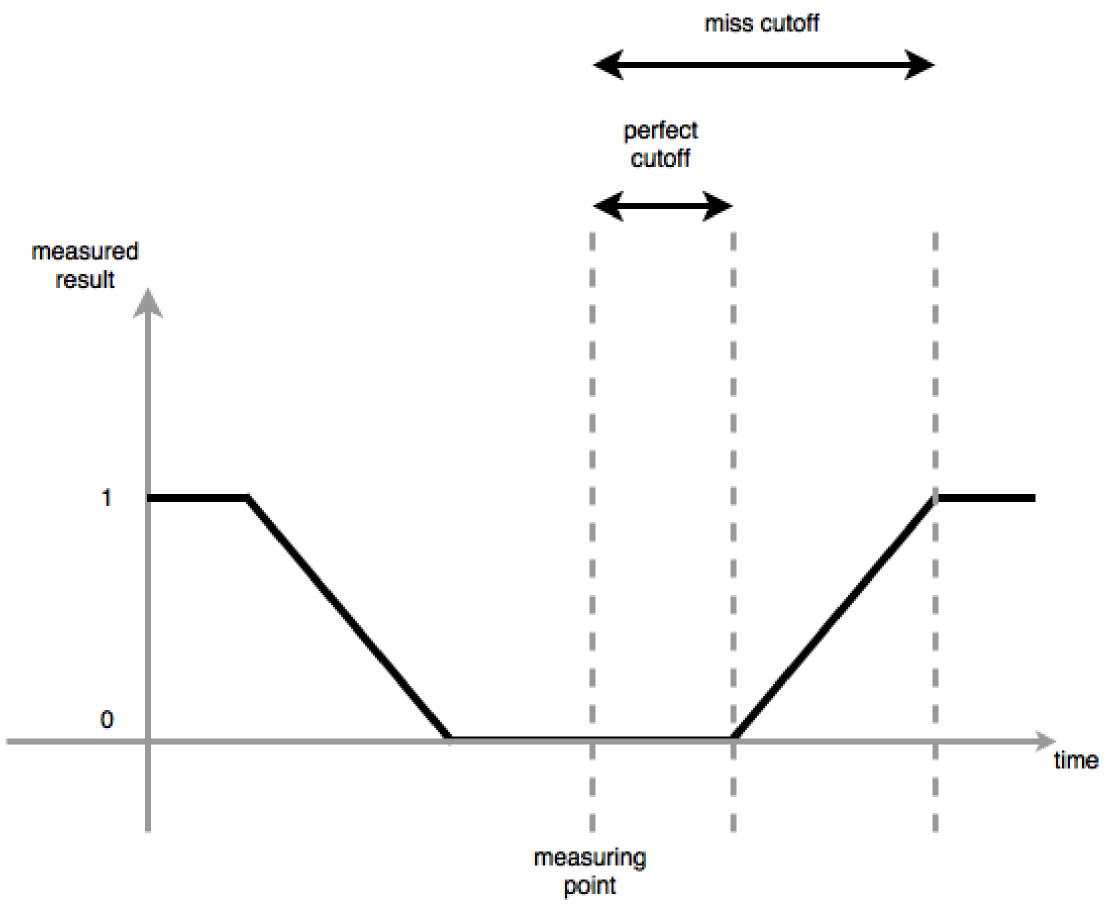
Parameters:

1. Song offset (ms)
2. Song BPM - *Beats per minute*
3. Bar divisor - *how many beats in a bar*
4. Perfect cutoff (ms)
5. Miss cutoff (ms)
6. Checkout maximum bars count
7. Event Action Invoker number - *how many event action invoker object are needed for initialize*

Measuring mechanism:

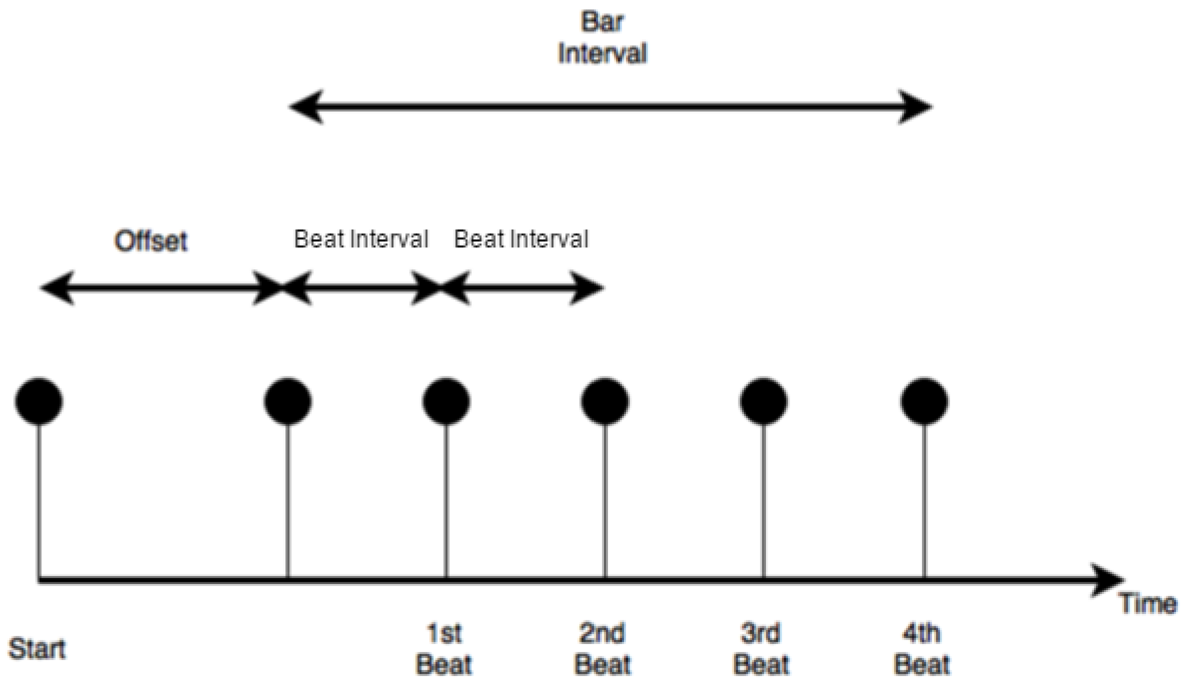
Measuring point: Input timing of the periodic beat.

Perfect cutoff: Time duration between the ending point of perfect and measuring point.



Miss cutoff: Time duration between the starting point of miss and measuring point.

The measured result would be returned using the "measure" function in Judge. For example, when input is "Perfect", it returns 0. Similarly, when input is "Miss", it returns 1. The results between the timing of "Perfect" and "Miss" will be linear interpolated.



The above diagram shows the rundown of the periodic measures if $bar\ divisor\ =\ 4$. Bar divisor defines the number of beats in a bar, for example, if $bar\ divisor\ =\ 2$, there would be one beat located on median timing of the bar and one beat located on the end of the bar. Offset means the time gap before measuring started (Silence time of the song beginning). Beat interval means the duration between beats and bar interval means the duration of a bar.

Beat interval is calculated by:

$$Beat\ Interval = \frac{60000}{bpm} \cdot \frac{4}{Bar\ Divisor}$$

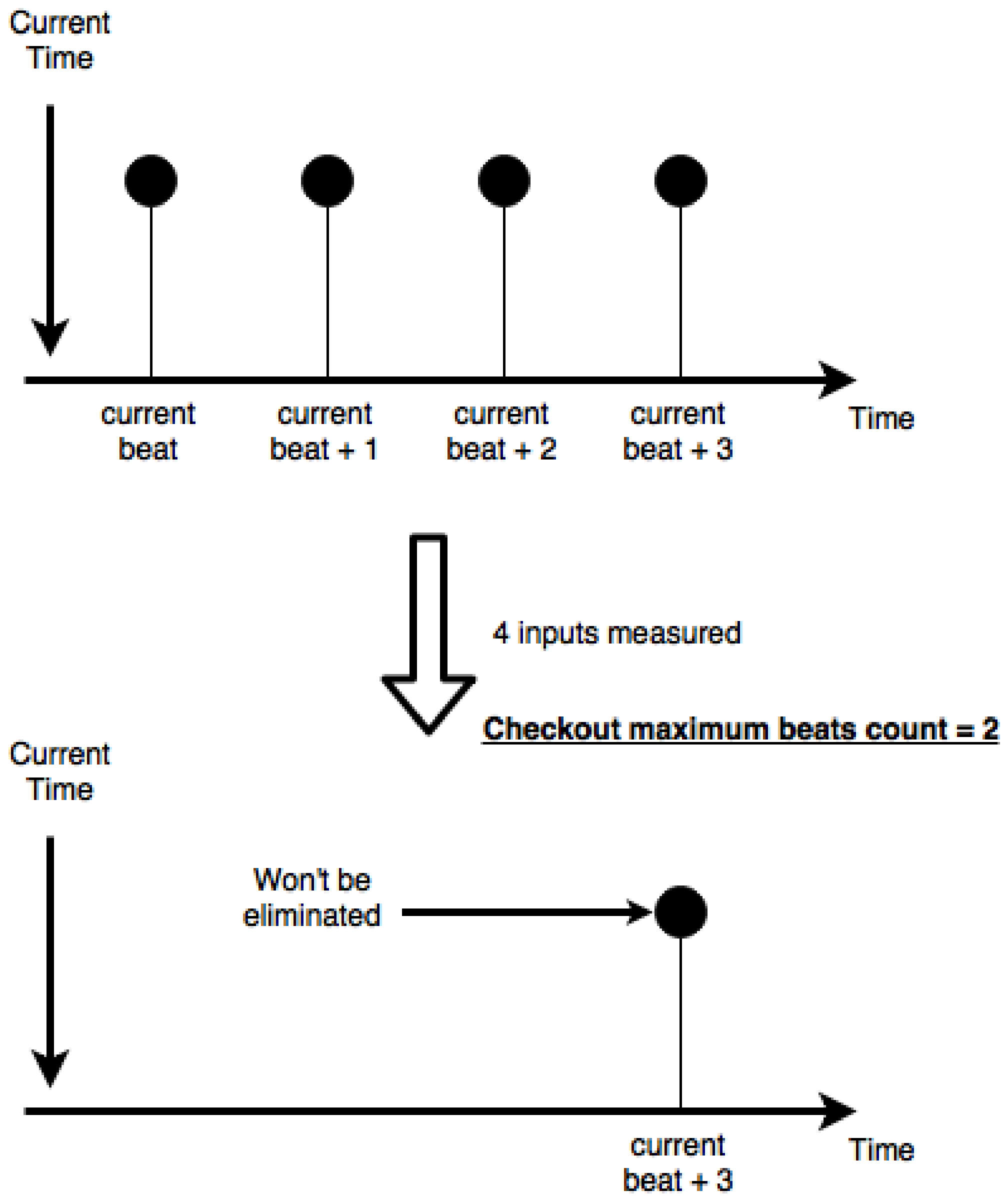
Notice that it should be, in general,

$$Miss\ Cutoff < \frac{Critical\ Interval}{2}$$

Checkout mechanism:

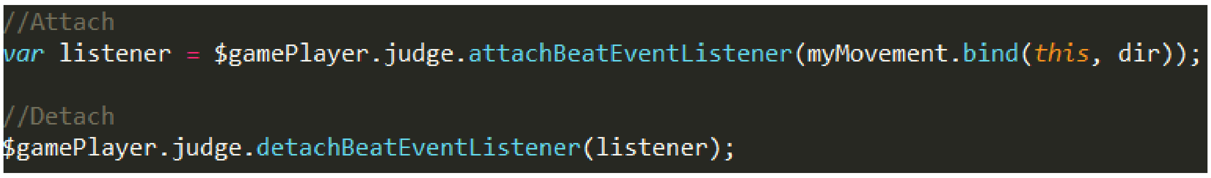
The beat will be eliminated once measured. You can define the maximum beat numbers that located after the current beat.

The following example shows the situation after 4 input measures at current time when $checkout\ maximum\ beats\ count\ =\ 2.$



Invoking event functions when measured:

In order to make events acting with beats, you can define the functions that would be automatically invoked after checkout. Those function should be attached on the **$gamePlayer.judge** object using **Judge.prototype.attachBeatEventListener()** and it will return the object of function listener that can be used to detach the attachment. Don't forget to bind the object that calls the attached function.



```
//Attach
var listener = $gamePlayer.judge.attachBeatEventListener(myMovement.bind(this, dir));

//Detach
$gamePlayer.judge.detachBeatEventListener(listener);
```

Miss checker:

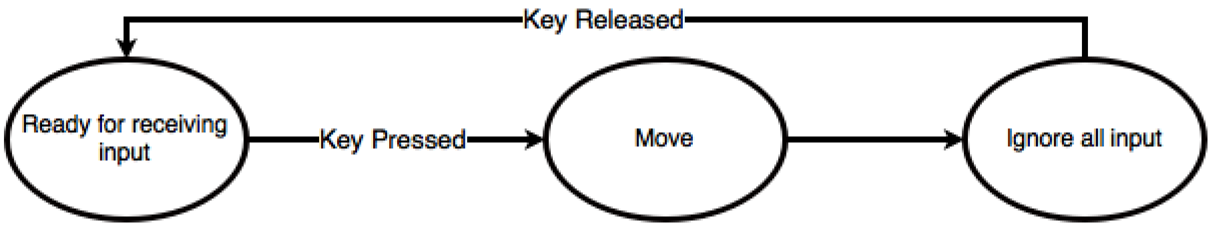
When the judging system start working, the miss checker function will be automatically bound by the setTimeout() function with fixed delay interval, which means **Judge.prototype.missCheck** will be called each time when the $current\ time\ =\ measuring\ point\ timing\ +\ miss\ cutoff$.

If the player hasn't input on the current beat, a measure would be taken automatically and returns $measured\ result\ =\ 1$.

The reason of using a sequence of setTimeout() instead of one setInterval() is to prevent influence from the first delay of setInterval() called. In my approach, the delay would be compensated each time of using setTimeout().

Input

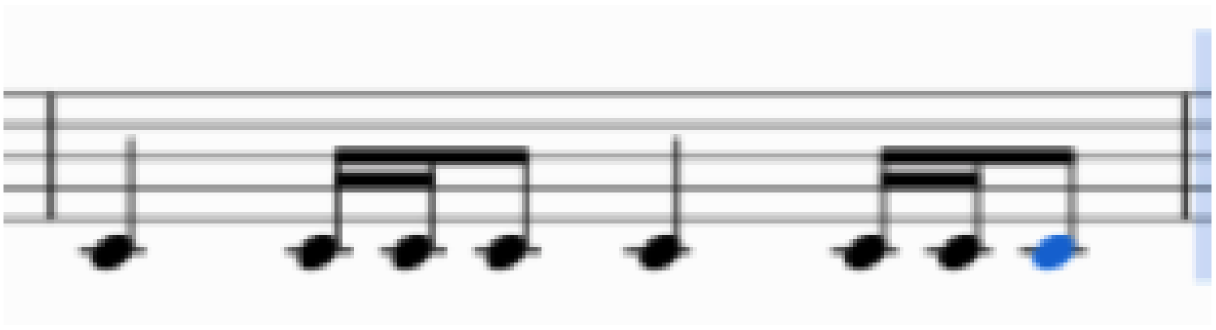
This system handles arrow keys input only.



The above state diagram shows how the system handles input.

Event Action Invoker

When you need the events act with the rhythm of the song, and you want to define a sequence of beats they act, you can use event action invoker to achieve.



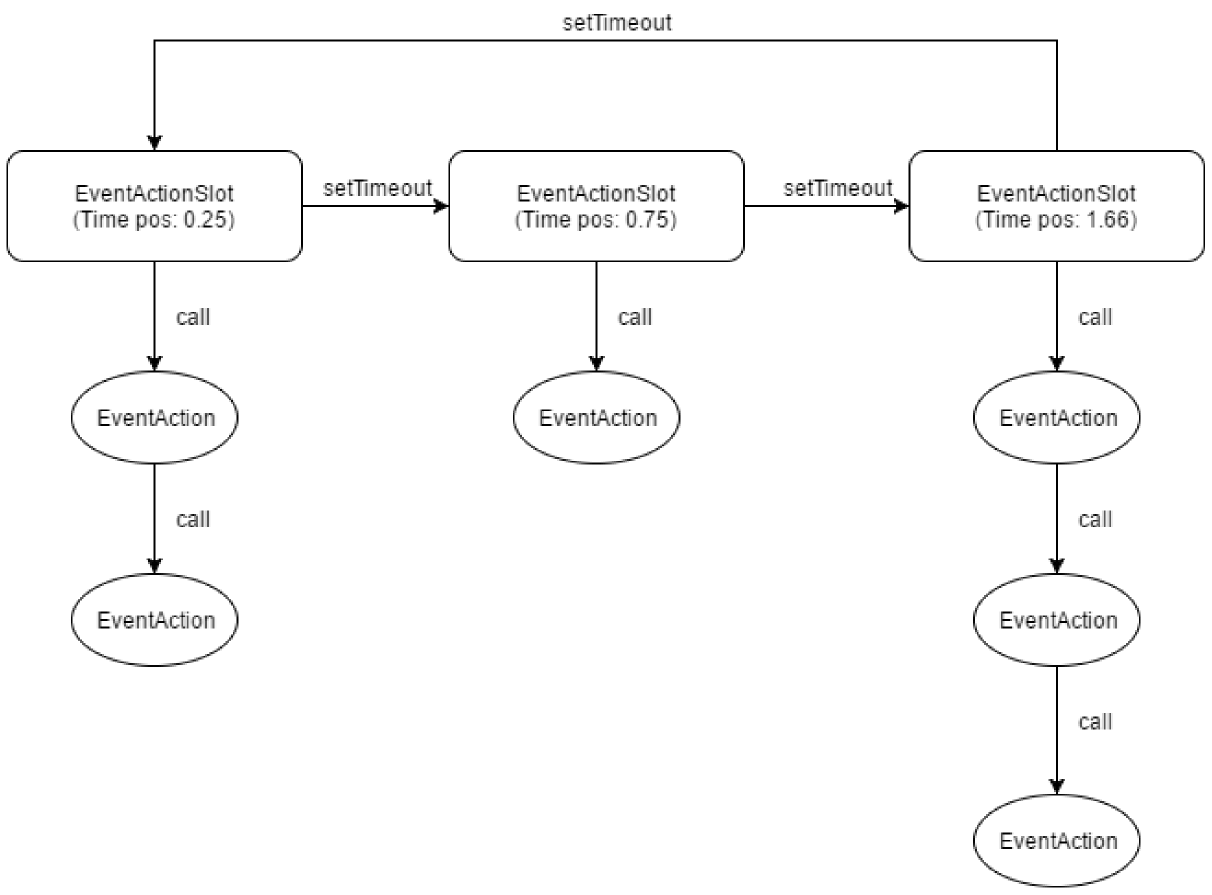
You can attach the functions of events to the event action invoker, similar to the **Judge.prototype.attachBeatEventListener**, and they will act as the sequence of timing you defined.

The array of Event Action Invoker object is the member of class Judge. As default, one event action invoker object would be initialized when object Judge init. The length of Event Action Invoker array can be defined by parameter: Event Action Invoker number in Judge.

The first Event Action Invoker object is a global timer, which means its timer will be started once the judge starts. And the following objects' timer in event action invoker array are started only when **EventActionInvoker.prototype.startTimer()** is called. So, for example, you can define the events start to act only when specified switches triggered.

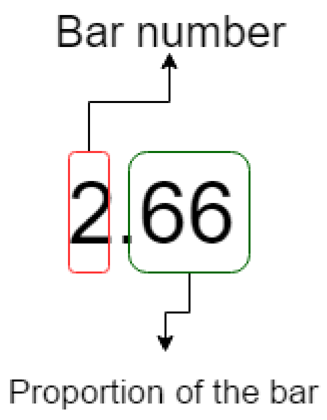
Event Action Invoker structure:

Each of Event Action Invoker object has an array of Event Action Slot, and the actions with the same time position would be located in the same slot. When processing a slot, that slot would call all the actions belongs to it. The following picture is an example of the flow when an Event Action Invoker object is running.

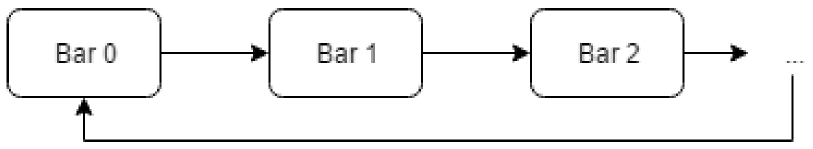


Time position:

The time positions of Event Actions defined as proportional to a bar's duration. For instance, time position = 0.5 means the action located at the median of bar number 0.



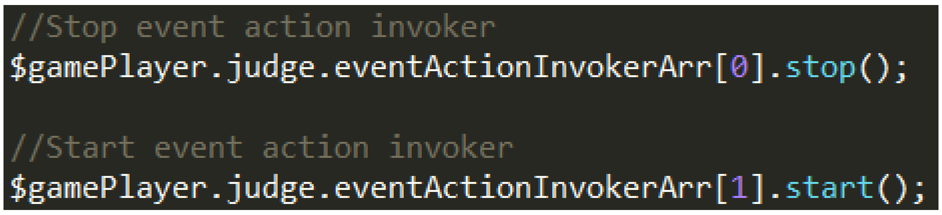
The number in front of the decimal point defines the bar that action located, starting from 0.



The bars will be running one by one. It will be back to the initial bar after running the last bar you defined. Of course, there can be only one bar, bar number 0, in the Event Action Invoker.

Notice that the maximum bar number only valid in the same **EventActionInvoker.prototype.addActions()**

Activate or deactivate Event Action Invoker object:



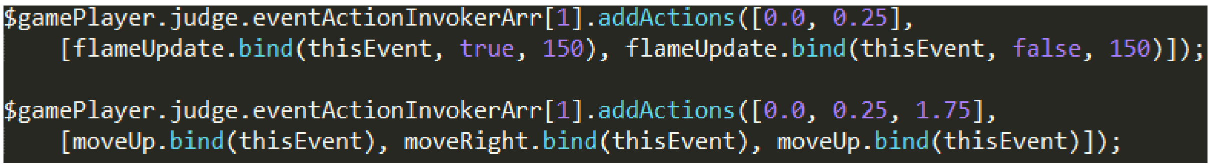
```
//Stop event action invoker
$gamePlayer.judge.eventActionInvokerArr[0].stop();

//Start event action invoker
$gamePlayer.judge.eventActionInvokerArr[1].start();
```

Since the activating time may not exactly follow the player's beats, the deviated time between beats and activating time will be compensated when the first setTimeout() called.

Adding actions in Event Action Invoker:

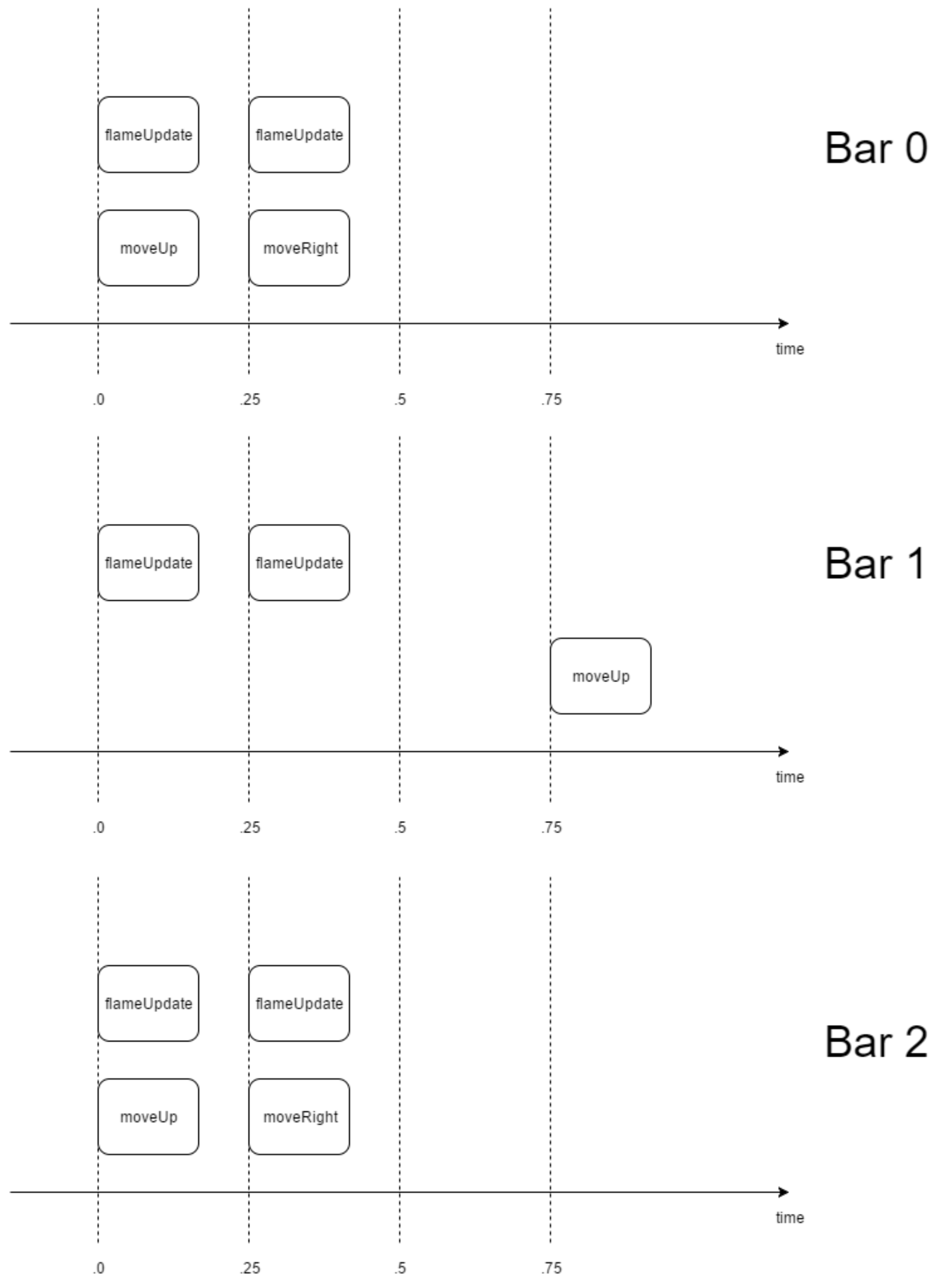
The first argument is a time position array and the second argument is the functions corresponding to time positions.

The following example shows how to add actions to Event Action Invoker object.



```
$gamePlayer.judge.eventActionInvokerArr[1].addActions([0.0, 0.25],
    [flameUpdate.bind(thisEvent, true, 150), flameUpdate.bind(thisEvent, false, 150)]);

$gamePlayer.judge.eventActionInvokerArr[1].addActions([0.0, 0.25, 1.75],
    [moveUp.bind(thisEvent), moveRight.bind(thisEvent), moveUp.bind(thisEvent)]);
```

And the above rundown would be:



▪

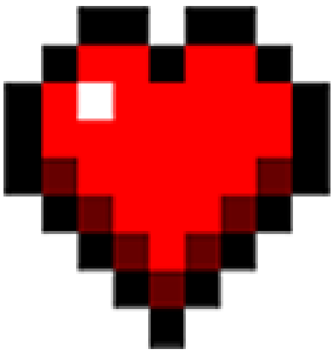
▪

▪

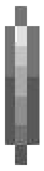
▪

UI Manager

The UI Manager occupied the first $(1 + pool\ size * 2)$ pictures in object: $gameScreen. If you want to show pictures, you should not use those id.



Measure area



notes

The notes’ x-coordinate determined by the game player’s judge stopwatch time and the measure area picture scales with the miss cutoff point of judge, which means the it wouldn’t be a “miss” when the note lies on the measure area picture.

The scaling of measure area (picture size = 64px*64px) would be:

$$scaling = \frac{miss\ cutoff}{beat\ interval} \frac{408 * UI\_speed}{64} * 100$$

The original scaling is 100.

You can define the UI_speed in range [1, 100], which determines the notes’ traveling speed.

The images should stores in /img/pictures/

Stage Manager

The class manages the count down time before the game starts and the song name of the background music. Notice that you should not bind the background music via the RPG Maker system since I observed some delay problems when using that.

I tried to deal with the delay problems but different devices seems would have different delay time on playing the BGM. You can set the global offset in Stage Manager to cope with this problem.

You would need to call **$stage.startCountDown()** when player loaded into the map and don’t forget to call **$stage.finalize()** before changing map.