Throttling

Throttling limits the rate at which a function is executed. It ensures that the function is called at most once per a specified time interval, regardless of how frequently the triggering events occur within that interval. Throttling is useful when you want to ensure that a function is executed at a steady rate and you want to avoid overloading the system with excessive function calls.

Example use case: Scrolling events on a web page where you want to load new content gradually as the user scrolls.

Debouncing

Debouncing, on the other hand, delays the execution of a function until after a specified time interval has passed since the last triggering event. It's used to ensure that the function is executed only after a pause in events, effectively grouping multiple events into a single execution. Debouncing is helpful when you want to handle rapidly occurring events (e.g., user typing) and want to ensure that the function is only called once the user has finished the action.

Example use case: Handling user input in a search box where you want to fetch search results only after the user has stopped typing.

Which One to Choose

The choice between throttling and debouncing depends on your specific use case and the desired user experience:

Use throttling when you want to limit the rate of function calls to a certain interval, ensuring a consistent and controlled execution rate. This is suitable when you need the function to be called at regular intervals.

Use debouncing when you want to delay the execution of a function until a pause in events occurs, consolidating multiple events into a single execution. This is suitable when you want to optimize performance and avoid excessive function calls.

In general, if you're dealing with events that can occur very frequently (such as mousemove or scroll events), throttling might be a better choice to prevent overwhelming the system. If you're dealing with user input that requires some processing (such as filtering or searching), debouncing can help reduce unnecessary function calls and provide a smoother user experience.

Ultimately, the decision should be based on your understanding of the specific interaction and the behavior you want to achieve in your application.

```
const inputRef = document.querySelector('input');

const debounce = (fn, delay)=>
{
  let timeOutId;
  return ()=>
  {
    if(timeOutId)
    {
      clearTimeout(timeOutId);
    }

    timeOutId = setTimeout(()=>
    {
      fn();
    }, delay);
  }
}


var count = 0;

const delay = 5000;

const fn = ()=>
{
  count++;
  console.log(inputRef.value, count);
}

const debounceFn = debounce(fn, delay);
inputRef.addEventListener('keyup', debounceFn);
```

```
function throttle(fn, delay)
{
  let flag = true;
  return function()
  {
      if (flag)
      {
        flag = false;
        setTimeout(() =>
        {
          fn();
          flag = true;
        }, delay);
      }
  }
}

let count = 0;

const fn = function()
{
  count++;
  console.log(count);
};

const throttleFn = throttle(fn, 2000);


const buttonRef = document.querySelector('button');
buttonRef.addEventListener('click', throttleFn);
```