



Tower of Hanoi in JavaScript



Nasir Darwish

6 Jan 2016 CPOL

The article describes a JavaScript-based solution to Tower of Hanoi problem

[Download source - 3.9 KB](#)



Introduction

The purpose of this post is to present a JavaScript-based solution to the famous *Tower of Hanoi* problem. The problem calls for moving a set of disks from one tower to another with the restriction that at no time a disk is placed on top of a smaller disk. The problem has an associated well-known recursive algorithm.

The **proposed solution** (HTML and JavaScript all within one HTML file) shows a possible animation of the algorithm using JavaScript `setInterval()` function.

The idea for animating the recursive tower-of-Hanoi algorithm the way it is described here is not new. I did implement it some 15 years back in Visual Basic. A video of this is available at [this link](#).

A Recursive Algorithm for Tower of Hanoi

The well-known recursive algorithm for the tower of Hanoi problem can be expressed by the following JavaScript function:

```
function Hanoi(n, from, to , via)
{
  if (n==0) return;

  Hanoi(n-1, from, via , to);

  moveDisk(from,to);

  // callStack.push([from,to]); // save parameters to callStack array

  Hanoi(n-1, via, to , from);
}
```

The input to the algorithm are 4 integer parameters, as follows:

1. *n*: number of disks; serves as the problem size for recursion
2. *from*: the "from" tower is where the disks are placed
3. *to*: the "to" tower is where the disks must be finally placed
4. *via*: the "via" tower is that used as an intermediate location as disks are moved between the towers *from* and *to*.

Normally (if we denote the towers with 0,1,2), the initial call for *n* disks will be **Hanoi(*n*,0,2,1)**.

A recursive algorithm normally tries to find subproblems (instances of the original problem but with smaller problem size). Then express the solution for the original problem in terms of the solutions to the subproblems. The recursive algorithm for the tower of Hanoi is based on observing that the **top *n*-1 disks at the "from" tower** (together with the other two towers) represent a smaller-size instance of the original problem and, thus, can be solved by the call **Hanoi(*n*-1, 0,1,2)**. This moves the disks to the middle tower (#1) using the other tower (#2) as intermediate. After this, we can move the *n*-th disk from tower #0 to tower #2 and then move all the *n*-1 disks from the middle tower to the last tower using tower #0 as intermediate by the call **Hanoi(*n*-1, 1,2,0)**.

The above algorithm encodes the preceding description but with the parameters for the towers kept as variables instead of being specific values.

Animation Handling in JavaScript is Problematic

Normally animation code can utilize JavaScript timer functions: **setInterval()** or **setTimer()**. However, things are not that easy. Suppose, in the above code, the call to the function **moveDisk()** is animated using **setInterval()**. A problem arises because code in the caller will continue executing and interfere (miss up shared data) with the animation. One might suggest that the caller sleeps for a while to allow for animation to complete. However, JavaScript lacks a "real" delay or sleep mechanism. If we are to implement the delay by a spin-loop, it would bring things to a stand-still because the whole of JavaScript code executes on one thread.

It seems that JavaScript animation uses the pattern: **if you start animation, do not do anything else**.

For the tower-of-Hanoi algorithm, we want to animate one call to **moveDisk()**, return to the caller until we encounter the next call to **moveDisk()**, animate it, and so on. But with JavaScript animation, somehow, we have to follow the pattern: animate **moveDisk()**, animate the next **moveDisk()**, ... and so on. In other words, every call to **moveDisk()** should issue (at the time when it is done) the next call to **moveDisk()**. But then how to find all these **moveDisk()** calls. After some thought, I was asking myself: Why not use a stack (**callStack**) to save the "from,to" call-parameters to **moveDisk()**. Using a stack will also preserve the order of the calls. This is essentially what I did, that is, in the above code comment the line **moveDisk()**; and uncomment the line **callStack.push([from,to]);**. Now the modified **Hanoi()** function is called from the function **executeHanoi()** given below.

The following listing shows the complete source code of my script.

```

var callStack;

function executeHanoi()
{ // Some initialization code goes here

    callStack=[]; // callStack array is global

    Hanoi(diskCount, 0,2,1);

    moveDisk(); // moveDisk takes its parameters from callStack
}

function moveDisk()
{ if (callStack.length==0) return;

    var param = callStack.shift(); // Get call parameters from callStack
    // Note: throughout the code, I use fromBar, toBar to refer to towers
    fromBar = param[0];
    toBar = param[1];
    // find top element in fromBar
    var elem = document.getElementById(barsInfo[fromBar].disks.pop());

    moveInfo = { elem: elem,
                  fromBar: fromBar,
                  toBar: toBar,
                  whichPos: "top", // element position property for movement
                  dir: -1, // 1 or -1
                  state: "up", // move upward
                  endPos:60 // end position (in pixels) for move upward
                }

    myTimer = setInterval(animateMove,speed); // Start animation
}

```

After the call to Hanoi(), the callStack will have an entry for each moveDisk() call. The processing of these entries is handled by the function moveDisk(). The function moveDisk() retrieves an entry from callStack and sets up an object **moveInfo** for passing data needed for the **animateMove()** function. The animation is started by the line **myTimer = setInterval(animateMove,speed)**.

The following code listing shows the code for the animateMove() function.

```

function animateMove()
{ var elem = moveInfo.elem;
  var dir = moveInfo.dir;
  var styleInfo = window.getComputedStyle(elem);
  var pos = parseInt(styleInfo[moveInfo.whichPos]);

  if (((dir==1) && (pos >= moveInfo.endPos)) ||
      ((dir== -1) && (pos <= moveInfo.endPos)))
  { // alert(moveInfo.state);
    if (moveInfo.state == "up")
    { moveInfo.state = "hor";
      moveInfo.whichPos = "left";
      moveInfo.dir = 1;
      if ( moveInfo.fromBar > moveInfo.toBar) moveInfo.dir = -1;
      //alert("toBar:" + moveInfo.toBar);
      var toBar = document.getElementById("bar"+ moveInfo.toBar);
      // alert(toBar.offsetLeft);

      moveInfo.endPos = toBar.offsetLeft -
        Math.floor(elem.offsetWidth/2)+15; // 15 is half of tower width
      return;
    }
  }
}

```

```

else if (moveInfo.state == "hor" ) // move down
{
    // alert("here");
    moveInfo.state = "down";
    moveInfo.whichPos = "top";
    moveInfo.dir = 1;
    //alert(elem.offsetHeight);
    moveInfo.endPos = document.getElementById("bottombar").offsetTop -
        (barsInfo[moveInfo.toBar].disks.length+1)*elem.offsetHeight;
    return;
}

else // end of current call to moveDisk, issue next call
{
    clearInterval(myTimer); // cancel timer
    barsInfo[moveInfo.toBar].disks.push(elem.id);
    moveDisk();
    return;
}
}

pos = pos + dir*moveInc;
elem.style[moveInfo.whichPos] = pos+ "px";
}

```

The `animateMove()` function essentially moves a disk up then right or left (if `toBar < fromBar`) then down and finally issues the next call to `moveDisk()`. Note that the timer is cancelled (using the call `clearInterval(myTimer)`) before making the next call to `moveDisk()`. This is to prevent `animateMove()` from incorrectly using the `moveInfo` object.

Limitations, suggestions

The presented code was tested in latest versions of popular browsers. It works perfectly in IE (version 11), Chrome (version 30), and Firefox (version 24). The animation in our solution is limited by the timer accuracy of JavaScript `setInterval()` function, which cannot be smaller than one millisecond. It may be possible to get faster animation using `window.requestAnimationFrame()` function. Finally, the graphics could be enhanced by using 3-D images for the disks that get partially covered by the tower during upward and downward movements.

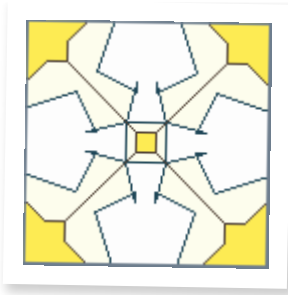
History

- 7th November, 2013: Initial version.
- 15th November, 2013: Updated with better graphics.
- 7th November, 2015: A slight update to program code to have the disks move to the leftmost column whenever "Start" button is clicked.

License


This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

About the Author



Nasir Darwish

Instructor / Trainer KFUPM


Saudi Arabia 

Nasir Darwish is an associate professor with the Department of Information and Computer Science, King Fahd University of Petroleum and Minerals (KFUPM), Saudi Arabia.

Developed some practical tools including COPS (Cooperative Problem Solving), PageGen (a tool for automatic generation of web pages), and an English/Arabic full-text search engine. The latter tools were used for the Global Arabic Encyclopedia and various other multimedia projects.

Recently, developed **TilerPro** which is a web-based software for construction of symmetric curves and their utilization in the design of aesthetic tiles. For more information, visit [Tiler website](#).

Comments and Discussions

 **9 messages** have been posted for this article Visit <https://www.codeproject.com/Articles/679651/Tower-of-Hanoi-in-JavaScript> to post and view comments on this article, or click [here](#) to get a print view with messages.

[Permalink](#)
[Advertise](#)
[Privacy](#)
[Cookies](#)
[Terms of Use](#)

Article Copyright 2013 by Nasir Darwish
Everything else Copyright © [CodeProject](#), 1999-2020

Web01 2.8.20201015.1