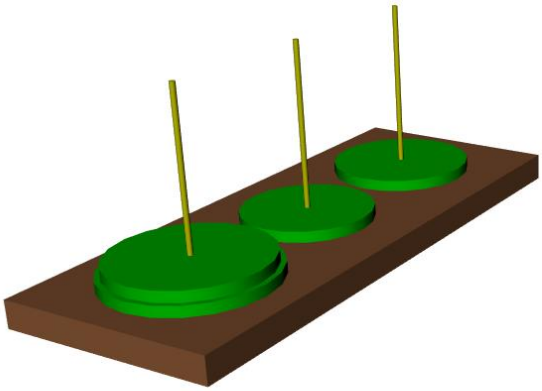
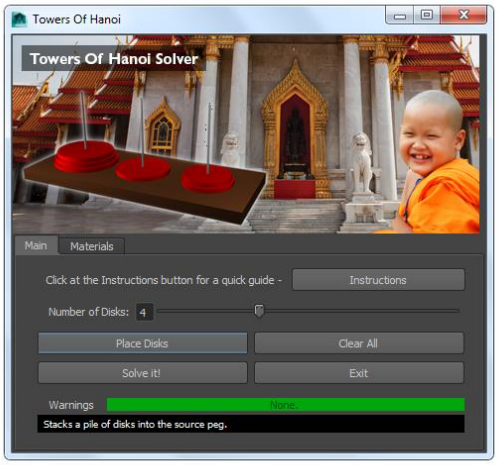
Towers Of Hanoi Maya

— Application Report —





Introduction

The towers of Hanoi puzzle consists on displacing all the disks from a source peg (which normally is the left-most one) to the destination peg (right-most one). The rules are that we can just one disk at time and that we cannot stack a bigger disk on top of a smaller one, so we need a third peg which acts as auxiliary position to solve the puzzle.

I considered two ways of solving the problem. I read it from *Wikipedia*. The first way it was too abstract for trying to write a script for it, it consisted on displacing the smaller disk always to the right if the height of the tower was even alternating each move of the smallest with the only other legal move you could do. If the height was odd, you would do the same but displacing the smallest one to the left alternating again with legal moves inbetween. For instance, put the example we have 4 disks (even number), so we would do:

```
Move smallest one step to the right.
Make another legal move. (Excluding the smallest disk, just 1 possible)
Move smallest one step to the right.
Make another legal move. (Excluding the smallest disk, just 1 possible)
Move smallest one step to the right. (In this case as it would go back to the
first peg, because there is no more pegs on the right)
```

I did not take this way. In the end I wanted to practise with recursion, because it is the basis for the fractal generation, and I really feel passion for fractal geometry. Our *Computing for Graphics* lecturer told us about the recursive way, so I just had to find my notes and refresh my memory.

The recursive way can be understood as following. Each *peg* is set a flag *A*, *B* or *C*, each flag will be changing as the recursion takes place, they are dynamic and they are reassigned. In the starting point we can say the left-most peg is the *A* peg, and the right-most the *C* peg being the middle one the *B* peg, we can set the flags `A --> flag src (source)`, `B --> flag aux (auxiliary)`, `C --> flag dest (destination)`. This flags are dynamic and will change as following.

What we want to do is to displace *height-1* disks from *peg A* (**src**) to *peg B* (**dest**) using *peg C* as **aux** then, move the remaining disk (largest one) from *peg A* (**src**) to *peg C* (**dest**) and we move the disks we left in the *peg B* (**src**) to *peg C* (**dest**) using this time *peg A* as **aux**. This is the idea.

For displacing *height-1* disks, you first have to displace *(height-1)-1* disks, hence before you can do this movement you need to displace (*(height-1)-1)-1*… You need to apply this until you reach a number **ONE**, the base case (this is recursion!). *Why do you need the reach the base case?* Because the rules of this game tells you that you just can move 1 disk at time.

The highlighted statement is the base case. The base case occurs when recursion reaches the lowest number, in this case, when a variable called *diskNum* equals to zero, so:

```
If diskNum == 0:
   # BASE CASE: Move from A to C.
```

Implementation

So now that I explained the way I would be using for solving the puzzle I will talk about the implementation for it. I will go over the main functions as well.

The recursion I explained previously it happens in a method called `towersOfHanoi()`. I wrote a simplified version as well which instead of geometric objects, the disks are treated as strings. It is very useful for understanding recursion if you debug it using IDEs that allow you to such as *PyCharm* which I would really recommend. This file is called `simplifiedPuzzle.py` and can be found under the root directory of my project.

To understand what the method `towersOfHanoi(___, ___, ___, diskNum)` exactly does you have to understand that what is in the first gap is the *source peg*, on the second one the *auxiliary peg* and the third gap contains the *destination peg*. Each peg will be treated as a Python **list**.

Moreover the `diskNum` will decrease by one every time we call again the function itself from inside the method. When this variable reaches one, it will run the base case (statements under the `if` condition block). This would be the skeleton:

```
def towersOfHanoi( A, B, C, diskNum ):
   if diskNum == 1:
      print "Move disk from", A, "to", C
   else:
      towersOfHanoi( A, C, B, diskNum-1 )
      print "Move disk from", A, "to", C
      towersOfHanoi( B, A, C, diskNum-1 )
```

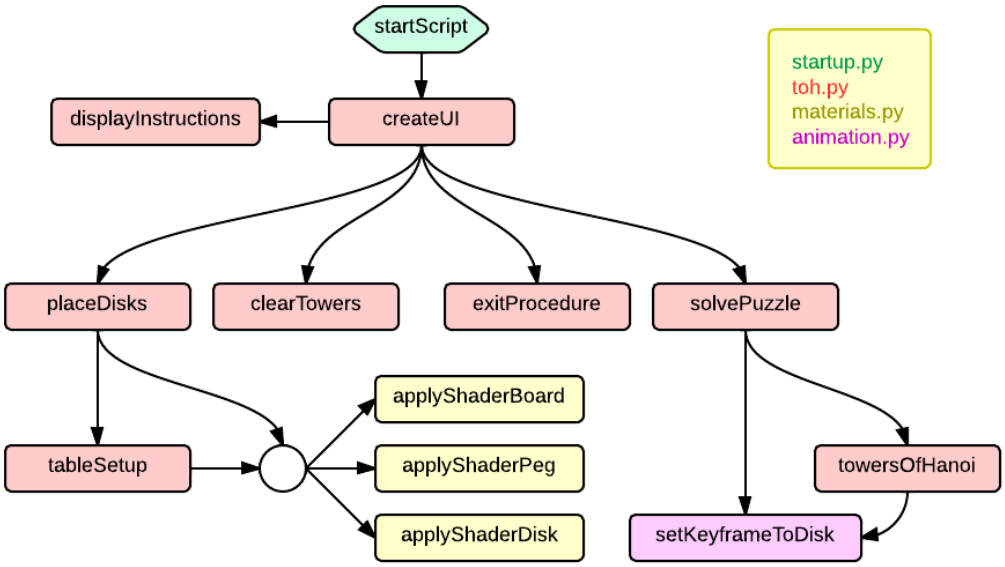
In my code the only thing I did was translating the print statements to `maya.cmds` transform commands. In addition to it I used lists a lot. Every time a disk was created it was appended to the concurrent list. So for instance, in the initial 3-disk setup:

```
print A
['disk3', 'disk2', 'disk1']
```

`disk1` refers to the smallest one, the top-most. So if we need to move the top-most disk, we access it through `A(len(A)-1)` (the last element of the list). This is quite handy because it is close to reality: the right-farthest is the top-most, the lower the disk number, the smaller the radius.

The functions that I used for the list is mainly `list.append()` and `list.pop()`

Now that I showed you how I used the recursion and the lists I would like to comment other functions I wrote. But first I should display the flowchart so that we have an idea about the method hierarchy.



Once the `startup.py` script has appended the location pointed by the user to the locations where Python searches for modules the user interface is created. It has an *Instructions* sub-UI too. When the user presses the *Place Disks* button various functions are called:

- `tableSetup()` method: Empty peg lists are created.
- `tableSetup()` method: Position lists are created. Position lists are used to tell where in the world space the disk should be moved to.
- `tableSetup()` method: The board geometry and the pegs are created.
- `placeDisks()` method: Disks are created and placed to the previously created lists. It also calls the **`animation.py`** module and sets keyframes using `setKeyframeToDisk()`
- All the materials have been concurrently created with the geometry using the functions highlighted in yellow (`applyShaderBoard`, `applyShaderPeg`, `applyShaderDisk`)

If the user presses *Clear All* button the `clearTowers()` method is called:

- Iterates over the pegs lists, selects the disks and deletes them.
- Lists are reset and emptied and ready to be used again.

By pressing *Solve it!* the `solvePuzzle()` method is called. Then it calls the `towersOfHanoi()` function which does the recursion and sets the keyframes by calling `setKeyframeToDisk()` included in the `animation.py` module. It is worth pointing out that the script prints out lines as the commands are executed, so it can be useful as sort of a debugging tool, so I strongly recommend taking a look at the **script editor** while running the script.

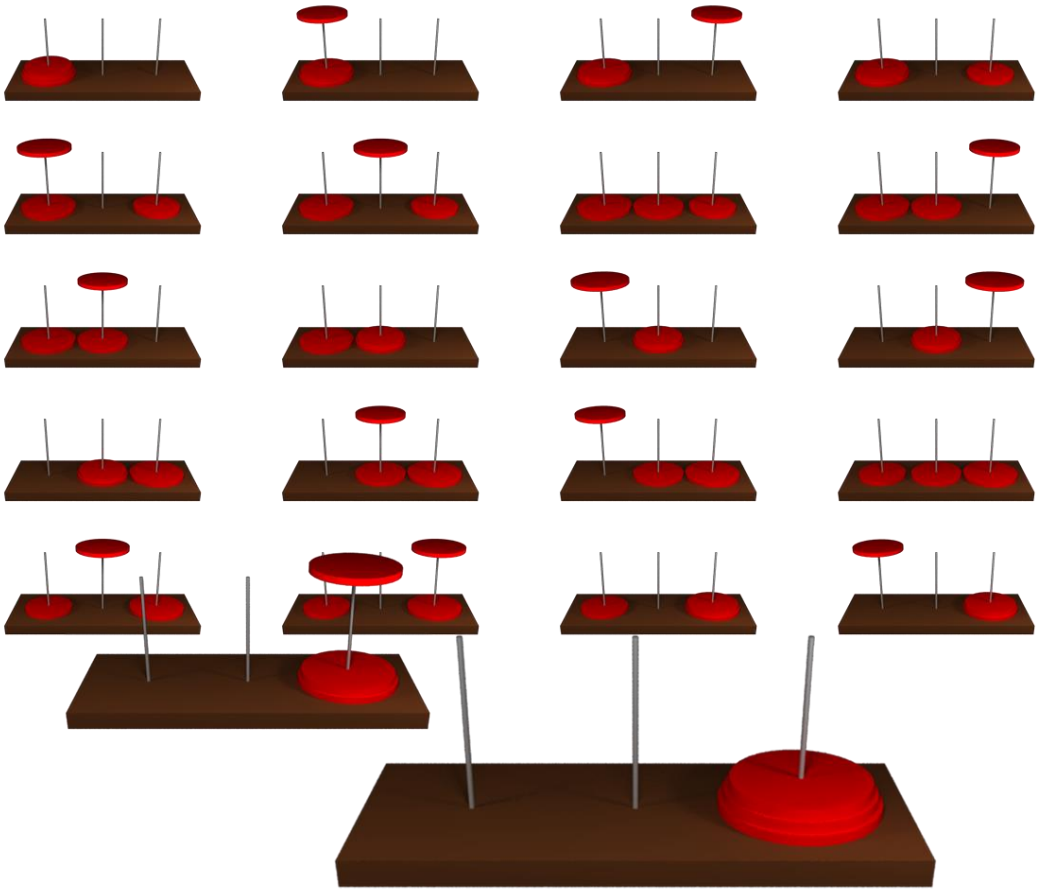
If the user presses *Exit*, the `exitProcedure()` is run. When a material or geometric element is created, it is appended to a list which in my script is called *superList*. This method iterates over this list and removes each item of it. This is a good alternative to the simple fact of creating a new Maya Scene...

If you have read through my script you might be wondering what it is exactly the `KFTime` variable. `KFTime` stands for *KeyFraming Time*. It is initialised to one and each time the `setKeyframe()` function is called it is incremented by one, so the next keyframe won't overlap the one that has been created.

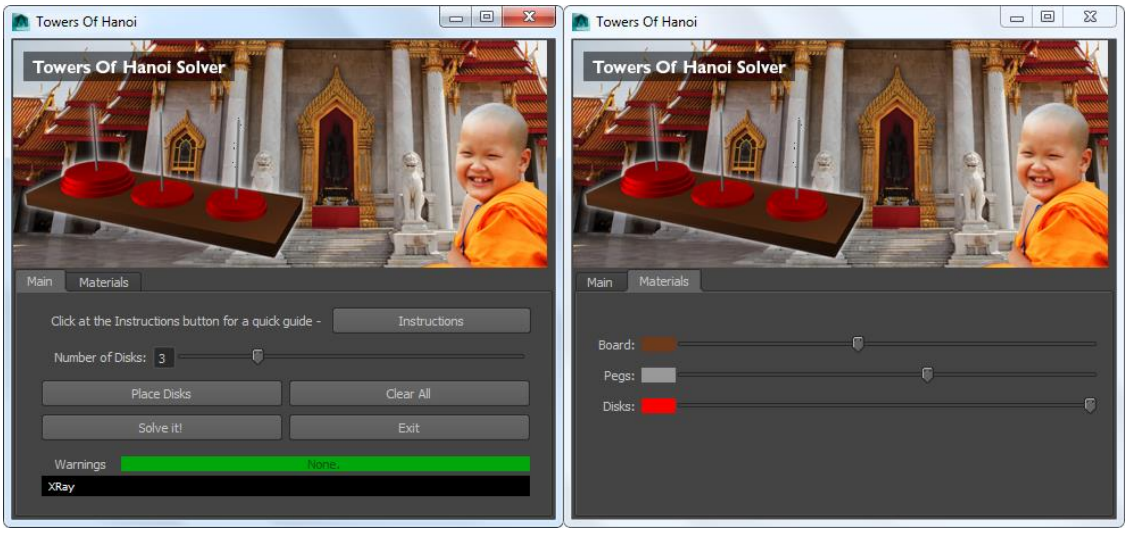
How are positions keyframed? Well as I comment in my code I use a position lists. They are called `pLeft, pMiddle, pRight, pLeftHigh, pMiddleHigh, pRightHigh`. These three first ones store the x, y and z value of where the disks should be in the world space. The ones with the world *High* on them they are nothing but the 3 first ones I enumerated plus an increment of 4 in the y-axis. That just tries to give a smoother result when animating the disks, before arriving to the position I keyframe them there but a little bit higher before landing onto the stack.

Results

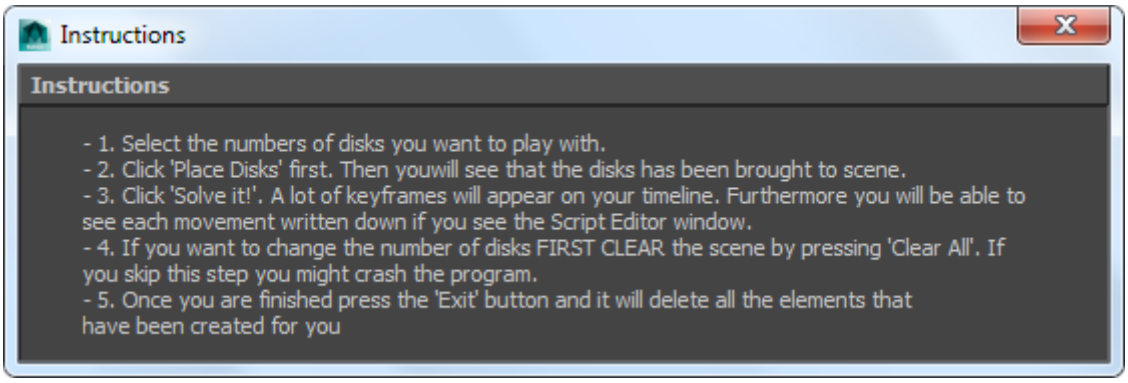
In this example I run the script using 3 disks. As you can appreciate I also keyframe the position in which the disk is about to be inserted. That gives more continuity to the movement.



The graphics user interface consists on two tabs. One which include all the commands for the puzzle itself plus a warnings bar and a help line (some annotations appear when you move your mouse over the buttons) and the other one which includes information for the RGB colour values for the disks, pegs and board.



Furthermore an *Instructions* mini-ui is included in the *Main* tab in the script. It briefly displays the steps to be done to get it working properly avoiding crashes and unexpected problems.



Thank you very much for taking your time reading my report. If you have any questions feel free to contact me. – *Ramon Blanquer (NCCA)*