UNIVERSITY OF CAPE TOWN

DEPARTMENT OF COMPUTER SCIENCE

CS/IT Honours Final Paper 2019

Title: 3 Visual Ways to Teach Recursion

Author: Shakeel Mohamed

Project Abbreviation: BBRV

Supervisors(s): Jecton Anyango, Hussein Suleman

Category	*Min*	*Max*	Chosen
Requirement Analysis and Design	*0*	*20*	20
Theoretical Analysis	*0*	*25*	0
Experiment Design and Execution	*0*	*20*	0
System Development and Implementation	*0*	*20*	20
Results, Findings and Conclusion	*10*	*20*	10
Aim Formulation and Background Work	*10*	*15*	10
Quality of Paper Writing and Presentation	*10*		10
Quality of Deliverables	*10*		10
Overall General Project Evaluation (*this section allowed only with motivation letter from supervisor*)	*0*	*10*	
Total marks		**80**	

ABSTRACT

Recursion is an important and powerful computational problem-solving tool, however many students find it hard to understand conceptually and challenging to utilize in a practical coding environment when applied to a given problem. This paper aims to outline the design and development of a stack simulation software tool that can be used to help students visualize the execution of their recursive solutions to a specific set of recursion-based problems. The simulation aims to facilitate the learning experience and provide learners with a deeper and more intuitive understanding of recursion with emphasis on the active and passive flow of the call stack that occurs behind the scenes. The tool was found to be useful in helping students better understand what the call stack is and how their recursive programs execute.

KEY CONCEPTS

Active and passive flow of recursion.

1. INTRODUCTION

Recursion is one of the most important programming concepts in computer science. It allows the creation of extremely simple algorithmic solutions to certain problems that would otherwise be unsolvable or inefficient with any other type of approach. It is a fundamental concept in computer science, whether it is understood as a programming technique or even a mathematical concept [1]. It is regarded as a challenging topic to learn for students being introduced to the world of computer science [8]. Educators often find it a difficult topic to teach as well [2]. Most people are first introduced to an iterative or loop based style of programming to solve problems, which shares many similarities with recursion and this often bewilders the beginner. Its applications in computer programming cannot be understated and many students fail to grasp the concept as it is taught in lectures and textbooks and thus may find it difficult to cope with more advanced topics taught later in many courses. Many classic examples are taught, such as factorial, Towers of Hanoi, Fibonacci numbers and binary search. However, simply learning the code or algorithms do not promote or illustrate the concept of recursive thinking in a visual or interactive way. Visualization is a highly efficient method for demonstrating difficult to learn concepts [3]. It is well known that concrete conceptual models can be better understood than abstract conceptual models [4]. Being able to see and understand when and how each recursive call is executed can be invaluable to one's understanding of recursion but this can be taken further.

Using visuals to teach programming concepts is not uncommon and has proven to be a fun and engaging way to promote learning [5]. Teaching programming visually is something that has been explored, however there are very few visualizations that are specific to recursion [5, 6, 7]. Few attempts have been made to visualize the concept of recursion especially when the payoff for achieving this can be tremendous [2]. Being able to easily teach such a challenging topic and promote interest in it could substantially reduce the initial confusion many people have and remove the misconception that recursion is a daunting method of solving problems where an iterative solution could be easier (and possibly slower).

This paper aims to outline the design and development process of a call stack simulator program designed to help learners understand the concept of recursion in an interactive environment. The program produces a visualization of the call stack of a user's recursive solution to a given recursive problem. The user enters their coded solution into a built-in text editor and is able to play a call stack simulation of what their code does. Section 2 discusses related software's that perform similar functions to what was developed. Section 3 details the user requirements gathered during an interview process with learners while section 4 explains the development process. Finally, section 5 reveals the results of user testing.

2. RELATED WORK

Recursion is a method of problem-solving that involves the decomposition of a problem into subproblem(s) of the same nature until a base case is reached. The composition of these problems solves the original. Many students fail to understand the "active" and "passive" flow of recursion and the use of the stack for backtracking [8]. The active flow refers to when the program explicitly calls the recursive function and places it on the stack. The passive flow refers to the backward "popping" of function calls on the stack. The passive flow begins when the active flow has reached its base case. More specifically, they visualize recursion as a loop structure and each recursive call as iteration, which is not true. It is well known that visualization can play an important role in understanding abstract concepts [4]. Visual analogies of recursion do exist and research into real-life examples of recursion has been done as a number of papers have been written in this regard. Examples such as parking cars [9], delegating tasks [10] and the Cargo-Bot game [5] aim to give context to recursion in the real world. Pirolli and Anderson [11] claim that the fact that there are very few analogies for recursive problems is what makes it difficult to learn. Kurland and Pea [12] discovered that students often develop an incorrect or skewed mental understanding of recursion through standard classroom and textbook learning. Kahney and Eisenstadt [13] examined inexperienced students' judgments of given recursive solutions to problems and came to the conclusion that they developed one of several mental conceptions of recursion, which they named "copies", "loop", "odd", "null", and "syntactic magic". All these models except for the "copies model" are considered to be incorrect.

One of the most important aspects of recursive programming that needs to be considered and understood is the role of the stack. The stack is a LIFO (Last In First Out) data structure, that is, it retrieves information in the reverse order that it was stored in [18]. It stores the recursive functions' parameters and local variables at the time it is called and thus saves the state of the function at the specific point during which it was executed. When the recursive function eventually returns via its base case, the stack unwinds.

Many popular IDE's provide functionality to display the current state of the call stack and allow the user to manually step through their code to see the pushing and popping of the recursive calls as their program executes. Figure 1 is an example of the way the call stack and variables are displayed from Oracle's IDE, "Netbeans" [17] (highlighted in red).

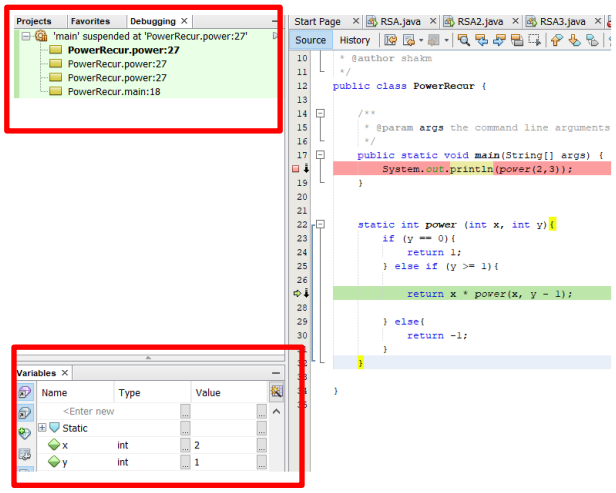


Figure 1: Screenshot of the Netbeans IDE call stack window[17]

The call stack displayed by Netbeans is not very visual in nature and only displays each recursive call with its input parameters. When a call is popped from the stack it simply disappears rather than display what value or string it returned in the process.

Fred Bartels [16] developed a visualization of the stack in order to better help students understand how calls are pushed and popped during the execution of a recursive program. The simulation made use of Google SketchUp to build a block tower with each block representing a call to the recursive function with its input parameter displayed on top of it. The simulation was premade and did not use actual code as input, however the way the stack was presented was simple and easy to understand.

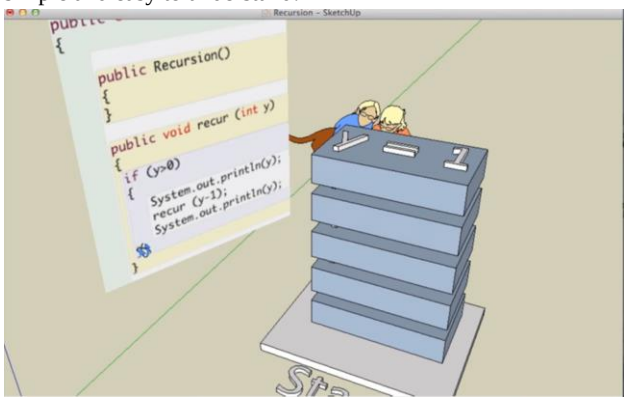


Figure 2: Stack visualization by Fred Bartels [28]

MUPPETS (Multi-User Programming Pedagogy for Enhancing Traditional Study) is a game where students are tasked with interacting with 3D objects in a virtual game environment [14]. Java is the required language for this game. Students can create their own code, edit existing code, compile and run said code and have direct feedback just like a traditional IDE would provide with the additional perk of having the changes appear in the game. The problem with this approach is that it barely differs from traditional coding assignments and simply adds a visualization to the students' completed code.

With regards to alternative methods of assessment, rather than the typical "code your solution in a regular IDE" type, an example is given below of the game "EleMental: The Recurrence" [15]. In EleMental, a code editor is provided as well as a simulated game world that visualizes the execution of the written code. The player is expected to edit the pre-written code in order to traverse a tree in this game. This game allows the user to see a direct relationship between their code and the visual simulation. The UI design allows for easy interaction with the code editor and provides a wide view of the game world.

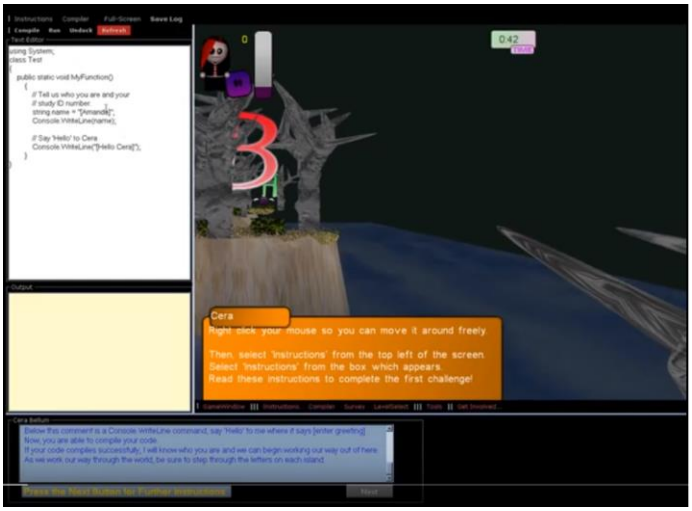


Figure 3: EleMental: The Recurrence screenshot [15]

3. REQUIREMENTS ANALYSIS

For requirements gathering, an interview was conducted with first year computer science students from the University of Cape Town. At the time the interviews took place, the students had already been taught recursion in their CSC1015F class. The software tool is aimed at these students which is the reason why the interviews were conducted with them. 10 first year students were interviewed.

The students were asked two questions relating to the stack simulation aspect of the tool to be designed. The first question asked if students found the concept of recursion challenging, and if so, what they found difficult about it. The second question asked if

the students would find a visual simulation of the call stack useful, and if so, what would they like to see in terms of functionality and visuals. Their responses were written down during the interview and summarized below:

80% of students responded saying that the way recursion was taught was very abstract and taught too fast. While the other 20% felt they understood the concept as it was taught. A student made the point that they found it difficult to shift thinking in terms of loops to thinking recursively. 50% of students made the claim that they needed more practice with the concept to familiarize themselves with it. At least 70% of students said they found it difficult to visualize what was happening within their code and not many good analogies were given to help them. The different parts of a recursive solution were hard to figure out such as what the base case should be or how the problem should get smaller with each iteration.

Due to the student's familiarity with standard IDE's such as Netbeans [17], there needed to be a text coding interface with similar functions or visual look.

In terms of what students would like in a visual stack simulation, there seemed to be an even split between students who preferred a 2D and 3D simulation. This seemed to be entirely up to preference as no strong opinions were expressed for either one. 60% of respondents claimed to make use of the debugger on their favourite IDE's (such as Netbeans [17]) in order to see some indication of the call stack but many claimed that it could be difficult to understand especially for more complex solutions. The simulation needs to be simple to understand. Students wanted to see the state of the parameters being passed to each recursive call in order to get a clearer picture of the shrinking problem size. Almost all students described some sort of 3D/2D render of an animated stack of books/blocks/plates growing and shrinking with the execution of their program when asked to come up with a visualization idea. It was suggested that there be an option to slow down or speed up the animation so students could understand it at their own pace.

The simulator should therefore be able to take a students' coded solution (in python) as input and generate a stack simulation based on how their program executes, specifically displaying how each recursive call is put on the stack (active flow) as well as popped from the stack (passive flow). The student should be able to see the parameters change with each call and what is returned. The simulation should have the option to manually step through the calls or have it play out automatically. There should be some form of error handling in the event of an incorrect solution being compiled.

Section 3.1 describes the student's interaction with the system (use case narrative). Describing what pre and post conditions are required for the successful use of the simulation. As well as the typical and alternate course of actions a student is able to take when interacting with the software.

3.1 USE CASE NARRATIVE

Expanded Use Case	Play a simulation
Actors	Student
Brief Description	Student compiles the given code and plays the simulation
Preconditions	Student must complete the necessary missing code. No stack overflow errors
Post conditions	Solution must be checked for accuracy and user notified

Typical Course of Events	
Student Action	**Simulator Response**
1. Complete correct coding solution	
2. Compile code	Generate .py and execute it. Play simulation
3. Click replay with step enabled	Replay simulation with manual step
Restart simulation	Reload entire question

Alternate Course of Events	
Student Action	**Simulator Response**
1. Complete incorrect coding solution	
2. Compile code	Generate .py and execute it. Display error
3. Edit code with correct solution	
4. Compile code	Generate .py and execute it. Play simulation

3.2 NON-FUNCTIONAL REQUIREMENTS

3.2.1 PERFORMANCE

The simulation should operate efficiently and smoothly with no drops in frame rate or stuttering. The student's code should be compiled quickly and output displayed.

3.2.2 USABILITY

The simulator should be easy and intuitive to make use of. All UI elements should have a clear function to avoid confusion. All user interaction should provide visual feedback.

3.2.3 COMPATIBILITY

The simulator program should be compatible with any standard Windows computer. No additional hardware is required for a smooth experience with the software.

Figure 4 is a use case diagram, outlining the basic functions the student is able to use while using the stack simulator as well as an activity diagram showing the series of actions the student can take.

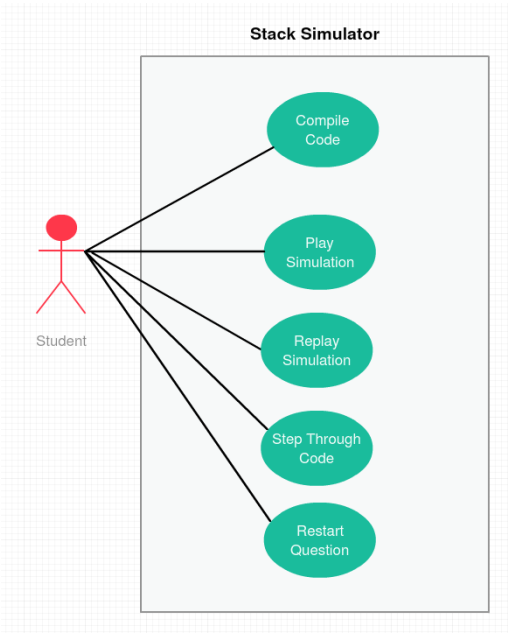


Figure 4: Use case diagram

students solution from the text editor; The question classes, which are responsible for modifying the students completed solution in order to get accurate information on the state of their variables during runtime (done using trace statements); and the stack creator, which makes use of the output of their program in order to produce the stack animation. A class diagram can be seen in Figure 5.

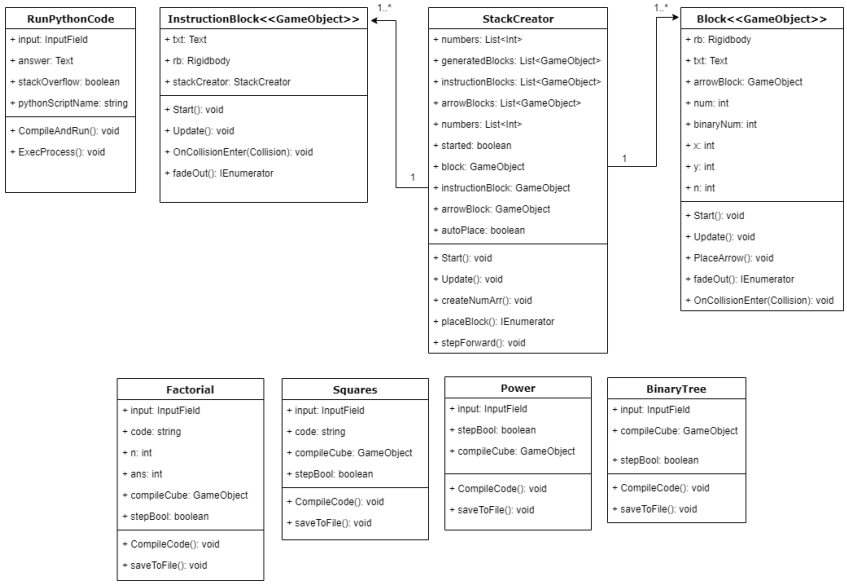


Figure 5: Stack simulation class diagram

4. SYSTEM DESIGN AND IMPLEMENTATION

This section outlines the development process that was taken and details the different components of the software and how they interact to produce the stack simulation.

4.1 SOFTWARE DEVELOPMENT PROCESS

An agile type software development approach was used with multiple iterations spanning one week each. Integration with the other components of the final software only needed to be done at the end and the stack simulation did not depend on any other aspect of the software and thus the development process went smoothly. The contents of each iteration can be seen in section 5.6.

4.2 HIGH LEVEL DESIGN

The design of the stack simulator can be broken up into 3 component: The python interpreter, which compiles and runs the

The structure and relationship between the classes needed to run the simulation are relatively basic. A brief explanation for each of the classes is described below:

RunPythonCode.cs – Takes the students input and compiles it using the New Process Initialization construct provided by C# as well as the installed python interpreter on the computer its running on. The output of the program is displayed to the student via the UI and the answer is checked to see whether it is correct or not or if there has been a stack overflow. A notification is displayed depending on this outcome. A text file containing the output of the trace statements is also produced.

StackCreator.cs – This class is responsible for reading in the text file created by the compiled python code and creating an array to store this information. If it was successful in reading this information, a coroutine is started which creates the stack animation using the block and instruction block game objects to do so. The animation runs until it is complete unless the student has selected the step toggle, which allows the stack to be created one block at a time.

Block.cs – When a block game object is instantiated by the stack creator, it checks what information needs to be displayed on it by accessing the array mentioned previously from the StackCreator class. The class also is responsible for the destruction/fading of the block when it gets popped from the stack.

InstructionBlock.cs – Similarly to the Block class, once instantiated by the stack creator, the information to be displayed on it is determined by its corresponding block game object's information (the block adjacent to it).

Factorial.cs, Square.cs, Power.cs and BinaryTree.cs – These classes are grouped together as they all do essentially the same thing. They are responsible for taking the students input and adding the necessary trace statements at the appropriate break points and saving them to the corresponding .py file that RunPythonCode.cs compiles.

Figure 6 below describes the flow of actions the user can perform. The simulator can be reset at any point.

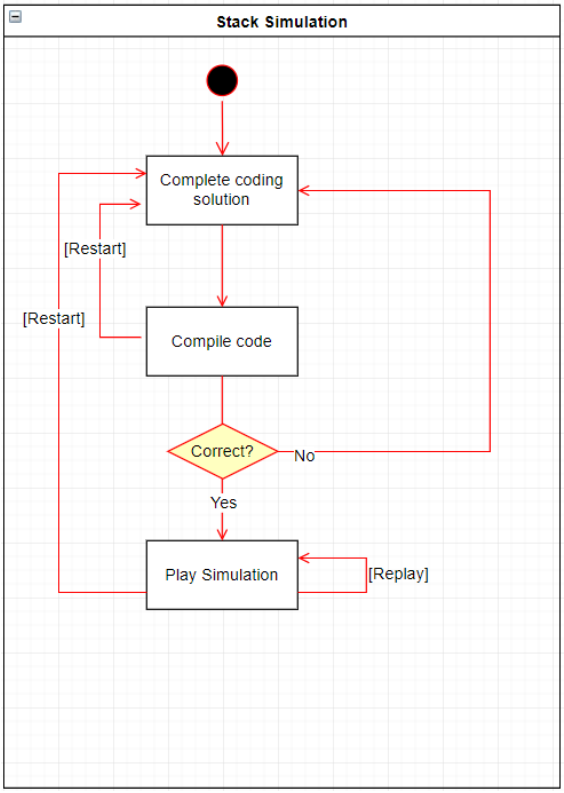


Figure 6: Stack simulation activity diagram

Additional system UML diagrams can be found on this project's website [20].

4.3 RECURSION QUESTIONS

The type of recursive problems decided on were based on a set criterion. Due to the software being aimed at first year students who would have only recently been introduced to the concept of recursion, the questions chosen are basic and number based. The number questions chosen are:

- Factorial – Calculating the factorial of a specific number recursively.
- Power – Calculating the power of an expression given a base and exponent recursively.
- Square – Calculating the square of a given number recursively.

An additional more complex question was added as a bonus exercise. This question requires performing a recursive post order traversal of a given binary tree.

Scaffolding code is provided to the students to use as a basis for their solution. This approach aims to limit the complexity of the problem and provides a starting point [15].

The solutions for each of the problems can be seen below. The omitted part of the solution that the student needs to complete are indicated by a red square. The code below was written in the Wing IDE.

Factorial:

```
#RECURSIVE FACTORIAL PYTHON

def factorial( n ):
        if n <1:   # base case
                return 1
        else:
                ans = n * factorial( n - 1 ) # recursive call

                return ans

#MAIN METHOD

print (factorial (3))
```

Power:

```
#Calculate the power

def power(x, y):
    if y == 0:

        return 1

    if y >= 1:


        return x * power(x, y - 1)


#MAIN METHOD

print(power(2,3))
```

Square:

```
(bottom)
def square(n):
    if n == 1:
        return 5

    else:
        return 5 + square(n-1)


print(square(abs(-5)))
```

Binary Post Order Traversal:

```
# Binary Tree

# A function to do postorder tree traversal
def printPostorder(root):

    if root:
        # First recur on left child
        printPostorder(root.left)

        # the recur on right child
        printPostorder(root.right)

        # now print the data of node
        print(root.val, end = ' ')

# Driver code

printPostorder(root)
```

It may be important to once again note that the simulator is aimed at first year students who most likely do not have much experience with recursion let alone programming in general. This was the reason the chosen questions are relatively basic as well as the fact that the solutions to these problems are simple. Students only have to complete a maximum of two lines of code. The emphasis is placed on the understanding of the active and passive flow of recursion rather than the coding of the solution.

4.4 RUNNING PYTHON SCRIPTS

The student's solution to the given question is written in python and therefore needs to be compiled by a python interpreter. It may be important to note that python needs to be installed on the machine that runs the software as the game makes use of the install path in order to compile the written code.

The process of compilation is as follows:

1) The written code is first saved to a string and written to a text file with a .py extension.
2) The saved python script is called using the New Process Initialization method provided by C#. This method requires an external python interpreter to be installed. The output of executing this script is saved.
3) Output is displayed to the student via the UI. If there is a stack overflow error (this is likely to be the most common error for recursive solutions other than syntax errors), a visual warning is displayed.

The alternative to using the basic New Process Initilization method is to use the IronPython interpreter (an open source implementation of the python programming language integrated with the .NET framework). It was decided not to use IronPython as the extra functionality it provides was not necessary due to the expected low complexity of the compiled code.

4.5 STACK CREATION

The creation and simulation of the stack is done at compile time. The stack needs to show various information relating to each recursive call such as the parameters passed and the return statement. This information is retrieved from the compiled python script in the following manner:

Once the student compiles their code, the contents of the text input field (see section 4.6) is first saved to an array with each element in the array corresponding to a line of code. This array is then modified and additional trace statements are added at specific points in the code, namely, before each return statement. Python provides the inspect and sys modules, which can be used to access variables used or maintained by the interpreter. These modules are used to attain the state of variables at different points during the execution of the program. The variables are written to a text file, which can then be accessed later in order to provide the necessary information for the creation of the stack. An example of the appended power.py file with these trace statements is shown in Figure 7 as well as the text file it produces in figure 8. The red indicates the lines of code that are added after the student has compiled the solution.

```
1  #Calculate the power
2
3  import inspect, sys
4  f = open("Assets\PythonScripts\PowerFile.txt", "w")
5  f.close()
6  def power(x, y):
7      if y == 0:
8          global out
9          out += str(x) + "," + str(y)
10         f = open("Assets\PythonScripts\PowerFile.txt", "a")
11         f.write("\n" + out)
12         f.close()
13         return 1
14
15     if y >= 1:
16
17
18         current_frame = inspect.currentframe()
19         calframe = inspect.getouterframes(current_frame, 2)
20         frame_object = calframe[0][0]
21         f = open("Assets\PythonScripts\PowerFile.txt", "a")
22         out += str(x) + "," + str(y)
23         f.write("\n" + out)
24         out = ""
25         f.close()
26
27         return x * power(x, y - 1)
28
29
30 #MAIN METHOD
31 global out
32 out = ""
33
34 print(power(2,3))
```

Figure 7: Power.py with trace statements

PowerFile - Notepad

File Edit Format View Help

```
2,3
2,2
2,1
2,0
```

Figure 8: Text file output of Power.py

The contents of the output file are stored in an array, which is used to put the necessary information on each stack block. This can be seen in figure 9.

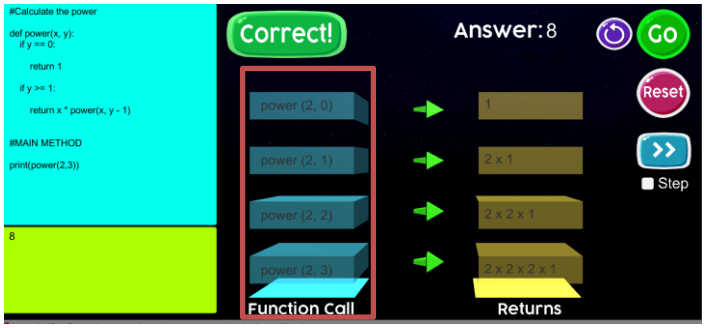


Figure 9: Fully built stack for recursive power

4.6 ERROR HANDLING

In order for the simulator to be robust, there needed to be sufficient error handling for when students enter the incorrect solution to the problem. Error handling is dealt with the python interpreter and displayed to the user via the output field. When dealing with problems that require a recursive solution, a common error students may encounter is the "Stack overflow" error. This error occurs when a program tries to use more memory space than the calls stack has available. A visual error in the case of a stack overflow is displayed as shown below.

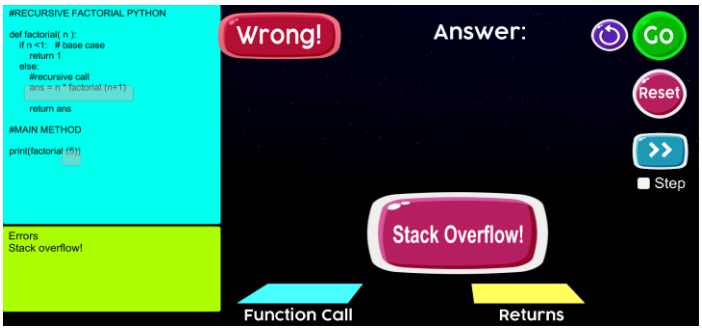
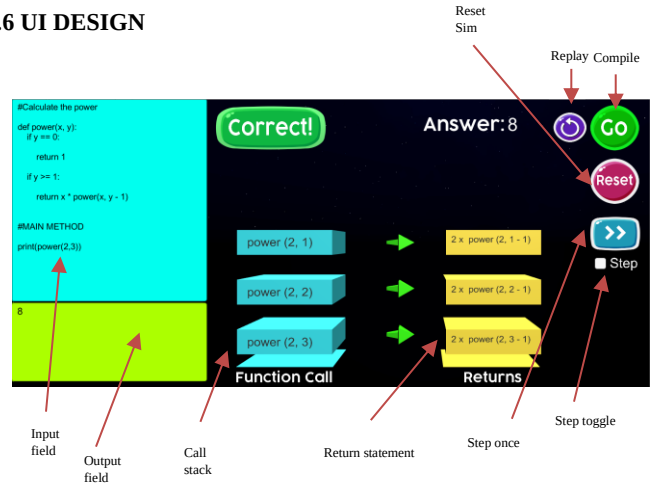


Figure 10: Stack overflow error

4.6 UI DESIGN



Main stack simulation UI

The UI was designed to be simple and easy to use, with emphasis on the actual stack animation. Simple and minimalistic colours and UI elements were chosen to not be a distraction from the simulation. The UI provides a basic button layout with an input field and output field. A large "correct" or "wrong" notification is displayed once the students' solution is compiled. The output is displayed in the output field as well as at the top of the simulation. An explanation for the function of each button can be seen below:

Compile button – Takes the student's input and compiles it.
Reset button – Resets the entire simulation and code.
Replay button – Only resets the stack simulation. Code remains the same.
Step toggle – Enables or disables the ability to slowly step through the stack creation.
Step button – Iterates once through the stack.

4.7 DESIGN PROCESS

The design process took a total of 7 iterations spanning one week each. What was achieved in each iteration is specified below:

Iteration 1: Developing a method of compiling python code in C# using built in libraries. This was described in section 4.4.

Iteration 2: Developing the general stack animation. This included the animated blocks being pushed and popped onto the stack.

Iteration 3: Designing the UI. This included gathering assets from the Unity Asset store specifically for the button designs.

Iteration 4: Designing the factorial and power question and creating a stack simulation for it.

Iteration 5: Designing the square and binary question and creating a stack simulation for it.

Iteration 6: Dedicated to bug fixes and tweaking visual elements of the UI.

Iteration 7: Integration with the other components of the system.

4.8 UNIT TESTING

Each python question was unit tested to ensure correctness. This was done using pythons built in "unittest" library. An example for the factorial unit test can be seen in Figure 11. Test cases for power, binaryTree and square share a similar format.

```
#Test case for factorial
class MyTest(unittest.TestCase):
   def test(self):
      self.assertEqual(factorial(5), 120)
```

Figure 11: Factorial 5 Unit Test Case

5. EVALUATION AND RESULTS

Evaluation was done with 20 first year computer science students currently completing their CS1016S course. The students were given a smaller version of the final software tool and tasked with completing a subset of the questions and, afterward, a questionnaire based on the popular standard Game Experience Questionnaire [17] to evaluate their user experience. The key aspects we aimed to look at are: usability, learning and gameplay experience. The students were asked to complete the factorial question only for this stack simulation section.

The usability of the text editor interface had a mostly positive response. Almost 90% of students claimed it was easy to use to type their own code. Syntax highlighting was a feature requested by one of the students. One student requested that it be clearer where they should type their own code, while another student wanted the specific line of code that was being executed to be highlighted as the code ran.

In terms of their interaction with the simulation itself, the simple block style of the simulation was a positive aspect about 50% of students enjoyed it with the other half not having any issues with it. One student wanted the speed of the animation to be variable as they found it to be slow. However, the step function solved this issue by allowing the student to go through the stack progress at their own pace.

90% of students rated the factorial question as beginner level and reported that the low difficulty of the question was a good idea so their focus could be on the actual stack simulation rather than their problem-solving ability. One student found it difficult to interpret the preexisting code and would have preferred to be able to write their entire solution to the problem themselves.

Overall, 80% of students claimed the software to be extremely helpful in supplementing their understanding of recursion and the stack. These student's claimed that such software would have been useful to have when they first started learning recursion as it demystified the concept of the call stack which they claimed was not easily grasped.

5.1 GAME EXPERIENCE QUESTIONNAIRE

The Game Experience Questionnaire (GEQ) is a standardized questionnaire developed to assess a users' experiences with various aspects of a given game [19]. There are 7 sections with each evaluated with a score out of 5. The scores for each section can be seen below.

Component	Score*
Competence	3.64
Sensory/Imaginative Immersion	4.11
Flow	3.65
Tension/Annoyance	1.3
Challenge	2.01

Negative affects	1.29
Positive affects	4.52

*1 – Not at all 2 – Slightly 3 – Moderately 4 – Fairly 5 - Extremely

Figure 12: GEQ Scores

The competence component assessed whether students felt they could use the simulator and answer the questions easily or not. Sensory and imaginative immersion involves the aesthetics of the simulator and whether users enjoyed the look and feel of it. The flow component aims to assess the difficulty curve of the questions and how the student's experience with the simulator changed as they progressed through it. The challenge component indicates whether the students felt challenged when answering the questions and using the simulator. The tension component assesses any difficulties or negative experiences with using the simulator. The negative and positive components are indicative of how boring, tiresome or fun and engaging using the simulator is.

The scores recorded in Figure 11 show that overall, the simulator is highly usable and enjoyable to use for students hoping to better understand the role of the stack when it comes to their recursive programs.

6. DISCUSSION

Based on the results from the students and literature review on visualization, games and the gamification of difficult to learn content in the context of computer science, this type of software tool can be hugely beneficial for students. Although gamification and visualization are used in many aspects when it comes to teaching programming, we find that not many attempts have been made to gamify or simulate recursion in a visual way, even though it is a fundamental concept especially for new computer science students. Students' interest and enthusiasm to engage with the content was apparent and many found it extremely useful to have an additional resource in the form of a simulation to help them understand recursion as a problem-solving method. Visualization was a key factor for learners in helping them understand the abstract concepts relating to the active and passive flow of recursion.

The usability of the software itself was largely positive. Even though the UI was largely simplistic, it was well received as students quickly understood what to do and what functionality was provided to them. The students could therefore focus on understanding the growing and shrinking of the stack as the simulator executed.

7. CONCLUSION AND FUTURE WORK

There is a clear opportunity to explore alternative methods of teaching recursion in the computer science classroom. Lectures and lab assessments are the current standard but there are clearly more ways of engaging students with programming material. Gamification and visualization are well received by students and many report how useful it can be to take a fresh new approach to learning a concept. Recursion, being one of the first and fundamental concepts students learn in the early part of any CS degree, is clearly a good option to explore and expand upon in terms of variety of teaching methods. The benefits of this cannot be understated as a good fundamental understanding of recursion is extremely useful for understanding more advanced problem-solving methods later.

A visualization of the call stack for recursive problems is not something that is deeply explored when the tradeoff of a good fully functional stack simulator could clear up many of the misconceptions of how recursion works behind the scenes, that being, the active and passive flow of the execution of a recursive program. The stack simulator that was developed and described in this paper shows that the concept can be visualized even if it only operates for a certain set of questions. The feedback from new computer science students was overwhelmingly positive and it's clear that if such a software was introduced on a larger scale in lectures to supplement lectures on recursion, the benefits could be substantial in aiding students understanding. Having a solid fundamental understanding of recursion at an early stage can have significant benefits for students later on in their computer science careers.

A more general simulator that works for any type of program (not just recursive ones) that can take the form of an IDE plug-in or IDE itself could be the future of visual programming. The simulator described in this paper could form the basis of a better, more robust and general stack simulator that could be used for more than just recursive programs. The main limitation of the simulator described in this paper is the fact that it is limited to 4 recursive questions and was designed specifically with these questions in mind. As a result, any other type of question's solution may not function correctly in the simulator. With the use of Unity as the development platform, future iterations may be ported for use in a web browser using Unity's built in Web Player. This would allow easier and more cross platform access.

It is hoped that any deeper exploration into the topic of visualizing and simulating recursive programs may find the stack simulator developed in this paper to be an inspiration for any future software tools developed for this purpose.

REFERENCES

1. McCracken, D.D. Ruminations on Computer Science Curricula. Communications of the ACM. 30, 1: (January 1987), 3-5.
2. Eagle, M., & Barnes, T. (2009). Experimental evaluation of an educational game for improved learning in introductory computing. ACM SIGCSE Bulletin, 2009, 321-325.
3. Thomas L. Naps. Rudolf Fliesher. Myles McNally. 2002. Exploring the role of visualization and engagement in computer science education.

4. John Stasko, Albert Badre. Clayton Lewis. 1993. Do Algorithm Animations Assist Learning? An Empirical Study and Analysis.
5. Tessler, J., Beth, B., & Lin, C. (2013). Using cargo-bot to provide contextualized learning of recursion. Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research - ICER '13. doi:10.1145/2493394.2493411
6. Kazimoglu, C., Kiernan, M., Bacon, L., & Mackinnon, L. (2012). A Serious Game for Developing Computational Thinking and Learning Introductory Computer Programming. Procedia - Social and Behavioral Sciences, 47, 1991–1999. doi:10.1016/j.sbspro.2012.06.938
7. Ginat, D, Shifoni, E. (1999) Teaching Recursion in a Procedural Environment - How much should we emphasize the Computing Model? ACM SIGCSE Bulletin 35(1), 346 doi:10.1145/792548/612004
8. Tamarisk Lurlyn Scholtz and Ian Sanders. Mental models of recursion: Investigating students' understanding of recursion. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '10, pages 103–107, New York, NY, USA, 2010. ACM.
9. Wirth, M.: Introducing Recursion by Parking Cars. SIGCSE Bulletin 40(4), 52–55 (2008)
10. Edgington, J.: Teaching and Viewing Recursion as Delegation. J. Computing Sciences in Colleges 23(1), 241–246 (2007)
11. Pirolli, P.L. and Anderson, J. R'. The role of learning from examples in the acquisition of recursive programming skills. Canadian Journal of Psychology 39 (1985), 240-272.
12. Kurland, D. M. and Pea, R. D. Children's mental models of recursive LOGO programs. In Proceedings of the 5th Annual Conference of the Cognitive Science Society (1983), Session 4,,1-5.
13. Kahney, H. (1983). What do novice programmers know about recursion. Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI '83. doi:10.1145/800045.801618
14. Bierre , Kevin J. and Andrew M. Phelps. The use of MUPPETS in an introductory java programming course, SIGITE 2004, October 28-30, 2004, Salt Lake City, UT, USA.
15. EleMental: The Recurrence Andrew Hicks, Katelyn Doran, Graduate Advisor: Amanda Chaffin, Mentor: Dr. Tiffany Barnes
16. Fred Bartels (April 2012) *Recursion and the Stack.* Available from: https://www.youtube.com/watch?v=s8JpA5MjYac
17. Oracle's Netbeans IDE [Computer Software], (2018) Version 8.2.Retrieved from www.netbeans.org/downloads/8.2/
18. Harry Fairhead (March 2019) The LIFO Stack – A Gentle Guide, viewed 20 August 2019 <https://www.i-programmer.info/babbages-bag/263-stacks.html?start=1>
19. IJsselsteijn, W. A., De Kort, Y. A. W., & Poels, K. (2013). The game experience questionnaire. Eindhoven: Technische Universiteit Eindhoven.
20. Honours Project Website: http://projects.cs.uct.ac.za/honsproj/cgi-bin/vuew/2019/tony_shak_raeez.zip/csa/index.html.