Application of Procedural Generation as a Medical Training Tool

J. Duffy [2] and Z. Wang [1, 2, *]

[1] College of Information Technology, Taiyuan University of Technology, Taiyuan, Shanxi, China

[2] Department of Math/CS, Virginia Wesleyan College, Norfolk, Virginia, USA

Abstract: *Procedural Generation is an algorithmic process that enables the creation of a variety of content without the need to create assets specific for each case. The purpose of this project is to explore a practical application procedural generation through the use of a basic technique of this process, a pseudo random number generator (PRNG), to create a simple medical training simulation. The simulation is able to create a variety of cases with unique details (such as ages and names), and has the potential to be scaled up to accommodate additional information or assets as needed. Resulting cases so far disassociate background information from the actual symptoms generated for a case, but with the implementation of an input system for a user's answers it serves as a functioning programming example of the procedural generation process.*

Keywords: Information systems, medical informatics, procedure generation

1. Introduction

"Procedural Generation" is a programming technique that allows for the creation of content through the use of an algorithm, rather than having to manually create it all beforehand. It is used in various media, some of the most common being computer graphics and electronic games. One approach to this method is to create the resources and assets for the program to choose from, and then having "assemble" content in a way specified by the programmer, which is the demonstrated method in this project. While procedural generation has gained an interest in computer graphics and games [1], there are much fewer mentions of it being used outside of those fields.

* Corresponding author. Email: zwang@vwc.edu

An interest in finding a practical use for procedural generation, as well as interest in the topic itself, led to the creation of this research project. This study explores the practical applications of procedural generation techniques. In particular, their use as a medical training tool [2]. By using a technique of procedural generation, the pseudorandom number generator, a basic medical diagnostic simulation was constructed in the Java programming language. The technique allows for the creation of many different cases without having to create a large amount of assets, reflecting the variation a medical professional would encounter in their line of work. The program is able to create a large variety of cases as well as prompt the user for diagnostic input. The program then checks to see if the input is correct based on the generated patient's case, and gives feedback accordingly.

The paper is organized as follows. The system overview is presented in the next section. The case study for most of the simulation programs is discussed in the section 3. A conclusion is followed in the last section.

2. System Overview

The goal of the project was to find a simple yet practical application of procedural generation. This was achieved by planning and creating a very basic (medically speaking) training program for those studying or practicing in the medical field.

The PRNG: The "PsuedoRandom Number Generator," (referred to as the "PRNG") is one of the elementary procedural generation techniques, and was selected due to its relative ease of implementation in the project. They are called "psuedorandom" because they rely on something predictable in order to create a random number. For this project, the PRNG relies on the system's internal clock time in milliseconds to create the numbers. The program itself then formats

those numbers into a form it uses to create a patient case.

The Patient: Given the complexity and the sheer amount of information that it used to create an actual, "real life" medical case and project time constraints, the patient cases presented by the program will have only the basic details. A generated case will contain a patient's first and last name, blood type, and age, as well as basic medical symptoms that they are presenting.

User Input: The program generates a multiple choice prompt for the user based upon the details contained within the patient's data. This method of user input was selected as it would be a familiar format for the user and its ease of use, in addition to it suiting the simplistic nature (medically speaking) of the cases.

3. Case Study

3.1 PRNG.JAVA

The PRNG (**P**seudo**R**andom **N**umber **G**enerator) class uses the system's hardware clock in milliseconds to create a random number, and is one of many procedural generation techniques. The PRNG class truncates the number it generates to four digits, which are used by the program itself to determine the characteristics of the Patient case. A summary of the PRNG code, including its methods and variables (excluding those used for testing) is as follows. An example is shown in Figure 1.

Variables

Random pseudo – A Random number generator that uses the method *System.currentTimeMillis()* to create a random double.

int seeded – a four element array that is used to store four digits of the generated number

Methods

double *prng()* – returns the double created by the number generator pseudo.

void *numGen()* – uses *prng()* to generate a number, converts that number to a string, creates a substring then converts that substring to integer digits that are stored in seeded.

int *getDigit*(int e) – returns the integer stored at element e in seeded.

int *getDubDigit(*int a, int b) – returns the digits stored in elements a and b in seeded as a double digit integer.

```
Our truncated random number: 1752
Our truncated random number: 3239
Our truncated random number: 4625
Our truncated random number: 0151
Our truncated random number: 4356
Our truncated random number: 2706
Our truncated random number: 0903
Our truncated random number: 0029
Our truncated random number: 4119
```

Figure 1 The numbers generated by the PRNG are truncated to 4 digits, which determine the characteristics of the patient

3.2 SYMPTOM.JAVA

The Symptom data class is used in the construction of the Patient case, and is used as part of the user prompt in the Main method file. The actual Symptom object itself stores an ID number, its name and description as well as its solution. However, this solution is not used by the object but rather in the user prompt. The class contains several "get" and "set" methods used to respectively return and change the data contained within a Symptom object. However, most of these methods are almost identical in structure and were mostly used for debugging and testing, and as such will be excluded from the following summary. Additionally, only the full constructor is listed for similar reasons:

Variables:

int id – an integer that serves as the Symptom's identification number

String desc – contains the Symptom's description

String name – the Symptom's name

String ans – stores the Symptom's answer, which is used in the Main method

Constructor:

Symptom(int i, String n, String d, String a) – instantiates a Symptom object with the specified data. If no data is set, the program uses the default constructor.

Methods:

void *SetData(*int i, String n, String d, String a *)* – sets the data of a Symptom object with the specified data

String *toString()* – returns a String containing the name, ID number and description of the Symptom object, shown in Figure 2.

```
public String toString()
{ return name+" (SYMPTOM ID: "+id+"): "+desc;}
```

Figure 2 The toString() method illustrates the object's data structure

3.3 SYMPDAT.JAVA

SympDat, short for "Symptom Data," is a class used primarily in the construction of the data matrix to be used in the Main method file (shown in Figure 3). The two constructors allow it to take either the same data used to instantiate two Symptom objects or two, actual Symptom objects. The SympDat class itself is not overly complex, and contains many of the same methods used in the Symptom class. The only difference is that there are redundant methods that only apply to a specific Symptom object stored within a SympDat object, but otherwise contains the same "get" and "set" methods as well as a *toString()* method that returns a String containing Symptom data in a similar fashion.

```
public SympDat(Symptom S1, Symptom S2)
{
  sy1 = S1;
  sy2 = S2;
}
```

Figure 3 A SympDat object contains two Symptom objects, for later use in the Main.java file

3.4 PATIENT.JAVA

The Patient class is one of the two major components to the program itself, as it contains the actual data itself used to create a Patient case and utilizes other classes to a large extent. It instantiates a PRNG object and then uses the numbers generated to determine which details are used to construct a Patient object. The data includes details such as names and blood type, with other variables being set through the use of the included methods.

Variables:

String fName – first name of a Patient

String lName – last name of a Patient

String bType – blood type of a Patient

int age – age of a Patient

Symptom s1 through s4 – the four Symptom variables store two Symptoms that are to be used in Main as correct answers and two that are used as wrong answers

Data:

String[] *firstName* – contains all of the first name Strings used by the program (contains 10 Strings).

String[] *lastName* – contains all of the last name Strings used by the program (contains 10 Strings).

String[] *bloodType* – contains all 8 medically known human blood types as Strings.

Constructor: See Figure 4.

```
public Patient(String fN, String lN, String bT, int a, Symptom o, Symptom t, Symptom th, Symptom f)
{
    fName = fN;
    lName = lN;
    bType = bT;
    age = a;
    s1 = o;
    s2 = t;
    s3 = th;
    s4 = f;
}
```

Figure 4 The Patient Constructor

The Patient object takes three Strings, (first and last name, blood type) an int (age) and four Symptom objects to fully instantiate. The default constructor creates a Patient without any names or blood type, sets age to 20 and manually sets the Symptom ID numbers. The purpose for this object is to create the cases in Main and to demonstrate the procedural generation aspect of the program as a whole.

Methods:

void *createSymptoms()* – takes two digits given by PRNG, formats them into a form useable by the method, and then sets the ID numbers of two the first two Symptom objects. Then it sets the ID number of the Symptoms used for the wrong answers. Included within the method's programming are checks that ensure no two ID numbers are alike, ensuring that they are all unique and that there are no overlaps.

void *grabName()* – takes two digits given by PRNG and selects those respective elements from the *firstName* and *lastName* String arrays, then sets fName and lName.

void *genBlood()* – takes one digit given by PRNG and uses it to select a blood type from the *bloodType* array. Since there are only 8 medically accepted blood types, there is also a check to ensure that the method doesn't attempt to select an element outside of the array's index.

void *createPatient()* – a combination of *createSymptoms()*, *grabName()*, and *genBlood()*.

There is also a function to set the age of the Patient, achieved by using the PRNG's *getDubDigit()* method.

String *toString()* – this method is particularly important, as it is used by Main when outputting cases to the screen. It returns a formatted String containing the first and last name of a patient, the blood type, age, and the first two Symptom objects. It does not output the last two Symptom objects, as they are intended for use in Main to create the wrong answers.

3.5 MAIN.JAVA

This is the code that creates the cases, asks for user input, and then prompts for user input. There are no methods defined in this class, but instead contains fully instantiated Symptom data to be used by the cases themselves. It is able to create a new case for the user each time the program is run or loops. The code for Main.java is summarized in Figure 5.

```
Scanner s = new Scanner(System.in);
String a = "y";
Boolean again = true;

//Our symptom data
Symptom s1 = new Symptom(0, "Ache", "Patient has aching muscles", "Advise mild pain medication
Symptom s2 = new Symptom(1, "Cold", "Patient is showing signs of a basic cold", "Advise increa
Symptom s3 = new Symptom(2, "Bruise", "Patient has visible bruising from a sport injury", "App
Symptom s4 = new Symptom(3, "Cut", "Patient has a visible, mildly bleeding cut", "Clean the af
Symptom s5 = new Symptom(4, "High Temperature", "Patient's body temp. is higher than average",

Symptom[] data = {s1, s2, s3, s4, s5}; //an array of our symptoms - to fill the data matrix ea
SympDat[][] sd = new SympDat[5][5]; //The data matrix

Patient test = new Patient(); //Our "patient"
```

Figure 5 Data and variables used in Main.java

The beginning code block (Figure 6) contains the data and variables used for the prompt later on in the program. Note the structure of a fully instantiated Symptom object contains its ID, a description and its solution.

In Figure 7, the program initializes the data for the SympData matrix. Once completed, the program then continues into the main code loop. It begins by setting the data for the Patient, and then setting the data for the Symptom objects contained within Patient with information from the matrix.

```
//This loop initializes the data in the SympDat matrix
 for(int c = 0; c<5; c++)
 {
        for(int d = 0; d<5; d++)
        {
            if(d != c)
            {
            sd[c][d] = new SympDat(data[c], data[d]);
            }
        }
 }

while(again) //A while loop in place in case the user wants to run
{
test.createPatient(); //create the patient

//set the data for the Symptom contained in the Patient object
for(int z = 0; z < 5; z++)
{
        int o = test.getSympOneID();
        int t = test.getSympTwoID();
        int th = test.getSympThreeID();
        int f = test.getSympFourID();

        if (o == data[z].getID())
            test.setSympOne(data[z]);
        else if (t == data[z].getID())
            test.setSympTwo(data[z]);
        else if (th == data[z].getID())
            test.setWrongOne(data[z]);
        else if (f == data[z].getID())
            test.setWrongTwo(data[z]);
```

Figure 6 The initializing code

In Figure 8, the code creates a case and user prompt based entirely on the information contained within the Patient object. The prompt asks for two user integer inputs, which correspond to the numbers presented by the program in a "multiple choice" format. Note that the "*wrongOne()*" and "*wrongTwo()*" methods are the same as a "get" method but under a different name, hence why they were not mentioned earlier.

Finally, the program performs some logic checks to see if the user inputs were correct. It should be noted that not all of the logic coding is presented in the above picture, but the program is able to determine whether the user inputs are correct or incorrect and even specify which answers are incorrect. It then asks the user if they would like to run the code again and try another case, ending if they do not.

```
System.out.println(test.toString());

int correct1 = test.getSympOneID();
int correct2 = test.getSympTwoID();
int wrong1 = test.getSympThreeID();
int wrong2 = test.getSympFourID();


//The list of choices for the prompt
System.out.println("\nYour following treatment options: ");
System.out.println(test.getSympOneID()+": "+test.getAnsOne());
System.out.println(test.getSympTwoID()+": "+test.getAnsTwo());
System.out.println(test.getSympThreeID()+": "+test.wrongOne());
System.out.println(test.getSympFourID()+": "+test.wrongTwo());

//Prompt user to enter their answers
System.out.println("Enter the number corresponding to correct treatment: ");
int ans1 = s.nextInt();
System.out.println("Enter the second number corresponding to correct treatment: ");
int ans2 = s.nextInt();
```

Figure 7 Creating the user prompt

```
else if(ans2 == wrong1 || ans2 == wrong2)
{
    System.out.println("INCORRECT: Your second answer is wrong");

    if(ans1 == wrong1 || ans1 == wrong2)
      System.out.println("INCORRECT: Your first answer is also wrong");
}

//Ask if they would like to do another case
System.out.println("Try another case? (y/n)");
a = s.next();

//Check to see if they said "y"es
if(a.equalsIgnoreCase("y"))
 again = true;
else
 again = false;

}
```

Figure 8 A sample of the logic coding and a prompt asking if the user wants to run the program again

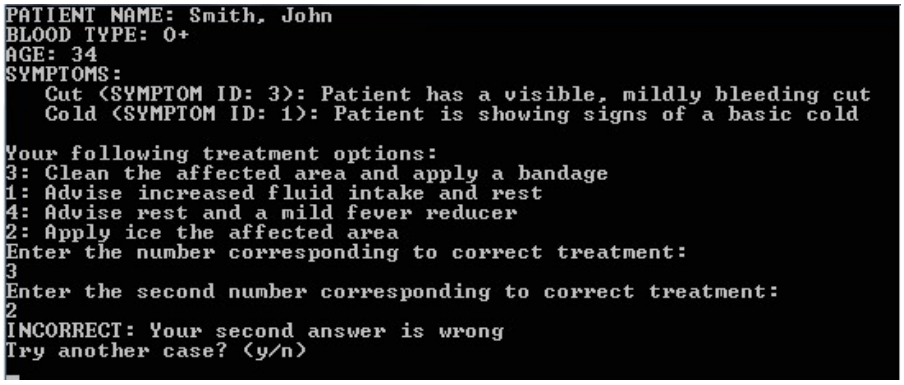

Figure 9 Example of demonstration displayed

Figure 9 demonstrates what the end user sees when they run the program, and what a fully generated case and prompt look like. For this particular case, our patient's full name is John Smith, aged 34, and has a O+ blood type. His symptoms include a bleeding cut and he appears to have a cold. The treatment options are based upon the Symptom objects contained within the Patient object, hence why it stores four but only displays two of them, as the other two are used to create possible incorrect answers.

4. Conclusion

Through the creation of a small amount of assets, the program is able to create a large amount of cases through the use of algorithms, demonstrating its practical use in a training scenario where such a number of cases can occur in the field. None of the cases are manually created beforehand, and if they were, there would need to be 2.4 million premade cases – a task that would take a large amount of time through traditional methods and create a large data file. The data used in the project can be easily increased to allow for many more cases to be generated by the program. Furthermore, additions to the program can increase its worth as a training tool, such as including the patient's temperature or having the patient's medical details influence what symptoms they present, and would be possible with more time and testing.

5. Reference

[1] H. Smith. Procedural Trees and Procedural Fire in a Virtual World, White Paper, Intel Software and Services Group, 2008. https://software.intel.com/sites/default/files/Procedural-Trees-and-Procedural-Fire-Wp.pdf

[2] E. H. Shortliffe, et al. Medical Informatics, Computer Application in Health Care and Biomedicine. 2nd ed. Springer-Verlag. 2001.