Procedural Generation and the Creation of a Pseudo-Infinite Universe

J O A K I M  R A S M U S O N

Bachelor of Science Thesis
Stockholm, Sweden 2012
Procedural Generation and the Creation of a Pseudo-Infinite Universe

JOAKIM RASMUSON

DD144X, Bachelor’s Thesis in Computer Science (30 ECTS credits)
Degree Progr. in Computer Science and Engineering 300 credits
Royal Institute of Technology year 2012
Supervisor at CSC was Björn Thuresson
Examiner was Olle Bälter

TRITA-CSC-E 2012:059
ISRN-KTH/CSC/E-12/059--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden
URL: www.kth.se/csc
Abstract

Procedural generation refers to the generation of media and content by means of algorithms and functions instead of manual labor. The many techniques of procedural generation are ways to get a lot out of a little and can therefore be of great importance for software projects when it comes to saving storage, memory, man hours and more.

During the course of this project I have analyzed and implemented different techniques of procedural generation within three different categories of content. The topic of the first chapter is natural language generation, in which two types of generators are created. One for generating natural-sounding words by use of a Markov chain, and another for building sentences. For the second chapter I have looked at the simplified genetics of Richard Dawkins’ virtual creatures called Biomorphs. A basic evolutionary process is here created where genomes are generated, mutated, and at times selected for reproduction.

In the third and last chapter I make use of procedural generation to generate a pseudo-infinite universe. For this I use a pseudo-random number generator and a system of seeds to keep the universe intact when it enters and exits memory, even though it is never stored. Some of the concepts of earlier chapters show up again for this part, and working together they show a great little example of the possibilities that are present while working with procedural generation.
Procedurell generering och skapandet av ett pseudoändligt universum

Begreppet procedurell generering syftar till skapandet av media genom algoritmer och funktioner i stället för manuellt arbete. Procedurell generering ger möjligheten att få ut väldigt mycket ur väldigt lite och kan därför vara av stor vikt för mjukvaruprojekt när man vill spara in på bland annat lagring, minne och arbetstid.


I det tredje och sista kapitlet används procedurell generering för skapandet av ett pseudoändligt universum. Under genomförandet används en pseudoslumptalsgenerator och ett system av tillstånd i skapandet och återskapandet av stjärnor, solar, planeter och andra himlakroppar. Vissa av de koncept som behandlats i tidigare kapitel kommer här åter dyka upp, och tillsammans med de nya ska de visa exempel på de möjligheter som procedurell generering för med sig.
Contents

Introduction 1
  Tools ....................................................... 1

Text generation 2
  Word-based generator .................................. 2
  Markov chain text generator .......................... 4
  Combining methods .................................... 6

Biomorphs 8
  Analysis ................................................ 8
  A basic implementation ............................... 9
  Another dimension and more freedom ............... 11

Pseudo-infinite universe 14
  Elite .................................................... 14
  Building a universe ................................... 14
  Visualization .......................................... 18
  One possible addition ................................ 18

Discussion 20

References 22
Introduction

The aim of this project is to explore a few of the different types of procedural generation that are used today in computer game development. The implementation of this project will contain two principal parts: one analytical and one practical, although these will be integrated into each other. For the methods and techniques studied in the analytical part of the project, there will be a corresponding section in the practical part where an implementation of the subject in question is made.

One final aim is to have a set of tools in the end, a set of techniques all within the subject of procedural generation, and combine these in one application. For this reason the topics have been picked based partially on their relative diversity. No two algorithms will be studied that handle the same type of content, at least not in detail. The goal with this is to in the end of the project have a non-overlapping repertoire of algorithms that can work together.

Tools

During this project I will work with Apple’s Xcode, Objective-C, and for anything visual, the iPhone SDK with OpenGL ES. The primary reason for this is the iPhone itself. It is a powerful device, with a lot of functionality, such as accelerometer, compass, camera, microphone, internet connection, etc.

Objective-C has a syntax that differs from many languages and looks confusing at first, but it is powerful and since you can write pure C in it there are a lot of possibilities. It is also well integrated in Xcode which will make the use of it even easier.
Text generation

Everything should have a name. And when content is generated procedurally then the number of elements in need of naming can grow pretty great, pretty fast. The theme of the first chapter is text generation, and I will go about it looking at two different techniques. Seventh Sanctum\(^1\) is a website run by Steven Savage. Basically it’s a very large collection of generators for different purposes, character names with all kinds of themes, and even various kinds of descriptions. Along with a large library of generators Steven Savage also publishes a lot of source code and articles on techniques used in his creations.

Word-based generator

Many of Steven Savage’s generators use a database of words and predefined algorithms for combining these. The themes of these range from wrestling moves and military units to space phenomena and science-fiction medical tools. In the spirit of being an Erasmus student I chose to make my word-based generator spit out descriptions for places to go to on a night on town - a nightclub theme generator.

Example analysis

First off: examples, in this case anything that could answer the question *So where are we going tonight?* Possible candidates might include:

- Rock bar
- American style pub
- Up beat electro club
- Modern fusion club
- Hip nightclub

Here we use a system borrowed from Steven Savage, his DOA system - descriptor-object-actor system. It’s very simple, and makes use of patterns very common in naming conventions in general. Basically the system divides all usable words into three categories, letting names be generated usually as a two or three word combination of these. The descriptors modify both objects and actors. Often these will be adjectives but it can vary. Objects are often nouns and can be paired with an actor. Actors being the most important part of the generated phrase are usually placed at the end. The above examples subdivided with the DOA system would result in the following:

\(^1\)http://www.seventhsanctum.com/
With the data categorized it’s time to define some patterns of occurrence. Of course a word-based generator can be as advanced or even more so than the language it handles, but to keep ourselves within the time and resources of this project, we will stick to three fairly simple and clear rules:

- Descriptors may modify objects and actors, placed before these
- Objects may modify actors, placed before these
- A generated phrase always ends with an actor

**Implementation**  When analysis is done then so is most of the work, at least when we’re handling such a simplified model of natural language. First off we place our data, being our words, in three different arrays, one for each category. Based on our above mentioned rule set we will allow three different sample subspaces: Descriptor-object-actor, descriptor-actor and object-actor. Since I didn’t have a very large number of examples to analyze I chose to give each one of these a one in three probability. After that I simply let the program pick from the data arrays in a random manner to generate the outcomes.

**Listing 1: Output from Word-based generator**

```
TextGenerator[540:903] relaxed bar
TextGenerator[540:903] house bar
TextGenerator[540:903] irish disco
TextGenerator[540:903] american style house disco
TextGenerator[540:903] american style house disco
TextGenerator[540:903] modern club
TextGenerator[540:903] modern disco
TextGenerator[540:903] funk pub
TextGenerator[540:903] hip house bar
TextGenerator[540:903] fusion pub
TextGenerator[540:903] modern funk pub
TextGenerator[540:903] relaxed techno pub
TextGenerator[540:903] irish pub
TextGenerator[540:903] relaxed rock pub
TextGenerator[540:903] relaxed club
TextGenerator[540:903] fusion bar
```
Some of the generated output looks decent, *Irish pubs*, *fusion bars* and *relaxed rock pubs* are all places that wouldn’t be too hard to find in a big city. Then again other generated results don’t have as good a ring to it, I hear *Irish discos* aren’t very popular and the paradoxical concept of a *relaxed techno pub* is probably unheard of. Of course this generator doesn’t take into account the actual meaning of the words it combines. This artificial intelligence will generate a string which is grammatically correct, but it won’t exclude bad ideas. My implementation of Steven Savage’s *wrestling move generator*\(^2\) generates more cohesive results, simply because there’s only one theme: hurt.

Listing 2: Output from Wrestling move generator

<table>
<thead>
<tr>
<th>TextGenerator</th>
<th>move</th>
</tr>
</thead>
<tbody>
<tr>
<td>1420:903</td>
<td>half ankle chop</td>
</tr>
<tr>
<td>1420:903</td>
<td>double rasmuson lock</td>
</tr>
<tr>
<td>1420:903</td>
<td>quadruple ankle twister</td>
</tr>
<tr>
<td>1420:903</td>
<td>reverse mandible lock</td>
</tr>
<tr>
<td>1420:903</td>
<td>triple nerve nelson</td>
</tr>
<tr>
<td>1420:903</td>
<td>trapezius twister</td>
</tr>
<tr>
<td>1420:903</td>
<td>ankle lock</td>
</tr>
</tbody>
</table>

**Markov chain text generator**

One technique for generating names is by means of a Markov chain. It is an interesting one because as opposed to a generator that combines words from a given resource in a random manner a Markov chain text generator can create new words.

A Markov chain text generator generates words by probability. By means of a two-dimensional matrix, or a table, containing the probabilities that any given letter will succeed the last letter generated. To start off a source text needs to be analyzed. The source can be written in any language so long as it can be written with some kind of phonetic alphabet. Of course languages with one character for one word can be used for this kind of text generation too, but naturally you will need a very large source text to get useful probabilities and the result will be a sequence of words, not one of letters.

**Source text analysis** For analysis we start at the first pair of letters \(ij\) in the source text, we go to our table of probabilities, row \(i\), column \(j\), and add one for one occurrence. We move one step in our source text to the next pair \(jk\), that is the second and third letter, after which we add one to row \(j\), column \(k\) of our table. This procedure continues until the end of the source text is reached, and depending on the size of the source, we should then have a pretty valid table of probabilities.

\(^2\)http://www.seventhsanctum.com/generate.php?Gensave=wrestling
Text generation  Now that we have a table of probabilities, all we need is a pseudo-random number feed. First we want a starting letter, and for that we’ll use the column of probabilities for the space character. We summarize all the probabilities of this column and generate a random number \( r \) between one and this number. We’ll iterate through the column, subtracting the elements we pass from \( r \) until \( r \leq 0 \). The correspondent letter \( l \) will be the start of our word. Next we perform these steps again, but now in the column \( l \) of our table. This procedure is repeated until we once again reach the space character, or until we reach a previously defined letters-per-word maximum. Of course to avoid unnecessary calculations it is preferred to use floating point values instead of integers in the probability matrix, and divide each element by the sum of its row. This way the program can ask for a random number between 0 and 1 avoiding to have to add up all the row elements before iterating.

As is shown in the following output of my program Markovian List some generated words may look natural, like *marmengar, lernera* and *dext*, some even happen to be real words like *öra* and *kakan*, but most don’t make much sense - *bexttegrkäntt*. As source I used the article Så Skriver Man\(^3\), written in Swedish.

Listing 3: Output from Markovian List(ä and ö replaced with AE and OE)

```
MarkovianList [29446:903] String : marmengar
MarkovianList [29446:903] String : et
MarkovianList [29446:903] String : kesOEkAER
MarkovianList [29446:903] String : lernera
MarkovianList [29446:903] String : er
MarkovianList [29446:903] String : OEra
MarkovianList [29446:903] String : datvuban
MarkovianList [29446:903] String : omett
MarkovianList [29446:903] String : kettAEs
MarkovianList [29446:903] String : kedint
MarkovianList [29446:903] String : sAEdid
MarkovianList [29446:903] String : it
MarkovianList [29446:903] String : dext
MarkovianList [29446:903] String : da
MarkovianList [29446:903] String : hior
MarkovianList [29446:903] String : t
MarkovianList [29446:903] String : t
MarkovianList [29446:903] String : bexttegrkAEntt
MarkovianList [29446:903] String : in
MarkovianList [29446:903] String : kakan
MarkovianList [29446:903] String : kar
```

\(^3\)http://www.fritext.se/svenska/texttyp/skarm.html
Second order generator  A Markov chain text generator can be implemented in many ways. The above implementation is of the first order, meaning we only look at the single last letter before we generate the next. A stronger model would be a generator of the second order, with a table of succession probabilities of any letter after any pair of letters. The steps of implementation don’t differ a lot between the first and second order generators: Instead of a $N \times N$ matrix ($N$ being the number of supported characters in the alphabet used) we use either a $(N \times N) \times N$ matrix or a three-dimensional $N \times N \times N$ matrix. With a larger number of possible states we should analyze a bigger source for the probabilities. For my implementation of the second order generator I used Wikipedia’s article on The United States\textsuperscript{4}, written in English. Furthermore I added a lower letters per word limit, causing words that come out too short to be discarded.

As can be seen in the output below the second order generator is far superior to the first one. Many words produced look and sound very natural, and unnatural letter combinations like $xtt$ and $ntt$ are far less frequent.

Listing 4: Output from Markovian List Second Order

| MarkovianList2o [712:903] String | poplevol |
| MarkovianList2o [712:903] String | alope |
| MarkovianList2o [712:903] String | knortna |
| MarkovianList2o [712:903] String | spac |
| MarkovianList2o [712:903] String | fludica |
| MarkovianList2o [712:903] String | homen |
| MarkovianList2o [712:903] String | ime |
| MarkovianList2o [712:903] String | wortica |
| MarkovianList2o [712:903] String | fir |
| MarkovianList2o [712:903] String | coleve |
| MarkovianList2o [712:903] String | tedin |
| MarkovianList2o [712:903] String | culat |
| MarkovianList2o [712:903] String | treented |
| MarkovianList2o [712:903] String | abourad |
| MarkovianList2o [712:903] String | couni |
| MarkovianList2o [712:903] String | comence |
| MarkovianList2o [712:903] String | havembel |
| MarkovianList2o [712:903] String | enother |
| MarkovianList2o [712:903] String | amer |

Combining methods

For different purposes different kinds of generators will be suitable. We have created two types of generators, each type with two implementations. On

\textsuperscript{4}http://en.wikipedia.org/wiki/The_United_States
the case of the Markov chain generators we made a generator of the first
order, and later we improved it by making it second order, giving it a third
dimension to its data matrix. An improvement that help generate much
better results. The two word-based generators actually consist of the exact
same functions. Between them the big difference is the data we load.

Below I have made use of both techniques along with some manual writ-
ing to generate a sequence of a fight commentary. Of course all this could be
generated, why stop at words or expressions when we could generate whole
sentences? Since time is limited we stop here, and even with the manual
insertion the following output can still be considered generated natural lan-
guage.

Listing 5: Code for Combined Text Generator

```objective-c
NSString *player1 = [m2 generateString];
NSString *player2 = [m2 generateString];

NSLog(@"n

. . . Round %i! %@ opens with a %@ and
follows with a %@. %@ replies with a swiftly
performed %@, but is overrun by a %@, a %@ and four
consecutive %@s. One gently placed %@ by %@ and %@
is down for the count!\n\n", rand() % 10, player1,
[tg generateString], [tg generateString], player2,
[tg generateString], [tg generateString], [tg
generateString], [tg generateString], [tg
generateString], player1, player2);
```

Listing 6: Output from Combined Text Generator

```
2012−02−24 16:37:23.133 TextGenerator [2162:903]

... Round 1! unit opens with a inverse twister and
follows with a half chop. reash replies with a
swiftly performed half chop, but is overrun by a
reverse chop, a inverse trapezium nelson and four
consecutive inverse locks. One gently placed
inverse lock by unit and reash is down for the
count!
```

A great deal of content can be created with these techniques, especially
when they work together. In a roleplaying game a category of quest or
mission can be created with randomized parameters, and techniques like the
ones analyzed in this chapter can be used to present to the player her new
goals and targets in a more colorful way. Not necessarily does everything
in a game or application have to be named using these techniques, but they
can also work as a great source of inspiration. Ask the generators for a
thousand strings and read through while having a cup of coffee.
Biomorphs

For this chapter the research subject has been provided by Richard Dawkins. In his book The Blind Watchmaker\(^5\) Richard Dawkins describes a computer simulation with its most important element being a virtual creature with a finite and very simplistic set of genes. These creatures are referred to sometimes as bodies, or trees (because of their appearance), but mainly they’re called biomorphs.

Analysis

The idea behind Dawkins’ biomorph is simple but the implementation can be made quite extensive. The above shown image is from The Blind Watchmaker and it shows a very basic implementation of the biomorph algorithm. Depicted are nine biomorphs. In the middle, the parent (sometimes referred to as Eve) and around it its offspring, all mutated from the parent. Their genome consists of a set of integers, and these decide how the biomorphs are drawn. The mutations that have taken place have been somewhat controlled in this implementation. As opposed to most implementations here only one gene is modified. By having a look at the second row which shows the parent accompanied by two biomorphs that have had gene number five decreased and increased respectively, it is not far-fetched to draw the conclusion that

gene five controls vertical scale in some manner, perhaps the angle of some of the branches. Looking at column two it seems like gene number nine controls depth of recursion.

It is not necessary to go into more detail about what the specific genes do, it is not important. As stated in The Blind Watchmaker “Genes only start to mean something when they are translated, via protein synthesis, into growing rules for a developing embryo.” The same set of genes can of course be translated into totally different things, depending on interpretation. Great! That leaves some space for creative thinking during the implementation. Furthermore, we need to work out how these genes will mutate. To conclude, we need to define the following three sets:

- Genome - a set of values
- A mutation algorithm - rules for how a parents genes are mutated and translated to its offspring
- A drawing algorithm - rules for interpretation of the genome

**A basic implementation**

For a greater range of our algorithm - a greater number of possible mutations - I will use values of type `float` instead of `int`, and round the numbers wherever an integer is needed, for example if I want to control depth of recursion.

Dawkins’ biomorphs are symmetric about a vertical line. This saves memory by cutting down on the number of genes needed, but another reason is Dawkins’ wish to evolve animal-like shapes and according to him “most animal bodies are pretty symmetrical”. This implementation however, in honor of all that is asymmetrical, will have one set of genes for each side of the biomorph. Our set of genes will consist of seven values with a range between 0.1 and 1, and there’s actually not more to the genome than that. So next up: drawing rules, and let’s start with a rule for recursion. A value within our range of [0.1, 1] makes a lot of sense for many things, but not for depth of recursion. Therefore we will let one gene - `genome[0]` if you will - represent proportion of a given maximum number of recursions instead of number of recursions. The rest of our genome will specify the following:

- `genome[1]` - angle between branch and body, to the left
- `genome[2]` - angle to the right
- `genome[3]` - point along body in which to branch out to the left
- `genome[4]` - point for branching out to the right
- `genome[5]` - length of left branch, in proportion to body
Figure 2: Three generations of biomorphs

- genome[6] - length of right branch

So the only thing that’s left to define is the rules of mutation. Let’s say we add to each gene a random generated number with possible range \([-r, r]\), where \(r\) is a constant (of course not letting our values exit our range of \([0.1, 1]\)).

The figure shown above is a series of three screenshots each showing one generation of biomorphs. The first generation (to the left) have been generated in a random manner, which is why they look so different from each other. Here the user is asked to pick a survivor, a biomorph that will be the parent of the second generation. I liked the look of the biomorph in the bottom left corner so I tapped it, thereby making it the parent of a generation and its competitors but a memory. In the middle image you can see that all the biomorphs belong to the same family. Again I played God by picking a new survivor, letting its genes mutate into the third generation (on the right). One could argue that there’s no natural selection going on here, since there’s no interaction between the biomorphs and I’m doing all the selection. My choices are not completely without reason, though. Usually I pick the one most pleasing to my eye, and that could be considered natural selection.

The algorithms in the basic implementation often create beautiful shapes and patterns, within what is actually a pseudorandom range of diversity. Still there are many limiting factors. All biomorphs will be created with two branches or limbs sticking out from their main body, one to the left and one to the right. Depending on the level of recursion these two branches might have their own pairs of branches sticking out of them but in every new pair
there will always be one to the left and one to the right. Furthermore the creatures aren’t just projected in 2D, their whole existence is within two dimensions.

**Another dimension and more freedom**

To give the biomorph some freedom a third dimension is added in the second implementation. There’s not too much work involved. The 2D models of the first implementation consisted of line segments between pairs of vertices, each vertex positioned according to its two coordinates. Let’s hold on to this model for drawing the new biomorphs, and just add one more coordinate to each vertex. Rotation is applied each new frame to make the 3D effect more visible. It is not visible in this report due to the low frame rate of ... paper, but I’ve tried to visualize it by overlapping screenshots differing in time in the figure below.

The new and improved biomorph should also have new possibilities when it comes to evolution, and to limit the limitations of its genome the pair of subbranches per branch will replaced with a set of subbranches of $n$ size with rules for how each new branch will behave compared to the others. We create the following variables:
• **int branches** - the number of subbranches on each branch

• **float verAngle** - the angle between the branch and its parent branch

• **float horAngle** - the rotation about the parent branch

• **float modifier** - a scalar to be applied to the values of each new branch

The variable or gene **modifier** has a similar task to one of the genes in the basic implementation, which decided the length of each new branch in proportion to its parent, although here the scalar is applied to more values, such as the angles and even the starting points of the branches. Furthermore it will be applied a different number of times for the different branches, rendering every branch unique, even though they share some variables. For more vividness genes deciding the colors of the biomorphs are added, of course with rules for evolving.

The figure above shows three generations of biomorphs from the second implementation. Once again in the first generation (to the left) all genes of all biomorphs have been randomized. The middle screenshot depicts the offspring of the biomorph in the top-right corner, out of which the middle-right one then is picked to produce the third generation (to the right).

Biomorphs show a great example of the power of procedural generation. By using techniques that are inspired by nature itself we can have algorithms and data evolve, so to speak, and improve. This can be done with or without human interaction. The implementations of this chapter use human interaction for selection, but what is to say we couldn’t have created
a test procedure to automate that process. If the goal for example was to create a fast algorithm using these techniques, we would have just created an automated test and then let the fastest algorithm continue to evolve.
Pseudo-infinite universe

Elite

Back in 1984 British programmers David Braben and Ian Bell published Elite, a space trading video game set in a procedurally generated universe. The game universe contained eight galaxies each containing 256 planets, and each of these with their own set of properties. Very small parts of this universe were kept on disk and in runtime very small parts of it were kept in memory. The universe was generated using a random seed which in turn generated more seeds for galaxies, and planets etc. Using this technique the programmers were able to create something much bigger than what the system with its storage and memory limitations otherwise would allow.

The techniques used by Braben and Bell were groundbreaking and they allowed for something bigger to fit into something smaller. Today these techniques are used to varying extent in large numbers of games and other digital applications. The jist of it is to from one seed generate new seeds, creating whatever you want to create in the manner of a growing tree, giving each new branch its own seed. When a newly created branch has got its seed, it can be left to work alone. From there it can do whatever it needs to do with its own seed without affecting the ongoing processes in the rest of the tree. This technique has inspired this third and final chapter of this project, in which my goal is to implement my own procedurally generated universe.

Building a universe

What I want to do is pick a block of space anywhere in our new universe and then go and see if there’s anything there. Let’s say that each block of space has the same width, height and depth. Then a unique block can be specified by it’s three starting coordinates along the three axes. From these numbers we can create one single number to use as the seed for this block.

It’s a good idea right now to take the PRNG (pseudo random number generator) into consideration. The PRNG I will use, that of C - rand() - returns an int on request, between zero and the upper limit of an int. It is possible to give the PRNG a seed to start from by a call to srand(unsigned int), and no matter when or where you feed a number a into the PRNG the next call to rand() will generate a number b where b is the subsequent number of a in the feed. The function srand(unsigned int) is key here. It makes it possible for us to activate and deactivate more than one feed of numbers, by storing for each seed the last number generated. Had it not been for this we could have still been able to generate our universe, but it would have had to be in the same order each time. We will let each sun, planet, moon, etc have it’s own seed, and that way we can load and delete however we want into and out of memory.
As seed we have been requested an unsigned int, which gives us 32 bits to play with. For now, let’s give ten to each axis, giving our universe a 1024x1024x1024 cubic units. For now we can spare the remaining two bits. They can be good to have later in case we feel like creating an underverse, or better yet an antiverse, to our universe.

So we pick or generate a triple of coordinates. Our position along the x axis is shifted using multiplication by 1024 raised to power 2, that along the y axis by 1024, and our z coordinate stays as it is. Sum it up and we have our seed, s, as follows:

\[ s = x \cdot 1024^2 + y \cdot 1024^1 + z \cdot 1024^0 \] (1)

That’s all we need! Or ... well, depends on how you see it, but with this seed fed to the PRNG we can get all the data to be used in the construction of our universe. The problem is that we haven’t got the functions to receive the data, to create something with it. To use the semantics of genetics, we have the genotype but not the phenotype.

So let’s build something with all this data, let’s give it meaning by parsing it. Our universe should have suns/stars with revolving planets, and these should have moons in orbit around them. Maybe the the moons will have satellites, and so on. So, a pattern emerges, a kind of hierarchy, and since we’re not going to take into consideration all the laws of physics in this simplified universe, a tree seems to be the best data structure for the job. For simplicity each block of space will contain either one or no star. If a block contains a star then it will be a solar system with a set of bodies arranged in a tree structure, where the sun will be the root. A body’s position and rotation will not be affected by all mass of it’s surroundings, because it wouldn’t fit into this project, but a decent-looking approximation can be made by making each body rotate about it’s parent.

Suns, moons, planets, and the rest of the celestial objects can all fit in a class called Body. When a body is created it should first be given it’s seed, and a couple of variables of information about its parent body. From that point all exterior work is done, for the power of procedural generation in this case comes from within. The exterior caller, or perhaps parent body, can just leave a seed and move on to its other tasks without worrying about keeping track of a new seed. The newly instantiated body can now figure out its details using the seed, such as its radius and rotational speed. A suitable set of instance variables could look like this:

Listing 7: Body class instance variables

```cpp
float radius_of_body;
float distance_to_parent;
float axis_rotation_about_self[3];
float axis_rotation_about_parent[3];
```
float rotational_speed_about_self;
float rotational_speed_about_parent;
float initial_angle_about_self;
float initial_angle_about_parent;

NSString *name;
NSString *type;

NSMutableArray *children;

Most of the variables are quite obvious; a body will have a radius, a
distance to the body about which it will rotate, a pair of rotational axes,
etc. I also added a variable for type, which for the purpose of this project
will suffice being a string. It will simply be assigned from a list of possible
values ordered by size, beginning with red giant, supernova and white dwarf
and ending with asteroid and satellite. Selection is simply made using the
radius of the body. For the name of the body I use the Markov chain text
generator created in the first chapter, initiated with the body's own seed.

Of course there's no end to how many more of these variables you can
add. We could for example add a variable keeping track of a planet’s popu-
lation(if it has one), but for now, let's stick to basics.

Each body will get, during its creation, the chance to create its own
offspring - a set of bodies in orbit. This way each solar system will be
generated recursively, by use of a single initialization method:

Listing 8: Body class initialization method
− (id)initWithParentRadius:(float)radius
    andSeed:(int)seed
    andRecursion:(int)n;

Here each object will be given three parameters. The radius of the
parent body which will be used to determine upper and lower limits for the
generation of all dimensional values such as radius and distance to parent.
The seed will be used for the generation of all values for this body, and finally
to generate new seeds for its offspring. n will decrease for each recursion
and the poor bodies that get it when it hits zero will be denied having
children. Without this limit our universe would probably become slightly
overpopulated by planets and such in some areas, and my iPhone would not
be happy about it.

So we call this method and store the result in memory, and now what?
We have a bunch of so called bodies consisting of a bunch of values in
memory, but we have no way of viewing them. We need a way to visualize our
data, and for this we’ll use OpenGL. For navigation within the application
we will use Apple's standard library of GUI tools.
Figure 5: Graphical user interface

[object name]

Type  -
Radius -
Children -

Warp
Visualization

The screenshot shows the user interface that will be used in the application. It’s not much but it will be enough to visit every corner of our universe. There’s one slider for each spatial axis. These will be used to pick a destination solar system. After picking a triplet of coordinates the user can press Warp to rapidly move to the new system. At this moment all the procedural generation will take place, and the solar system will be created on the go. Typically the result will be a star/sun with zero or more children. This sun will be centered on the screen directly after the jump. Since someone or something has named each and every object in this universe it would be a waste not to put in a label showing the name of the currently centered body, and underneath it there’re some fields for general information. The field named Children will be showing the number of objects that are in direct orbit around the centered object. If the centered object will have any children then the down arrow button will take you to one of them, making the child the new centered object. If you have centered an object that has any siblings, that is that there are more objects with the same parent, then the right arrow button will cycle through these. Upon centering a new object of course the data in the information fields will update.

From here on most of the work is made with OpenGL, or more specifically OpenGL ES which is a lighter version of OpenGL, made specifically for more compact devices such as smart phones. This has nothing to do with procedural generation though, so I will not go into this in any detail. The above triplet of screenshots is from the finished application. To the left a welcome screen, with a short introduction on how to proceed to explore this new little universe. The controls are already there, ready for take-off. In the middle we’re visiting a solar system with a sun named Theries containing, as shown in the interface of the application, a total of 490 objects. Another coordinate vector generates(or takes us to) the system on the right. The total number of objects in this solar system is not shown in this image, because the navigational buttons have been used to travel to one of the subsystems. The interface here informs us that there’s a total of seven objects in this subsystem, which is the size of the tree data structure that has the current object - Amerv - as its root.

One possible addition

The current implementation is made with a PRNG that produces values and takes seeds of type int. I adjusted the input values for this, so that each place in this universe would get a unique seed. There are now 1024 available spots along each axis, which makes for a little over a billion different areas, all generated which a unique seed. The int would actually have granted me another couple of bits for the seeds, but I chose to keep each axis within ten
bits for aesthetic reasons. Our set of over a billion solar systems will make for quite some exploration, and as mentioned earlier, maybe the extra two bits will come in handy at some point.

Would however our set of a billion solar systems feel too plain then all we need to do is make a new PRNG, that for example could work with numbers of type `long` instead. With this PRNG we could actually go from billions to quintillions of unique areas, using the same techniques as those displayed here.
Discussion

There is a great number of chores we perform daily, at work, in school, at home, etc, that can be automated. Some are more obvious than others, for example if I want to invite all my friends to a party via email, I will not write one separate email for each person. I’ll write one email and then I’ll add all recipients in the address field. The need for this feature was probably recognized very early in the life of emailing due to its frequent appearance, and when this happened it probably didn’t take long to implement a solution.

There are some chores however that at first glance seem hard to implement, often because of vague patterns in their execution, or patterns too complex. One area where this is the case is creativity, a fuzzy matter that is hard to map into a simple rule set, probably because often in creativity the unexpected is to be expected (or vice versa). Sending the same email to multiple recipients is a pretty straight-forward task, one that is easily explained to a computer, but if we want to create a program that for example can paint or compose music, then we need to take additional parameters into consideration, and this is where automation becomes procedural generation.

In the first chapter of this report language was analyzed for the creation of new words. Language is not a small subject and neither is it easily dissected, but still there are ways to mimic a language without fully understanding it. The Markov chain text generator doesn’t know anything about grammar, nor does it understand the meaning of any of the words it analyzes, but it manages to output words that feel natural to us humans. This is because we found a useful pattern, in the shape of probability. The word-based generator works in a similar way, by a simplified rule set derived from a complex natural language.

At times creativity also includes iteration, being a sort of trial-and-error process (imagine a writer by his desk, next to him a waste basket full of crinkled up paper). While we in chapter one excluded human interaction entirely from the creation of words and phrases, the product of the second chapter works in a different way. The natural selection of the biomorph evolution is exclusively managed by the user, letting procedural generation create for us a set of biomorphs to choose from. This technique can be used to perform tasks that are doable by humans but too boring or time-consuming, or tasks that are too complex for us to perform, yet where comprehensive results can be generated procedurally.

The creation of a universe in chapter three is an example of the power procedural generation holds when it comes to volume. Billions of unique stars and planets are generated showing the endless (or pseudo-endless) potential of combining different techniques of procedural generation.

Patterns are key. We study creativity, searching for the set of rules that makes it creativity. Where the rules are too advanced to map into a simple process, we look for ways to simplify them. The applications created
during the course of this project, and the techniques analyzed, can all be considered to mimic human creativity. The downside of this is that as long as the applications stay unchanged so will the range of their output, the creativity will be limited. The upside however is that this range, if the algorithms are thought through, can be vast, allowing for volumes of content to be generated in seconds, that a human wouldn’t be able to think up in a lifetime.
References


